This chapter covers a few alternative methods for preconditioning a linear system. These methods are suitable when the desired goal is to maximize parallelism. The simplest approach is the diagonal (or Jacobi) preconditioning. Often, this preconditioner is not very useful, since the number of iterations of the resulting iteration tends to be much larger than the more standard variants, such as ILU or SSOR. When developing parallel preconditioners, one should beware that the benefits of increased parallelism are not outweighed by the increased amount of computations. The main question to ask is whether or not it is possible to find preconditioning techniques that have a high degree of parallelism, as well as good intrinsic qualities.

12.1

As seen in the previous chapter, a limited amount of parallelism can be extracted from the standard preconditioners such as ILU and SSOR. Fortunately, a number of alternative techniques can be developed that are specifically targeted at parallel environments. These are preconditioning techniques that would normally not be used on a standard machine, but only for parallel computers. There are at least three such types of techniques discussed in this chapter. The simplest approach is to use a Jacobi or, even better, a block Jacobi approach. In the simplest case, a Jacobi preconditioner may consist of the diagonal or a block-diagonal of $A$. To enhance performance, these preconditioners can themselves be accelerated by polynomial iterations, i.e., a second level of preconditioning called polynomial preconditioning.

A different strategy altogether is to enhance parallelism by using graph theory algorithms, such as graph-coloring techniques. These consist of coloring nodes such that two adjacent nodes have different colors. The gist of this approach is that all unknowns associated with the same color can be determined simultaneously in the forward and backward sweeps of the ILU preconditioning operation.

Finally, a third strategy uses generalizations of “partitioning” techniques, which can
be put in the general framework of “domain decomposition” approaches. These will be covered in detail in the next chapter.

Algorithms are emphasized rather than implementations. There are essentially two types of algorithms, namely, those which can be termed coarse-grain and those which can be termed fine-grain. In coarse-grain algorithms, the parallel tasks are relatively big and may, for example, involve the solution of small linear systems. In fine-grain parallelism, the subtasks can be elementary floating-point operations or consist of a few such operations. As always, the dividing line between the two classes of algorithms is somewhat blurred.

### 12.2 BLOCK-JACOBI PRECONDITIONERS

Overlapping block-Jacobi preconditioning consists of a general block-Jacobi approach as described in Chapter 4, in which the sets \( S_i \) overlap. Thus, we define the index sets

\[
S_i = \{ j \mid l_i \leq j \leq r_i \}
\]

with

\[
l_i = 1, \\
r_p = n, \\
r_i > l_{i+1}, \quad 1 \leq i \leq p - 1
\]

where \( p \) is the number of blocks. Now use the block-Jacobi method with this particular partitioning, or employ the general framework of additive projection processes of Chapter 5, and use an additive projection method onto the sequence of subspaces

\[
K_i = \text{span}\{V_i\}, \quad V_i = [e_{l_i}, e_{l_i+1}, \ldots, e_{r_i}].
\]

Each of the blocks will give rise to a correction of the form

\[
\xi_i^{(k+1)} = \xi_i^{(k)} + A_i^{-1}V_i^T (b - Ax^{(k)}).
\]

One problem with the above formula is related to the overlapping portions of the \( x \) variables. The overlapping sections will receive two different corrections in general. According to the definition of “additive projection processes” seen in Chapter 5, the next iterate can be defined as

\[
x_{k+1} = x_k + \sum_{i=1}^{p} V_i A_i^{-1} V_i^T r_k
\]

where \( r_k = b - Az_k \) is the residual vector at the previous iteration. Thus, the corrections for the overlapping regions simply are added together. It is also possible to weigh these contributions before adding them up. This is equivalent to redefining (12.1) into

\[
\xi_i^{(k+1)} = \xi_i^{(k)} + D_i A_i^{-1} V_i^T (b - Az_k)
\]

in which \( D_i \) is a nonnegative diagonal matrix of weights. It is typical to weigh a nonoverlapping contribution by one and an overlapping contribution by \( 1/k \) where \( k \) is the number
of times the unknown is represented in the partitioning.

![Diagram of block-Jacobi matrix]

**Figure 12.1** The block-Jacobi matrix with overlapping blocks.

The block-Jacobi iteration is often over- or under-relaxed, using a relaxation parameter \( \omega \). The iteration can be defined in the form

\[
x_{k+1} = x_k + \sum_{i=1}^{p} \omega_i V_i A_i^{-1} V_i^T r_k.
\]

Recall that the residual at step \( k + 1 \) is then related to that at step \( k \) by

\[
r_{k+1} = \left[ I - \sum_{i=1}^{p} \omega_i A_i \left( V_i^T A_i \right)^{-1} V_i^T \right] r_k.
\]

The solution of a sparse linear system is required at each projection step. These systems can be solved by direct methods if the subblocks are small enough. Otherwise, iterative methods may be used. The outer loop accelerator should then be a flexible variant, such as FGMRES, which can accommodate variations in the preconditioners.
12.3 POLYNOMIAL PRECONDITIONERS

In polynomial preconditioning the matrix $M$ is defined by

$$M^{-1} = s(A)$$

where $s$ is a polynomial, typically of low degree. Thus, the original system is replaced by the preconditioned system

$$s(A)x = s(A)b$$

(12.2)

which is then solved by a conjugate gradient-type technique. Note that $s(A)$ and $A$ commute and, as a result, the preconditioned matrix is the same for right or left preconditioning. In addition, the matrix $s(A)$ or $As(A)$ does not need to be formed explicitly since $As(A)v$ can be computed for any vector $v$ from a sequence of matrix-by-vector products.

Initially, this approach was motivated by the good performance of matrix-vector operations on vector computers for long vectors, e.g., the Cyber 205. However, the idea itself is an old one and has been suggested by Stiefel [204] for eigenvalue calculations in the mid 1950s. Next, some of the popular choices for the polynomial $s$ are described.

12.3.1 NEUMANN POLYNOMIALS

The simplest polynomial $s$ which has been used is the polynomial of the Neumann series expansion

$$I + N + N^2 + \cdots + N^s$$

in which

$$N = I - \omega A$$

and $\omega$ is a scaling parameter. The above series comes from expanding the inverse of $\omega A$ using the splitting

$$\omega A = I - (I - \omega A).$$

This approach can also be generalized by using a splitting of the form

$$\omega A = D - (D - \omega A)$$

where $D$ can be the diagonal of $A$ or, more appropriately, a block diagonal of $A$. Then,

$$(\omega A)^{-1} = \left[D(I - (I - \omega D^{-1} A))\right]^{-1}$$

$$= \left[I - (I - \omega D^{-1} A)\right]^{-1} D^{-1}. $$

Thus, setting

$$N = I - \omega D^{-1} A$$
results in the approximate $s$-term expansion
\[(\omega A)^{-1} \approx M^{-1} \equiv [I + N + \cdots + N^s] D^{-1}.\]  
(12.3)

Since $D^{-1} A = \omega^{-1} [I - N]$, note that
\[M^{-1} A = \frac{1}{\omega} [I + N + \cdots + N^s] (I - N)\]
\[= \frac{1}{\omega} (I - N^{s+1}).\]

The matrix operation with the preconditioned matrix can be difficult numerically for large $s$. If the original matrix is Symmetric Positive Definite, then $M^{-1} A$ is not symmetric, but it is self-adjoint with respect to the $D$-inner product; see Exercise 1.

### 12.3.2 CHEBYSHEV POLYNOMIALS

The polynomial $s$ can be selected to be optimal in some sense, and this leads to the use of Chebyshev polynomials. The criterion that is used makes the preconditioned matrix $s(A)A$ as close as possible to the identity matrix in some sense. For example, the spectrum of the preconditioned matrix can be made as close as possible to that of the identity. Denoting by $\sigma(A)$ the spectrum of $A$, and by $P_k$ the space of polynomials of degree not exceeding $k$, the following may be solved.

Find $s \in P_k$ which minimizes:
\[\max_{\lambda \in \sigma(A)} |1 - \lambda s(\lambda)|.\]  
(12.4)

Unfortunately, this problem involves all the eigenvalues of $A$ and is harder to solve than the original problem. Usually, problem (12.4) is replaced by the problem

Find $s \in P_k$ which minimizes:
\[\max_{1 \leq s \leq k} |1 - \lambda s(\lambda)|,\]  
(12.5)

which is obtained from replacing the set $\sigma(A)$ by some continuous set $E$ that encloses it. Thus, a rough idea of the spectrum of the matrix $A$ is needed. Consider first the particular case where $A$ is Symmetric Positive Definite, in which case $E$ can be taken to be an interval $[\alpha, \beta]$ containing the eigenvalues of $A$.

A variation of Theorem 6.4 is that for any real scalar $\gamma$ such with $\gamma \leq \alpha$, the minimum
\[\min_{p \in P_k, p(\gamma) = 1} \max_{t \in [\alpha, \beta]} |p(t)|\]
is reached for the shifted and scaled Chebyshev polynomial of the first kind,
\[\hat{C}_k(t) \equiv C_k \left( 1 + 2 \frac{\alpha - \gamma}{\beta - \alpha} \right).\]
Of interest is the case where $\gamma = 0$ which gives the polynomial
\[
T_k(t) \equiv \frac{1}{\sigma_k} C_k \left( \frac{\beta + \alpha - 2t}{\beta - \alpha} \right) \quad \text{with} \quad \sigma_k \equiv C_k \left( \frac{\beta + \alpha}{\beta - \alpha} \right).
\]
Denote the center and mid-width of the interval $[\alpha, \beta]$, respectively, by
\[
\theta = \frac{\beta + \alpha}{2}, \quad \delta = \frac{\beta - \alpha}{2}.
\]
Using these parameters instead of $\alpha, \beta$, the above expressions then become
\[
T_k(t) \equiv \frac{1}{\sigma_k} C_k \left( \frac{\theta - t}{\delta} \right) \quad \text{with} \quad \sigma_k \equiv C_k \left( \frac{\theta}{\delta} \right).
\]
The three-term recurrence for the Chebyshev polynomials results in the following three-term recurrences:
\[
\sigma_{k+1} = 2 \frac{\theta}{\delta} \sigma_k - \sigma_{k-1}, \quad k = 1, 2, \ldots,
\]
with
\[
\sigma_1 = \frac{\theta}{\delta}, \quad \sigma_0 = 1,
\]
and
\[
T_{k+1}(t) \equiv \frac{1}{\sigma_{k+1}} \left[ \frac{\theta - t}{\delta} \sigma_k T_k(t) - \sigma_{k-1} T_{k-1}(t) \right]
\]
\[
= \frac{\sigma_k}{\sigma_{k+1}} \left[ \frac{\theta - t}{\delta} T_k(t) - \frac{\sigma_{k-1}}{\sigma_k} T_{k-1}(t) \right], \quad k \geq 1,
\]
with
\[
T_1(t) = 1 - \frac{t}{\theta}, \quad T_0(t) = 1.
\]
Define
\[
\rho_k \equiv \frac{\sigma_k}{\sigma_{k+1}}, \quad k = 1, 2, \ldots \quad \text{(12.6)}
\]
Note that the above recurrences can be put together as
\[
\rho_k = \frac{1}{2\sigma_1 - \rho_{k-1}} \quad \text{(12.7)}
\]
\[
T_{k+1}(t) = \rho_k \left[ 2(\sigma_1 - \frac{1}{\delta}) T_k(t) - \rho_{k-1} T_{k-1}(t) \right], \quad k \geq 1. \quad \text{(12.8)}
\]
Observe that formulas (12.7–12.8) can be started at $k = 0$ provided we set $T_{-1} \equiv 0$ and $\rho_{-1} \equiv 0$, so that $\rho_0 = 1/(2\sigma_1)$.

The goal is to obtain an iteration that produces a residual vector of the form $r_{k+1} = T_{k+1}(A)r_0$ where $T_k$ is the polynomial defined by the above recurrence. The difference between two successive residual vectors is given by
\[
r_{k+1} - r_k = (T_{k+1}(A) - T_k(A))r_0.
\]
The identity $1 = (2\sigma_1 - \rho_{k-1})\rho_k$ and the relations (12.8) yield
\[
T_{k+1}(t) - T_k(t) = T_{k+1}(t) - (2\sigma_1 - \rho_{k-1})\rho_k T_k(t)
\]
Define
\[ d_k \equiv x_{k+1} - x_k, \]
and note that \( r_{k+1} - r_k = Ad_k \). If \( x_{k+1} = x_0 + s_k(A)r_0 \), then \( r_{k+1} = (I - As_k(A))r_0 \), and \( d_k = A^{-1}(T_{k+1}(A) - T_k(A))r_0 \). Therefore the relation (12.9) translates into the recurrence,
\[ d_k = \rho_k \left[ \rho_{k-1} d_{k-1} - \frac{2}{\delta} r_k \right]. \]
Finally, the following algorithm is obtained.

**Algorithm 12.1: Chebyshev Acceleration**

1. \( r_0 = b - Ax_0; \sigma_1 = \delta/\theta; \)
2. \( \rho_0 = 1/\sigma_1; d_0 = \frac{1}{\delta} r_0; \)
3. For \( k = 0, \ldots, \) until convergence Do:
   4. \( x_{k+1} = x_k + d_k \)
   5. \( r_{k+1} = r_k - Ad_k \)
   6. \( \rho_{k+1} = (2\sigma_1 - \rho_k)^{-1}; \)
   7. \( d_{k+1} = \rho_{k+1} \rho_k d_k - \frac{2\rho_{k+1}}{\delta} r_{k+1} \)
8. EndDo

Lines 7 and 4 can also be recast into one single update of the form
\[ x_{k+1} = x_k + \rho_k \left[ \rho_{k-1} (x_k - x_{k-1}) - \frac{2}{\delta} (b - Ax_k) \right]. \]

It can be shown that when \( \alpha = \lambda_1 \) and \( \beta = \lambda_N \), the resulting preconditioned matrix minimizes the condition number of the preconditioned matrices of the form \( As(A) \) over all polynomials \( s \) of degree \( \leq k - 1 \). However, when used in conjunction with the Conjugate Gradient method, it is observed that the polynomial which minimizes the total number of Conjugate Gradient iterations is far from being the one which minimizes the condition number. If instead of taking \( \alpha = \lambda_1 \) and \( \beta = \lambda_N \), the interval \( [\alpha, \beta] \) is chosen to be slightly inside the interval \( [\lambda_1, \lambda_N] \), a much faster convergence might be achieved. The true optimal parameters, i.e., those that minimize the number of iterations of the polynomial preconditioned Conjugate Gradient method, are difficult to determine in practice.

There is a slight disadvantage to the approaches described above. The parameters \( \alpha \) and \( \beta \), which approximate the smallest and largest eigenvalues of \( A \), are usually not available beforehand and must be obtained in some dynamic way. This may be a problem mainly because a software code based on Chebyshev acceleration could become quite complex.
To remedy this, one may ask whether the values provided by an application of Gershgorin’s theorem can be used for \( \alpha \) and \( \beta \). Thus, in the symmetric case, the parameter \( \alpha \), which estimates the smallest eigenvalue of \( A \), may be nonpositive even when \( A \) is a positive definite matrix. However, when \( \alpha \leq 0 \), the problem of minimizing (12.5) is not well defined, since it does not have a unique solution due to the non-strict-convexity of the uniform norm. An alternative uses the \( L_2 \)-norm on \( [\alpha, \beta] \) with respect to some weight function \( w(\lambda) \). This “least-squares” polynomials approach is considered next.

### 12.3.3 LEAST-SQUARES POLYNOMIALS

Consider the inner product on the space \( \mathbb{P}_k \):

\[
\langle p, q \rangle = \int_{\alpha}^{\beta} p(\lambda)q(\lambda)w(\lambda)d\lambda
\]

where \( w(\lambda) \) is some non-negative weight function on \( (\alpha, \beta) \). Denote by \( \|p\|_w \) and call \( w \)-norm, the 2-norm induced by this inner product.

We seek the polynomial \( s_{k-1} \) which minimizes

\[
\|1 - \lambda s(\lambda)\|_w
\]

over all polynomials \( s \) of degree \( \leq k - 1 \). Call \( s_{k-1} \) the least-squares iteration polynomial, or simply the least-squares polynomial, and refer to \( R_k(\lambda) \equiv 1 - \lambda s_{k-1}(\lambda) \) as the least-squares residual polynomial. A crucial observation is that the least squares polynomial is well defined for arbitrary values of \( \alpha \) and \( \beta \). Computing the polynomial \( s_{k-1}(\lambda) \) is not a difficult task when the weight function \( w \) is suitably chosen.

**Computation of the least-squares polynomials** There are three ways to compute the least-squares polynomial \( s_k \) defined in the previous section. The first approach is to use an explicit formula for \( R_k \), known as the kernel polynomials formula,

\[
R_k(\lambda) = \frac{\sum_{i=0}^{k} q_i(0)q_i(\lambda)}{\sum_{i=0}^{k} q_i(0)^2}
\]

in which the \( q_i \)'s represent a sequence of polynomials orthogonal with respect to the weight function \( w(\lambda) \). The second approach generates a three-term recurrence satisfied by the residual polynomials \( R_k(\lambda) \). These polynomials are orthogonal with respect to the weight function \( \lambda w(\lambda) \). From this three-term recurrence, we can proceed exactly as for the Chebyshev iteration to obtain a recurrence formula for the sequence of approximate solutions \( x_k \).

Finally, a third approach solves the Normal Equations associated with the minimization of (12.11), namely,

\[
\langle 1 - \lambda s_{k-1}(\lambda), \lambda Q_j(\lambda) \rangle = 0, \ j = 0, 1, 2, \ldots, k - 1
\]

where \( Q_j, j = 1, \ldots, k - 1 \) is any basis of the space \( P_{k-1} \) of polynomials of degree \( \leq k - 1 \).

These three approaches can all be useful in different situations. For example, the first approach can be useful for computing least-squares polynomials of low degree explicitly. For high-degree polynomials, the last two approaches are preferable for their better numer-
ical behavior. The second approach is restricted to the case where $\alpha \geq 0$, while the third is more general.

Since the degrees of the polynomial preconditioners are often low, e.g., not exceeding 5 or 10, we will give some details on the first formulation. Let $q_i(\lambda), i = 0, 1, \ldots, n, \ldots,$ be the orthonormal polynomials with respect to $w(\lambda)$. It is known that the least-squares residual polynomial $R_k(\lambda)$ of degree $k$ is determined by the kernel polynomials formula (12.12). To obtain $s_{k-1}(\lambda)$, simply notice that

$$s_{k-1}(\lambda) = \frac{1 - R_k(\lambda)}{\lambda} = \frac{\sum_{i=0}^{k} q_i(0) t_i(\lambda)}{\sum_{i=0}^{k} q_i(0)^2}, \quad \text{with} \quad (12.13)$$

$$t_i(\lambda) = \frac{q_i(0) - q_i(\lambda)}{\lambda}. \quad (12.14)$$

This allows $s_{k-1}$ to be computed as a linear combination of the polynomials $t_i(\lambda)$. Thus, we can obtain the desired least-squares polynomials from the sequence of orthogonal polynomials $q_i$ which satisfy a three-term recurrence of the form:

$$\beta_{i+1} q_{i+1}(\lambda) = (\lambda - \alpha_i) q_i(\lambda) - \beta_i q_{i-1}(\lambda), \quad i = 1, 2, \ldots.$$ 

From this, the following recurrence for the $t_i$'s can be derived:

$$\beta_{i+1} t_{i+1}(\lambda) = (\lambda - \alpha_i) t_i(\lambda) - \beta_i t_{i-1}(\lambda) + q_i(0), \quad i = 1, 2, \ldots.$$ 

The weight function $w$ is chosen so that the three-term recurrence of the orthogonal polynomials $q_i$ is known explicitly and/or is easy to generate. An interesting class of weight functions that satisfy this requirement is considered next.

**Choice of the weight functions** This section assumes that $\alpha = 0$ and $\beta = 1$. Consider the Jacobi weights

$$w(\lambda) = \lambda^{\mu - 1} (1 - \lambda)^{\nu}, \quad \text{where} \quad \mu > 0 \quad \text{and} \quad \nu \geq -\frac{1}{2}. \quad (12.15)$$

For these weight functions, the recurrence relations are known explicitly for the polynomials that are orthogonal with respect to $w(\lambda)$. This allows the use of any of the methods described in the previous section for computing $s_{k-1}(\lambda)$. Moreover, it has been shown [129] that the preconditioned matrix $A_s(A)$ is Symmetric Positive Definite when $A$ is Symmetric Positive Definite, provided that $\mu - 1 \geq \nu \geq -\frac{1}{2}$.

The following explicit formula for $R_k(\lambda)$ can be derived easily from the explicit expression of the Jacobi polynomials and the fact that $\{R_k\}$ is orthogonal with respect to the weight $\lambda w(\lambda)$:

$$R_k(\lambda) = \sum_{j=0}^{k} \kappa_j^{(k)} (1 - \lambda)^{k-j} (-\lambda)^j \quad (12.16)$$

$$\kappa_j^{(k)} = \binom{k}{j} \prod_{i=0}^{j-1} \frac{k - i + \nu}{i + 1 + \mu}.$$ 

Using (12.13), the polynomial $s_{k-1}(\lambda) = (1 - R_k(\lambda))/\lambda$ can be derived easily “by hand” for small degrees; see Exercise 4.
Example 12.1 As an illustration, we list the least-squares polynomials \( s_k \) for \( k = 1, \ldots, 8 \), obtained for the Jacobi weights with \( \mu = \frac{1}{2} \) and \( \nu = -\frac{1}{2} \). The polynomials listed are for the interval \([0, 4]\) as this leads to integer coefficients. For a general interval \([0, \beta]\), the best polynomial of degree \( k \) is \( s_k(4\lambda/\beta) \). Also, each polynomial \( s_k \) is rescaled by \((3 + 2k)/4\) to simplify the expressions. However, this scaling factor is unimportant if these polynomials are used for preconditioning.

<table>
<thead>
<tr>
<th>( s_k )</th>
<th>( 1 )</th>
<th>( \lambda )</th>
<th>( \lambda^2 )</th>
<th>( \lambda^3 )</th>
<th>( \lambda^4 )</th>
<th>( \lambda^5 )</th>
<th>( \lambda^6 )</th>
<th>( \lambda^7 )</th>
<th>( \lambda^8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>5</td>
<td>(-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_2 )</td>
<td>14</td>
<td>(-7)</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_3 )</td>
<td>55</td>
<td>(-77)</td>
<td>44</td>
<td>(-11)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_4 )</td>
<td>91</td>
<td>(-182)</td>
<td>156</td>
<td>(-65)</td>
<td>13</td>
<td>(-1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_5 )</td>
<td>140</td>
<td>(-378)</td>
<td>450</td>
<td>(-275)</td>
<td>90</td>
<td>(-15)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_6 )</td>
<td>204</td>
<td>(-714)</td>
<td>1122</td>
<td>(-935)</td>
<td>442</td>
<td>(-119)</td>
<td>17</td>
<td>(-1)</td>
<td></td>
</tr>
<tr>
<td>( s_7 )</td>
<td>285</td>
<td>(-1254)</td>
<td>2508</td>
<td>(-2717)</td>
<td>1729</td>
<td>(-665)</td>
<td>152</td>
<td>(-19)</td>
<td>1</td>
</tr>
</tbody>
</table>

We selected \( \mu = \frac{1}{2} \) and \( \nu = -\frac{1}{2} \) only because these choices lead to a very simple recurrence for the polynomials \( q_i \), which are the Chebyshev polynomials of the first kind.

