Chapter 10
Incomplete Factorizations

They [incomplete factorizations] can be thought of as approximating the exact LU factorization of a given matrix A (e.g. computed via Gaussian elimination) by disallowing certain fill-ins. As opposed to other PDE-based preconditioners such as multigrid and domain decomposition, this class of preconditioners are primarily algebraic in nature and can in principle be applied to any sparse matrices. When applied to PDE problems, they are usually not optimal ... On the other hand, they are often quite robust. – Chan & van der Vorst (1997).

Having introduced incomplete factorization preconditioners in the previous chapter, the focus in this chapter is on different ways to compute such factorizations and their relationship to the complete factorizations used in sparse direct methods. We denote the incomplete factors by $\widetilde{L}$ and $\widetilde{U}$; in the SPD case, $\widetilde{U} = \widetilde{L}^T$. We assume that the sparsity patterns of A and its incomplete factors always include the positions of the diagonal entries.

10.1 ILU(0) Factorization

The simplest sparsity pattern for an incomplete factorization is $\mathcal{S}\{\widetilde{L} + \widetilde{U}\} = \mathcal{S}\{A\}$, that is, no entries in $\widetilde{L}$ or $\widetilde{U}$ are allowed outside the sparsity pattern of A and only entries in positions $(i, j) \in \mathcal{S}\{A\}$ are retained in the (incomplete) elimination matrices. The resulting incomplete factorization is called an ILU(0) factorization (or an IC(0) factorization if A is SPD).

Motivation for considering a sparsity pattern that is a superset of $\mathcal{S}\{A\}$ is given by the following straightforward but important result.

Theorem 10.1 (Chan & van der Vorst 1997; van der Vorst 2003) *Consider the incomplete LU factorization $A + E = \widetilde{L}\widetilde{U}$ with sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. The entries of the error matrix E are zero at positions $(i, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$.*

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_10

Proof The result clearly holds for $j = 1$. Let $(i, j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ and assume without loss of generality that $i > j > 1$. The (i, j) entry of $\widetilde{L}$ is computed as

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\, \tilde{u}_{kj} \right) / \tilde{u}_{jj}$$

with the sums over k implying $(i, k) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ and $(k, j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$. This gives

$$a_{ij} = \widetilde{L}_{i,1:j-1}\widetilde{U}_{1:j-1,j} + \tilde{l}_{ij}\tilde{u}_{jj} = \widetilde{L}_{i,1:j}\widetilde{U}_{1:j,j} = L_{i,1:j}U_{1:j,j},$$

and the corresponding entry of E is zero. □

A consequence of Theorem 10.1 is that extending $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ gives a larger set of entries of A for which the error is zero. This is attractive provided the incomplete factorization can still be computed and employed cheaply and does not require prohibitive amounts of memory. In some situations, there are straightforward ways to extend $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$. For example, consider a simple discretization of a PDE on a rectangular grid. The sparsity pattern of the corresponding SPD matrix A and its graph $\mathcal{G}(A)$ together with the first three steps of the Cholesky factorization of A (in which variables 1, 2, and 3 are eliminated in turn) are given in Figure 10.1. A has entries on the diagonal and four of its subdiagonals and the fill-in lies within $band(A)$. A natural choice is to allow $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ to include fill-in along a few additional diagonals within the band.

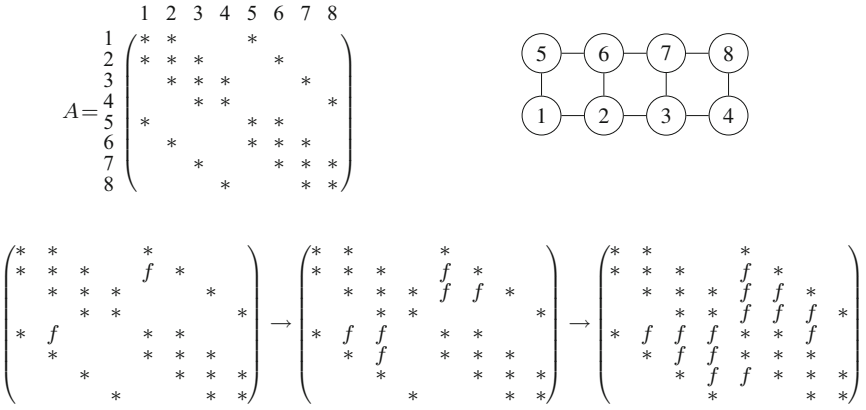

Figure 10.1 An 8×8 banded sparse SPD matrix A and its graph $\mathcal{G}(A)$. The first three steps of a Cholesky factorization are shown. Filled entries are denoted by f.

10.2 Basic Incomplete Factorizations

We start with the two basic incomplete factorizations. Here and elsewhere, section notation is used but operations are performed only on nonzero entries. The Crout variant given in Algorithm 10.1 computes $\widetilde{U}$ row-by-row and $\widetilde{L}$ column-by-column and sparsifies each row and column as soon as they are computed using a target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. The widely used variant outlined in Algorithm 10.2 constructs both $\widetilde{L}$ and $\widetilde{U}$ by rows. Prescribing an appropriate sparsity pattern in advance can be difficult. If it is not supplied, sparsification can be applied inside the k loops (for instance, entries with absolute value less than a chosen tolerance may be dropped) and the sparsity patterns of the factors updated as the factorization proceeds.

