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

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

whose row sum is zero.

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

10.4 Threshold Strategies and ILUT

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

10.4.1 The ILUT Approach

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

ALGORITHM **10.6** *ILUT*

```
1.    For $i = 1, \ldots, n$ Do:
2.       $w := a_{i*}$
3.       For $k = 1, \ldots, i-1$ and when $w_k \neq 0$ Do:
4.          $w_k := w_k / a_{kk}$
```

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

Now consider the operations involved in the above algorithm. Line 7 is a sparse update operation. A common implementation of this is to use a full vector for w and a companion pointer which points to the positions of its nonzero elements. Similarly, lines 11 and 12 are sparse-vector copy operations. The vector w is filled with a few nonzero elements after the

completion of each outer loop i, and therefore it is necessary to zero out those elements at the end of the Gaussian elimination loop as is done in line 13. This is a sparse *set-to-zero* operation.

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

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

1. In line 5, an element w_k is dropped (i.e., replaced by zero) if it is less than the relative tolerance τ_i obtained by multiplying τ by the original norm of the i-th row (e.g., the 2-norm).

2. In line 10, a dropping rule of a different type is applied. First, drop again any element in the row with a magnitude that is below the relative tolerance τ_i. Then, keep only the p largest elements in the L part of the row and the p largest elements in the U part of the row in addition to the diagonal element, which is always kept.

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

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

10.4.2 Analysis

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

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

$$\begin{aligned}
& l_{ik} := u_{ik}^{k}/u_{kk} && (10.22)\\
& \text{if } |l_{ik}| \quad \text{small enough set} \quad l_{ik} = 0 &&\\
& \text{else:} &&\\
& u_{i,j}^{k+1} := u_{i,j}^{k} - l_{ik}u_{k,j} - r_{ij}^{k} \quad j = k+1, \ldots, n && (10.23)
\end{aligned}$$

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

$$u_{i,*} \equiv u_{i-1,*}^{i} \qquad (10.24)$$

and the following relation is satisfied:

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

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

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

- **Drop Strategy Modification.** For any $i < n$, let a_{i,j_i} be the element of largest modulus among the elements $a_{i,j},\ j = i+1, \ldots n$, in the original matrix.

Then elements generated in position (i, j_i) during the ILUT procedure are not subject to the dropping rule.

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

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

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

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

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

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

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

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

Theorem 10.8 *If the matrix A is a diagonally dominant $\hat{M}$ matrix, then the rows $u_{i,*}^k, k = 0, 1, 2, \ldots, i$ defined by (10.23) starting with $u_{i,*}^0 = 0$ and $u_{i,*}^1 = a_{i,*}$ satisfy the following relations for $k = 1, \ldots, l$*

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

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

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

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

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

in which $l_{ik} \le 0, u_{k,j} \le 0$. Either r_{ij}^k is zero which yields $u_{ij}^{k+1} \le u_{ij}^k \le 0$, or r_{ij}^k is nonzero which means that u_{ij}^{k+1} is being dropped, i.e., replaced by zero, and therefore again $u_{ij}^{k+1} \le 0$. This establishes (10.28). Note that by this argument $r_{ij}^k = 0$ except when the j-th element in the row is dropped, in which case $u_{ij}^{k+1} = 0$

and $r^k_{ij} = u^k_{ij} - l_{ik}u_{k,j} \leq 0$. Therefore, $r^k_{ij} \leq 0$, always. Moreover, when an element in position (i, j) is not dropped, then

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

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

$$u^{k+1}_{i,j_i} \leq u^k_{i,j_i} \tag{10.31}$$

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

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

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

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

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

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

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

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

10.4.3 Implementation Details

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

1. Generation of the linear combination of rows of A (Line 7 in Algorithm 10.6).

2. Selection of the p largest elements in L and U.

3. Need to access the elements of L in increasing order of columns (in line 3 of Algorithm 10.6).

For (1), the usual technique is to generate a full row and accumulate the linear combination of the previous rows in it. The row is zeroed again after the whole loop is finished using a sparse set-to-zero operation. A variation on this technique uses only a full integer array $jr(1:n)$, the values of which are zero except when there is a nonzero element. With this full row, a short real vector $w(1:maxw)$ must be maintained which contains the real values of the row, as well as a corresponding short integer array $jw(1:maxw)$ which points to the column position of the real values in the row. When a nonzero element resides in position j of the row, then $jr(j)$ is set to the address k in w, jw where the nonzero element is stored. Thus, $jw(k)$ points to $jr(j)$, and $jr(j)$ points to $jw(k)$ and $w(k)$. This is illustrated in Figure 10.12.