Theoretical considerations An interesting theoretical question is whether the least-squares residual polynomial becomes small in some sense as its degree increases. Consider first the case \( 0 < \alpha < \beta \). Since the residual polynomial \( R_k \) minimizes the norm \( \|R\|_w \) associated with the weight \( w \), over all polynomials \( R \) of degree \( \leq k \) such that \( R(0) = 1 \), the polynomial \((1 - (\lambda/\theta))^{k} \) with \( \theta = (\alpha + \beta)/2 \) satisfies

\[
\|R_k\|_w \leq \left\| \left(1 - \frac{\lambda}{\theta}\right)^{k}\right\|_w \leq \left\| \frac{b - a}{b + a}\right\|_w^{k} = \kappa \left[\frac{\beta - \alpha}{\beta + \alpha}\right]^{k}
\]

where \( \kappa \) is the \( w\)-norm of the function unity on the interval \([\alpha, \beta]\). The norm of \( R_k \) will tend to zero geometrically as \( k \) tends to infinity, provided \( \alpha > 0 \).

Consider now the case \( \alpha = 0, \beta = 1 \) and the Jacobi weight (12.15). For this choice of the weight function, the least-squares residual polynomial is known to be \( p_k(\lambda)/p_k(0) \) where \( p_k \) is the \( k^{th} \) degree Jacobi polynomial associated with the weight function \( w(\lambda) = \lambda^\mu(1 - \lambda)^\nu \). It can be shown that the 2-norm of such a residual polynomial with respect to this weight is given by

\[
\|p_k/p_k(0)\|_{w'}^2 = \frac{\Gamma^2(\mu + 1)\Gamma(k + \nu + 1)}{(2k + \mu + \nu + 1)\Gamma(k + \mu + \nu + 1)} \frac{\Gamma(k + 1)}{\Gamma(k + \mu + 1)}
\]

in which \( \Gamma \) is the Gamma function. For the case \( \mu = \frac{1}{2} \) and \( \nu = -\frac{1}{2} \), this becomes

\[
\|p_k/p_k(0)\|_{w'}^2 = \frac{\Gamma\left(\frac{3}{2}\right)^2}{(2k + 1)(k + \frac{1}{2})} = \frac{\pi}{2(2k + 1)^2}.
\]

Therefore, the \( w\)-norm of the least-squares residual polynomial converges to zero like \( 1/k \) as the degree \( k \) increases (a much slower rate than when \( \alpha > 0 \). However, note that the condition \( p(0) = 1 \) implies that the polynomial must be large in some interval around the
Given a set of approximate eigenvalues of a nonsymmetric matrix $A$, a simple region $E$ can be constructed in the complex plane, e.g., a disk, an ellipse, or a polygon, which encloses the spectrum of the matrix $A$. There are several choices for $E$. The first idea uses an ellipse $E$ that encloses an approximate convex hull of the spectrum. Consider an ellipse centered at $\theta$, and with focal distance $\delta$. Then as seen in Chapter 6, the shifted and scaled Chebyshev polynomials defined by
\[
T_k(\lambda) = \frac{C_k \left( \frac{\theta - \lambda}{\delta} \right)}{C_k \left( \frac{\theta}{\delta} \right)}
\]
are nearly optimal. The use of these polynomials leads again to an attractive three-term recurrence and to an algorithm similar to Algorithm 12.1. In fact, the recurrence is identical, except that the scalars involved can now be complex to accommodate cases where the ellipse has foci not necessarily located on the real axis. However, when $A$ is real, then the symmetry of the foci with respect to the real axis can be exploited. The algorithm can still be written in real arithmetic.

An alternative to Chebyshev polynomials over ellipses employs a polygon $H$ that contains $\sigma(A)$. Polygonal regions may better represent the shape of an arbitrary spectrum. The best polynomial for the infinity norm is not known explicitly but it may be computed by an algorithm known in approximation theory as the Remez algorithm. It may be simpler to use an $L_2$-norm instead of the infinity norm, i.e., to solve (12.11) where $w$ is some weight function defined on the boundary of the polygon $H$.

Now here is a sketch of an algorithm based on this approach. We use an $L_2$-norm associated with Chebyshev weights on the edges of the polygon. If the contour of $H$ consists of $k$ edges each with center $\theta_i$ and half-length $\delta_i$, then the weight on each edge is defined by
\[
w_i(\lambda) = \frac{2}{\pi} |\delta_i - (\lambda - \theta_i)^2|^{-1/2}, \quad i = 1, \ldots, k.
\]  
Using the power basis to express the best polynomial is not a safe practice. It is preferable to use the Chebyshev polynomials associated with the ellipse of smallest area containing $H$. With the above weights or any other Jacobi weights on the edges, there is a finite procedure which does not require numerical integration to compute the best polynomial. To do this, each of the polynomials of the basis (namely, the Chebyshev polynomials associated with the ellipse of smallest area containing $H$) must be expressed as a linear combination of the Chebyshev polynomials associated with the different intervals $[\theta_i - \delta_i, \theta_i + \delta_i]$. This redundancy allows exact expressions for the integrals involved in computing the least-squares solution to (12.11).

Next, the main lines of a preconditioned GMRES algorithm are described based on least-squares polynomials. Eigenvalue estimates are obtained from a GMRES step at the beginning of the outer loop. This GMRES adaptive corrects the current solution and the eigenvalue estimates are used to update the current polygon $H$. Correcting the solution at this stage is particularly important since it often results in a few orders of magnitude
improvement. This is because the polygon \( H \) may be inaccurate and the residual vector is dominated by components in one or two eigenvectors. The GMRES step will immediately annihilate those dominating components. In addition, the eigenvalues associated with these components will now be accurately represented by eigenvalues of the Hessenberg matrix.

**Algorithm 12.2: Polynomial Preconditioned GMRES**

1. **Start or Restart:**
2. Compute current residual vector \( r := b - Ax \).
3. **Adaptive GMRES step:**
4. Run \( m_1 \) steps of GMRES for solving \( Ad = r \).
5. Update \( x \) by \( x := x + d \).
7. Compute new polynomial.
8. Compute new polynomial:
9. Refine \( H \) from previous hull \( H \) and new eigenvalue estimates.
10. Get new best polynomial \( s_k \).
11. **Polynomial Iteration:**
12. Compute the current residual vector \( r = b - Ax \).
13. Run \( m_2 \) steps of GMRES applied to \( s_k (A) Ad = s_k (A)r \).
14. Update \( x \) by \( x := x + d \).
15. Test for convergence.
16. If solution converged then Stop; else GoTo 1.

**Example 12.2** Table 12.1 shows the results of applying GMRES(20) with polynomial preconditioning to the first four test problems described in Section 3.7.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Iters</th>
<th>Kflops</th>
<th>Residual</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2DA</td>
<td>56</td>
<td>2774</td>
<td>0.22E-05</td>
<td>0.51E-06</td>
</tr>
<tr>
<td>F3D</td>
<td>22</td>
<td>7203</td>
<td>0.18E-05</td>
<td>0.22E-05</td>
</tr>
<tr>
<td>ORS</td>
<td>78</td>
<td>4454</td>
<td>0.16E-05</td>
<td>0.32E-08</td>
</tr>
<tr>
<td>F2DB</td>
<td>100</td>
<td>4432</td>
<td>0.47E-05</td>
<td>0.19E-05</td>
</tr>
</tbody>
</table>

Table 12.1 A test run of ILU(0)-GMRES accelerated with polynomial preconditioning.

See Example 6.1 for the meaning of the column headers in the table. In fact, the system is preconditioned by ILU(0) before polynomial preconditioning is applied to it. Degree 10 polynomials (maximum) are used. The tolerance for stopping is \( 10^{-7} \). Recall that \( \text{Iters} \) is the number of matrix-by-vector products rather than the number of GMRES iterations. Notice that, for most cases, the method does not compare well with the simpler ILU(0) example seen in Chapter 10. The notable exception is example F2DB for which the method converges fairly fast in contrast with the simple ILU(0)-GMRES; see Example 10.2. An attempt to use the method for the fifth matrix in the test set, namely, the FIDAP matrix FID, failed because the matrix has eigenvalues on both sides of the imaginary axis and the code tested does not handle this situation.
It is interesting to follow the progress of the algorithm in the above examples. For the first example, the coordinates of the vertices of the upper part of the first polygon \( H \) are

<table>
<thead>
<tr>
<th>( \Re(c_i) )</th>
<th>( \Im(c_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.06492</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.17641</td>
<td>0.02035</td>
</tr>
<tr>
<td>0.29340</td>
<td>0.03545</td>
</tr>
<tr>
<td>0.62858</td>
<td>0.04977</td>
</tr>
<tr>
<td>1.18052</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

This hull is computed from the 20 eigenvalues of the \( 20 \times 20 \) Hessenberg matrix resulting from the first run of GMRES(20). In the ensuing GMRES loop, the outer iteration converges in three steps, each using a polynomial of degree 10, i.e., there is no further adaptation required. For the second problem, the method converges in the 20 first steps of GMRES, so polynomial acceleration was never invoked. For the third example, the initial convex hull found is the interval \([0.06319, 1.67243]\) of the real line. The polynomial preconditioned GMRES then converges in five iterations. Finally, the initial convex hull found for the last example is

<table>
<thead>
<tr>
<th>( \Re(c_i) )</th>
<th>( \Im(c_i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.17131</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.39337</td>
<td>0.10758</td>
</tr>
<tr>
<td>1.43826</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

and the outer loop converges again without another adaptation step, this time in seven steps.

**MULTICOLORING**

**12.4**

The general idea of multicoloring, or graph coloring, has been used for a long time by numerical analysts. It was exploited, in particular, in the context of relaxation techniques both for understanding their theory and for deriving efficient algorithms. More recently, these techniques were found to be useful in improving parallelism in iterative solution techniques. This discussion begins with the 2-color case, called *red-black* ordering.

**12.4.1 RED-BLACK ORDERING**

The problem addressed by multicoloring is to determine a coloring of the nodes of the adjacency graph of a matrix such that any two adjacent nodes have different colors. For a 2-dimensional finite difference grid (5-point operator), this can be achieved with two
colors, typically referred to as “red” and “black.” This red-black coloring is illustrated in Figure 12.2 for a $6 \times 4$ mesh where the black nodes are represented by filled circles.

![Red-black coloring of a $6 \times 4$ grid. Natural labeling of the nodes.](image)

Assume that the unknowns are labeled by listing the red unknowns first together, followed by the black ones. The new labeling of the unknowns is shown in Figure 12.3.

![Red-black coloring of a $6 \times 4$ grid. Red-black labeling of the nodes.](image)

Since the red nodes are not coupled with other red nodes and, similarly, the black nodes are not coupled with other black nodes, the system that results from this reordering will have the structure

$$
\begin{pmatrix}
D_1 & F \\
E & D_2
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2
\end{pmatrix}
=
\begin{pmatrix}
b_1 \\
b_2
\end{pmatrix},
$$

(12.18)
in which $D_1$ and $D_2$ are diagonal matrices. The reordered matrix associated with this new labeling is shown in Figure 12.4.

Two issues will be explored regarding red-black ordering. The first is how to exploit this structure for solving linear systems. The second is how to generalize this approach for systems whose graphs are not necessarily 2-colorable.
The easiest way to exploit the red-black ordering is to use the standard SSOR or ILU(0) preconditioners for solving the block system (12.18) which is derived from the original system. The resulting preconditioning operations are highly parallel. For example, the linear system that arises from the forward solve in SSOR will have the form

\[
\begin{pmatrix}
D_1 & O \\
E & D_2
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2
\end{pmatrix} = \begin{pmatrix} b_1 \\
b_2
\end{pmatrix}.
\]

This system can be solved by performing the following sequence of operations:

1. Solve \( D_1 x_1 = b_1 \).
2. Compute \( b_2 := b_2 - E x_1 \).
3. Solve \( D_2 x_2 = b_2 \).

This consists of two diagonal scalings (operations 1 and 3) and a sparse matrix-by-vector product. Therefore, the degree of parallelism, is at least \( n/2 \) if an atomic task is considered to be any arithmetic operation. The situation is identical with the ILU(0) preconditioning. However, since the matrix has been reordered before ILU(0) is applied to it, the resulting LU factors are not related in any simple way to those associated with the original matrix. In fact, a simple look at the structure of the ILU factors reveals that many more elements are dropped with the red-black ordering than with the natural ordering. The result is that the number of iterations to achieve convergence can be much higher with red-black ordering than with the natural ordering.

A second method that has been used in connection with the red-black ordering solves the reduced system which involves only the black unknowns. Eliminating the red unknowns from (12.18) results in the reduced system:

\[(D_2 - ED_1^{-1}F)x_2 = b_2 - ED_1^{-1}b_1.\]
Note that this new system is again a sparse linear system with about half as many unknowns. In addition, it has been observed that for “easy problems,” the reduced system can often be solved efficiently with only diagonal preconditioning. The computation of the reduced system is a highly parallel and inexpensive process. Note that it is not necessary to form the reduced system. This strategy is more often employed when $D_1$ is not diagonal, such as in domain decomposition methods, but it can also have some uses in other situations. For example, applying the matrix to a given vector $x$ can be performed using nearest-neighbor communication, and this can be more efficient than the standard approach of multiplying the vector by the Schur complement matrix $D_2 - ED_1^{-1}F$. In addition, this can save storage, which may be more critical in some cases.

### 12.4.3 Multicoloring for General Sparse Matrices

Chapter 3 discussed a general greedy approach for multicoloring a graph. Given a general sparse matrix $A$, this inexpensive technique allows us to reorder it into a block form where the diagonal blocks are diagonal matrices. The number of blocks is the number of colors. For example, for six colors, a matrix would result with the structure shown in Figure 12.5 where the $D_i$'s are diagonal and $E, F$ are general sparse. This structure is obviously a generalization of the red-black ordering.

![Figure 12.5 A six-color ordering of a general sparse matrix.](image)

Just as for the red-black ordering, ILU(0), SOR, or SSOR preconditioning can be used on this reordered system. The parallelism of SOR/SSOR is now of order $n/p$ where $p$ is the number of colors. A loss in efficiency may occur since the number of iterations is likely to increase.

A Gauss-Seidel sweep will essentially consist of $p$ scalings and $p - 1$ matrix-by-vector products, where $p$ is the number of colors. Specifically, assume that the matrix is stored in the well known Ellpack-Itpack format and that the block structure of the permuted matrix is defined by a pointer array $iptr$. The index $iptr(j)$ is the index of the first row in the $j$-th block. Thus, the pair $A(n1 : n2, *)$, $JA(n1 : n2, *)$ represents the sparse matrix consisting of the rows $n1$ to $n2$ in the Ellpack-Itpack format. The main diagonal of $A$ is assumed to
be stored separately in inverted form in a one-dimensional array $\text{diag}$. One single step of the multicolor SOR iteration will then take the following form.

**ALGORITHM 12.3: Multicolor SOR Sweep in the Ellpack Format**

1. Do $col = 1, ncol$
2. $n1 = \text{iptr}(col)$
3. $n2 = \text{iptr}(col+1) - 1$
4. $y(n1:n2) = \text{rhs}(n1:n2)$
5. Do $j = 1, n\text{diag}$
6. Do $i = n1, n2$
7. $y(i) = y(i) - a(i,j)*y(ja(i,j))$
8. EndDo
9. EndDo
10. $y(n1:n2) = \text{diag}(n1:n2) * y(n1:n2)$
11. EndDo

In the above algorithm, $ncol$ is the number of colors. The integers $n1$ and $n2$ set in lines 2 and 3 represent the beginning and the end of block $col$. In line 10, $y(n1:n2)$ is multiplied by the diagonal $D^{-1}$ which is kept in inverted form in the array $\text{diag}$. The outer loop, i.e., the loop starting in line 1, is sequential. The loop starting in line 6 is vectorizable/parallelizable. There is additional parallelism which can be extracted in the combination of the two loops starting in lines 5 and 6.

**MULTI-ELIMINATION ILU**

**12.5**

The discussion in this section begins with the Gaussian elimination algorithm for a general sparse linear system. Parallelism in sparse Gaussian elimination can be obtained by finding unknowns that are independent at a given stage of the elimination, i.e., unknowns that do not depend on each other according to the binary relation defined by the graph of the matrix. A set of unknowns of a linear system which are independent is called an independent set. Thus, independent set orderings can be viewed as permutations to put the original matrix in the form

$$
\begin{pmatrix}
D & E \\
F & C
\end{pmatrix}
$$

in which $D$ is diagonal, but $C$ can be arbitrary. This amounts to a less restrictive form of multicoloring, in which a set of vertices in the adjacency graph is found so that no equation in the set involves unknowns from the same set. A few algorithms for finding independent set orderings of a general sparse graph were discussed in Chapter 3.

The rows associated with an independent set can be used as pivots simultaneously. When such rows are eliminated, a smaller linear system results, which is again sparse. Then we can find an independent set for this reduced system and repeat the process of
reduction. The resulting second reduced system is called the second-level reduced system. The process can be repeated recursively a few times. As the level of the reduction increases, the reduced systems gradually lose their sparsity. A direct solution method would continue the reduction until the reduced system is small enough or dense enough to switch to a dense Gaussian elimination to solve it. This process is illustrated in Figure 12.6. There exists a number of sparse direct solution techniques based on this approach.

Figure 12.6 Illustration of two levels of multi-elimination for sparse linear systems.

After a brief review of the direct solution method based on independent set orderings, we will explain how to exploit this approach for deriving incomplete LU factorizations by incorporating drop tolerance strategies.

12.5.1 MULTI-ELIMINATION

We start by a discussion of an exact reduction step. Let \( A_j \) be the matrix obtained at the \( j \)-th step of the reduction, \( j = 0, \ldots, nlev \) with \( A_0 = A \). Assume that an independent set ordering is applied to \( A_j \) and that the matrix is permuted accordingly as follows:

\[
P_jA_jP_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix}
\]  

(12.20)

where \( D_j \) is a diagonal matrix. Now eliminate the unknowns of the independent set to get the next reduced matrix,

\[
A_{j+1} = C_j - E_jD_j^{-1}F_j.
\]  

(12.21)

This results, implicitly, in a block LU factorization

\[
P_jA_jP_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix} = \begin{pmatrix} I & O \\ E_jD_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ O & A_{j+1} \end{pmatrix}
\]

with \( A_{j+1} \) defined above. Thus, in order to solve a system with the matrix \( A_j \), both a forward and a backward substitution need to be performed with the block matrices on the right-hand side of the above system. The backward solution involves solving a system with the matrix \( A_{j+1} \).

This block factorization approach can be used recursively until a system results that is small enough to be solved with a standard method. The transformations used in the elimination process, i.e., the matrices \( E_jD_j^{-1} \) and the matrices \( F_j \) must be saved. The permutation
matrices $P_j$ can also be saved. Alternatively, the matrices involved in the factorization at each new reordering step can be permuted explicitly.

12.5.2 ILUM

The successive reduction steps described above will give rise to matrices that become more and more dense due to the fill-ins introduced by the elimination process. In iterative methods, a common cure for this is to neglect some of the fill-ins introduced by using a simple dropping strategy as the reduced systems are formed. For example, any fill-in element introduced is dropped, whenever its size is less than a given tolerance times the 2-norm of the original row. Thus, an “approximate” version of the successive reduction steps can be used to provide an approximate solution $M^{-1}v$ to $A^{-1}v$ for any given $v$. This can be used to precondition the original linear system. Conceptually, the modification leading to an “incomplete” factorization replaces (12.21) by

$$A_{j+1} = (C_j - E_jD_j^{-1}F_j) - R_j$$

(12.22)

in which $R_j$ is the matrix of the elements that are dropped in this reduction step. Globally, the algorithm can be viewed as a form of incomplete block LU with permutations.

Thus, there is a succession of block ILU factorizations of the form

$$P_jA_jP_j^T = \begin{pmatrix} D_j & F_j \\ E_j & C_j \end{pmatrix}$$

$$= \begin{pmatrix} I & 0 \\ E_jD_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ O & A_{j+1} \end{pmatrix} + \begin{pmatrix} O & O \\ O & R_j \end{pmatrix}$$

with $A_{j+1}$ defined by (12.22). An independent set ordering for the new matrix $A_{j+1}$ will then be found and this matrix is reduced again in the same manner. It is not necessary to save the successive $A_j$ matrices, but only the last one that is generated. We need also to save the sequence of sparse matrices

$$B_{j+1} = \begin{pmatrix} D_j & F_j \\ E_jD_j^{-1} & O \end{pmatrix}$$

(12.23)

which contain the transformation needed at level $j$ of the reduction. The successive permutation matrices $P_j$ can be discarded if they are applied to the previous $B_j$ matrices as soon as these permutation matrices are known. Then only the global permutation is needed, which is the product of all these successive permutations.

An illustration of the matrices obtained after three reduction steps is shown in Figure 12.7. The original matrix is a 5-point matrix associated with a $15 \times 15$ grid and is therefore of size $N = 225$. Here, the successive matrices $B_j$ (with permutations applied) are shown together with the last $A_j$ matrix which occupies the location of the $O$ block in (12.23).
Illustration of the processed matrices obtained from three steps of independent set ordering and reductions.

We refer to this incomplete factorization as ILUM (ILU with Multi-Elimination). The preprocessing phase consists of a succession of \( n \text{lev} \) applications of the following three steps: (1) finding the independent set ordering, (2) permuting the matrix, and (3) reducing it.

**ALGORITHM 12.4: ILUM: Preprocessing Phase**

1. Set \( A_0 = A \).
2. For \( j = 0, 1, \ldots, n \text{lev} - 1 \) Do:
3. Find an independent set ordering permutation \( P_j \) for \( A_j \);
4. Apply \( P_j \) to \( A_j \) to permute it into the form (12.20);
5. Apply \( P_j \) to \( B_1, \ldots, B_j \);
6. Apply \( P_j \) to \( P_0, \ldots, P_{j-1} \);
7. Compute the matrices \( A_{j+1} \) and \( B_{j+1} \) defined by (12.22) and (12.23).
8. EndDo

In the backward and forward solution phases, the last reduced system must be solved but not necessarily with high accuracy. For example, we can solve it according to the level of tolerance allowed in the dropping strategy during the preprocessing phase. Observe that if the linear system is solved inaccurately, only an accelerator that allows variations in the preconditioning should be used. Such algorithms have been discussed in Chapter 9. Alternatively, we can use a fixed number of multicolor SOR or SSOR steps or a fixed polynomial iteration. The implementation of the ILUM preconditioner corresponding to
this strategy is rather complicated and involves several parameters.