Algorithms 10.1 and 10.2 are straightforward to implement using sparse data structures. At major step i, Algorithm 10.2 computes $\widetilde{L}_{i,1:i-1}$ and $\widetilde{U}_{i,i+1:n}$; both rows can be held using a single auxiliary vector. Note that, in Algorithm 10.1, sparsification of the partially computed vectors is performed outside the k loops, whereas in Algorithm 10.2 it is inside the k loop. In practice, either approach can be used, leading to slightly different variants.

ALGORITHM 10.1 Crout incomplete LU factorization

Input: Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

for $j = 1 : n$ **do**
  $\tilde{l}_{jj} = 1$, $\widetilde{L}_{j+1:n,j} = A_{j+1:n,j}$
  $\widetilde{U}_{j,j:n} = A_{j,j:n}$
  for $k = 1 : j-1$ such that $(j,k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
    $\widetilde{U}_{j,j:n} = \widetilde{U}_{j,j:n} - \tilde{l}_{jk}\,\widetilde{U}_{k,j:n}$ ▷ Sparse linear combination
  end for
  Sparsify $\widetilde{U}_{j,j+1:n}$ ▷ Drop entries from row j of $\widetilde{U}$
  for $k = 1 : j-1$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
    $\widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j} - \tilde{u}_{kj}\,\widetilde{L}_{j+1:n,k}$ ▷ Sparse linear combination
  end for
  Sparsify $\widetilde{L}_{j+1:n,j}$ ▷ Drop entries from column j of $\widetilde{L}$
  $\widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j}/\tilde{u}_{jj}$
end for

ALGORITHM 10.2 Row incomplete LU factorization

Input: Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

for $i = 1 : n$ **do**
  $\tilde{l}_{ii} = 1$, $\widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$
  $\widetilde{U}_{i,i:n} = A_{i,i:n}$
  Sparsify $\widetilde{L}_{1,1:i-1}$ and $\widetilde{U}_{i,i+1:n}$
  for $k = 1 : i-1$ such that $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
    $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
    $\widetilde{L}_{i,k+1:i-1} = \widetilde{L}_{i,k+1:i-1} - \tilde{l}_{ik}\,\widetilde{U}_{k,k+1:i-1}$
    Sparsify $\widetilde{L}_{i,k+1:i-1}$
    $\widetilde{U}_{i,i:n} = \widetilde{U}_{i,i:n} - \tilde{l}_{ik}\,\widetilde{U}_{k,i:n}$
    Sparsify $\widetilde{U}_{i,i+1:n}$
  end for
end for

10.3 Incomplete Factorizations Based on the Shortest Fill-Paths

We next consider an incomplete LU factorization that uses a structure-based dropping strategy. Entries of the factors that correspond to nonzero entries of A are assigned the level 0, while each potential filled entry in position (i, j) is assigned a level as follows:

$$level(i,j) = \min_{1 \le k < \min\{i,j\}} (level(i,k) + level(k,j) + 1). \tag{10.1}$$

Given $\ell \ge 0$, during the factorization, a filled entry is permitted at position (i, j) provided $level(i, j) \le \ell$. The resulting **level-based** incomplete factorization is denoted by ILU(ℓ) (or IC(ℓ)); the basic row variant is given in Algorithm 10.3.

Figure 10.2 depicts $\mathcal{S}\{\widetilde{L} + \widetilde{L}^T\}$ for the IC(ℓ) factorization of A from the discretized Laplace equation on a square grid (see the smaller problem in (9.14)) and for a matrix with a more general symmetric sparsity structure. The fill-in is typically generated irregularly throughout the factorization: initially few updates are needed, but later steps involve many updates, leading to large amounts of dropping. Furthermore, the amount of fill-in can grow quickly with increasing ℓ and, as a result, ℓ is typically small and level-based dropping is often combined with threshold-based dropping or with sparsifying A before the factorization commences (for example, by discarding entries of A with small absolute values).

ALGORITHM 10.3 Level-based incomplete LU factorization
Input: Matrix A and the level parameter $\ell \geq 0$.
Output: ILU(ℓ) factorization $A \approx \widetilde{L}\widetilde{U}$.

1: Initialise $level$ to 0 for nonzeros and diagonal entries of A and to $n+1$ otherwise
2: **for** $i = 1 : n$ **do** ▷ Loop over rows
3: $\tilde{l}_{ii} = 1$, $\widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$ and $\widetilde{U}_{i,i:n} = A_{i,i:n}$ ▷ Initialise row i of $\widetilde{L}$ and $\widetilde{U}$
4: **for** $k = 1 : i-1$ such that $level(i,k) \leq \ell$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
6: **for** $j = k+1 : i-1$ **do**
7: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ and update $level(i,j)$
8: **end for**
9: **for** $j = i : n$ **do**
10: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ and update $level(i,j)$
11: **end for**
12: **end for**
13: **for** $k = 1 : i-1$ **do** ▷ Drop entries in row i for which $level$ is too high
14: **if** $level(i,k) > \ell$ **then** $\tilde{l}_{ik} = 0$
15: **end for**
16: **for** $k = i : n$ **do**
17: **if** $level(i,k) > \ell$ **then** $\tilde{u}_{ik} = 0$
18: **end for**
19: **end for**

The level-based strategy comes from observing that in practical examples the absolute values of the entries in the factors in positions for which $level$ is large are often small. This is the case for model problems arising from discretized PDEs. A closer look shows a surprising connection between the level-based ILU factorization and the complete factorization: entries with large values of $level$ correspond to long fill-paths. This is expressed in Theorem 10.2, which allows the sparsity patterns of the incomplete factors to be determined a priori.