In order to describe the forward and backward solution, we introduce some notation. We start by applying the “global permutation,” i.e., the product

\[ P_{rd} \cdot e_{v-1} \cdot P_{rd} \cdot e_{v-2} \cdots P_0 \]

to the right-hand side. We overwrite the result on the current solution vector, an \( N \)-vector called \( x_0 \). Now partition this vector into

\[ x_0 = \begin{pmatrix} y_0 \\ x_1 \end{pmatrix} \]

according to the partitioning (12.20). The forward step consists of transforming the second component of the right-hand side as

\[ x_1 := x_1 - E_0 D_0^{-1} y_0. \]

Now \( x_1 \) is partitioned in the same manner as \( x_0 \) and the forward elimination is continued the same way. Thus, at each step, each \( x_j \) is partitioned as

\[ x_j = \begin{pmatrix} y_j \\ x_{j+1} \end{pmatrix}. \]

A forward elimination step defines the new \( x_{j+1} \) using the old \( x_{j+1} \) and \( y_j \) for \( j = 0, \ldots, nlev - 1 \) while a backward step defines \( y_j \) using the old \( y_j \) and \( x_{j+1} \), for \( j = nlev - 1, \ldots, 0 \). Algorithm 12.5 describes the general structure of the forward and backward solution sweeps. Because the global permutation was applied at the beginning, the successive permutations need not be applied. However, the final result obtained must be permuted back into the original ordering.

**Algorithm 12.5: ILUM: Forward and Backward Solutions**

1. Apply global permutation to right-hand-side \( b \) and copy into \( x_0 \).
2. For \( j = 0, 1, \ldots, nlev - 1 \) Do: [Forward sweep]
   3. \( x_{j+1} := x_{j+1} - E_j D_j^{-1} y_j \)
   4. EndDo
3. Solve with a relative tolerance \( \epsilon \):
   5. For \( j = nlev - 1, \ldots, 1, 0 \) Do: [Backward sweep]
   6. \( y_j := D_j^{-1} (y_j - F_j x_{j+1}) \)
   7. EndDo
4. Permute the resulting solution vector back to the original
5. ordering to obtain the solution \( x \).

Computer implementations of ILUM can be rather tedious. The implementation issues are similar to those of parallel direct-solution methods for sparse linear systems.
This section describes parallel variants of the block Successive Over-Relaxation (BSOR) and ILU(0) preconditioners which are suitable for distributed memory environments. Chapter 11 briefly discussed distributed sparse matrices. A distributed matrix is a matrix whose entries are located in the memories of different processors in a multiprocessor system. These types of data structures are very convenient for distributed memory computers and it is useful to discuss implementations of preconditioners that are specifically developed for them. Refer to Section 11.5.6 for the terminology used here. In particular, the term subdomain is used in the very general sense of subgraph. For both ILU and SOR, multicoloring or level scheduling can be used at the macro level, to extract parallelism. Here, macro level means the level of parallelism corresponding to the processors, or blocks, or subdomains.

12.6.1 DISTRIBUTED SPARSE MATRICES

In the ILU(0) factorization, the LU factors have the same nonzero patterns as the original matrix $A$, so that the references of the entries belonging to the external subdomains in the ILU(0) factorization are identical with those of the matrix-by-vector product operation with the matrix $A$. This is not the case for the more accurate ILU($p$) factorization, with $p > 0$. If an attempt is made to implement a wavefront ILU preconditioner on a distributed memory computer, a difficulty arises because the natural ordering for the original sparse problem may put an unnecessary limit on the amount of parallelism available. Instead, a two-level ordering is used. First, define a “global” ordering which is a wavefront ordering for the subdomains. This is based on the graph which describes the coupling between the subdomains: Two subdomains are coupled if and only if they contain at least a pair of coupled unknowns, one from each subdomain. Then, within each subdomain, define a local ordering.

To describe the possible parallel implementations of these ILU(0) preconditioners, it is sufficient to consider a local view of the distributed sparse matrix, illustrated in Figure 12.8. The problem is partitioned into $p$ subdomains or subgraphs using some graph partitioning technique. This results in a mapping of the matrix into processors where it is assumed that the $i$-th equation (row) and the $i$-th unknown are mapped to the same processor. We distinguish between interior points and interface points. The interior points are those nodes that are not coupled with nodes belonging to other processors. Interface nodes are those local nodes that are coupled with at least one node which belongs to another processor. Thus, processor number 10 in the figure holds a certain number of rows that are local rows. Consider the rows associated with the interior nodes. The unknowns associated with these nodes are not coupled with variables from other processors. As a result, the rows associated with these nodes can be eliminated independently in the ILU(0) process. The rows associated with the nodes on the interface of the subdomain will require more attention. Recall that an ILU(0) factorization is determined entirely by the order in which the rows are processed. The interior nodes can be eliminated first. Once this is done, the interface
rows can be eliminated *in a certain order*. There are two natural choices for this order. The first would be to impose a global order based on the labels of the processors. Thus, in the illustration, the interface rows belonging to Processors 2, 4, and 6 are processed before those in Processor 10. The interface rows in Processor 10 must in turn be processed before those of Processors 13 and 14. The local order, i.e., the order in which we process the interface rows in the same processor (e.g. Processor 10), may not be as important. This global order based on PE-number defines a natural priority graph and parallelism can be exploited easily in a data-driven implementation.

![Diagram of processor labels and interface points](image)

*Figure 12.8 A local view of the distributed ILU(0).*

It is somewhat unnatural to base the ordering just on the processor labeling. Observe that a proper order can also be defined for performing the elimination by *replacing the PE-numbers with any labels, provided that any two neighboring processors have a different label*. The most natural way to do this is by performing a multicoloring of the subdomains, and using the colors in exactly the same way as before to define an order of the tasks. The algorithms will be written in this general form, i.e., with a label associated with each processor. Thus, the simplest valid labels are the PE numbers, which lead to the PE-label-based order. In the following, we define $Lab_j$ as the label of Processor number $j$.

**Algorithm 12.6:** Distributed ILU(0) factorization

1. In each processor $P_i$, $i = 1, \ldots, p$ Do:
2. Perform the ILU(0) factorization for interior local rows.
3. Receive the factored rows from the adjacent processors $j$ with $Lab_j < Lab_i$.
4. Perform the ILU(0) factorization for the interface rows with pivots received from the external processors in step 3.
5. Perform the ILU(0) factorization for the boundary nodes, with pivots from the interior rows completed in step 2.
6. Send the completed interface rows to adjacent processors $j$ with
Step 2 of the above algorithm can be performed in parallel because it does not depend on data from other subdomains. Once this distributed ILU(0) factorization is completed, the preconditioned Krylov subspace algorithm will require a forward and backward sweep at each step. The distributed forward/backward solution based on this factorization can be implemented as follows.

**ALGORITHM 12.7: Distributed Forward and Backward Sweep**

1. In each processor $P_i$, $i = 1, \ldots, p$ Do:
2. Forward solve:
3. Perform the forward solve for the interior nodes.
4. Receive the updated values from the adjacent processors $j$ with $L_{ab_j} < L_{ab_i}$.
5. Perform the forward solve for the interface nodes.
6. Send the updated values of boundary nodes to the adjacent processors $j$ with $L_{ab_j} > L_{ab_i}$.
7. Backward solve:
8. Receive the updated values from the adjacent processors $j$ with $L_{ab_j} > L_{ab_i}$.
9. Perform the backward solve for the boundary nodes.
10. Send the updated values of boundary nodes to the adjacent processors, $j$ with $L_{ab_j} < L_{ab_i}$.
11. Perform the backward solve for the interior nodes.
12. EndDo

As in the ILU(0) factorization, the interior nodes do not depend on the nodes from the external processors and can be computed in parallel in lines 3 and 15. In the forward solve, the solution of the interior nodes is followed by an exchange of data and the solution on the interface. The backward solve works in reverse in that the boundary nodes are first computed, then they are sent to adjacent processors. Finally, interior nodes are updated.

**OTHER TECHNIQUES**

This section gives a brief account of other parallel preconditioning techniques which are sometimes used. The next chapter also examines another important class of methods, which were briefly mentioned before, namely, the class of Domain Decomposition methods.
12.7.1 APPROXIMATE INVERSES

Another class of preconditioners that require only matrix-by-vector products, is the class of approximate inverse preconditioners. Discussed in Chapter 10, these can be used in many different ways. Besides being simple to implement, both their preprocessing phase and iteration phase allow a large degree of parallelism. Their disadvantage is similar to polynomial preconditioners, namely, the number of steps required for convergence may be large, possibly substantially larger than with the standard techniques. On the positive side, they are fairly robust techniques which can work well where standard methods may fail.

12.7.2 ELEMENT-BY-ELEMENT TECHNIQUES

A somewhat specialized set of techniques is the class of Element-By-Element (EBE) preconditioners which are geared toward finite element problems and are motivated by the desire to avoid assembling finite element matrices. Many finite element codes keep the data related to the linear system in unassembled form. The element matrices associated with each element are stored and never added together. This is convenient when using direct methods since there are techniques, known as frontal methods, that allow Gaussian elimination to be performed by using a few elements at a time.

It was seen in Chapter 2 that the global stiffness matrix $A$ is the sum of matrices $A^{[e]}$ associated with each element, i.e.,

$$A = \sum_{e=1}^{\text{Net}} A^{[e]}.$$  

Here, the matrix $A^{[e]}$ is an $n \times n$ matrix defined as

$$A^{[e]} = P_e A_K e P_e^T$$

in which $A_K$ is the element matrix and $P_e$ is a Boolean connectivity matrix which maps the coordinates of the small $A_K$ matrix into those of the full matrix $A$. Chapter 2 showed how matrix-by-vector products can be performed in unassembled form. To perform this product in parallel, note that the only potential obstacle to performing the matrix-by-vector product in parallel, i.e., across all elements, is in the last phase, i.e., when the contributions are summed to the resulting vector $y$. In order to add the contributions $A^{[e]} x$ in parallel, group elements that do not have nodes in common. Referring to Equation (2.35), the contributions

$$y_e = A_K e (P_e^T x)$$

all can be computed in parallel and do not depend on one another. The operations

$$y := y + P_e y_e$$

can be processed in parallel for any group of elements that do not share any vertices. This grouping can be found by performing a multicoloring of the elements. Any two elements which have a node in common receive a different color. Using this idea, good performance can be achieved on vector computers.
EBE preconditioners are based on similar principles and many different variants have been developed. They are defined by first normalizing each of the element matrices. In the sequel, assume that \( \tilde{A} \) is a Symmetric Positive Definite matrix. Typically, a diagonal, or block diagonal, scaling is first applied to \( \tilde{A} \) to obtain a scaled matrix \( \tilde{A} \).

\[
\tilde{A} = D^{-1/2} AD^{-1/2}.
\]

This results in each matrix \( A^{[e]} \) and element matrix \( A_{K_e} \) being transformed similarly:

\[
\tilde{A}^{[e]} = D^{-1/2} A^{[e]} D^{-1/2} = D^{-1/2} P_e A_{K_e} D^{-1/2} = P_e (P^T_e D^{-1/2} P_e) A^{[e]} (P_e D^{-1/2} P^T_e) = P_e \tilde{A}_{K_e} P_e^T.
\]

The second step in defining an EBE preconditioner is to regularize each of these transformed matrices. Indeed, each of the matrices \( \tilde{A}^{[e]} \) is of rank \( O \) at most, where \( O \) is the size of the element matrix \( A_{K_e} \), i.e., the number of nodes which constitute the \( e \)-th element. In the so-called Winget regularization, the diagonal of each \( \tilde{A}^{[e]} \) is forced to be the identity matrix. In other words, the regularized matrix is defined as

\[
\tilde{A}^{[e]} = I + \tilde{A}^{[e]} - \text{diag}(\tilde{A}^{[e]}).
\]

These matrices are positive definite; see Exercise 8.

The third and final step in defining an EBE preconditioner is to choose the factorization itself. In the EBE Cholesky factorization, the Cholesky (or Crout) factorization of each regularized matrix \( \tilde{A}^{[e]} \) is performed,

\[
\tilde{A}^{[e]} = L_e D_e L_e^T.
\]

The preconditioner from it is defined as

\[
M = \prod_{e=1}^{n_e} L_e \times \prod_{e=1}^{n_e} D_e \times \prod_{e=n_e}^{1} L_e^T.
\]

Note that to ensure symmetry, the last product is in reverse order of the first one. The factorization (12.26) consists of a factorization of the small \( p_e \times p_e \) matrix \( \tilde{A}_{K_e} \). Performing the preconditioning operations will therefore consist of a sequence of small \( p_e \times p_e \) backward or forward solves. The gather and scatter matrices \( P_e \), defined in Chapter 2 must also be applied for each element. These solves are applied to the right-hand side in sequence. In addition, the same multicoloring idea as for the matrix-by-vector product can be exploited to perform these sweeps in parallel.

One of the drawbacks of the EBE Cholesky preconditioner is that an additional set of element matrices must be stored. That is because the factorizations (12.26) must be stored for each element. In EBE/SSOR, this is avoided. Instead of factoring each \( \tilde{A}^{[e]} \), the usual splitting of each \( \tilde{A}^{[e]} \) is exploited. Assuming the Winget regularization, we have

\[
\tilde{A}^{[e]} = I - E_e - E_e^T
\]

in which \(-E_e\) is the strict-lower part of \( \tilde{A}^{[e]} \). By analogy with the SSOR preconditioner,
the EBE-SSOR preconditioner is defined by

\[ M = \prod_{e=1}^{\text{nel}} (I - \omega E_e) \times \prod_{e=1}^{\text{nel}} D_e \times \prod_{e=\text{nel}}^{1} (I - \omega E_e^T). \] (12.29)

### 12.7.3 PARALLEL ROW PROJECTION PRECONDITIONERS

One of the attractions of row-projection methods seen in Chapter 8 is their high degree of parallelism. In Cimmino’s method, the scalars \( \delta_i \) as well as the new residual vector can be computed in parallel. In the Gauss-Seidel-NE (respectively Gauss-Seidel-NR), it is also possible to group the unknowns in such a way that any pair of rows (respectively columns) have disjointed nonzero patterns. Updates of components in the same group can then be performed in parallel. This approach essentially requires finding a multicolor ordering for the matrix \( B = AA^T \) (respectively \( B = A^TA \)).

It is necessary to first identify a partition of the set \( \{1, 2, \ldots, N\} \) into subsets \( S_1, \ldots, S_k \) such that the rows (respectively columns) whose indices belong to the same set \( S_i \) are structurally orthogonal to each other, i.e., have no nonzero elements in the same column locations. When implementing a block SOR scheme where the blocking is identical with that defined by the partition, all of the unknowns belonging to the same set \( S_i \) can be updated in parallel. To be more specific, the rows are reordered by scanning those in \( S_1 \) followed by those in \( S_2 \), etc.. Denote by \( A_i \) the matrix consisting of the rows belonging to the \( i \)-th block. We assume that all rows of the same set are orthogonal to each other and that they have been normalized so that their 2-norm is unity. Then a block Gauss-Seidel sweep, which generalizes Algorithm 8.1, follows.

**Algorithm 12.8: Forward Block NE-Gauss-Seidel Sweep**

1. Select an initial \( x_0 \).
2. For \( i = 1, 2, \ldots, k \) Do:
   3. \( d_i = b_i - A_i x \)
   4. \( x := x + A_i^T d_i \)
3. EndDo

Here, \( x_i \) and \( b_i \) are subvectors corresponding to the blocking and \( d_i \) is a vector of length the size of the block, which replaces the scalar \( \delta_i \) of Algorithm 8.1. There is parallelism in each of the steps 3 and 4.

The question that arises is how to find good partitions \( S_i \). In simple cases, such as block-tridiagonal matrices, this can easily be done; see Exercise 7. For general sparse matrices, a multicoloring algorithm on the graph of \( AA^T \) (respectively \( A^TA \)) can be employed. However, these matrices are never stored explicitly. Their rows can be generated, used, and then discarded.
1. Let \( A \) be a Symmetric Positive Definite matrix and consider \( N = I - D^{-1} A \) where \( D \) is a block diagonal of \( A \).
   
   a. Show that \( D \) is a Symmetric Positive Definite matrix. Denote by \((.,.)_D\) the associated inner product.
   
   b. Show that \( N \) is self-adjoint with respect to \((.,.)_D\).
   
   c. Show that \( N^k \) is self-adjoint with respect to \((.,.)_D\) for any integer \( k \).
   
   d. Show that the Neumann series expansion preconditioner defined by the right-hand side of (12.3) leads to a preconditioned matrix that is self-adjoint with respect to the \( D \)-inner product.
   
   e. Describe an implementation of the preconditioned CG algorithm using this preconditioner.

2. The development of the Chebyshev iteration algorithm seen in Section 12.3.2 can be exploited to derive yet another formulation of the conjugate algorithm from the Lanczos algorithm. Observe that the recurrence relation (12.8) is not restricted to scaled Chebyshev polynomials.
   
   a. The scaled Lanczos polynomials, i.e., the polynomials \( p_k(t)/p_k(0) \), in which \( p_k(t) \) is the polynomial such that \( v_{k+1} = p_k(A)v_1 \) in the Lanczos algorithm, satisfy a relation of the form (12.8). What are the coefficients \( a \) and \( b \) in this case?
   
   b. Proceed in the same manner as in Section 12.3.2 to derive a version of the Conjugate Gradient algorithm.

3. Show that \( \rho_k \) as defined by (12.7) has a limit \( \rho \). What is this limit? Assume that Algorithm 12.1 is to be executed with the \( \rho_k \)'s all replaced by this limit \( \rho \). Will the method converge? What is the asymptotic rate of convergence of this modified method?

4. Derive the least-squares polynomials for \( \alpha = -\frac{1}{2}, \beta = \frac{1}{2} \) for the interval \([0, 1]\) for \( k = 1, 2, 3, 4 \). Check that these results agree with those of the table shown at the end of Section 12.3.3.

5. Consider the mesh shown below. Assume that the objective is to solve the Poisson equation with Dirichlet boundary conditions.

   a. Consider the resulting matrix obtained (before boundary conditions are applied) from ordering the nodes from bottom up, and left to right (thus, the bottom left vertex is labeled 1 and the top right vertex is labeled 13). What is the bandwidth of the linear system? How many memory locations would be needed to store the matrix in Skyline format? (Assume that the matrix is nonsymmetric so both upper and lower triangular parts must be stored.)
b. Is it possible to find a 2-color ordering of the mesh points? If so, show the ordering, or otherwise prove that it is not possible.

c. Find an independent set of size 5. Show the pattern of the matrix associated with this independent set ordering.

d. Find a multicolor ordering of the mesh by using the greedy multicolor algorithm. Can you find a better coloring (i.e., a coloring with fewer colors)? If so, show the coloring [use letters to represent each color].

6. A linear system $Ax = b$ where $A$ is a 5-point matrix, is reordered using red-black ordering as

\[
\begin{pmatrix}
D_1 & F \\
E & D_2
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix} =
\begin{pmatrix}
f \\
g
\end{pmatrix}.
\]

a. Write the block Gauss-Seidel iteration associated with the above partitioned system (where the blocking in block Gauss-Seidel is the same as the above blocking).

b. Express the $y$ iterates, independently of the $x$ iterates, i.e., find an iteration which involves only $y$-iterates. What type of iteration is the resulting scheme?

7. Consider a tridiagonal matrix $T = \text{tridiag} \{a_i, b_i, c_i\}$. Find a grouping of the rows such that rows in each group are structurally orthogonal, i.e., orthogonal regardless of the values of the entries. Find a set of three groups at most. How can this be generalized to block tridiagonal matrices such as those arising from 2-D and 3-D centered difference matrices?

8. Why are the Winget regularized matrices $\tilde{A}^{[k]}$ defined by (12.25) positive definite when the matrix $\tilde{A}$ is obtained from $A$ by a diagonal scaling from $A$?

Notes and References. As vector processing appeared in the middle to late 1970s, a number of efforts were made to change algorithms, or implementations of standard methods, to exploit the new architectures. One of the first ideas in this context was to perform matrix-by-vector products by diagonals [133]. Matrix-by-vector products using this format can yield excellent performance. Hence, came the idea of using polynomial preconditioning. Polynomial preconditioning was exploited independently of supercomputing, as early as 1952 in a paper by Lanczos [141], and later for eigenvalue problems by Stiefel who employed least-squares polynomials [204], and Rutishauser [171] who combined the QD algorithm with Chebyshev acceleration. Dubois et al. [75] suggested using polynomial preconditioning, specifically, the Neumann series expansion, for solving Symmetric Positive Definite linear systems on vector computers. Johnson et al. [129] later extended the idea by exploiting Chebyshev polynomials, and other orthogonal polynomials. It was observed in [129] that least-squares polynomials tend to perform better than those based on the uniform norm, in that they lead to a better overall clustering of the spectrum. Moreover, as was already observed by Rutishauser [171], in the symmetric case there is no need for accurate eigenvalue estimates: It suffices to use the simple bounds that are provided by Gershgorin’s theorem. In [175] it was also observed that in some cases the least-squares polynomial approach which requires less information than the Chebyshev approach tends to perform better.

The use of least-squares polynomials over polygons was first advocated by Smolarski and Saylor [200] and later by Saad [176]. The application to the indefinite case was examined in detail in [174]. Still in the context of using polygons instead of ellipses, yet another attractive possibility proposed by Fischer and Reichel [91] avoids the problem of best approximation altogether. The polygon can be conformally transformed into a circle and the theory of Faber polynomials yields a simple way of deriving good polynomials from exploiting specific points on the circle.

Although only approaches based on the formulation (12.5) and (12.11) have been discussed, there are other lesser known possibilities based on minimizing $\|1/\lambda - \sigma(\lambda)\|_\infty$. There has been
very little work on polynomial preconditioning or Krylov subspace methods for highly non-normal matrices; see, however, the recent analysis in [207]. Another important point is that polynomial preconditioning can be combined with a subsidiary relaxation-type preconditioning such as SSOR [2, 153]. Finally, polynomial preconditionings can be useful in some special situations such as that of complex linear systems arising from the Helmholtz equation [93].

Multicoloring has been known for a long time in the numerical analysis literature and was used in particular for understanding the theory of relaxation techniques [232, 213] as well as for deriving efficient alternative formulations of some relaxation algorithms [213, 110]. More recently, it became an essential ingredient in parallelizing iterative algorithms, see for example [4, 2, 82, 155, 154, 164]. It is also commonly used in a slightly different form — coloring elements as opposed to nodes — in finite elements techniques [23, 217]. In [182] and [69], it was observed that $k$-step SOR preconditioning was very competitive relative to the standard ILU preconditioners. Combined with multicolor ordering, multiple-step SOR can perform quite well on supercomputers. Multicoloring is especially useful in Element-By-Element techniques when forming the residual, i.e., when multiplying an unassembled matrix by a vector [123, 88, 194]. The contributions of the elements of the same color can all be evaluated and applied simultaneously to the resulting vector. In addition to the parallelization aspects, reduced systems can sometimes be much better conditioned than the original system, see [83].