Theorem 10.2 (Hysom & Pothen 2002) *Consider the ILU(ℓ) factorization of A. $level(i,j) = k$ for some $k \leq \ell$ if and only if there is a shortest fill-path $i \Longrightarrow j$ of length $k+1$ in the adjacency graph $\mathcal{G}(A)$.*

Algorithm 10.4 outlines finding the pattern of row i of $\widetilde{U}$; finding the pattern of columns of $\widetilde{L}$ is analogous. Only $\mathcal{G}(A)$ is required, and hence the sparsity pattern of each row in the factor can be computed independently, in parallel. The algorithm operates via a simple breadth-first search that finds a shortest path between vertex

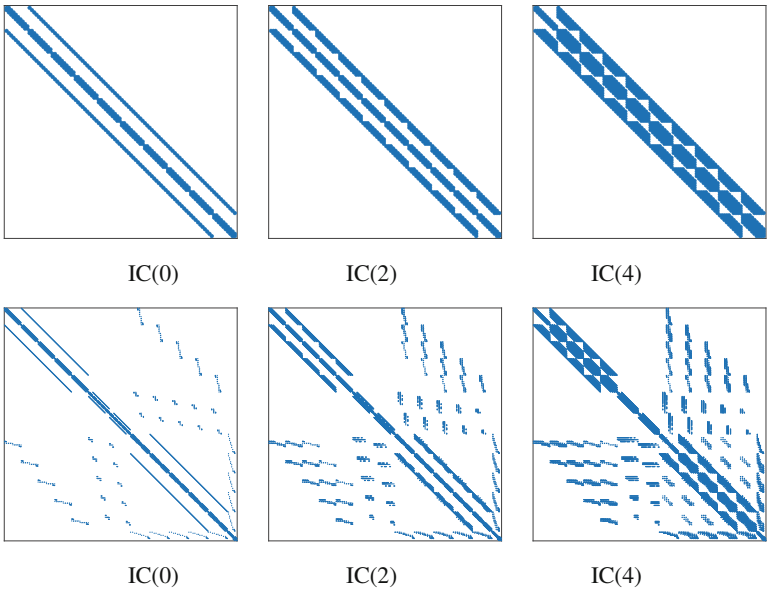

Figure 10.2 The sparsity patterns of the IC(ℓ) factors of A from the discretized Laplace equation on a square grid (top) and a more general symmetric sparse matrix (bottom).

i and vertices reachable from i via a graph traversal of $l + 1$ or fewer edges. The correctness of the algorithm follows from Theorem 10.2.

10.4 Modifications Based on Maintaining Row Sums

We assume in this section that the target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ contains $\mathcal{S}\{A\}$. **Modified incomplete factorizations** (MILU or MIC in the SPD case) seek to maintain equality between the row sums of A and $\widetilde{L}\widetilde{U}$, that is, $\widetilde{L}\widetilde{U}e = Ae$ (e is the vector of all ones). Rather than discarding potential fill-in outside the target sparsity pattern, the approach subtracts it from the diagonal entries of $\widetilde{U}$; this is outlined in Algorithm 10.5. Note that an MILU factorization may break down. If the target sparsity pattern corresponds to that of an ILU(ℓ) factorization, then an MILU(ℓ) factorization is computed.

Equality of the row sums of A and $\widetilde{L}\widetilde{U}$ can be seen as follows. If all the filled entries are retained (that is, $\mathcal{S}\{\widetilde{L} + \widetilde{U}\} = \mathcal{S}\{L + U\}$), then the claim holds trivially. Now assume some filled entries are not kept. If an entry in column j of row i of A belongs to the target sparsity pattern, then its value is modified in Step 8 if $i \leq j$ or in Step 15 if $i > j$. Otherwise, the i-th diagonal entry of $\widetilde{U}$ is modified (Step 10 or Step 17). In each case, $\tilde{l}_{ik}\,\tilde{u}_{kj}$ is subtracted from entries of the i-th row of the incomplete factors. Consider row i of $\widetilde{L}\widetilde{U}$. This product is given by

ALGORITHM 10.4 Find the sparsity pattern of row i of the ILU(ℓ) factor $\widetilde{U}$ of A
Input: Graph $\mathcal{G}(A)$, the level parameter $\ell \geq 0$ and row index i.
Output: Sparsity pattern $\mathcal{S}\{\widetilde{U}_{i,i:n}\}$ of row i of the ILU(ℓ) factorization $A \approx \widetilde{L}\widetilde{U}$.

```
S{Ũ_{i,i:n}} = {i}, Q = {i}                          ▷ Queue holds i initially
length(i) = 0
visited(i) = i
while Q is not empty do
    pop(Q, k)                                        ▷ Take k from the queue
    for j ∈ adj_{G(A)}(k) with visited(j) ≠ i do
        visited(j) = i
        if j < i and length(k) < ℓ then
            append(Q, j)                             ▷ Add j to the queue
            length(j) = length(k) + 1
        else if j > i then
            S{Ũ_{i,i:n}} = S{Ũ_{i,i:n}} ∪ {j}        ▷ Add j to the sparsity pattern of row i
        end if
    end for
end while
```

$$\sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j}^{n} \tilde{u}_{jk} = \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jj} + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} + \sum_{k=i}^{n} \tilde{u}_{ik} =$$

$$= \sum_{j=1}^{i-1} \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\tilde{u}_{kj} \right) + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} + \sum_{k=i}^{n} \left(a_{ik} - \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jk} \right)$$

$$= \sum_{j=1}^{n} a_{ij} + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} - \left(\sum_{j=1}^{i-1} \sum_{k=1}^{j-1} \tilde{l}_{ik}\tilde{u}_{kj} + \sum_{k=i}^{n} \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jk} \right).$$

Rearranging the indices in the double summations, the last three sums cancel out. Moreover, the added double summation is the sum of all the modification terms $\tilde{l}_{ik}\,\tilde{u}_{kj}$ in Algorithm 10.5, and the sum of the two subtracted double summations also comprises all the modification terms. Consequently, the row sums of A are preserved in the product of the incomplete factors.

Theorem 10.3 provides motivation for maintaining constant row sums in the case of a model PDE problem. The result is also valid for Neumann or mixed boundary conditions, and there are extensions to three-dimensional problems and MIC(ℓ)

ALGORITHM 10.5 Modified incomplete factorization (MILU)

Input: Matrix $A = L_A + D_A + U_A$ (see (9.6)) and a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ containing $\mathcal{S}\{A\}$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

1: $\tilde{l}_{ij} = (I + L_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{L})$ ▷ $\mathcal{S}(L_A) \subseteq \mathcal{S}(\widetilde{L})$
2: $\tilde{u}_{ij} = (D_A + U_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{U})$ ▷ $\mathcal{S}(U_A) \subseteq \mathcal{S}(\widetilde{U})$
3: **for** $k = 1 : n - 1$ **do**
4: **for** $i = k + 1 : n$ such that $(i, k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$ ▷ Check that $\tilde{u}_{kk}$ is nonzero
6: **for** $j = i : n$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
7: **if** $(i, j) \in \mathcal{S}\{\widetilde{U}\}$ **then**
8: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
9: **else**
10: $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ ▷ Modify diagonal instead of creating fill-in
11: **end if**
12: **end for**
13: **for** $j = k + 1 : i - 1$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
14: **if** $(i, j) \in \mathcal{S}\{\widetilde{L}\}$ **then**
15: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
16: **else**
17: $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ ▷ Modify diagonal instead of creating fill-in
18: **end if**
19: **end for**
20: **end for**
21: **end for**

with $\ell > 0$. However, although Theorem 10.1 holds for MILU factorizations, the approach may not be useful for general A.

Theorem 10.3 (Gustafsson 1978; Bern et al. 2006) *Let A come from a discretized Poisson problem on a uniform two-dimensional rectangular grid with Dirichlet boundary conditions and discretization parameter h. Then the condition number $\kappa((\widetilde{L}\widetilde{U})^{-1}A)$ for the level-based MIC(0) preconditioner is $O(h^{-1})$.*

Optionally, in Steps 10 and 17 of Algorithm 10.5, the update term $\tilde{l}_{ik}\,\tilde{u}_{kj}$ may be multiplied by a parameter θ ($0 < \theta < 1$) before it is subtracted from the diagonal entry $\tilde{u}_{ii}$. The introduction of θ was proposed as a practical way to extend MILU to linear systems not coming from discretized PDEs. Clearly, using $\theta < 1$ reduces the amount by which the diagonal entries are modified.

10.5 Dynamic Compensation

As discussed in Section 9.4.1, dropping entries can lead to breakdown. One way to avoid this (in exact arithmetic) is to dynamically modify the computed entries; this is outlined as Algorithm 10.6. Instead of accepting a filled entry in position (i, j), the idea is to add a (weighted) multiple of its absolute value to the corresponding diagonal entries $\tilde{u}_{ii}$ and $\tilde{u}_{jj}$. Provided the number of modifications is small, this can be useful if A is a diagonally dominant matrix and scaled so that its diagonal entries are nonnegative. The parameter ω controls the amount by which the diagonal entries of $\widetilde{U}$ are modified, but if $\omega < 1$, then breakdown can still occur. Dynamic compensation can be successful when incorporated into an IC factorization of

ALGORITHM 10.6 ILU factorization with dynamic compensation

Input: Matrix $A = L_A + D_A + U_A$ (see (9.6)), a target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and parameter ω $(0 \le \omega \le 1)$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

$\tilde{l}_{ij} = (I + L_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{L})$
$\tilde{u}_{ij} = (D_A + U_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{U})$
for $k = 1 : n - 1$ **do**
  for $i = k + 1 : n$ such that $(i, k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
    $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
    for $j = i : n$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
      if $(i, j) \in \mathcal{S}\{\widetilde{U}\}$ **then**
        $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
      else
        $\rho = (\tilde{u}_{ii}/\tilde{u}_{jj})^{1/2}$
        $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega\rho\,|\tilde{l}_{ik}\,\tilde{u}_{kj}|, \quad \tilde{u}_{jj} = \tilde{u}_{jj} + \omega|\tilde{l}_{ik}\,\tilde{u}_{kj}|\,/\rho, \quad \tilde{u}_{ij} = 0.$
      end if
    end for
    for $j = k + 1 : i - 1$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
      if $(i, j) \in \mathcal{S}\{\widetilde{L}\}$ **then**
        $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
      else
        $\rho = (\tilde{u}_{ii}/\tilde{u}_{jj})^{1/2}$
        $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega\rho\,|\tilde{l}_{ik}\,\tilde{u}_{kj}|, \quad \tilde{u}_{jj} = \tilde{u}_{jj} + \omega|\tilde{l}_{ik}\,\tilde{u}_{kj}|\,/\rho, \quad \tilde{l}_{ij} = 0.$
      end if
    end for
  end for
end for

an SPD matrix A because the resulting local modifications correspond to adding positive semidefinite matrices to A. In practice, the behaviour of the resulting preconditioner can be very different from that computed using the MIC approach of the previous section.