Independent set orderings have been used mainly in the context of parallel direct solution techniques for sparse matrices [66, 144, 145] and multifrontal techniques [77] can be viewed as a particular case. The gist of all these techniques is that it is possible to reorder the system in groups of equations which can be solved simultaneously. A parallel direct solution sparse solver based on performing several successive levels of independent set orderings and reduction was suggested in [144] and in a more general form in [65].
As multiprocessing technology is steadily gaining ground, new classes of numerical methods that can take better advantage of parallelism are emerging. Among these techniques, domain decomposition methods are undoubtedly the best known and perhaps the most promising for certain types of problems. These methods combine ideas from Partial Differential Equations, linear algebra, mathematical analysis, and techniques from graph theory. This chapter is devoted to “decomposition” methods, which are based on the general concepts of graph partitionings.

INTRODUCTION

13.1 Domain decomposition methods refer to a collection of techniques which revolve around the principle of divide-and-conquer. Such methods have been primarily developed for solving Partial Differential Equations over regions in two or three dimensions. However, similar principles have been exploited in other contexts of science and engineering. In fact, one of the earliest practical uses for domain decomposition approaches was in structural engineering, a discipline which is not dominated by Partial Differential Equations. Although this chapter considers these techniques from a purely linear algebra viewpoint, the basic concepts, as well as the terminology, are introduced from a model Partial Differential Equation.
Consider the problem of solving the Laplace Equation on an L-shaped domain $\Omega$ partitioned as shown in Figure 13.1. Domain decomposition or substructuring methods attempt to solve the problem on the entire domain $\Omega = \bigcup_{i=1}^{s} \Omega_i$, from problem solutions on the subdomains $\Omega_i$. There are several reasons why such techniques can be advantageous. In the case of the above picture, one obvious reason is that the subproblems are much simpler because of their rectangular geometry. For example, fast solvers can be used on each subdomain in this case. A second reason is that the physical problem can sometimes be split naturally into a small number of subregions where the modeling equations are different (e.g., Euler’s equations on one region and Navier-Stokes in another). Substructuring can also be used to develop “out-of-core” solution techniques. As already mentioned, such techniques were often used in the past to analyze very large mechanical structures. The original structure is partitioned into $s$ pieces, each of which is small enough to fit into memory. Then a form of block-Gaussian elimination is used to solve the global linear system from a sequence of solutions using $s$ subsystems. More recent interest in domain decomposition techniques has been motivated by parallel processing.

13.1.1 NOTATION

In order to review the issues and techniques in use and to introduce some notation, assume that the following problem is to be solved:

$$\Delta u = f \text{ in } \Omega$$
Domain decomposition methods are all implicitly or explicitly based on different ways of handling the unknown at the interfaces. From the PDE point of view, if the value of the solution is known at the interfaces between the different regions, these values could be used in Dirichlet-type boundary conditions and we will obtain uncoupled Poisson equations. We can then solve these equations to obtain the value of the solution at the interior points. If the whole domain is discretized by either finite elements or finite difference techniques, then this is easily translated into the resulting linear system.

Now some terminology and notation will be introduced for use throughout this chapter. Assume that the problem associated with domain shown in Figure 13.1 is discretized with centered differences. We can label the nodes by subdomain as shown in Figure 13.3. Note that the interface nodes are labeled last. As a result, the matrix associated with this problem will have the structure shown in Figure 13.4. For a general partitioning into subdomains, the linear system associated with the problem has the following structure:

\[
\begin{bmatrix}
B_1 & E_1 \\
B_2 & E_2 \\
\vdots & \vdots \\
B_s & E_s \\
F_1 & F_2 & \cdots & F_s & C
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_s \\
y
\end{bmatrix}
= 
\begin{bmatrix}
f_1 \\
f_2 \\
\vdots \\
f_s \\
g
\end{bmatrix}
\]

(13.1)

where each \( x_i \) represents the subvector of unknowns that are interior to subdomain \( \Omega_i \), and \( y \) represents the vector of all interface unknowns. It is useful to express the above system in the simpler form,

\[
A \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} \quad \text{with} \quad A = \begin{bmatrix} B & E \\ F & C \end{bmatrix}.
\]

(13.2)

Thus, \( E \) represents the subdomain to interface coupling seen from the subdomains, while \( F \) represents the interface to subdomain coupling seen from the interface nodes.

### 13.1.2 TYPES OF PARTITIONINGS

When partitioning a problem, it is common to use graph representations. Since the subproblems obtained from a given partitioning will eventually be mapped into distinct processors, there are some restrictions regarding the type of partitioning needed. For example, in Element-By-Element finite element techniques, it may be desirable to map elements into processors instead of vertices. In this case, the restriction means no element should be split between two subdomains, i.e., all information related to a given element is mapped to the same processor. These partitionings are termed element-based. A somewhat less restrictive class of partitionings are the edge-based partitionings, which do not allow edges to be split between two subdomains. These may be useful for finite volume techniques where computations are expressed in terms of fluxes across edges in two dimensions. Finally, vertex-based partitionings work by dividing the origin vertex set into subsets of vertices and have no restrictions on the edges, i.e., they allow edges or elements to straddle between subdomains. See Figure 13.2, (a), (b), and (c).
Figure 13.2  (a) Vertex-based, (b) edge-based, and (c) element-based partitioning of a $4 \times 3$ mesh into two subregions.

13.1.3 TYPES OF TECHNIQUES

The interface values can be obtained by employing a form of block-Gaussian elimination which may be too expensive for large problems. In some simple cases, using FFT’s, it is possible to explicitly obtain the solution of the problem on the interfaces inexpensively.

Other methods alternate between the subdomains, solving a new problem each time, with boundary conditions updated from the most recent subdomain solutions. These methods are called Schwarz Alternating Procedures, after the Swiss mathematician who used the idea to prove the existence for a solution of the Dirichlet problem on irregular regions.

The subdomains may be allowed to overlap. This means that the $\Omega_i$’s are such that

$$\Omega = \bigcup_{i=1}^{n} \Omega_i, \quad \Omega_i \cap \Omega_j \neq \emptyset.$$  

For a discretized problem, it is typical to quantify the extent of overlapping by the number of mesh-lines that are common to the two subdomains. In the particular case of Figure 13.3, the overlap is of order one.
The various domain decomposition techniques are distinguished by four features:

1. **Type of Partitioning.** For example, should partitioning occur along edges, or along
vertices, or by elements? Is the union of the subdomains equal to the original domain or a superset of it (fictitious domain methods)?

2. **Overlap.** Should sub-domains overlap or not, and by how much?

3. **Processing of interface values.** For example, is the Schur complement approach used? Should there be successive updates to the interface values?

4. **Subdomain solution.** Should the subdomain problems be solved exactly or approximately by an iterative method?

The methods to be discussed in this chapter will be classified in four distinct groups. First, direct methods and the substructuring approach are useful for introducing some definitions and for providing practical insight. Second, among the simplest and oldest techniques are the Schwarz Alternating Procedures. Then, there are methods based on preconditioning the Schur complement system. The last category groups all the methods based on solving the linear system with the matrix \( A \), by using a preconditioning derived from Domain Decomposition concepts.

### 13.2 DIRECT SOLUTION AND THE SCHUR COMPLEMENT

One of the first divide-and-conquer ideas used in structural analysis exploited the partitioning (13.1) in a direct solution framework. This approach, which is covered in this section, introduces the Schur complement and explains some of its properties.

#### 13.2.1 BLOCK GAUSSIAN ELIMINATION

Consider the linear system written in the form (13.2), in which \( B \) is assumed to be nonsingular. From the first equation the unknown \( x \) can be expressed as

\[
x = B^{-1} (f - Ey).
\]

Upon substituting this into the second equation, the following **reduced system** is obtained:

\[
(C - FB^{-1}E)y = g - FB^{-1}f.
\]

The matrix

\[
S = C - FB^{-1}E
\]

is called the **Schur complement** matrix associated with the \( y \) variable. If this matrix can be formed and the linear system (13.4) can be solved, all the interface variables \( y \) will become available. Once these variables are known, the remaining unknowns can be computed, via (13.3). Because of the particular structure of \( B \), observe that any linear system solution with it decouples in \( s \) separate systems. The parallelism in this situation arises from this natural decoupling.

A solution method based on this approach involves four steps:
1. Obtain the right-hand side of the reduced system (13.4).
2. Form the Schur complement matrix (13.5).
3. Solve the reduced system (13.4).
4. Back-substitute using (13.3) to obtain the other unknowns.

One linear system solution with the matrix $B$ can be saved by reformulating the algorithm in a more elegant form. Define

$$E' = B^{-1}E \quad \text{and} \quad f' = B^{-1}f.$$  

The matrix $E'$ and the vector $f'$ are needed in steps (1) and (2). Then rewrite step (4) as

$$x = B^{-1}f - B^{-1}Ey = f' - E'y,$$

which gives the following algorithm.

**Algorithm 13.1: Block-Gaussian Elimination**

1. Solve $BE' = B$, and $Bf' = f$ for $E'$ and $f'$, respectively
2. Compute $g' = g - Ff'$
3. Compute $S = C - FE'$
4. Solve $Sy = g'$
5. Compute $x = f' - E'y$.

In a practical implementation, all the $B_i$ matrices are factored and then the systems $B_iE'_i = E_i$ and $B_if'_i = f_i$ are solved. In general, many columns in $E_i$ will be zero. These zero columns correspond to interfaces that are not adjacent to subdomain $i$. Therefore, any efficient code based on the above algorithm should start by identifying the nonzero columns.

### 13.2.2 Properties of the Schur Complement

Now the connections between the Schur complement and standard Gaussian elimination will be explored and a few simple properties will be established. Start with the block-LU factorization of $A$,

$$
\begin{pmatrix}
B & E \\
F & C
\end{pmatrix} =
\begin{pmatrix}
I & 0 \\
FB^{-1} & I
\end{pmatrix}
\begin{pmatrix}
B & E \\
O & S
\end{pmatrix}
$$

which is readily verified. The Schur complement can therefore be regarded as the $(2,2)$ block in the $U$ part of the block-LU factorization of $A$. From the above relation, note that if $A$ is nonsingular, then so is $S$. Taking the inverse of $A$ with the help of the above equality yields

$$
\begin{pmatrix}
B & E \\
F & C
\end{pmatrix}^{-1} =
\begin{pmatrix}
B^{-1} & -B^{-1}ES^{-1} \\
O & S^{-1}
\end{pmatrix}
\begin{pmatrix}
I & 0 \\
-FB^{-1} & I
\end{pmatrix}
= 
\begin{pmatrix}
B^{-1} + B^{-1}ES^{-1}FB^{-1} & -B^{-1}ES^{-1} \\
-S^{-1}FB^{-1} & S^{-1}
\end{pmatrix}
.$$  

(13.7)
Observe that \( S^{-1} \) is the (2,2) block in the block-inverse of \( A \). In particular, if the original matrix \( A \) is Symmetric Positive Definite, then so is \( A^{-1} \). As a result, \( S \) is also Symmetric Positive Definite in this case.

Although simple to prove, the above properties are nonetheless important. They are summarized in the following proposition.

**PROPOSITION 13.1** Let \( A \) be a nonsingular matrix partitioned as in (13.2) and such that the submatrix \( B \) is nonsingular and let \( R_y \) be the restriction operator onto the interface variables, i.e, the linear operator defined by

\[
R_y \begin{pmatrix} x \\ y \end{pmatrix} = y.
\]

Then the following properties are true.

1. The Schur complement matrix \( S \) is nonsingular.
2. If \( A \) is SPD, then so is \( S \).
3. For any \( y \), \( S^{-1} y = R_y A^{-1} \begin{pmatrix} 0 \\ y \end{pmatrix} \).

The first property indicates that a method that uses the above block Gaussian elimination algorithm is feasible since \( S \) is nonsingular. A consequence of the second property is that when \( A \) is positive definite, an algorithm such as the Conjugate Gradient algorithm can be used to solve the reduced system (13.4). Finally, the third property establishes a relation which may allow preconditioners for \( S \) to be defined based on solution techniques with the matrix \( A \).

### 13.2.3 SCHUR COMPLEMENT FOR VERTEX-BASED PARTITIONINGS

The partitioning used in Figure 13.3 is edge-based, meaning that a given edge in the graph does not straddle two subdomains. If two vertices are coupled, then they must belong to the same subdomain. From the graph theory point of view, this is perhaps less common than vertex-based partitionings in which a vertex is not shared by two partitions (except when domains overlap). A vertex-based partitioning is illustrated in Figure 13.5.

We will call interface edges all edges that link vertices that do not belong to the same subdomain. In the case of overlapping, this needs clarification. An overlapping edge or vertex belongs to the same subdomain. Interface edges are only those that link a vertex to another vertex which is not in the same subdomain already, whether in the overlapping portion or elsewhere. Interface vertices are those vertices in a given subdomain that are adjacent to an interface edge. For the example of the figure, the interface vertices for subdomain one (bottom, left subsquare) are the vertices labeled 10 to 16. The matrix shown at the bottom of Figure 13.5 differs from the one of Figure 13.4, because here the interface nodes are not relabeled the last in the global labeling as was done in Figure 13.3. Instead, the interface nodes are labeled as the last nodes in each subdomain. The number of interface nodes is about twice that of the edge-based partitioning.
Consider the Schur complement system obtained with this new labeling. It can be written similar to the edge-based case using a reordering in which all interface variables are listed last. The matrix associated with the domain partitioning of the variables will have
a natural $s$-block structure where $s$ is the number of subdomains. For example, when $s = 3$ (as is the case in the above illustration), the matrix has the block structure defined by the solid lines in the figure, i.e.,

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}. \tag{13.8}$$

In each subdomain, the variables are of the form

$$z_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix},$$

where $x_i$ denotes interior nodes while $y_i$ denotes the interface nodes associated with subdomain $i$. Each matrix $A_i$ will be called the local matrix. The structure of $A_i$ is as follows:

$$A_i = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix} \tag{13.9}$$

in which, as before, $B_i$ represents the matrix associated with the internal nodes of subdomain $i$ and $E_i$ and $F_i$ represent the couplings to/from external nodes. The matrix $C_i$ is the local part of the interface matrix $C$ defined before, and represents the coupling between local interface points. A careful look at the matrix in Figure 13.5 reveals an additional structure for the blocks $A_{ij}$, $j \neq i$. Each of these blocks contains a zero sub-block in the part that acts on the variable $x_j$. This is expected since $x_i$ and $x_j$ are not coupled. Therefore,

$$A_{ij} = \begin{pmatrix} 0 \\ E_{ij} \end{pmatrix}. \tag{13.10}$$

In addition, most of the $E_{ij}$ matrices are zero since only those indices $j$ of the subdomains that have couplings with subdomain $i$ will yield a nonzero $E_{ij}$.

Now write the part of the linear system that is local to subdomain $i$, as

$$B_i x_i + E_i y_i + F_i x_i + C_i y_i + \sum_{j \in N_i} E_{ij} y_j = f_i \quad \Rightarrow \quad S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i. \tag{13.11}$$

The term $E_{ij} y_j$ is the contribution to the equation from the neighboring subdomain number $j$, and $N_i$ is the set of subdomains that are adjacent to subdomain $i$. Assuming that $B_i$ is nonsingular, the variable $x_i$ can be eliminated from this system by extracting from the first equation $x_i = B_i^{-1} (f_i - E_i y_i)$ which yields, upon substitution in the second equation,

$$S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - F_i B_i^{-1} f_i, \quad i = 1, \ldots, s \tag{13.12}$$

in which $S_i$ is the “local” Schur complement

$$S_i = C_i - F_i B_i^{-1} E_i. \tag{13.13}$$

When written for all subdomains $i$, the equations (13.12) yield a system of equations which involves only the interface points $y_j, j = 1, 2, \ldots, s$ and which has a natural block structure
associated with these vector variables

\[
S = \begin{pmatrix}
S_1 & E_{12} & E_{13} & \cdots & E_{1s} \\
E_{21} & S_2 & E_{23} & \cdots & E_{2s} \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
E_{s1} & E_{s2} & E_{s3} & \cdots & S_s
\end{pmatrix}.
\]

(13.14)

The diagonal blocks in this system, namely, the matrices \( S_i \), are dense in general, but the offdiagonal blocks \( E_{ij} \) are sparse and most of them are zero. Specifically, \( E_{ij} \neq 0 \) only if subdomains \( i \) and \( j \) have at least one equation that couples them.

A structure of the global Schur complement \( S \) has been unraveled which has the following important implication: For vertex-based partitionings, the Schur complement matrix can be assembled from local Schur complement matrices (the \( S_i \)'s) and interface-to-interface information (the \( E_{ij} \)'s). The term “assembled” was used on purpose because a similar idea will be exploited for finite element partitionings.

### 13.2.4 SCHUR COMPLEMENT FOR FINITE-ELEMENT PARTITIONINGS

In finite-element partitionings, the original discrete set \( \Omega \) is subdivided into \( s \) subsets \( \Omega_i \), each consisting of a distinct set of elements. Given a finite element discretization of the domain \( \Omega \), a finite dimensional space \( V_h \) of functions over \( \Omega \) is defined, e.g., functions that are piecewise linear and continuous on \( \Omega \), and that vanish on the boundary \( \Gamma \) of \( \Omega \). Consider now the Dirichlet problem on \( \Omega \) and recall that its weak formulation on the finite element discretization can be stated as follows (see Section 2.3):

Find \( u \in V_h \) such that \( a(u, v) = (f, v), \ \forall \ v \in V_h \),

where the bilinear form \( a(\cdot, \cdot) \) is defined by

\[
a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \ dx = \int_{\Omega} \left( \frac{\partial u}{\partial x_1} \frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2} \frac{\partial v}{\partial x_2} \right) \ dx.
\]

It is interesting to observe that since the set of the elements of the different \( \Omega_i \)'s are disjoint, \( a(\cdot, \cdot) \) can be decomposed as

\[
a(u, v) = \sum_{i=1}^{s} a_i(u, v),
\]

where

\[
a_i(u, v) = \int_{\Omega_i} \nabla u \cdot \nabla v \ dx.
\]

In fact, this is a generalization of the technique used to assemble the stiffness matrix from element matrices, which corresponds to the extreme case where each \( \Omega_i \) consists of exactly one element.

If the unknowns are ordered again by subdomains and the interface nodes are placed
last as was done in Section 13.1, immediately the system shows the same structure,
\[
\begin{pmatrix}
B_1 & E_1 & x_1 \\
B_2 & E_2 & x_2 \\
\vdots & \vdots & \vdots \\
B_s & E_s & x_s
\end{pmatrix}
\begin{pmatrix}
f_1 \\
f_2 \\
\vdots \\
f_s
\end{pmatrix}
= 
\begin{pmatrix}
f_1 \\
f_2 \\
\vdots \\
f_s
\end{pmatrix},
\]
where each \( B_i \) represents the coupling between interior nodes and \( E_i \) and \( F_i \) represent the coupling between the interface nodes and the nodes interior to \( \Omega_i \). Note that each of these matrices has been assembled from element matrices and can therefore be obtained from contributions over all subdomain \( \Omega_j \) that contain any node of \( \Omega_i \).

In particular, assume that the assembly is considered only with respect to \( \Omega_i \). Then the assembled matrix will have the structure
\[
A_i = \begin{pmatrix}
B_i & E_i \\
F_i & C_i
\end{pmatrix},
\]
where \( C_i \) contains only contributions from local elements, i.e., elements that are in \( \Omega_i \). Clearly, \( C \) is the sum of the \( C_i \)'s.

\[
C = \sum_{i=1}^{s} C_i.
\]

The Schur complement associated with the interface variables is such that
\[
S = C - F B^{-1} E
\]
\[
= C - \sum_{i=1}^{s} F_i B_i^{-1} E_i
\]
\[
= \sum_{i=1}^{s} C_i - \sum_{i=1}^{s} F_i B_i^{-1} E_i
\]
\[
= \sum_{i=1}^{s} [C_i - F_i B_i^{-1} E_i].
\]

Therefore, if \( S_i \) denotes the local Schur complement
\[
S_i = C_i - F_i B_i^{-1} E_i
\]
then the above proves that,
\[
S = \sum_{i=1}^{s} S_i,
\] (13.16)
showing again that the Schur complement can be obtained easily from smaller Schur complement matrices.

Another important observation is that the stiffness matrix \( A_k \), defined above by restricting the assembly to \( \Omega_k \), solves a Neumann-Dirichlet problem on \( \Omega_k \). Indeed, consider the problem
\[
\begin{pmatrix}
B_k & E_k \\
F_k & C_k
\end{pmatrix}
\begin{pmatrix}
x_k \\
y_k
\end{pmatrix}
= 
\begin{pmatrix}
b_k \\
g_k
\end{pmatrix},
\] (13.17)
The elements of the submatrix $C_k$ are the terms $a_k(\phi_i, \phi_j)$ where $\phi_i, \phi_j$ are the basis functions associated with nodes belonging to the interface $\Gamma_k$. As was stated above, the matrix $C$ is the sum of these submatrices. Consider the problem of solving the Poisson equation on $\Omega_k$ with boundary conditions defined as follows: On $\Gamma_{k0}$, the part of the boundary which belongs to $\Gamma_k$, use the original boundary conditions; on the interfaces $\Gamma_{kj}$ with other subdomains, use a Neumann boundary condition. According to Equation (2.36) seen in Section 2.3, the $j$-th equation will be of the form,

$$\int_{\Omega_k} \nabla u \cdot \phi_j \, dx = \int_{\Omega_k} f \phi_j \, dx + \int_{\Gamma_k} \phi_j \frac{\partial u}{\partial n} \, ds. \quad (13.18)$$

This gives rise to a system of the form (13.17) in which the $g_k$ part of the right-hand side incorporates the Neumann data related to the second integral on the right-hand side of (13.18).

It is interesting to note that if a problem were to be solved with all-Dirichlet conditions, i.e., if the Neumann conditions at the interfaces were replaced by Dirichlet conditions, the resulting matrix problem would be of the form,

$$\begin{pmatrix} B_k & E_k \\ 0 & I \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix} = \begin{pmatrix} b_k \\ g_k \end{pmatrix} \quad (13.19)$$

where $g_k$ represents precisely the Dirichlet data. Indeed, according to what was seen in Section 2.3, Dirichlet conditions are handled simply by replacing equations associated with boundary points by identity equations.

### SCHWARZ ALTERNATING PROCEDURES

#### 13.3

The original alternating procedure described by Schwarz in 1870 consisted of three parts: alternating between two overlapping domains, solving the Dirichlet problem on one domain at each iteration, and taking boundary conditions based on the most recent solution obtained from the other domain. This procedure is called the Multiplicative Schwarz procedure. In matrix terms, this is very reminiscent of the block Gauss-Seidel iteration with overlap defined with the help of projectors, as seen in Chapter 5. The analogue of the block-Jacobi procedure is known as the Additive Schwarz procedure.

#### 13.3.1 MULTIPLICATIVE SCHWARZ PROCEDURE

In the following, assume that each pair of neighboring subdomains has a nonvoid overlapping region. The boundary of subdomain $\Omega_i$ that is included in subdomain $j$ is denoted by $\Gamma_{i,j}$. 
This is illustrated in Figure 13.6 for the L-shaped domain example. Each subdomain extends beyond its initial boundary into neighboring subdomains. Call $\Gamma_i$ the boundary of $\Omega_i$ consisting of its original boundary (which is denoted by $\Gamma_{i,0}$) and the $\Gamma_{i,j}$'s, and denote by $u_{ji}$ the restriction of the solution $u$ to the boundary $\Gamma_{ji}$. Then the Schwarz Alternating Procedure can be described as follows.

**Algorithm 13.2: SAP**

1. Choose an initial guess $u$ to the solution
2. Until convergence Do:
3. For $i = 1, \ldots, s$ Do:
4. Solve $\Delta u = f$ in $\Omega_i$ with $u = u_{ij}$ in $\Gamma_{ij}$
5. Update $u$ values on $\Gamma_{ji}$, $\forall j$
6. EndDo
7. EndDo

The algorithm sweeps through the $s$ subdomains and solves the original equation in each of them by using boundary conditions that are updated from the most recent values of $u$. Since each of the subproblems is likely to be solved by some iterative method, we can take advantage of a good initial guess. It is natural to take as initial guess for a given subproblem the most recent approximation. Going back to the expression (13.11) of the local problems, observe that each of the solutions in line 4 of the algorithm will be translated into an update of the form

$$u_i := u_i + \delta_i,$$
where the correction \( \delta_i \) solves the system

\[
A_i \delta_i = r_i.
\]

Here, \( r_i \) is the local part of the most recent global residual vector \( b - Ax \), and the above system represents the system associated with the problem in line 4 of the algorithm when a nonzero initial guess is used in some iterative procedure. The matrix \( A_i \) has the block structure (13.9). Writing

\[
u_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad \delta_i = \begin{pmatrix} \delta_{x,i} \\ \delta_{y,i} \end{pmatrix}, \quad r_i = \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix},
\]

the correction to the current solution step in the algorithm leads to

\[
\begin{pmatrix} x_i \\ y_i \end{pmatrix} := \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}^{-1} \begin{pmatrix} r_{x,i} \\ r_{y,i} \end{pmatrix}.
\]

(13.20)

After this step is taken, normally a residual vector \( r \) would have to be computed again to get the components associated with domain \( M_{t+1} \) and to proceed with a similar step for the next subdomain. However, only those residual components that have been affected by the change of the solution need to be updated. Specifically, employing the same notation used in equation (13.11), we can simply update the residual \( r_{y,j} \) for each subdomain \( j \) for which \( i \in N_j \) as

\[
r_{y,j} := r_{y,j} - E_{ji} \delta_{y,i}.
\]

This amounts implicitly to performing Step 5 of the above algorithm. Note that since the matrix pattern is assumed to be symmetric, then the set of all indices \( j \) such that \( i \in N_j \), i.e., \( N^* = \{ j \mid i \in N_j \} \), is identical to \( N_i \). Now the loop starting in line 3 of Algorithm 13.2 and called domain sweep can be restated as follows.

**ALGORITHM 13.3: Multiplicative Schwarz Sweep – Matrix Form**

1. For \( i = 1, \ldots, s \) Do:
2. Solve \( A_i \delta_i = r_i \)
3. Compute \( x_i := x_i + \delta_{x,i}, y_i := y_i + \delta_{y,i} \), and set \( r_i := 0 \)
4. For each \( j \in N_i \) Compute \( r_{y,j} := r_{y,j} - E_{ji} \delta_{y,i} \)
5. EndDo

Considering only the \( y \) iterates, the above iteration would resemble a form of Gauss-Seidel procedure on the Schur complement matrix (13.14). In fact, it is mathematically equivalent, provided a consistent initial guess is taken. This is stated in the next result established by Chan and Goovaerts [48]:

**THEOREM 13.1** Let the guess \( \begin{pmatrix} x_i^{(0)} \\ y_i^{(0)} \end{pmatrix} \) for the Schwarz procedure in each subdomain be chosen such that

\[
x_i^{(0)} = B_i^{-1} [f_i - E_i y_i^{(0)}].
\]

(13.21)

Then the \( y \) iterates produced by the Algorithm 13.3 are identical to those of a Gauss-Seidel sweep applied to the Schur complement system (13.12).
Proof. We start by showing that with the choice (13.21), the $y$ components of the initial residuals produced by the algorithm are identical to those of the Schur complement system (13.12). Refer to Section 13.2.3 and the relation (13.10) which defines the $E_{ij}$’s from the block structure (13.8) of the global matrix. Observe that $A_{ij}u_{j} = E_{ij}y_{j}$ and note from (13.11) that for the global system the $y$ components of the initial residual vectors are

$$r_{y,i}^{(0)} = g_{i} - F_{i}x_{i}^{(0)} - C_{i}y_{i}^{(0)} - \sum_{j \in N_{i}} E_{ij}y_{j}^{(0)}$$

$$= g_{i} - F_{i}B^{-1}[f_{i} - E_{ij}y_{i}^{(0)}] - C_{i}y_{i}^{(0)} - \sum_{j \in N_{i}} E_{ij}y_{j}^{(0)}$$

$$= g_{i} - F_{i}B^{-1}f_{i} - S_{i}y_{i}^{(0)} - \sum_{j \in N_{i}} E_{ij}y_{j}^{(0)}.$$  

This is precisely the expression of the residual vector associated with the Schur complement system (13.12) with the initial guess $y_{i}^{(0)}$.

Now observe that the initial guess has been selected so that $r_{x,i}^{(0)} = 0$ for all $i$. Because only the $y$ components of the residual vector are modified, according to line 4 of Algorithm 13.3, this property remains valid throughout the iterative process. By the updating equation (13.20) and the relation (13.7), we have

$$y_{i} := y_{i} + S_{i}^{-1}r_{y,i},$$

which is precisely a Gauss-Seidel step associated with the system (13.14). Note that the update of the residual vector in the algorithm results in the same update for the $y$ components as in the Gauss-Seidel iteration for (13.14).

It is interesting to interpret Algorithm 13.2, or rather its discrete version, in terms of projectors. For this we follow the model of the overlapping block-Jacobi technique seen in the previous chapter. Let $\mathcal{S}_{i}$ be an index set

$$\mathcal{S}_{i} = \{j_{1}, j_{2}, \ldots, j_{n_{i}}\},$$

where the indices $j_{k}$ are those associated with the $n_{i}$ mesh points of the interior of the discrete subdomain $\Omega_{i}$. Note that as before, the $\mathcal{S}_{i}$’s form a collection of index sets such that

$$\bigcup_{i=1, \ldots, \sigma} \mathcal{S}_{i} = \{1, \ldots, n\},$$

and the $\mathcal{S}_{i}$’s are not necessarily disjoint. Let $R_{i}$ be a restriction operator from $\Omega$ to $\Omega_{i}$. By definition, $R_{i}x$ belongs to $\Omega_{i}$ and keeps only those components of an arbitrary vector $x$ that are in $\Omega_{i}$. It is represented by an $n_{i} \times n$ matrix of zeros and ones. The matrices $R_{i}$ associated with the partitioning of Figure 13.4 are represented in the three diagrams of Figure 13.7, where each square represents a nonzero element (equal to one) and every other element is a zero. These matrices depend on the ordering chosen for the local problem. Here, boundary nodes are labeled last, for simplicity. Observe that each row of each $R_{i}$ has exactly one nonzero element (equal to one). Boundary points such as the nodes 36 and 37 are represented several times in the matrices $R_{1}, R_{2},$ and $R_{3}$ because of the overlapping of the boundary points. Thus, node 36 is represented in matrices $R_{1}$ and $R_{2}$, while 37 is represented in all three matrices.
Patterns of the three matrices $R_i$ associated with the partitioning of Figure 13.4.

From the linear algebra point of view, the restriction operator $R_i$ is an $n_i \times n$ matrix formed by the transposes of columns $e_j$ of the $n \times n$ identity matrix, where $j$ belongs to the index set $\mathcal{S}_i$. The transpose $R_i^T$ of this matrix is a prolongation operator which takes a variable from $\Omega_i$ and extends it to the equivalent variable in $\Omega$. The matrix

$$A_i = R_i AR_i^T$$

of dimension $n_i \times n_i$ defines a restriction of $A$ to $\Omega_i$. Now a problem associated with $A_i$ can be solved which would update the unknowns in the domain $\Omega_i$. With this notation, the multiplicative Schwarz procedure can be described as follows:

1. For $i = 1, \ldots, s$ Do
2. \[ x := x + R_i^T A_i^{-1} R_i (b - Ax) \]
3. EndDo

We change notation and rewrite step 2 as

$$x_{\text{new}} = x + R_i^T A_i^{-1} R_i (b - Ax).$$

(13.22)

If the errors $d = x_\ast - x$ are considered where $x_\ast$ is the exact solution, then notice that $b - Ax = A (x_\ast - x)$ and, at each iteration the following equation relates the new error $d_{\text{new}}$ and the previous error $d$,

$$d_{\text{new}} = d - R_i^T A_i^{-1} R_i Ad.$$
13.3 SCHWARZ ALTERNATING PROCEDURES

Starting from a given $x_0$ whose error vector is $d_0 = x_0 - x$, each sub-iteration produces an error vector which satisfies the relation

$$d_i = d_{i-1} - R_i^T A_i^{-1} R_i A d_{i-1},$$

for $i = 1, \ldots, s$. As a result,

$$d_i = (I - P_i) d_{i-1},$$

in which

$$P_i = R_i^T A_i^{-1} R_i A.$$ (13.23)

Observe that the operator $P_i = R_i^T A_i^{-1} R_i A$ is a projector since

$$(R_i^T A_i^{-1} R_i A)^2 = R_i^T A_i^{-1} (R_i A R_i^T) A_i^{-1} R_i A = R_i^T A_i^{-1} R_i A.$$

Thus, one sweep produces an error which satisfies the relation

$$d_s = (I - P_s)(I - P_{s-1}) \ldots (I - P_1) d_0.$$ (13.24)

In the following, we use the notation

$$Q_s \equiv (I - P_s)(I - P_{s-1}) \ldots (I - P_1).$$ (13.25)

13.3.2 MULTIPlicative SCHwarz PREconditioning

Because of the equivalence of the multiplicative Schwarz procedure and a block Gauss-Seidel iteration, it is possible to recast one Multiplicative Schwarz sweep in the form of a global fixed-point iteration of the form $x_{new} = Gx + f$. Recall that this is a fixed-point iteration for solving the preconditioned system $M^{-1} A x = M^{-1} b$ where the preconditioning matrix $M$ and the matrix $G$ are related by $G = I - M^{-1} A$. To interpret the operation associated with $M^{-1}$, it is helpful to identify the result of the error vector produced by this iteration with that of (13.24), which is $x_{new} - x_0 = Q_s(x - x_0)$. This comparison yields,

$$x_{new} = Q_s x + (I - Q_s) x_0,$$

and therefore,

$$G = Q_s \quad f = (I - Q_s) x_0.$$

Hence, the preconditioned matrix is $M^{-1} A = I - Q_s$. This result is restated as follows.

**Proposition 13.2** The multiplicative Schwarz procedure is equivalent to a fixed-point iteration for the “preconditioned” problem

$$M^{-1} A x = M^{-1} b,$$

in which

$$M^{-1} A = I - Q_s$$ (13.26)

$$M^{-1} b = (I - Q_s) x_0 = (I - Q_s) A^{-1} b.$$ (13.27)
The transformed right-hand side in the proposition is not known explicitly since it is expressed in terms of the exact solution. However, a procedure can be found to compute it. In other words, it is possible to operate with \( M^{-1} \) without invoking \( A^{-1} \). Note that \( M^{-1} = (I - Q_s)A^{-1} \). As the next lemma indicates, \( M^{-1} \), as well as \( M^{-1}A \), can be computed recursively.

**Lemma 13.1** Define the matrices

\[
Z_i = I - Q_i \tag{13.28}
\]
\[
M_i = Z_iA^{-1} \tag{13.29}
\]
\[
T_i = P_iA^{-1} = R_i^T A^{-1} R_i \tag{13.30}
\]

for \( i = 1, \ldots, s \). Then \( M^{-1} = M_s, M^{-1}A = Z_s \), and the matrices \( Z_i \) and \( M_i \) satisfy the recurrence relations

\[
Z_i = P_1, \\
Z_i = Z_{i-1} + P_i(I - Z_{i-1}), \quad i = 2, \ldots, s 
\]

and

\[
M_i = T_1, \\
M_i = M_{i-1} + T_i(I - AM_{i-1}), \quad i = 2, \ldots, s. 
\]

**Proof.** It is clear by the definitions (13.28) and (13.29) that \( M_s = M^{-1} \) and that \( M_1 = T_1, Z_1 = P_1 \). For the cases \( i > 1 \), by definition of \( Q_i \) and \( Q_{i-1} \),

\[
Z_i = I - (I - P_i)(I - Z_{i-1}) = P_i + Z_{i-1} - P_i Z_{i-1}, \tag{13.33}
\]

which gives the relation (13.31). Multiplying (13.33) to the right by \( A^{-1} \) yields,

\[
M_i = T_i + M_{i-1} - P_i M_{i-1}. 
\]

Rewriting the term \( P_i \) as \( T_iA \) above yields the desired formula (13.32).

Note that (13.31) yields immediately the important relation

\[
Z_i = \sum_{j=1}^{i} P_j Q_{j-1}. \tag{13.34}
\]

If the relation (13.32) is multiplied to the right by a vector \( v \) and if the vector \( M_iA^{-1}v \) is denoted by \( z_i \), then the following recurrence results.

\[
z_i = z_{i-1} + T_i(v - Az_{i-1}).
\]

Since \( z_s = (I - Q_s)A^{-1}v = M^{-1}v \), the end result is that \( M^{-1}v \) can be computed for an arbitrary vector \( v \), by the following procedure.

**Algorithm 13.4: Multiplicative Schwarz Preconditioner**

1. **Input:** \( v \); **Output:** \( z = M^{-1}v \).
2. \( z := T_1v \)
3. For \( i = 2, \ldots, s \) Do:
4. \( z := z + T_i(v - Az) \)
5. \textbf{EndDo}

By a similar argument, a procedure can be found to compute vectors of the form \( z = M^{-1}Av \). In this case, the following algorithm results:

\textbf{ALGORITHM 13.5: Multiplicative Schwarz Preconditioned Operator}

1. \textit{Input:} \( v \), \textit{Output:} \( z = M^{-1}Av \).
2. \( z := P_1v \)
3. \textbf{For} \( i = 2, \ldots, s \) \textbf{Do}
4. \( z := z + P_i(v - z) \)
5. \textbf{EndDo}

In summary, the Multiplicative Schwarz procedure is equivalent to solving the “preconditioned system”

\[ (I - Q_s)x = g \]  \hspace{1cm} (13.35)

where the operation \( z = (I - Q_s)v \) can be computed from Algorithm 13.5 and \( g = M^{-1}b \) can be computed from Algorithm 13.4. Now the above procedures can be used within an accelerator such as GMRES. First, to obtain the right-hand side \( g \) of the preconditioned system (13.35), Algorithm 13.4 must be applied to the original right-hand side \( b \). Then GMRES can be applied to (13.35) in which the preconditioned operations \( I - Q_s \) are performed by Algorithm 13.5.

Another important aspect of the Multiplicative Schwarz procedure is that multicoloring can be exploited in the same way as it is done traditionally for block SOR. Finally, note that symmetry is lost in the preconditioned system but it can be recovered by following the sweep 1, 2, \ldots, \( s \) by a sweep in the other direction, namely, \( s - 1, s - 2, \ldots, 1 \). This yields a form of the block SSOR algorithm.

\subsection*{13.3.3 Additive Schwarz Procedure}

The additive Schwarz procedure is similar to a block-Jacobi iteration and consists of updating all the new (block) components from the same residual. Thus, it differs from the multiplicative procedure only because the components in each subdomain are not updated until a whole cycle of updates through all domains are completed. The basic Additive Schwarz iteration would therefore be as follows:

1. \textbf{For} \( i = 1, \ldots, s \) \textbf{Do}
2. \textit{Compute} \( \delta_i = R_i^T A_i^{-1} R_i (b - Ax) \)
3. \textbf{EndDo}
4. \( x_{\text{new}} = x + \sum_{i=1}^{s} \delta_i \)

The new approximation (obtained after a cycle of the \( s \) substeps in the above algorithm
are applied) is
\[ x_{\text{new}} = x + \sum_{i=1}^{s} R_i^T A_i^{-1} R_i (b - A x). \]

Each instance of the loop redefines different components of the new approximation and there is no data dependency between the subproblems involved in the loop.

The preconditioning matrix is rather simple to obtain for the additive Schwarz procedure. Using the matrix notation defined in the previous section, notice that the new iterate satisfies the relation
\[ x_{\text{new}} = x + \sum_{i=1}^{s} T_i (b - A x) = \left( I - \sum_{i=1}^{s} P_i \right) x + \sum_{i=1}^{s} T_i b. \]

Thus, using the same analogy as in the previous section, this iteration corresponds to a fixed-point iteration \[ x_{\text{new}} = G z + f \] with
\[ G = I - \sum_{i=1}^{s} P_i, \quad f = \sum_{i=1}^{s} T_i b. \]

With the relation \[ G = I - M^{-1} A, \] between \( G \) and the preconditioning matrix \( M \), the result is that
\[ M^{-1} A = \sum_{i=1}^{s} P_i, \]
and
\[ M^{-1} = \sum_{i=1}^{s} P_i A_i^{-1} = \sum_{i=1}^{s} T_i. \]

Now the procedure for applying the preconditioned operator \( M^{-1} \) becomes clear.

**ALGORITHM 13.6: Additive Schwarz Preconditioner**

1. **Input:** \( v \); **Output:** \( z = M^{-1} v \).
2. For \( i = 1, \ldots, s \) Do:
   3. **Compute** \( z_i := T_i v \)
   4. EndDo
3. **Compute** \( z := z_1 + z_2 \ldots + z_s \).

Note that the do loop can be performed in parallel. Step 5 sums up the vectors \( z_i \) in each domain to obtain a global vector \( z \). In the nonoverlapping case, this step is parallel and consists of just forming these different components since the addition is trivial. In the presence of overlap, the situation is similar except that the overlapping components are added up from the different results obtained in each subdomain.

The procedure for computing \( M^{-1} A v \) is identical to the one above except that \( T_i \) in line 3 is replaced by \( P_i \).
Throughout this section, it is assumed that $A$ is Symmetric Positive Definite. The projectors $P_i$ defined by (13.23) play an important role in the convergence theory of both additive and multiplicative Schwarz. A crucial observation here is that these projectors are orthogonal with respect to the $\tilde{A}$-inner product. Indeed, it is sufficient to show that $P_i$ is self-adjoint with respect to the $\tilde{A}$-inner product,

$$(P \tilde{x}, y)_A = (AR_i^T A_i^{-1} R_i A \tilde{x}, y) = (A \tilde{x}, R_i^T A_i^{-1} R_i A y) = (\tilde{x}, P_i y)_A.$$ Consider the operator,

$$A_J = \sum_{i=1}^{s} P_i.$$

Since each $P_j$ is self-adjoint with respect to the $A$-inner product, i.e., $A$-self-adjoint, their sum $A_J$ is also $A$-self-adjoint. Therefore, it will have real eigenvalues. An immediate consequence of the fact that the $P_i$’s are projectors is stated in the following theorem.

**Theorem 13.2**  The largest eigenvalue of $A_J$ is such that

$$\lambda_{\text{max}}(A_J) \leq s,$$

where $s$ is the number of subdomains.

**Proof.** For any matrix norm, $\lambda_{\text{max}}(A_J) \leq \|A_J\|$. In particular, if the $A$-norm is used, we have

$$\lambda_{\text{max}}(A_J) \leq \sum_{i=1}^{s} \|P_i\|_A.$$

Each of the $A$-norms of $P_i$ is equal to one since $P_i$ is an $A$-orthogonal projector. This proves the desired result.

This result can be improved substantially by observing that the projectors can be grouped in sets that have disjoint ranges. Graph coloring techniques seen in Chapter 3 can be used to obtain such colorings of the subdomains. Assume that $c$ sets of indices $\Theta_i, i = 1, \ldots, c$ are such that all the subdomains $\Omega_j$ for $j \in \Theta_i$ have no intersection with one another. Then,

$$P_{\Theta_i} = \sum_{j \in \Theta_i} P_j$$

is again an orthogonal projector.

This shows that the result of the previous theorem can be improved trivially into the following.

**Theorem 13.3**  Suppose that the subdomains can be colored in such a way that two subdomains with the same color have no common nodes. Then, the largest eigenvalue of $A_J$ is such that

$$\lambda_{\text{max}}(A_J) \leq c,$$
where \( c \) is the number of colors.

In order to estimate the lowest eigenvalue of the preconditioned matrix, an assumption must be made regarding the decomposition of an arbitrary vector \( x \) into components of \( \Omega_i \).

**Assumption 1.** There exists a constant \( K_0 \) such that the inequality
\[
\sum_{i=1}^{s} (Au_i, u_i) \leq K_0 (Au, u),
\]

is satisfied by the representation of \( u \in \Omega \) as the sum
\[
u = \sum_{i=1}^{s} u_i, \quad u_i \in \Omega_i.
\]

The following theorem has been proved by several authors in slightly different forms and contexts.

**THEOREM 13.4** If Assumption 1 holds, then \( \lambda_{\text{min}}(A_J) \geq \frac{1}{K_0} \).

**Proof.** Unless otherwise stated, all summations in this proof are from 1 to \( s \). Start with an arbitrary \( u \) decomposed as \( u = \sum u_i \) and write
\[
(u, u)_A = \sum (u_i, u)_A = \sum (P_i u_i, u)_A = \sum (u_i, P_i u)_A.
\]
The last equality is due to the fact that \( P_i \) is an \( A \)-orthogonal projector onto \( \Omega_i \), and it is therefore self-adjoint. Now, using Cauchy-Schwarz inequality, we get
\[
(u, u)_A \leq \left( \sum (u_i, u)_A \right)^{1/2} \left( \sum (P_i u, P_i u)_A \right)^{1/2}.
\]
By Assumption 1, this leads to
\[
||u||_A^2 \leq K_0 \sum (P_i u, P_i u)_A,
\]
which, after squaring, yields
\[
||u||_A^2 \leq K_0 \sum (P_i u, P_i u)_A.
\]
Finally, observe that since each \( P_i \) is an \( A \)-orthogonal projector, we have
\[
\sum (P_i u, P_i u)_A = \sum (P_i u, u)_A = \left( \sum P_i u, u \right)_A.
\]
Therefore, for any \( u \), the inequality
\[
(A_J u, u)_A \geq \frac{1}{K_0} (u, u)_A
\]
holds, which yields the desired upper bound by the min-max theorem. \( \blacksquare \)
13.3 SCHWARZ ALTERNATING PROCEDURES

Note that the proof uses the following form of the Cauchy-Schwarz inequality:
\[ \sum_{i=1}^{p} (x_i, y_i) \leq \left( \sum_{i=1}^{p} (x_i^2) \right)^{1/2} \left( \sum_{i=1}^{p} (y_i^2) \right)^{1/2}. \]
See Exercise 1 for a proof of this variation.