A related scheme, called **diagonally compensated reduction**, modifies A before the factorization begins by adding the values of all of its positive off-diagonal entries to the corresponding diagonal entries and then setting these off-diagonal entries to zero. If A is SPD, then the resulting matrix is a symmetric M-matrix and the incomplete factorization will not break down (Theorem 9.4). However, the modified matrix may be too far from A for its incomplete factors to be useful.

10.6 Memory-Limited Incomplete Factorizations

We next consider a more sophisticated modification scheme that introduces the use of intermediate memory that is employed during the construction of the incomplete factors but is then discarded. The aim is to obtain a high quality preconditioner while maintaining sparsity and allowing the user to control how much memory is used (both in the construction of the preconditioner and in the incomplete factor $\widetilde{L}$). Let the matrix A be SPD and consider the decomposition

$$A = (\widetilde{L} + \widetilde{R})\,(\widetilde{L} + \widetilde{R})^T - E.$$

Here the incomplete factor $\widetilde{L}$ is a lower triangular matrix with positive diagonal entries, $\widetilde{R}$ is a strictly lower triangular matrix with "small" entries, and the error matrix is $E = \widetilde{R}\widetilde{R}^T$. At each step, the next column of $\widetilde{L}$ is computed, and then the remaining Schur complement is modified. On step j of the incomplete factorization, the first column of the Schur complement $S^{(j)}$ is split into the sum

$$\widetilde{L}_{j:n,j} + \widetilde{R}_{j:n,j},$$

where $\widetilde{L}_{j:n,j}$ contains the entries that are retained in column j of the final incomplete factorization, $(\widetilde{R})_{jj} = 0$ and $\widetilde{R}_{j+1:n,j}$ contains the entries that are discarded. If a complete factorization was being computed, then the Schur complement would be updated by subtracting

$$(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})\,(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})^T.$$

However, the incomplete factorization discards the term

$$E^{(j)} = \widetilde{R}_{j+1:n,j}\,\widetilde{R}^T_{j+1:n,j}.$$

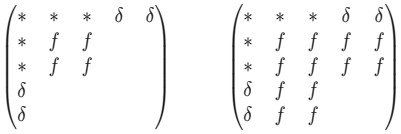

Figure 10.3 An illustration of the fill-in in a standard sparsification-based IC factorization (left) and in the approach that uses intermediate memory (right) after one step of the factorization. Entries with a small absolute value in row and column 1 are denoted by δ. The filled entries are denoted by f.

Figure 10.4 On the left is an SPD matrix with an entry of small absolute in positions $(1, 3)$ and $(3, 1)$. In the centre is $\mathcal{S}\{\widetilde{L}\}$ computed using a standard IC factorization that drops the small entry δ at position $(3, 1)$ (there are no filled entries in this case). On the right is the lower triangular part of the elimination matrix after the first step of the incomplete factorization using intermediate memory. The filled entry is denoted by f.

Thus, the matrix $E^{(j)}$ is implicitly added to A, and because $E^{(j)}$ is positive semidefinite, the approach is naturally breakdown-free.

The obvious choice for $\widetilde{R}_{j+1:n,j}$ is the smallest off-diagonal entries in the column (those that are smaller in absolute value than a chosen tolerance). Then implicitly adding $E^{(j)}$ is combined with the standard steps of an IC factorization, with entries dropped from $\widetilde{L}$ after the updates have been applied to the Schur complement.

Figure 10.3 depicts the first step of this approach. In the first row and column, $*$ and δ denote the entries of $\widetilde{L}_{1:n,1}$ and $\widetilde{R}_{1:n,1}$, respectively. Because a standard sparsification scheme does not store the smallest entries, using such a scheme gives no fill-in in the rows and columns corresponding to the discarded entries; this is shown on the left. The fill-in in the factorization that uses intermediate memory is depicted on the right. Clearly, more filled entries are used in constructing $\widetilde{L}$.

This strategy enables the structure of the complete factorization to be followed more closely than is possible using a standard approach. This is illustrated in Figure 10.4. If the small entries at positions $(1, 3)$ and $(3, 1)$ are not discarded, then there is a filled entry in position $(3, 2)$ and this allows the incomplete factorization using intermediate memory to involve the (large) off-diagonal entries in positions $(5, 2)$ and $(6, 2)$ in the second step of the IC factorization.

Unfortunately, because the column $\widetilde{R}_{j+1:n,j}$ must be retained to perform the updates on the next step, the total memory requirements are essentially as for a

ALGORITHM 10.7 Crout memory-limited IC factorization
Input: SPD matrix A, memory control parameters $lsize > 0$ and $rsize \geq 0$.
Output: Incomplete Cholesky factorization $A \approx \widetilde{L}\widetilde{L}^T$.