We now turn to the analysis of the Multiplicative Schwarz procedure. We start by recalling that the error after each outer iteration (sweep) is given by
\[ d = Q_s d_0. \]
We wish to find an upper bound for \( \|Q_s\|_A \). First note that (13.31) in Lemma 13.1 results in
\[ Q_i = Q_{i-1} - P_i Q_{i-1}, \]
from which we get, using the \( A \)-orthogonality of \( P_i \),
\[ \|Q_i v\|^2_A = \|Q_{i-1} v\|^2_A - \|P_i Q_{i-1} v\|^2_A. \]
The above equality is valid for \( i = 1 \) provided \( Q_0 = I \). Summing these equalities from \( i = 1 \) to \( s \) gives the result,
\[ \|Q_s v\|^2_A = \|v\|^2_A - \sum_{i=1}^{s} \|P_i Q_{i-1} v\|^2_A. \]  \hspace{1cm} (13.38)
This indicates that the \( A \)-norm of the error will not increase at each substep of the sweep.

Now a second assumption must be made to prove the next lemma.

**Assumption 2.** For any subset \( S \) of \( \{1, 2, \ldots, s\}^2 \) and \( u_i, v_j \in \Omega \), the following inequality holds:
\[ \sum_{(i,j) \in S} (P_i u_i, P_j v_j)_A \leq \sum_{i=1}^{s} \|P_i u_i\|^2_A \left( \sum_{j=1}^{s} \|P_j v_j\|^2_A \right)^{1/2}. \]  \hspace{1cm} (13.39)

**LEMMA 13.2** If Assumptions 1 and 2 are satisfied, then the following is true,
\[ \sum_{i=1}^{s} \|P_i v\|^2_A \leq (1 + K_1)^2 \sum_{i=1}^{s} \|P_i Q_{i-1} v\|^2_A. \]  \hspace{1cm} (13.40)

**Proof.** Begin with the relation which follows from the fact that \( P_i \) is an \( A \)-orthogonal projector,
\[ (P_i v, P_i v)_A = (P_i v, P_i Q_{i-1} v)_A + (P_i v, (I - Q_{i-1}) v)_A, \]
which yields, with the help of (13.34),
\[ \sum_{i=1}^{s} \|P_i v\|^2_A = \sum_{i=1}^{s} (P_i v, P_i Q_{i-1} v)_A + \sum_{i=1}^{s} \sum_{j=1}^{i-1} (P_i v, P_j Q_{j-1} v)_A. \]  \hspace{1cm} (13.41)
For the first term of the right-hand side, use the Cauchy-Schwarz inequality to obtain
\[
\sum_{i=1}^{s} (P_i v, P_i Q_{i-1} v)_A \leq \left( \sum_{i=1}^{s} \|P_i v\|_A^2 \right)^{1/2} \left( \sum_{i=1}^{s} \|P_i Q_{i-1} v\|_A^2 \right)^{1/2}.
\]
For the second term of the right-hand side of (13.41), use the assumption (13.39) to get
\[
\sum_{i=1}^{s} \sum_{j=1}^{i-1} (P_i v, P_j Q_{j-1} v)_A \leq K_1 \left( \sum_{i=1}^{s} \|P_i v\|_A^2 \right)^{1/2} \left( \sum_{j=1}^{s} \|P_j Q_{j-1} v\|_A^2 \right)^{1/2}.
\]
Adding these two inequalities, squaring the result, and using (13.41) leads to the inequality (13.40).

From (13.38), it can be deduced that if Assumption 2 holds, then,
\[
\|Q_s v\|_A^2 \leq \|v\|_A^2 - \frac{1}{(1 + K_1)^2} \sum_{i=1}^{s} \|P_i v\|_A^2.
\]  (13.42)
Assumption 1 can now be exploited to derive a lower bound on \(\sum_{i=1}^{s} \|P_i v\|_A^2\). This will yield the following theorem.

**THEOREM 13.5**  Assume that Assumptions 1 and 2 hold. Then,
\[
\|Q_s\|_A \leq \left[ 1 - \frac{1}{K_0 (1 + K_1)^2} \right]^{1/2}.
\]  (13.43)

**Proof.** Using the notation of Section 13.3.3, the relation \(\|P_i v\|_A^2 = (P_i v, v)_A\) yields
\[
\sum_{i=1}^{s} \|P_i v\|_A^2 = \left( \sum_{i=1}^{s} P_i v, v \right)_A = (A_J v, v)_A.
\]
According to Theorem 13.4, \(\lambda_{min}(A_J) \geq \frac{1}{K_0}\), which implies \((A_J v, v)_A \geq (v, v)_A / K_0\).

Thus,
\[
\sum_{i=1}^{s} \|P_i v\|_A^2 \geq \frac{(v, v)_A}{K_0},
\]
which upon substitution into (13.42) gives the inequality
\[
\frac{\|Q_s v\|_A^2}{\|v\|_A^2} \leq 1 - \frac{1}{K_0 (1 + K_1)^2}.
\]
The result follows by taking the maximum over all vectors \(v\).

This result provides information on the speed of convergence of the multiplicative Schwarz procedure by making two key assumptions. These assumptions are not verifiable from linear algebra arguments alone. In other words, given a linear system, it is unlikely that one can establish that these assumptions are satisfied. However, they are satisfied for equations originating from finite element discretization of elliptic Partial Differential Equations. For details, refer to Drya and Widlund [72, 73, 74] and Xu [230].
13.4 SCHUR COMPLEMENT APPROACHES

Schur complement methods are based on solving the reduced system (13.4) by some preconditioned Krylov subspace method. Procedures of this type involve three steps.

1. Get the right-hand side $g' = g - FB^{-1} f$.
2. Solve the reduced system $Sy = g'$ via an iterative method.
3. Back-substitute, i.e., compute $x$ via (13.3).

The different methods relate to the way in which step 2 is performed. First observe that the matrix $S$ need not be formed explicitly in order to solve the reduced system by an iterative method. For example, if a Krylov subspace method without preconditioning is used, then the only operations that are required with the matrix $S$ are matrix-by-vector operations $w = Sv$. Such operations can be performed as follows.

1. Compute $v' = Ev$.
2. Solve $Bz = v'$
3. Compute $w = Cv - Fz$.

The above procedure involves only matrix-by-vector multiplications and one linear system solution with $B$. Recall that a linear system involving $B$ translates into $s$-independent linear systems. Also note that the linear systems with $B$ must be solved exactly, either by a direct solution technique or by an iterative technique with a high level of accuracy.

While matrix-by-vector multiplications with $S$ cause little difficulty, it is much harder to precondition the matrix $S$, since this full matrix is often not available explicitly. There have been a number of methods, derived mostly using arguments from Partial Differential Equations to precondition the Schur complement. Here, we consider only those preconditioners that are derived from a linear algebra viewpoint.

13.4.1 INDUCED PRECONDITIONERS

One of the easiest ways to derive an approximation to $S$ is to exploit Proposition 13.1 and the intimate relation between the Schur complement and Gaussian elimination. This proposition tells us that a preconditioning operator $M$ to $S$ can be defined from the (approximate) solution obtained with $A$. To precondition a given vector $v$, i.e., to compute $w = M^{-1}v$, where $M$ is the desired preconditioner to $S$, first solve the system

$$A\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ v \end{pmatrix},$$

then take $w = y$. Use any approximate solution technique to solve the above system. Let $M_A$ be any preconditioner for $A$. Using the notation defined earlier, let $R_y$ represent the restriction operator on the interface variables, as defined in Proposition 13.1. Then the
preconditioning operation for $S$ which is induced from $M_A$ is defined by

$$M_S^{-1}v = R_yM_A^{-1}\begin{pmatrix} 0 \\ v \end{pmatrix} = R_yM_A^{-1}R_y^Tv.$$  

Observe that when $M_A$ is an exact preconditioner, i.e., when $M_A = A$, then according to Proposition 13.1, $M_S$ is also an exact preconditioner, i.e., $M_S = S$. This induced preconditioner can be expressed as

$$M_S = (R_yM_A^{-1}R_y^T)^{-1}. \quad (13.45)$$

It may be argued that this uses a preconditioner related to the original problem to be solved in the first place. However, even though the preconditioning on $S$ may be defined from a preconditioning of $A$, the linear system is being solved for the interface variables. That is typically much smaller than the original linear system. For example, GMRES can be used with a much larger dimension of the Krylov subspace since the Arnoldi vectors to keep in memory are much smaller. Also note that from a Partial Differential Equations viewpoint, systems of the form (13.44) correspond to the Laplace equation, the solutions of which are “Harmonic” functions. There are fast techniques which provide the solution of such equations inexpensively.

In the case where $M_A$ is an ILU factorization of $A$, $M_S$ can be expressed in an explicit form in terms of the entries of the factors of $M_A$. This defines a preconditioner to $S$ that is induced canonically from an incomplete LU factorization of $A$. Assume that the preconditioner $M_A$ is in a factored form $M_A = L_A U_A$, where

$$L_A = \begin{pmatrix} L_B & 0 \\ FU_B^{-1} & L_S \end{pmatrix}, \quad U_A = \begin{pmatrix} U_B & L_B^{-1}E \\ 0 & U_S \end{pmatrix}.$$  

Then, the inverse of $M_A$ will have the following structure:

$$M_A^{-1} = U_A^{-1}L_A^{-1}$$

$$= \begin{pmatrix} * & * & 0 \\ 0 & U_S^{-1} & * \\ * & * & U_S^{-1}L_S^{-1} \end{pmatrix}$$

where a star denotes a matrix whose actual expression is unimportant. Recall that by definition,

$$R_y = (0 \quad I),$$

where this partitioning conforms to the above ones. This means that

$$R_yM_A^{-1}R_y^T = U_S^{-1}L_S^{-1}$$

and, therefore, according to (13.45), $M_S = L_S U_S$. This result is stated in the following proposition.

**PROPOSITION 13.3** Let $M_A = L_A U_A$ be an ILU preconditioner for $A$. Then the preconditioner $M_S$ for $S$ induced by $M_A$, as defined by (13.45), is given by

$$M_S = L_S U_S, \quad \text{with} \quad L_S = R_y L_A R_y^T, \quad U_S = R_y U_A R_y^T.$$
In words, the proposition states that the L and U factors for \( M_S \) are the (2, 2) blocks of the L and U factors of the ILU factorization of \( A \). An important consequence of the above idea is that the parallel Gaussian elimination can be exploited for deriving an ILU preconditioner for \( S \) by using a general purpose ILU factorization. In fact, the \( L \) and \( U \) factors of \( M_A \) have the following structure:

\[
A = L_A U_A - R \quad \text{with,}
\]

\[
L_A = \begin{pmatrix}
    L_1 & & \\
    & L_2 & \\
    & & \ddots & \ddots & L_s \\
    F_1 U_1^{-1} & F_2 U_2^{-1} & \cdots & F_s U_s^{-1} & L
\end{pmatrix}
\]

\[
U_A = \begin{pmatrix}
    U_1 & & & & L_1^{-1} E_1 \\
    & U_2 & & & L_2^{-1} E_2 \\
    & & \ddots & & \vdots \\
    & & & U_s & L_s^{-1} E_s \\
    & & & & U
\end{pmatrix}
\]

Each \( L_i, U_i \) pair is an incomplete LU factorization of the local \( B_i \) matrix. These ILU factorizations can be computed independently. Similarly, the matrices \( L_i^{-1} E_i \) and \( F_i U_i^{-1} \) can also be computed independently once the LU factors are obtained. Then each of the matrices

\[
\tilde{S}_i = C_i - F_i U_i^{-1} L_i^{-1} E_i,
\]

which are the approximate local Schur complements, is obtained. Note that since an incomplete LU factorization is being performed, some drop strategy is applied to the elements in \( \tilde{S}_i \). Let \( T_i \) be the matrix obtained after this is done,

\[
T_i = \tilde{S}_i - R_i.
\]

Then a final stage would be to compute the ILU factorization of the matrix (13.14) where each \( S_i \) is replaced by \( T_i \).

### 13.4.2 PROBING

To derive preconditioners for the Schur complement, another general purpose technique exploits ideas used in approximating sparse Jacobians when solving nonlinear equations. In general, \( S \) is a dense matrix. However, it can be observed, and there are physical justifications for model problems, that its entries decay away from the main diagonal. Assume that \( S \) is nearly tridiagonal, i.e., neglect all diagonals apart from the main diagonal and the two codiagonals, and write the corresponding tridiagonal approximation to \( S \) as

\[
T = \begin{pmatrix}
a_1 & b_2 & & & \\
b_2 & a_2 & b_3 & & \\
& \ddots & \ddots & \ddots & \\
& & c_{m-1} & a_{m-1} & b_m \\
& & & c_m & a_m
\end{pmatrix}
\]
Then, it is easy to recover $T$ by applying it to three well-chosen vectors. Consider the three vectors

\[ w_1 = (1, 0, 0, 1, 0, 0, 1, 0, 0, \ldots)^T, \]
\[ w_2 = (0, 1, 0, 1, 0, 0, 1, 0, 0, \ldots)^T, \]
\[ w_3 = (0, 0, 1, 0, 0, 1, 0, 0, 1, \ldots)^T. \]

Then we have

\[ Tw_1 = (a_1, c_2, b_4, a_4, c_5, \ldots, b_{3i+1}, a_{3i+1}, c_{3i+2}, \ldots)^T, \]
\[ Tw_2 = (b_2, a_2, c_3, b_5, a_5, c_6, \ldots, b_{3i+2}, a_{3i+2}, c_{3i+3}, \ldots)^T, \]
\[ Tw_3 = (b_3, a_3, c_4, b_6, a_6, c_7, \ldots, b_{3i+3}, a_{3i+3}, c_{3i+4}, \ldots)^T. \]

This shows that all the coefficients of the matrix $T$ are indeed all represented in the above three vectors. The first vector contains the nonzero elements of the columns 1, 4, 7, \ldots, $3i + 1$, \ldots, in succession written as a long vector. Similarly, $Tw_2$ contains the columns 2, 5, 8, \ldots, and $Tw_3$ contains the columns 3, 6, 9, \ldots. We can easily compute $Sw_i, i = 1, 3$ and obtain a resulting approximation $T$ which can be used as a preconditioner to $S$. The idea can be extended to compute any banded approximation to $S$. For details and analysis see [49].

### 13.4.3 Preconditioning Vertex-Based Schur Complements

We now discuss some issues related to the preconditioning of a linear system with the matrix coefficient of (13.14) associated with a vertex-based partitioning. As was mentioned before, this structure is helpful in the direct solution context because it allows the Schur complement to be formed by local pieces. Since incomplete LU factorizations will utilize the same structure, this can be exploited as well.

Note that multicolor SOR or SSOR can also be exploited and that graph coloring can be used to color the interface values $y_i$ in such a way that no two adjacent interface variables will have the same color. In fact, this can be achieved by coloring the domains. In the course of a multicolor block-SOR iteration, a linear system must be solved with the diagonal blocks $S_i$. For this purpose, it is helpful to interpret the Schur complement. Call $P$ the canonical injection matrix from the local interface points to the local nodes. If $n_i$ points are local and if $m_i$ is the number of the local interface points, then $P$ is an $n_i \times m_i$ matrix whose columns are the last $m_i$ columns of the $n_i \times n_i$ identity matrix. Then it is easy to see that

\[ S_i = (P^TA_{loc,i}^{-1}P)^{-1}. \]  

(13.46)

If $A_{loc,i} = LU$ is the LU factorization of $A_{loc,i}$ then it can be verified that

\[ S_i^{-1} = P^TU^{-1}L^{-1}P = P^TU^{-1}P P^TL^{-1}P, \]  

(13.47)

which indicates that in order to operate with $P^TL^{-1}P$, the last $m_i \times m_i$ principal submatrix of $L$ must be used. The same is true for $P^TU^{-1}P$ which requires only a back-solve with the last $m_i \times m_i$ principal submatrix of $U$. Therefore, only the LU factorization of $A_{loc,i}$ is
needed to solve a system with the matrix $S_i$. Interestingly, approximate solution methods associated with incomplete factorizations of $A_{10c_1}$ can be exploited.

### 13.5 FULL MATRIX METHODS

We call any technique that iterates on the original system (13.2) a **full matrix method**. In the same way that preconditioners were derived from the LU factorization of $A$ for the Schur complement, preconditioners for $A$ can be derived from approximating interface values.

Before starting with preconditioning techniques, we establish a few simple relations between iterations involving $A$ and $S$.

**Proposition 13.4** Let

$$L_A = \begin{pmatrix} I & 0 \\ F B^{-1} & I \end{pmatrix}, \quad U_A = \begin{pmatrix} B & E \\ O & I \end{pmatrix} \quad (13.48)$$

and assume that a Krylov subspace method is applied to the original system (13.1) with left preconditioning $L_A$ and right preconditioning $U_A$, and with an initial guess of the form

$$\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} B^{-1}(f - E y_0) \\ y_0 \end{pmatrix}. \quad (13.49)$$

Then this preconditioned Krylov iteration will produce iterates of the form

$$\begin{pmatrix} x_m \\ y_m \end{pmatrix} = \begin{pmatrix} B^{-1}(f - E y_m) \\ y_m \end{pmatrix} \quad (13.50)$$

in which the sequence $y_m$ is the result of the same Krylov subspace method applied without preconditioning to the reduced linear system $S y = g'$ with $g' = g - F B^{-1} f$ starting with the vector $y_0$.

**Proof.** The proof is a consequence of the factorization

$$\begin{pmatrix} B & E \\ F & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ F B^{-1} & I \end{pmatrix} \begin{pmatrix} I & O \\ O & I \end{pmatrix} \begin{pmatrix} B & E \\ O & I \end{pmatrix}. \quad (13.51)$$

Applying an iterative method (e.g., GMRES) on the original system, preconditioned from the left by $L_A$ and from the right by $U_A$, is equivalent to applying this iterative method to

$$L^{-1}_A A U^{-1}_A = \begin{pmatrix} I & O \\ O & S \end{pmatrix} \equiv A' \quad (13.52)$$

The initial residual for the preconditioned system is

$$L^{-1}_A \begin{pmatrix} f \\ g \end{pmatrix} - (L^{-1}_A A U^{-1}_A) U_A \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$$

$$= \begin{pmatrix} I & 0 \\ -F B^{-1} & I \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix} - \begin{pmatrix} f \\ F B^{-1} (f - E y_0) + C y_0 \end{pmatrix}$$
As a result, the Krylov vectors obtained from the preconditioned linear system associated with the matrix \( \tilde{A} \) have the form

\[
\begin{pmatrix}
0 \\
r_0 \\
\vdots \\
S^{m-1}r_0
\end{pmatrix}
\]

and the associated approximate solution will be of the form

\[
\begin{pmatrix}
x_m \\
y_m
\end{pmatrix} = \begin{pmatrix}
x_0 \\
y_0
\end{pmatrix} + \begin{pmatrix}
B^{-1} & -B^{-1}E \\
O & I
\end{pmatrix} \begin{pmatrix}
0 \\
\sum_{i=0}^{m-1} \alpha_i S^i r_0
\end{pmatrix}
\begin{pmatrix}
y_m \\
y_m
\end{pmatrix}
= \begin{pmatrix}
B^{-1} (f - E y_0) \\
B^{-1} (f - E y_m)
\end{pmatrix}.
\]

Finally, the scalars \( \alpha_i \) that express the approximate solution in the Krylov basis are obtained implicitly via inner products of vectors among the vector sequence (13.53). These inner products are identical to those of the sequence \( r_0, Sr_0, \ldots, S^{m-1}r_0 \). Therefore, these coefficients will achieve the same result as the same Krylov method applied to the reduced system \( Sy = g' \), if the initial guess gives the residual guess \( r_0 \).

A version of this proposition should allow \( S \) to be preconditioned. The following result is an immediate extension that achieves this goal.

**Proposition 13.5** Let \( S = L_S U_S - R \) be an approximate factorization of \( S \) and define

\[
L_A = \begin{pmatrix}
I & O \\
FB^{-1} & L_S
\end{pmatrix}, \quad U_A = \begin{pmatrix}
B & E \\
O & U_S
\end{pmatrix}.
\]

Assume that a Krylov subspace method is applied to the original system (13.1) with left preconditioning \( L_A \) and right preconditioning \( U_A \), and with an initial guess of the form

\[
\begin{pmatrix}
x_0 \\
y_0
\end{pmatrix} = \begin{pmatrix}
B^{-1} (f - E y_0)
\end{pmatrix}.
\]

Then this preconditioned Krylov iteration will produce iterates of the form

\[
\begin{pmatrix}
x_m \\
y_m
\end{pmatrix} = \begin{pmatrix}
B^{-1} (f - E y_m)
\end{pmatrix}.
\]

Moreover, the sequence \( y_m \) is the result of the same Krylov subspace method applied to the reduced linear system \( Sy = g - FB^{-1} f \), left preconditioned with \( L_S \), right preconditioned with \( U_S \), and starting with the vector \( y_0 \).

**Proof.** The proof starts with the equality

\[
\begin{pmatrix}
B & E \\
F & C
\end{pmatrix} = \begin{pmatrix}
I & O \\
FB^{-1} & L_S
\end{pmatrix} \begin{pmatrix}
I & O \\
O & L_S^{-1}SU_S^{-1}
\end{pmatrix} \begin{pmatrix}
B & E \\
O & U_S
\end{pmatrix}.
\]

The rest of the proof is similar to that of the previous result and is omitted.
Also there are two other versions in which $S$ is allowed to be preconditioned from the left or from the right. Thus, if $M_S$ is a certain preconditioner for $S$, use the following factorizations
\[
\begin{pmatrix}
    B & E \\
    F & C
\end{pmatrix} = 
\begin{pmatrix}
    I & O \\
    FB^{-1} & M_S
\end{pmatrix}
\begin{pmatrix}
    I & O \\
    O & M_S^{-1}s
\end{pmatrix}
\begin{pmatrix}
    B & E \\
    O & I
\end{pmatrix},
\]
(13.58)
\[
= 
\begin{pmatrix}
    I & O \\
    FB^{-1} & I
\end{pmatrix}
\begin{pmatrix}
    I & O \\
    O & SM_S^{-1}
\end{pmatrix}
\begin{pmatrix}
    B & E \\
    O & M_S
\end{pmatrix},
\]
(13.59)
to derive the appropriate left or right preconditioners. Observe that when the preconditioner $M_S$ to $S$ is exact, i.e., when $M = S$, then the block preconditioner $L_A, U_A$ to $A$ induced from $M_S$ is also exact.