$w_i = 0, \quad 1 \leq i \leq n$
for $j = 1 : n$ **do**
  for $i = j : n$ such that $a_{ij} \neq 0$ **do**
    $w_i = a_{ij}$
  end for
  for $k < j$ such that $\tilde{l}_{jk} \neq 0$ **do**
    for $i = j : n$ such that $\tilde{l}_{ik} \neq 0$ **do**
      $w_i = w_i - \tilde{l}_{ik}\tilde{l}_{jk}$
    end for
    for $i = j : n$ such that $\tilde{r}_{ik} \neq 0$ **do**
      $w_i = w_i - \tilde{r}_{ik}\tilde{l}_{jk}$
    end for
  end for
  for $k < j$ such that $\tilde{r}_{jk} \neq 0$ **do**
    for $i = j : n$ such that $\tilde{l}_{ik} \neq 0$ **do**
      $w_i = w_i - \tilde{l}_{ik}\tilde{r}_{jk}$
    end for
  end for
  Copy into $\widetilde{L}_{j:n,j}$ the $lsize + nz(A_{j:n,j})$ entries of w of largest absolute value
  Copy into $\widetilde{R}_{j+1:n,j}$ the $rsize$ entries of w that are the next largest in absolute value
  Scale $\tilde{l}_{jj} = (w_j)^{1/2}$, $\widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j}/\tilde{l}_{jj}$, $\widetilde{R}_{j+1:n,j} = \widetilde{R}_{j+1:n,j}/\tilde{l}_{jj}$
  Reset entries of w to zero.
end for
Optionally discard $\widetilde{R}$ ▷ $\widetilde{R}$ is often only used in the construction of $\widetilde{L}$

complete factorization. However, the memory can be reduced by introducing two drop tolerances so that only entries of absolute value at least τ_1 are kept in $\widetilde{L}$ and entries smaller than τ_2 are dropped from $\widetilde{R}$. The factorization is no longer guaranteed to be breakdown-free. Furthermore, the numbers of entries in $\widetilde{L}$ and $\widetilde{R}$ are not known a priori.

An alternative idea that limits both the number of entries in the incomplete factor and the intermediate memory is to fix the maximum number of entries in each column of $\widetilde{L}$ and $\widetilde{R}$. This is outlined in Algorithm 10.7. Here $lsize \geq 0$ and $rsize \geq 0$ are the maximum number of filled entries in each column of $\widetilde{L}$ and the maximum number of entries in each column of $\widetilde{R}$, respectively, and $nz(A_{j:n,j})$

denotes the number of entries in the lower triangular part of column j of A. The number of entries in $\widetilde{L}$ is less than $nz(A)+(n-1)lsize$ (where $nz(A)$ is the number of entries in the lower triangular part of A) and $\widetilde{R}$ has at most $(n-1)rsize$ entries. If the parameter $rsize$ is set to 0, then no intermediate memory is used but in general choosing $rsize > 0$ leads to the computed $\widetilde{L}$ being a higher quality preconditioner. In case of breakdown, the algorithm can incorporate the use of a global shift; see Algorithm 9.1.

10.7 Fixed-Point Iterations for Computing ILU Factorizations

The fixed-point ILU algorithm is fundamentally different from Gaussian elimination-based approaches. Given the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, the goal is to iteratively generate incomplete factors fulfilling the ILU property

$$(\widetilde{L}\widetilde{U})_{ij} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$$

(see Theorem 10.1). The idea is appealing because the entries of $\widetilde{L}$ and $\widetilde{U}$ can be computed iteratively in parallel using the constraints

$$\sum_{\substack{k=1 \\ (i,k),(k,j)\in\mathcal{S}\{\widetilde{L}+\widetilde{U}\}}}^{\min(i,j)} \tilde{l}_{ik}\,\tilde{u}_{kj} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\},$$

and the normalization $\tilde{l}_{ii} = 1$. Using the relations

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\,\tilde{u}_{kj}\right) / \tilde{u}_{jj}, \quad i > j, \tag{10.2}$$

$$\tilde{u}_{ij} = a_{ij} - \sum_{k=1}^{i-1} \tilde{l}_{ik}\,\tilde{u}_{kj}, \quad i \le j, \tag{10.3}$$

the approach can be formulated as a fixed-point iteration method of the form $w^{k+1} = f(w^k)$, $k = 0, 1, \ldots$, where w is a vector containing the unknowns $\tilde{l}_{ij}$ and $\tilde{u}_{ij}$. Each fixed-point iteration is called a **sweep**. Algorithm 10.8 outlines the method.

An important question is how to choose initial values for the factor entries. In some applications, a natural initial guess is available. For example, in time-dependent problems, the $\widetilde{L}$ and $\widetilde{U}$ computed in the previous time step may provide appropriate initial guesses for the current time step. In other cases, a possible strategy is to symmetrically scale A to have a unit diagonal and then take the initial $\widetilde{L}$

ALGORITHM 10.8 Fixed-point ILU factorization

Input: Matrix A, the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, and initial incomplete factors $\widetilde{L}$ and $\widetilde{U}$.
Output: Updated incomplete factors.

```
for (i, j) ∈ S{L̃ + Ũ} do
    Set l̃_ij and ũ_ij to the given initial values
end for
for sweep = 1, 2, ... do
    for (i, j) ∈ S{L̃ + Ũ} do
        if i > j then
            Compute l̃_ij using (10.2)
        else
            Compute ũ_ij using (10.3)
        end if
    end for
end for
```

and $\widetilde{U}$ to be the lower and upper parts of the scaled matrix, respectively. In practice, a few sweeps may be sufficient to generate preconditioners that are competitive in terms of quality to those generated via classical incomplete Gaussian elimination algorithms.

The following features differentiate the fixed-point ILU algorithm from classical methods and make it attractive for parallel computations on modern architectures.

- The algorithm is fine-grained, allowing for scaling to a very large number of processors, limited only by the number of nonzero entries in the target sparsity patterns.
- Preordering A is not needed to enhance parallelism, and thus orderings that improve the accuracy of the incomplete factorization can be used.
- The algorithm can utilize an initial guess for the ILU factorization.

To enhance the preconditioner quality, it is possible to interleave employing Algorithm 10.8 with a strategy that dynamically adapts $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ to the problem characteristics. In an iterative process based on highly parallel building blocks, this allows threshold-based ILU factorizations to be computed on parallel shared-memory architectures and enables the efficient use of streaming-based architectures such as GPUs.

10.8 Ordering in Incomplete Factorizations

Ordering algorithms designed for sparse direct solvers (see Chapter 8) can have a positive effect on the robustness and performance of preconditioned Krylov subspace methods. However, the best choice of ordering for an incomplete factorization preconditioner may not be the same as for a complete factorization, and although the effects of orderings and how much fill-in is allowed have been widely demonstrated, they are not yet fully understood.

When the natural (lexicographic) ordering is used, the incomplete triangular factors resulting from a no-fill ILU factorization can be highly ill-conditioned, even if the matrix A is well-conditioned. Allowing more fill-in in the factors, for example, using ILU(1) instead of ILU(0), may solve the problem, but it is not guaranteed. In some cases, preordering A can lead to more stable factors, and hence more effective preconditioners, but, again, this is not understood.

Minimum degree orderings (Section 8.1.2) are popular for direct methods, but for incomplete factorizations care is needed to ensure the dropping strategy is compatible with the ordering. This is because the rows (and columns) of the permuted matrix can have significantly different counts. In this situation, using memory-based dropping in which the maximum allowable number of filled entries in a row of $\widetilde{L}$ is the same for all rows may not be a good approach. An alternative strategy is to specify that the permitted fill-in is proportional to that of the complete factorization (which can be computed using Algorithm 4.3).

A level set ordering that reduces the bandwidth or profile of a matrix can be employed (Section 8.2). For complete factorizations, the fill-in in the factors can be much greater than for nested dissection or minimum degree, but for incomplete factorizations they can be highly effective. In particular, using an RCM ordering (Algorithm 8.3) is often found to lead to a higher quality preconditioner than using the natural ordering. RCM-based orderings are generally inexpensive to compute and can provide good reuse of computer caches.

Global orderings based on a divide-and-conquer approach and, in particular, nested dissection (Section 8.4) are important for complete factorizations. But such orderings cut local connections within the graph of A and, when used with incomplete factorizations, can lead to poor quality preconditioners. A global ordering that specifically targets incomplete factorizations is a **red–black** (or checker board) ordering. Consider the graph $\mathcal{G}(A)$ of an SPD matrix A that arises from a simple 5-point discretization of a PDE on a regular two-dimensional grid and colour its vertices using two colours so that no vertices of the same colour are incident to the same edge (see Figure 10.5). Because no red vertex is adjacent to any other red vertex, the red vertices are an independent set; similarly, the black vertices are an independent set. The red vertices can be processed in any order, provided they are all processed before any of the black vertices. This can make red–black orderings convenient for parallel implementations and is the main reason that they are often employed with stationary iterative methods.

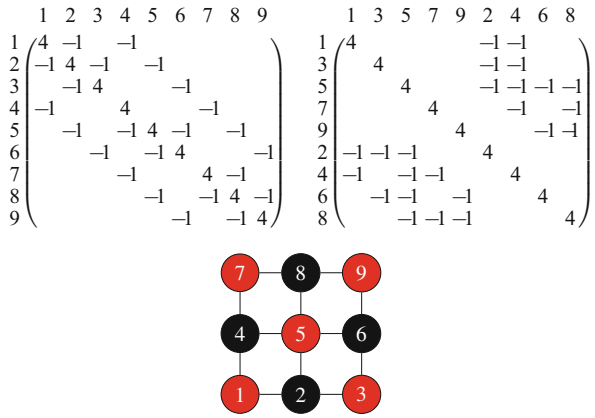

Figure 10.5 A model problem to illustrate a red–black ordering. The grid-based graph $\mathcal{G}(A)$ with coloured vertices is given together with the matrix A (left) and the symmetrically permuted matrix using the red–black ordering (right).

A bipartite graph is an undirected graph whose vertices can be partitioned into two disjoint sets such that each set is an independent set (Section 6.3.1). It follows that the red–black ordering exists if and only if $\mathcal{G}(A)$ is bipartite. The ordering is often generalized as follows. Start by finding a set of mutually non-adjacent vertices (that is, an independent set) and flag them as red vertices. After the elimination of the variables corresponding to the red vertices and employing a sparsification strategy, a Schur complement matrix is obtained. Proceed by finding a set of mutually non-adjacent vertices in this matrix, flag them as red vertices and continue recursively. This approach can lead to a significant decrease in the condition number of the preconditioned matrix. Another generalization for arbitrary graphs is to employ more colours (multicolouring). Again, the colouring can be exploited in parallel computations. For efficiency, load balancing of the coloured vertices needs to be considered. Because reordering the vertices can affect the convergence rate of an iterative solver, the potential gain in parallel performance at each iteration may be offset by a slower convergence rate.