Although the previous results indicate that a Preconditioned Schur Complement iteration is mathematically equivalent to a certain preconditioned full matrix method, there are some practical benefits in iterating with the nonreduced system. The main benefit involves the requirement in the Schur Complement techniques to compute $Sx$ exactly at each Krylov subspace iteration. Indeed, the matrix $S$ represents the coefficient matrix of the linear system, and inaccuracies in the matrix-by-vector operation may result in loss of convergence. In the full matrix techniques, the operation $Sx$ is never needed explicitly. In addition, this opens up the possibility of preconditioning the original matrix with approximate solves with the matrix $B$ in the preconditioning operation $L_A$ and $U_A$.

**13.6 GRAPH PARTITIONING**

The very first task that a programmer faces when solving a problem on a parallel computer, be it a dense or a sparse linear system, is to decide how to map the data into the processors. For shared memory and SIMD computers, directives are often provided to help the user input a desired mapping, among a small set of choices. Distributed memory computers are more general since they allow mapping the data in an arbitrary fashion. However, this added flexibility puts the burden on the user to find good mappings. In particular, when implementing Domain Decomposition ideas on a parallel computer, efficient techniques must be available for partitioning an arbitrary graph. This section gives an overview of the issues and covers a few techniques.

**13.6.1 BASIC DEFINITIONS**

Consider a general sparse linear system whose adjacency graph is $G = (V, E)$. There are two issues related to the distribution of mapping a general sparse linear system on a number of processors. First, a good partitioning must be found for the original problem. This translates into partitioning the graph $G$ into subgraphs and can be viewed independently from the underlying architecture or topology. The second issue, which is architecture dependent, is to find a good mapping of the subdomains or subgraphs to the processors, after
the partitioning has been found. Clearly, the partitioning algorithm can take advantage of a measure of quality of a given partitioning by determining different weight functions for the vertices, for vertex-based partitionings. Also, a good mapping could be found to minimize communication costs, given some knowledge on the architecture.

Graph partitioning algorithms address only the first issue. Their goal is to subdivide the graph into smaller subgraphs in order to achieve a good load balancing of the work among the processors and ensure that the ratio of communication over computation is small for the given task. We begin with a general definition.

![Mapping of a simple 4 × 3 mesh to 4 processors.](image)

**DEFINITION 13.1** We call a map of $V$, any set $V_1, V_2, \ldots, V_n$, of subsets of the vertex set $V$, whose union is equal to $V$:

$$V_i \subseteq V, \quad \bigcup_{i=1}^{n} V_i = V.$$  

When all the $V_i$ subsets are disjoint, the map is called a proper partition; otherwise we refer to it as an overlapping partition.

The most general way to describe a node-to-processor mapping is by setting up a list for each processor, containing all the nodes that are mapped to that processor. Three distinct classes of algorithms have been developed for partitioning graphs. An overview of each of these three approaches is given next.

### 13.6.2 GEOMETRIC APPROACH

The geometric approach works on the physical mesh and requires the coordinates of the mesh points to find adequate partitionings. In the simplest case, for a 2-dimensional rectangular grid, stripes in the horizontal and vertical direction can be defined to get square subregions which have roughly the same number of points. Other techniques utilize notions of moment of inertia to divide the region recursively into two roughly equal-sized subregions.

Next is a very brief description of a technique based on work by Miller, Teng, Thurston, and Vavasis [150]. This technique finds good separators for a mesh using projections.
into a higher space. Given a mesh in $\mathbb{R}^d$, the method starts by projecting the mesh points into a unit sphere centered at the origin in $\mathbb{R}^{d+1}$. Stereographic projection is used: A line is drawn from a given point $p$ in the plane to the North Pole $(0, \ldots, 0, 1)$ and the stereographic projection of $p$ is the point where this line intersects the sphere. In the next step, a centerpoint of the projected points is found. A centerpoint $c$ of a discrete set $S$ is defined as a point where every hyperplane passing through $c$ will divide $S$ approximately evenly. Once the centerpoint is found, the points of the sphere are rotated so that the centerpoint is aligned with the North Pole, i.e., so that coordinates of $c$ are transformed into $(0, \ldots, 0, 1)$. The points are further transformed by dilating them so that the centerpoint becomes the origin. Through all these transformations, the point $c$ remains a centerpoint. Therefore, if any hyperplane is taken that passes through the centerpoint which is now the origin, it should cut the sphere into two roughly equal-sized subsets. Any hyperplane passing through the origin will intersect the sphere along a large circle $S$. Transforming this circle back into the original space will give a desired separator. Notice that there is an infinity of circles to choose from. One of the main ingredients in the above algorithm is a heuristic for finding centerpoints in $\mathbb{R}^d$ space (actually, $\mathbb{R}^{d+1}$ in the algorithm). The heuristic that is used repeatedly replaces randomly chosen sets of $d + 2$ points by their centerpoint, which are easy to find in this case.

There are a number of interesting results that analyze the quality of geometric graph partitionings based on separators. With some minimal assumptions on the meshes, it is possible to show that there exist “good” separators. In addition, the algorithm discussed above constructs such separators. We start with two definitions.

**Definition 13.2** A $k$-ply neighborhood system in $\mathbb{R}^d$ is a set of $n$ closed disks $D_i$, $i = 1, \ldots, n$ in $\mathbb{R}^d$ such that no point in $\mathbb{R}^d$ is (strictly) interior to more than $k$ disks.

**Definition 13.3** Let $\alpha \geq 1$ and let $D_1, \ldots, D_n$ be a $k$-ply neighborhood system in $\mathbb{R}^d$. The $(\alpha, k)$-overlap graph for the neighborhood system is the graph with vertex set $V = \{1, 2, \ldots, n\}$ and edge set, the subset of $V \times V$ defined by

$$\{(i, j) : (D_i \cap (\alpha \cdot D_j) \neq \emptyset) \text{ and } (D_j \cap (\alpha \cdot D_i) \neq \emptyset)\}.$$}

A mesh in $\mathbb{R}^d$ is associated with an overlap graph by assigning the coordinate of the center $c_i$ of disk $i$ to each node $i$ of the graph. Overlap graphs model computational meshes in $d$ dimensions. Indeed, every mesh with bounded aspect ratio elements (ratio of largest to smallest edge length of each element) is contained in an overlap graph. In addition, any planar graph is an overlap graph. The main result regarding separators of overlap graphs is the following theorem [150].

**Theorem 13.6** Let $G$ be an $n$-vertex $(\alpha, k)$ overlap graph in $d$ dimensions. Then the vertices of $G$ can be partitioned into three sets $A$, $B$, and $C$ such that:

1. No edge joins $A$ and $B$.
2. $A$ and $B$ each have at most $n(d + 1)/(d + 2)$ vertices.
3. $C$ has only $O(\alpha k^{1/d} n^{(d-1)/d})$ vertices.

Thus, for $d = 2$, the theorem states that it is possible to partition the graph into two
subgraphs $A$ and $B$, with a separator $C$, such that the number of nodes for each of $A$ and $B$ does not exceed $\frac{3}{4}n$ vertices in the worst case and such that the separator has a number of nodes of the order $O(\alpha^{k/2}n^{1/2})$.

### 13.6.3 SPECTRAL TECHNIQUES

Spectral bisection refers to a technique which exploits some known properties of the eigenvectors of the Laplacean of a graph. Given an adjacency graph $G = (V, E)$, we associate to it a Laplacian matrix $\mathcal{L}$ which is a sparse matrix having the same adjacency graph and defined as follows:

$$l_{ij} = \begin{cases} -1 & \text{if } (v_i, v_j) \in E \text{ and } i \neq j \\ \deg(i) & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

There are some interesting fundamental properties of such matrices. Assuming the graph is undirected, the matrix is symmetric. It can easily be seen that it is also negative semi definite (see Exercise 9). Zero is an eigenvalue and it is the smallest one. An eigenvector associated with this eigenvalue is any constant vector, and this eigenvector bears little interest. However, the second smallest eigenvector, called the Fiedler vector, has the useful property that the signs of its components divide the domain into roughly two equal subdomains. To be more accurate, the Recursive Spectral Bisection (RSB) algorithm consists of sorting the components of the eigenvector and assigning the first half of the sorted vertices to the first subdomain and the second half to the second subdomain. The two subdomains are then partitioned in two recursively, until a desirable number of domains is reached.

**ALGORITHM 13.7: RSB (Recursive Spectral Bisection)**

1. Compute the Fiedler vector $f$ of the graph $G$.
2. Sort the components of $f$, e.g., increasingly.
3. Assign first $\lfloor n/2 \rfloor$ nodes to $V_1$, and the rest to $V_2$.
4. Apply RSB recursively to $V_1, V_2$, until the desired number of partitions is reached.

The main theoretical property that is exploited here is that the differences between the components of the Fiedler vector represent some sort of distance between the corresponding nodes. Thus, if these components are sorted they would be grouping effectively the associated node by preserving nearness. In addition, another interesting fact is that the algorithm will also tend to minimize the number $n_c$ of cut-edges, i.e., the number of edges $(v_i, v_j)$ such that $v_i \in V_1$ and $v_j \in V_2$. Let $p$ be a partition vector whose components are $+1$ or $-1$ in equal number, so that $e^TP = 0$ where $e = (1, 1, \ldots, 1)^T$. Assume that $V_1$ and $V_2$ are of equal size and that the components of $p$ are set to $+1$ for those in $V_1$ and $-1$ for those in $V_2$. Then notice that

$$\langle Lp, p \rangle = 4n_c, \quad \langle p, e \rangle = 0.$$
(p, e) = 0. Note that here p is a vector of signs. If, instead, the objective function
\((Lz, x)/(x, x)\) were minimized with respect to the constraint \((x, e) = 0\) for \(x\) real, the so-
lution would be the Fiedler vector, since \(e\) is the eigenvector associated with the eigenvalue
zero. The Fiedler vector is an eigenvector associated with the second smallest eigenvalue
of \(L\). This eigenvector can be computed by the Lanczos algorithm or any other method ef-
fective for large sparse matrices. Recursive Spectral Bisection gives excellent partitionings.
On the other hand, it is rather expensive because of the requirement to compute eigenvec-
tors.

13.6.4 GRAPH THEORY TECHNIQUES

There exist a number of other techniques which, like spectral techniques, are also based
on the adjacency graph only. The simplest idea is one that is borrowed from the technique
of nested dissection in the context of direct sparse solution methods. Refer to Chapter 3
where level set orderings are described. An initial node is given which constitutes the level
zero. Then, the method recursively traverses the \(k\)-th level \((k \geq 1)\), which consists of the
neighbors of all the elements that constitute level \(k - 1\). A simple idea for partitioning the
graph in two traverses enough levels to visit about half of all the nodes. The visited nodes
will be assigned to one subdomain and the others will constitute the second subdomain.
The process can then be repeated recursively on each of the subdomains. A key ingredient
for this technique to be successful is to determine a good initial node from which to start
the traversal. Often, a heuristic is used for this purpose. Recall that \(d(x, y)\) is the distance
between vertices \(x\) and \(y\) in the graph, i.e., the length of the shortest path between \(x\) and \(y\).
If the diameter of a graph is defined as
\[
\delta(G) = \max\{d(x, y) \mid x \in V, y \in V\}
\]
then, ideally, one of two nodes in a pair \((x, y)\) that achieves the diameter can be used as
a starting node. These peripheral nodes, are expensive to determine. Instead, a pseudo-
peripheral node, as defined through the following procedure, is often employed.

**ALGORITHM 13.8: Pseudo-Peripheral Node**

1. Select an initial node \(x\). Set \(\delta = 0\).
2. Do a level set traversal from \(x\).
3. Select a node \(y\) in the last level set, with minimum degree.
4. If \(d(x, y) > \delta\) then
5. \(\text{Set } x := y \text{ and } \delta := d(x, y)\)
6. \(\text{GoTo 2}\)
7. \(\text{Else Stop: } x \text{ is a pseudo-peripheral node.}\)
8. \(\text{EndIf}\)

The distance \(d(x, y)\) in line 5 is the number of levels in the level set traversal needed in Step
2. The algorithm traverses the graph from a node of the last level in the previous traversal,
until the number of levels stabilizes. It is easy to see that the algorithm does indeed stop
after a finite number of steps, typically small.
The above algorithm plays a key role in sparse matrix computations. It is very helpful in the context of graph partitioning as well. A first heuristic approach based on level set traversals is the recursive dissection procedure mentioned above and described next.

**ALGORITHM 13.9: Recursive Graph Bisection**

1. Set $G_\ast := G, S := \{G\}, n_{\text{dom}} := 1$
2. While $n_{\text{dom}} < s$ Do:
3.   Select in $S$ the subgraph $G_\ast$ with largest size.
4.   Find a pseudo-peripheral node $p$ in $G_\ast$ and
5.   Do a level set traversal from $p$. Let $\text{lev} := \text{number of levels}$.
6.   Let $G_1$ the subgraph of $G_\ast$ consisting of the first $\text{lev}/2$
7.   levels, and $G_2$ the subgraph containing the rest of $G_\ast$.
8.   Remove $G_\ast$ from $S$ and add $G_1$ and $G_2$ to it
9.   $n_{\text{dom}} := n_{\text{dom}} + 1$
10. EndWhile

The cost of this algorithm is rather small. Each traversal of a graph $G = (V,E)$ costs around $|E|$, where $|E|$ is the number of edges (assuming that $|V| = O(|E|)$). Since there are $s$ traversals of graphs whose size decreases by 2 at each step, it is clear that the cost is $O(|E|)$, the order of edges in the original graph.

As can be expected, the results of such an algorithm are not always good. Typically, two qualities that are measured are the sizes of the domains as well as the number of cut-edges. Ideally, the domains should be equal. In addition, since the values at the interface points should be exchanged with those of neighboring processors, their total number, as determined by the number of cut-edges, should be as small as possible. The first measure can be easily controlled in a recursive Graph Bisection Algorithm — for example, by using variants in which the number of nodes is forced to be exactly half that of the original sub-domain. The second measure is more difficult to control. Thus, the top part of Figure 13.9 shows the result of the RGB algorithm on a sample finite-element mesh. This is a vertex-based partitioning. The dashed lines are the cut-edges that link two different domains.

An approach that is competitive with the one described above is that of double striping. This method uses two parameters $p_1, p_2$ such that $p_1p_2 = s$. The original graph is first partitioned into $p_1$ large partitions, using one-way partitioning, then each of these partitions is subdivided into $p_2$ partitions similarly. One-way partitioning into $p$ subgraphs consists of performing a level set traversal from a pseudo-peripheral node and assigning each set of roughly $n/p$ consecutive nodes in the traversal to a different subgraph. The result of this approach with $p_1 = p_2 = 4$ is shown in Figure 13.9 on the same graph as before. As can be observed, the subregions obtained by both methods have elongated and twisted shapes. This has the effect of giving a larger number of cut-edges.
Figure 13.9 The RGB algorithm (top) and the double-stripping algorithm (bottom) for partitioning a graph into 16 subgraphs.
There are a number of heuristic ways to remedy this. One strategy is based on the fact that a level set traversal from \( k \) nodes can be defined instead of only one node. These \( k \) nodes are called the centers or sites. Each subdomain will expand from one of these \( k \) centers and the expansion will stop when it is no longer possible to acquire another point that is not already assigned. The boundaries of each domain that are formed this way will tend to be more “circular.” To smooth the boundaries of an initial partition, find some center point of each domain and perform a level set expansion from the set of points. The process can be repeated a few times.

**Algorithm 13.10: Multinode Level-Set Expansion Algorithm**

1. Find a partition \( S = \{G_1, G_2, \ldots, G_s\} \).
2. For \( \text{iter} = 1, \ldots, n_{\text{outer}} \) Do:
   3. For \( k = 1, \ldots, s \) Do:
      4. Find a center \( c_k \) of \( G_k \). Set \( \text{label}(c_k) = k \).
      5. EndDo
   6. Do a level set traversal from \( \{c_1, c_2, \ldots, c_s\} \). Label each child in the traversal with the same label as its parent.
   7. For \( k = 1, \ldots, s \) set \( G_k := \text{subgraph of all nodes having label } k \)
   8. EndDo

![Figure 13.10](image.png)  
*Multinode expansion starting with the partition obtained in Figure 13.9.*
For this method, a total number of cut-edges equal to 548 and a rather small standard deviation of 0.5 are obtained for the example seen earlier.

Still to be decided is how to select the center nodes mentioned in line 4 of the algorithm. Once more, the pseudo-peripheral algorithm will be helpful. Find a pseudo-peripheral node, then do a traversal from it until about one-half of the nodes have been traversed. Then, traverse the latest level set (typically a line or a very narrow graph), and take the middle point as the center.

A typical number of outer steps, nouter, to be used in line 2, is less than five. This heuristic works well in spite of its simplicity. For example, if this is applied to the graph obtained from the RGB algorithm, with nouter = 3, the partition shown in Figure 13.10 is obtained. With this technique, the resulting total number of cut-edges is equal to 441 and the standard deviation is 7.04. As is somewhat expected, the number of cut-edges has decreased dramatically, while the standard deviation of the various sizes has increased.

1. In the proof of Theorem 13.4, the following form of the Cauchy-Schwarz inequality was used:

\[ \sum_{i=1}^{p} (x_i, y_i) \leq \left( \sum_{i=1}^{p} (x_i, x_i) \right)^{1/2} \left( \sum_{i=1}^{p} (y_i, y_i) \right)^{1/2}. \]

(a) Prove that this result is a consequence of the standard Cauchy-Schwarz inequality. (b) Extend the result to the \( A \)-inner product. (c) Assume that the \( x_i \)'s and \( y_i \)'s are the columns of two \( n \times p \) matrix \( X \) and \( Y \). Rewrite the result in terms of these matrices.

2. Using Lemma 13.1, write explicitly the vector \( M^{-1}b \) for the Multiplicative Schwarz procedure, in terms of the matrix \( A \) and the \( R_i \)'s, when \( s = 2 \), and then when \( s = 3 \).

3. (a) Show that in the multiplicative Schwarz procedure, the residual vectors \( r_i = b - A x_i \) obtained at each step satisfy the recurrence,

\[ r_i = r_{i-1} - A R_i^T A^{-1} R_i r_{i-1}. \]

for \( i = 1, \ldots, s \). (b) Consider the operator \( Q_i \equiv A R_i^T A^{-1} R_i \). Show that \( Q_i \) is a projector. (c) Is \( Q_i \) an orthogonal projector with respect to the \( A \)-inner product? With respect to which inner product is it orthogonal?

4. The analysis of the Additive Schwarz procedure assumes that \( A_i^{-1} \) is “exact,” i.e., that linear systems \( A_i x = b \) are solved exactly, each time \( A_i^{-1} \) is applied. Assume that \( A_i^{-1} \) is replaced by some approximation \( \Theta_i^{-1} \). (a) Is \( P_i \) still a projector? (b) Show that if \( \Theta_i \) is Symmetric Positive Definite, then so is \( P_i \). (c) Now make the assumption that \( \lambda_{max}(P_i) \leq \omega_s \). What becomes of the result of Theorem 13.2?

5. In Element-By-Element (EBE) methods, the extreme cases of the Additive or the Multiplicative Schwarz procedures are considered in which the subdomain partition corresponds to taking \( \Omega_k \) to be an element. The advantage here is that the matrices do not have to be assembled. Instead, they are kept in unassembled form (see Chapter 2). Assume that Poisson’s equation is being solved.
(a) What are the matrices $A_i$? (b) Are they SPD? (c) Write down the EBE preconditioning corresponding to the multiplicative Schwarz procedure, its multicolor version, and the additive Schwarz procedure.

6. Theorem 13.1 was stated only for the multiplicative version of the Schwarz procedure. There is a similar result for the additive Schwarz procedure. State this result and prove it.

7. Show that the matrix defined by (13.37) is indeed a projector. Is it possible to formulate Schwarz procedures in terms of projection processes as seen in Chapter 5?

8. It was stated at the end of the proof of Theorem 13.4 that if

$$
(A_j v, v)_A \geq \frac{1}{C} (v, v)_A
$$

for any nonzero $v$, then $\lambda_{\text{min}}(A_j) \geq \frac{1}{C}$. (a) Prove this result without invoking the min-max theory. (b) Prove a version of the min-max theorem with the $A$-inner product, i.e., prove that the min-max theorem is valid for any inner product for which $A$ is self-adjoint.

9. Consider the Laplacean of a graph as defined in Section 13.6. Show that

$$
(Lx, x) = \sum_{(i,j) \in E} (x_i - x_j)^2.
$$

10. Consider a rectangular finite difference mesh, with mesh size $\Delta x = h$ in the $x$-direction and $\Delta y = h$ closest to the $y$-direction.

a. To each mesh point $p = (x_i, y_j)$, associate the closed disk $D_{ij}$ of radius $h$ centered at $p_i$. What is the smallest $k$ such that the family $\{D_{ij}\}$ is a $k$-ply system?

b. Answer the same question for the case where the radius is reduced to $h/2$. What is the overlap graph (and associated mesh) for any $\alpha$ such that

$$
\frac{1}{2} < \alpha < \frac{\sqrt{2}}{2}?
$$

What about when $\alpha = 2$?

11. Determine the cost of a level set expansion algorithm starting from $p$ distinct centers.

12. Write a FORTRAN subroutine (or C function) which implements the Recursive Graph Partitioning algorithm.

13. Write recursive versions of the Recursive Graph Partitioning algorithm and Recursive Spectral Bisection algorithm. [Hint: Recall that a recursive program unit is a subprogram or function, say $foo$, which calls itself, so $foo$ is allowed to make a subroutine call to $foo$ within its body. Recursivity is not allowed in FORTRAN but is possible in C or C++.] (a) Give a pseudo-code for the RGB algorithm which processes the subgraphs in any order. (b) Give a pseudo-code for the RGB algorithm case when the larger subgraph is to be processed before the smaller one in any dissection. Is this second version equivalent to Algorithm 13.9?

**Notes and References.** To start with, the original paper by Schwarz is the reference [193], but an earlier note appeared in 1870. In recent years, research on Domain Decomposition techniques has been very active and productive. This rebirth of an old technique has been in large part motivated by parallel processing. However, the first practical use of Domain Decomposition ideas has been in applications to very large structures; see [166, 29], and elasticity problems; see, e.g., [169, 205, 198, 51, 28] for references.
Two recent monographs that describe the use of Domain Decomposition approaches in structural mechanics are [143] and [87]. Recent survey papers include those by Keyes and Gropp [135] and another by Chan and Matthew [50]. The recent volume [136] discusses the various uses of “domain-based” parallelism in computational sciences and engineering.

The bulk of recent work on Domain Decomposition methods has been geared toward a Partial Differential Equations viewpoint. Often, there appears to be a dichotomy between this viewpoint and that of “applied Domain Decomposition,” in that the good methods from a theoretical point of view are hard to implement in practice. The Schwarz multiplicative procedure, with multicoloring, represents a compromise between good intrinsic properties and ease of implementation. For example, Venkatakrishnan concludes in [215] that although the use of global coarse meshes may accelerate convergence of local, domain-based, ILU preconditioners, it does not necessarily reduce the overall time to solve a practical aerodynamics problem.

Much is known about the convergence of the Schwarz procedure; refer to the work by Widlund and co-authors [30, 72, 73, 74, 46]. The convergence results of Section 13.3.4 have been adapted from Xu [230] as well as Hackbusch [116]. The result on the equivalence between Schwarz and Schur complement iterations stated in Theorem 13.1 seems to have been originally proved by Chan and Goovaerts [48]. The results on the equivalence between the full matrix techniques and the Schur matrix techniques seen in Section 13.5 have been adapted from results by S. E. Eisenstat, reported in [135]. These connections are rather interesting and useful in practice since they provide some flexibility on ways to implement a method. A number of preconditioners have also been derived using these connections in the PDE framework [32, 31, 33, 34, 35].

Research on graph partitioning is currently very active. So far, variations of the Recursive Spectral Bisection algorithm [165] seem to give the best results in terms of overall quality of the subgraphs. However, the algorithm is rather expensive, and less costly multilevel variations have been developed [119]. Alternatives of the same class as those presented in Section 13.6.4 may be quite attractive for a number of reasons, including cost, ease of implementation, and flexibility; see [107]. There is a parallel between the techniques based on level set expansions and the ideas behind Voronoi diagrams known in computational geometry. The description of the geometric partitioning techniques in Section 13.6.2 is based on the recent papers [105] and [150]. Earlier approaches have been developed in [55, 56, 57].
REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


219. V. V. Voevodin. The problem of a non-selfadjoint generalization of the conjugate gradient method has been closed. USSR Computational Mathematics and Mathematical Physics, 23:143–144, 1983.


INDEX

A
additive projection procedure, 136
ADI, 116
Peaceman-Rachford algorithm, 117
adjacency graph, 71
of PDE matrices, 71
adjoint of a matrix, 7
algebraic multiplicity, 15
Alternating Direction Implicit, see ADI
angle between a vector and a subspace, 130
anisotropic medium, 47
approximate inverse preconditioners, 297
column-oriented, 300
global iteration, 298
for improving a preconditioner, 308
approximate inverse techniques, 375
Arnoldi’s method, 146–157
basic algorithm, 146
breakdown of, 148
with Householder orthogonalization, 149
for linear systems, 151
lucky breakdown, 148
with Modified Gram-Schmidt, 148
practical implementation, 148
Arrow-Hurwicz’s Algorithm, 241
assembled matrix, 60
assembly process, 59
bidirectional matrices, 5
bilinear form, 56
biorthogonal bases, 35
biorthogonal vectors, 35, 205
biorthogonalization, 204
bipartite graph, 82, 112
block Arnoldi
algorithm, 196
Ruhe’s variant, 197
block diagonal matrices, 5
block FOM, 199
block Gaussian elimination, 385–388
algorithm, 388
block GMRES, 199–200
multiple right-hand sides, 199
block Gram-Schmidt, 197
block Jacobi, 102
as a preconditioner, 353
block Krylov subspace methods, 144, 196–200
block preconditioners, 309
block relaxation, 98
block tridiagonal matrices, 5, 309
preconditioning, 309
boundary conditions, 45, 46
Dirichlet, 46
mixed, 46
Neumann, 46
banded matrices, 5
bandwidth
of a bus, 327
of a matrix, 5
basis of a subspace, 10
BCG, 209–213
algorithm, 210
transpose-free variants, 213–226
BICGSTAB, 216
Biconjugate Gradient, see BCG

C
cache memory, 327
canonical form, 15
Jordan, 16
Schur, 17
Cauchy-Schwartz inequality, 6, 8
Cayley-Hamilton theorem, 144
cell-centered scheme, 64
cell-vertex scheme, 64
centered difference approximation, 48
centered difference formula, 48
centerpoint, 415
CG algorithm, see Conjugate Gradient algorithm
CG for normal equations, 236, 237
CGNE, 237
algorithm, 238
optimality, 238
CGNR, 236
algorithm, 236
optimality, 236
CGS, 214–216
algorithm, 216
characteristic polynomial, 3
Chebyshev acceleration, 358
Chebyshev polynomials, 186–192, 194, 356–364
complex, 188, 203
and ellipses, 188
optimality, 189–191
for preconditioning, 356
real, 187
Cimmino’s method, 233
Circuit switching, 328
coarse-grain, 353
coefficient matrix, 95
coloring vertices, 81
column reordering, 74
Compressed Sparse Column storage, see CSC
Compressed Sparse Row storage, see CSR
Concus, Golub, and Widlund algorithm, 260
condition number, 40
for normal equation systems, 230
condition numbers and CG, 180
Conjugate Gradient algorithm, 174–181
algorithm, 178
alternative formulations, 178
convergence, 191, 192
derivation, 174, 177
eigenvalue estimates, 180
for the normal equations, 236
preconditioned, 244
Conjugate Gradient Squared, see CGS
Conjugate Residual algorithm, 181
consistent matrix norms, 8
consistent orderings, 112–116
control volume, 63
convection-diffusion equation, 47
convergence factor, 105
general, 105
specific, 105
of GMRES, 193
of the Minimal Residual method, 135
rate, 105
of relaxation methods, 104
of Schwarz procedures, 402
COO storage scheme, 84
coordinate storage format, see COO
Courant characterization, 26
Craig’s method, 238
CRAY T3D, 329
CSC storage format, 85
matvecs in, 335
CSR storage format, 85, 272
matvecs in, 335
cut-edges, 416
Cuthill-McKee ordering, 77
D
data coherence, 327
data-parallel, 326
defective eigenvalue, 15
derogatory, 15
determinant, 3
diagonal storage format, 85, 338
matvecs in, 338
diagonal
compensation, 285
dominance, 108, 109
form of matrices, 16
matrices, 5
diagonal storage format, see DIA
diagonalizable matrix, 16
diagonally dominant matrix, 109
diagonally structured matrices, 85
diameter of a graph, 417
diameter of a triangle, 58
DIOM, 154–157, 175
algorithm, 156
direct IOM, see DIOM
direct sum of subspaces, 10, 33
directed graph, 71
Dirichlet boundary conditions, 45, 46
distributed computing, 325
ILU, 372
memory, 328
sparse matrices, 341, 373
divergence of a vector, 46
divergence operator, 46
INDEX

domain decomposition
  convergence, 402
  and direct solution, 387
full matrix methods, 411
induced preconditioners, 407
Schur complement approaches, 406
Schwarz alternating procedure, 394
domain sweep, 396
double orthogonalization, 148
double-striping, 418
DQGMRES, 168–172, 258
  algorithm, 169

E
  EBE preconditioner, 376
  EBE regularization, 377
  edge in a graph, 71
eigenspace, 10
eigenvalues, 3
  definition, 3
  from CG iteration, 180
  index, 16, 17
  of an orthogonal projector, 37
eigenvector, 3
  left, 4
  right, 4
Eisenstat’s implementation, 248, 263
Eisenstat’s trick, see Eisenstat’s implementation
Element-By-Element preconditioner, see EBE preconditioner
ELL storage format, 86
  matvecs in, 339
Ell storage format, 339
elliptic operators, 44
Ellpack-Itpack storage format, see ELL storage format
energy norm, 32, 236, 238
error projection methods, 129
Euclidean inner product, 6
Euclidean norm, 7

F
  Faber-Manteuffel theorem, 184
  factored approximate inverse, 306
  fast solvers, 47, 383
  FGMRES, 255–258
    algorithm, 256
  fictitious domain methods, 387
  Fiedler vector, 416
  field of values, 23
  fill-in elements, 275
  fine-grain algorithms, 353
  finite difference scheme, 47
    for 1-D problems, 50
    for 2-D problems, 54
    for the Laplacean, 49
    upwind schemes, 51
  finite element method, 44, 55
  finite volume method, 63
  flexible GMRES, see FGMRES
  flexible iteration, 255
  flux vector, 63
  FOM, 151
    algorithm, 152
    with restarting, 153
  Frobenius norm, 8
  frontal methods, 60, 376
  full matrix methods, 411–413
  Full Orthogonalization Method, see FOM

G
  Galerkin conditions, 124
  Gastinel’s method, 139
gather operation, 336
Gauss-Seidel iteration, 95
  backward, 97
  for normal equations, 231
  in parallel, 378
  symmetric, 97
  Gaussian elimination, 60, 176, 269–273, 278,
    282, 283, 285–287, 368, 369, 383
    block, 385
    frontal methods, 376
    IKJ variant, 271
    in IOM and DIOM, 156
    in Lanczos process, 176
    parallel, 409
    parallelism in, 71
    reordering in, 75
    in skyline format, 295
    sparse, 70
  GCR, 182–184
  Generalized Conjugate Residual, see GCR
  geometric multiplicity, 15
  Gershgorin discs, 110
  Gershgorin’s theorem, 109
global iteration, 298–300, 305
global reduction operations, 332
GMRES, 157–172, 184, 193–196
  algorithm, 158
  block algorithm, 199
breakdown, 163, 164
convergence, 193
flexible variant, 250, 255–258
Householder version, 158
lucky breakdown, 164
parallel implementation, 331
with polynomial preconditioning, 363
practical implementation, 160
relation with FOM, 164, 166
with restarting, 167
stagnation, 167
truncated, 168
grade of a vector, 144
Gram-Schmidt algorithm, 11–12, 314
block, 197
cancellations in, 148
modified, 11
standard, 11
graph, 71
bipartite, 82
coloring, 81, 403
directed, 71
edges, 71
Laplacean of a, 416
partitioning, 382, 413
goemetric, 414
graph theory techniques, 417
spectral techniques, 416
type, 384
undirected, 71
vertices, 71
idempotent, 10, 33
if and only if, 3
iff, see if and only if
ILQ
factorization, 315
preconditioning, 314
ILU, 268–297
distributed, 372
factorization, 268
instability in, 293, 297
general algorithm, 270
IKJ version, 272
ILUS, 294–297
algorithm, 296
modified, 285–286
preconditioner, 268
for Schur complement, 409
static pattern, 273
with threshold, see ILUT and ILUP
with multi-elimination, see ILUM
zero pattern, 270
ILU(0), 265, 268, 274–276
algorithm, 275
distributed factorization, 374
for distributed sparse matrices, 373
for red-black ordering, 366
ILU(1), 278
ILUM, 370
ILUT, 286–293
algorithm, 287
analysis, 288
implementation, 290
with pivoting, see ILUTP
ILUTP, 293
for normal equations, 312
incomplete
orthogonalization
algorithm, 154
incomplete factorization, 265, 268
Gram-Schmidt, 315
ILQ, 314, 315
QR, 315
incomplete Gram-Schmidt, 316
Incomplete LQ, see ILQ
Incomplete LU, see ILU
Incomplete Orthogonalization Method, see IOM
indefinite inner product, 207
independent set orderings, 79
independent sets, 79, 368
maximal, 80
INDEX

index of an eigenvalue, 16, 17
indirect addressing, 69
induced norm, 8
induced preconditioners, 407
inhomogeneous medium, 47
inner products, 5
    indefinite, 207
invariant subspace, 10, 130
inverse LU factors, 306
IOM, 154
    algorithm, 154
direct version, 154
irreducibility, 83
irreducible, 27
isometry, 7
iteration matrix, 102, 104

J
j-diagonal, 340
Jacobi iteration, 95
    for the normal equations, 233
JAD storage format, 340
    definition, 340
in level scheduling, 348
matvecs in, 341
jagged diagonal format, see JAD storage format
jagged diagonals, 340
Jordan block, 17
Jordan box, 17
Jordan canonical form, 16
Jordan submatrix, 17
Joukowski mapping, 188

K
kernel, 9, 10
Krylov subspace, 144
    dimension of a, 144
invariant, 145
methods, 143
Krylov subspace methods, 204

L
Lanczos algorithm, 172, 173
    algorithm, 173, 205
biorthogonalization, 204
breakdown, 206
incurable, 207
lucky, 207
serious, 207
for linear systems, 208
look-ahead version, 207
loss of orthogonality, 173
modified Gram-Schmidt version, 173
nonsymmetric, 204
and orthogonal polynomials, 173
partial reorthogonalization, 173
practical implementations, 207
selective reorthogonalization, 173
symmetric case, 172
Laplacean, see Laplacean operator
Laplacean operator, 46, 55
    of a graph, 416
least-squares polynomials, 359
least-squares problem, 229
left eigenvector, 4
left versus right preconditioning, 255
level of fill-in, 278
level scheduling, 345–348
    for 5-point matrices, 345
    for general matrices, 346
level set orderings, 76, 417
line relaxation, 99
linear mappings, 2
linear span, 9
linear system, 38, 95
    existence of a solution, 38
    right-hand side of a, 38
    singular, 38
    unknown of a, 38
linked lists, 88
local Schur complement, 393
Look-ahead Lanczos algorithm, 207
lower triangular matrices, 5
LQ factorization, 314
    algorithm, 315
lucky factorizations, 148

M
mask, 320
matrix, 1
    addition, 2
    adjoint of a, 7
    banded, 5
    bidiagonal, 5
canonical forms, 15
characteristic polynomial, 3
diagonal, 5
diagonal dominant, 108
diagonal form, 16
diagonalizable, 16
Hermitian, 4, 21, 24
Hessenberg, 5
irreducible, 83
Jordan canonical form, 16
multiplication, 2
nonnegative, 4, 26
nonsingular, 3
norm of a, 8
normal, 4, 21
orthogonal, 5
outer product, 5
positive definite, 30–32
powers of a, 19
reduction, 15
Schur form, 17
self-adjoint, 7, 403
singular, 3
skew-Hermitian, 4
skew-symmetric, 4
spectral radius, 4
spectrum, 3
square, 3
symmetric, 4
Symmetric Positive Definite, 31, 112
trace, 4
transpose, 2
transpose conjugate, 2
triangular, 5
tridiagonal, 5
unitary, 4
matrix-by-vector product, 334
dense matrices, 334
for distributed matrices, 344
in DIA format, 338
in Ellpack format, 339
in triad form, 339
mesh generation, 61
mesh size, 58
message passing, 328
MILU, 285–286
minimal degree ordering, 88
Minimal Residual iteration, 133
algorithm, 134
convergence, 135
min-max theorem, 24
mixed boundary conditions, 45, 46
$M$-matrix, 26, 269, 310
modified Gram-Schmidt, 148
Modified ILU, see MILU
Modified Sparse Row storage, see MSR
molecule, 48
moment matrix, 208
in Lanczos procedure, 208
MR iteration, see Minimal Residual iteration
MSR storage format, 85
multi-elimination, 368, 369
multicolor orderings, 81
multicoloring, 364–368
for general sparse matrices, 367
multifrontal methods, 381
multinode expansion algorithm, 420
multiple eigenvalue, 15
multiple vector pipelines, 325
multiplicative projection process, 138
multiplicative Schwarz preconditioning, 399
multiprocessing, 325
N
natural ordering, 54
near singularity, 40
nested-dissection ordering, 88
Neumann boundary conditions, 45, 46
Neumann polynomials, 355
nonnegative matrix, 4, 26
nonsingular matrix, 3
norm
Euclidean, 7
Hölder, 7
induced, 8
of matrices, 8
$p$-norm, 8
of vectors, 5
normal derivative, 56
normal equations, 229
normal matrix, 4, 21
null space, 9, 10
of a projector, 33
O
Object Oriented Programming, 334
oblique projection methods, 204
oblique projector, 35
operator
elliptic, 44
Laplacean, 46
optimality of projection methods, 126
order relation for matrices, 26
ORTHODIR, 182–184
orthogonal
complement, 10
matrix, 5
projector, 10, 35
vectors, 10
orthogonality, 10
between vectors, 10
of a vector to a subspace, 10
ORTHOMIN, 182–184
orthonormal, 10
outer product matrices, 5
overdetermined systems, 229
overlapping domains, 385
over-relaxation, 97

P
p-norm, 8
packet switching, 328
parallel architectures, 326
parallel sparse techniques, 72
parallelism, 324
forms of, 324
partial differential equations, 44
partial Schur decomposition, 18
partition, 100
partition vector, 416
partitioning, 384
PDE, see partial differential equations
PE, see Processing Element
Peaceman-Rachford algorithm, 117
peripheral node, 417
permutation matrices, 5, 73
permutations, 72
Perron-Frobenius theorem, 27
perturbation analysis, 39
Petrov-Galerkin conditions, 122–124
physical mesh versus graph, 72
pipelining, 324
polynomial approximation, 144
polynomial preconditioning, 352, 354–364
positive definite matrix, 6, 25, 30–32
positive matrix, 26
positive real matrix, see positive definite matrix
positive semidefinite, 25
preconditioned
CG, 244
efficient implementations, 248
left, 246
for the normal equations, 259
parallel implementation, 330
split, 247
symmetry in, 245
fixed-point iteration, 103
GMRES, 250
comparison, 253
flexible variant, 255, 256
left preconditioning, 250
right preconditioning, 252
split preconditioning, 253
preconditioner, 103
preconditioning, 102, 244
EBE, 376
incomplete LU, 268
induced, 407
Jacobi, 265
normal equations for normal equations, 311
polynomial, 354–364
with Chebyshev polynomials, 356
with least-squares polynomials, 359
with Neumann polynomials, 355
and relaxation scheme, 103
SOR, 265
SSOR, 265
probing, 409
Processing Element (PE), 325
profile, 79
projection
operator, see projector
orthogonal to, 33
parallel to, 33
projection methods, 122
additive, 136
approximate problem, 123
definitions, 122
error bounds, 129
geneneral, 123
matrix representation, 124
multiplicative, 138
oblique, 122, 204
one-dimensional, 131
optimality, 126
orthogonal, 122, 124
prototype, 124
residual, 127
theory, 126
projector, 10, 32–38, 101
existence, 34
matrix representation, 35
oblique, 35
orthogonal, 35
eigenvalues, 37
properties, 37
prolongation operator, 101, 398
property A, 112
pseudo-peripheral node, 417
### Q
- QMR, 209–213  
  - algorithm, 212  
  - approximation, 212  
- QR decomposition, 11  
- Quasi-GMRES, 168  
  - algorithm, 168  
- Quasi-Minimal Residual, see QMR  
- quasi-Schur form, 18  
- quick-split, in ILUT, 291  
- quotient graph, 72

### R
- range, 2, 9, 10  
  - of a projector, 33  
- rank, 10  
  - full, 10  
- Rayleigh quotient, 23, 24  
- real Schur form, 18  
- recursive graph bisection, 418  
- red-black ordering, 364  
- reduced system, 318, 387  
- reducible, 27  
- reduction of matrices, 15  
- reduction operations, 332  
- reflectors, 12  
- regular splitting, 107  
- regularization, 241  
- relaxation methods  
  - block, 98  
  - convergence, 104  
- reordering, 74  
- reordering rows, columns, 72  
- reorthogonalization, 11  
- residual norm steepest descent, 135  
- residual projection methods, 127  
- restarted FOM, 153  
- restriction operator, 101, 397  
- reverse communication, 333  
- right versus left preconditioning, 255  
- right-hand side, 38, 95  
  - multiple, 199  
- row projection methods, 231, 378  
  - parallel, 378  
- row reordering, 74  
- row sum, 285

### S
- saddle-point problems, 238  
- SAXPY, 131, 301, 332  
  - parallel, 332  
- sparse, 301  
- scatter and gather operations, 336–337  
- Schur complement, 387  
  - approaches, 406  
  - and direct solution, 387  
  - for finite-element partitionings, 392  
  - local, 391  
  - methods, 407  
  - properties, 388  
  - for vertex partitionings, 389  
- Schur form, 17  
  - example, 18  
  - nonuniqueness, 19  
  - partial, 18  
  - quasi, 18  
  - real, 18  
- Schwarz alternating procedure, 385, 394  
  - additive, 401  
  - algorithm, 395  
  - multiplicative, 394  
- search subspace, 122  
- section of an operator, 145  
- self preconditioning, 301  
  - convergence behavior, 303  
- self-adjoint, 7, 403  
- semisimple, 15  
- separators, 414  
- set decomposition, 100  
- shared memory computers, 326  
- similarity transformation, 15  
- simple eigenvalue, 15  
- singular matrix, 3  
- singular values, 9  
- sites (in graph partitioning), 420  
- skew-Hermitian  
  - matrices, 4, 21, 186  
  - part, 31  
- skew-symmetric matrices, 4  
- skyline solvers, 79  
- SOR, 97  
  - convergence, 112  
  - iteration, 95  
  - multicolor sweep, 368  
  - for SPD matrices, 112  
- span of q vectors, 9  
- sparse, 59  
- sparse Gaussian elimination, 70, 88  
- sparse matrices  
  - adjacency graph, 70, 71  
  - basic operations, 86  
  - direct methods, 88
INDEX

graph representation, 70
matrix-by-vector operation, 87
permutation and reordering, 72
storage, 83–86
sparse matrix-by-vector product, 87
sparse skyline storage format, see SSK
sparse triangular system solution, 87
sparse-sparse mode computations, 300
sparse-sparse mode computations, 300
sparsity, 68
SPARSKIT, 89–91
SPD, see Symmetric Positive Definite
spectral bisection, 416
spectral radius, 4
spectrum of a matrix, 3
splitting, 97
square matrices, 3
SSK storage format, 295
SSOR, 97
steepest descent, 131
stencil, 48
stereographic projection, 415
Stieltjes algorithm, 174
stiffness matrix, 59, 61
Stokes problem, 240
storage format
COO, 84
CSC, 85
CSR, 85, 272
ELL, 86
MSR, 85
SSK, 295
storage of sparse matrices, 83–86
structured sparse matrix, 69
subdomain, 373
subspace, 9
direct sum, 10
of approximants, 122
of constraints, 122
orthogonal, 10
sum, 10
Successive Over-Relaxation, see SOR
symbolic factorization, 88
symmetric Gauss Seidel, 97
symmetric matrices, 4
Symmetric Positive Definite, 31, 112
Symmetric SOR, see SSOR
symmetric squaring, 315
symmetry in preconditioned CG, 245
test problems, 88
TFQMR, 219
algorithm, 224
topological sorting, 346
trace, 4
Transpose-Free QMR, see TFQMR
triad operation, 339
triangular systems, 344
distributed, 375
level scheduling, 346
sparse, 344
tridiagonal matrices, 5
U
unassembled matrix, 60
under-determined, 230
undirected graph, 71
unitary matrices, 4
unstructured sparse matrix, 69
upper triangular matrices, 5
upwind schemes, 51
Uzawa's algorithm, 239
V
variable preconditioner, 255
vector
computers, 325
operations, 331
orthogonality, 10
processors, 325
of unknowns, 95
updates, 131, 332
parallel, 332
vertex (in a graph), 71
W
wavefronts, 346
weak formulation, 56
weakly diagonally dominant matrix, 109
Winget regularization, 377
Z
Zarantonello's lemma, 189