10.9 Exploiting Block Structure

Blocking methods for complete factorizations can be adapted to incomplete factorizations. The aim is to speed up the computation of the factors and to obtain more effective preconditioners. In a block factorization, scalar operations of the form

$$\tilde{l}_{ik} = a_{ik}/\tilde{u}_{kk}$$

are replaced by matrix operations

$$\widetilde{L}_{ib,kb} = A_{ib,kb}\widetilde{U}_{kb,kb}^{-1},$$

and scalar multiplications of entries of the factors are replaced by matrix–matrix products. When dropping entries, instead of considering the absolute values, simple norms of the block entries (such as the one norm, max norm, or Frobenius norm) are used.

An incomplete factorization can start with the supernodal structure of the complete factors. If dropping is applied to individual columns, this structure is generally lost. To try and retain it, the dropping strategy can be modified either to drop the set of nonzeros of a row in the current supernode or to keep it. To obtain sufficiently sparse incomplete factors, it may be necessary to subdivide each supernode, allowing greater flexibility on how many rows are dropped. It is also possible to relax blocking operations in such a way that the supernodes are not exact but are allowed to incur some fill-in.

10.10 Notes and References

Sparsity structure was the main ingredient of the first algebraic preconditioners that were developed in the late 1950s. The nonzero structure represented the stencils resulting from the discretization of PDEs on structured grids. The earliest contribution is Buleev (1959), and this was later generalized to three-dimensional problems. An independent derivation and its interpretation as an incomplete factorization for a sparse matrix coming from a simple 5-point stencil is given in Varga (1960); other early work is by Baker & Oliphant (1960). For an overview of early contributions and the motivations behind incomplete factorizations, see Il′in (1992); we also refer to the survey of Chan & van der Vorst (1997).

Important breakthroughs in the use of preconditioning using incomplete factorizations for practical problems came in two key papers. The first by Meijerink & van der Vorst (1977) recognized the importance of preconditioning for the conjugate gradient method. In the second, Kershaw (1978) proposed locally replacing pivots by a small positive number to prevent breakdown of the factorization. This paved the way for incomplete factorizations in which dropping is based solely on the size of the computed entries and which were introduced even earlier by Tuff & Jennings (1973).

The Crout incomplete LU factorization outlined in Algorithm 10.1 was implemented in a successful code for symmetric problems by Lin & Moré (1999), building on earlier ideas of Jones & Plassmann (1995) and Eisenstat et al. (1982) (see also Li et al., 2003 for later contributions to this approach). Algorithm 10.2

with a sparsification strategy that uses both a drop tolerance and a limit on the number of entries in each column of the incomplete factors was published in Saad (1994a) as the dual threshold ILUT method. For general nonsymmetric matrices, ILUT has proved very popular and has been developed further (see, for example, MacLachlan et al., 2012). But because it is based on the row factorization, it ignores symmetry in A and, if A is symmetric, the computed sparsity patterns of L and U^T are normally different. In this case, a Crout incomplete factorization may be preferable. The hierarchy of sparsity structures based on the concept of levels is introduced in Watts-III (1981). The initial work has since been significantly improved, notably for parallel implementations by Hysom & Pothen (2002). The Euclid library is a scalable implementation of a parallel level-based ILU algorithm that is available as part of the *hypre* library of linear solvers (see Falgout et al., 2006, 2021). Scalable means that the incomplete factorization and triangular solve timings remain nearly constant when the problem size n is scaled in proportion to the number of processors. Another parallel level-based ILU preconditioner that uses an adaptive block implementation is proposed in Hénon et al. (2008).

The modified incomplete factorizations of Section 10.4 are described in Saad (2003b). A proof of Theorem 10.3 can be found in Bern et al. (2006), but it is also of interest to follow earlier work on asymptotic bounds for the condition number of matrices preconditioned by modified incomplete factorizations given in Dupont et al. (1968), Axelsson (1972), and Gustafsson (1978), while an elegant description is in Meurant (1999).

Incomplete factorizations with dynamic compensation originally introduced by Ajiz & Jennings (1984) have been routinely employed in practice. However, memory-limited approaches based on relaxing the strategy of Tismenetsky (1991) often lead to more efficient preconditioners; see Kaporin (1998) for a row-based construction that has recently been used by Konshin et al. (2017, 2019) to solve challenging practical problems. Scott & Tůma (2014b) present a Crout construction of a sophisticated memory-limited incomplete factorization and provide a robust implementation for SPD systems as the package `HSL_MI28` within the HSL mathematical software library (Scott & Tůma, 2014a); a variant for symmetric saddle point systems is also included in HSL.

Using fixed-point iterations for the parallel computation of incomplete factorizations is a relatively new idea that was proposed and analysed by Chow & Patel (2015). Interleaving a fixed-point iteration with a procedure that adjusts the sparsity pattern is proposed by Anzt et al. (2018). Other attempts to compute and use ILU preconditioners in parallel that build on the software package ILUPACK (Bollhöfer et al., 2012) are presented in Aliaga et al. (2016, 2019). A different approach to parallelize incomplete factorizations by relaxing supernodes is given by Gupta & George (2010).

Significant attention has been devoted to using orderings of A to try and improve the quality of incomplete factorization preconditioners. An early and often quoted comparison of reorderings for SPD problems is by Duff & Meurant (1989). For more general matrices, see Benzi et al. (1999), Oliker et al. (2002), or Osei-Kuffuor

et al. (2015). Saad (1996a) and Saad & Zhang (1999) generalize red–black orderings and consider blocks and/or more colours; also of interest are the papers of Saad & Suchomel (2002), Li et al. (2003), and Carpentieri et al. (2014)).