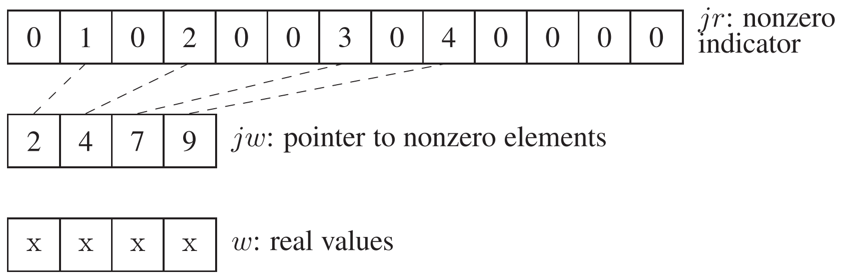

Figure 10.12: Illustration of data structure used for the working row in ILUT.

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

For (2), the natural technique is to employ a heap-sort strategy. The cost of this implementation would be $O(m + p \times \log_2 m)$, i.e., $O(m)$ for the heap construction and $O(\log_2 m)$ for each extraction. Another implementation is to use a modified quick-sort strategy based on the fact that sorting the array is not necessary.

Only the largest p elements must be extracted. This is a *quick-split* technique to distinguish it from the full quick-sort. The method consists of choosing an element, e.g., $x = w(1)$, in the array $w(1:m)$, then permuting the data so that $|w(k)| \leq |x|$ if $k \leq mid$ and $|w(k)| \geq |x|$ if $k \geq mid$, where mid is some split point. If $mid = p$, then exit. Otherwise, split *one of the left or right sub-arrays* recursively, depending on whether mid is smaller or larger than p. The cost of this strategy *on the average* is $O(m)$. The savings relative to the simpler bubble sort or insertion sort schemes are small for small values of p, but they become rather significant for large p and m.

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

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

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

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

If the total time to solve one linear system with A is considered, a typical curve of the total time required to solve a linear system when the *lfil* parameter varies would look like the plot shown in Figure 10.13. As *lfil* increases, a critical value is reached where the preprocessing time and the iteration time are equal. Beyond this critical point, the preprocessing time dominates the total time. If there are several linear systems to solve with the same matrix A, then it is advantageous to use a more accurate factorization, since the cost of the factorization will be amortized. Otherwise, a smaller value of *lfil* will result in a more efficient, albeit also less reliable, run.

10.4.4 The ILUTP Approach

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

1. The ILUT procedure encounters a zero pivot;
2. The ILUT procedure encounters an overflow or underflow condition, because of an exponential growth of the entries of the factors;

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

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

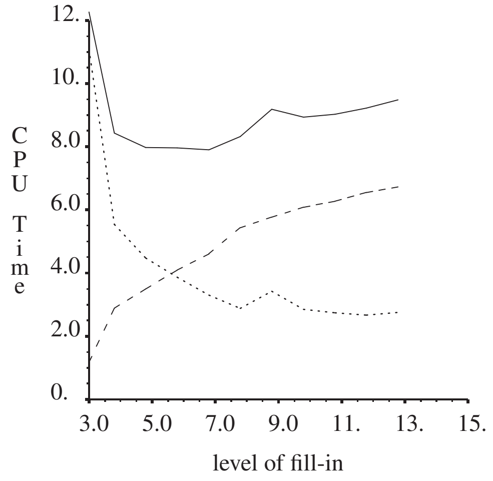

Figure 10.13: Typical CPU time as a function of lfil. Dashed line: ILUT. Dotted line: GMRES. Solid line: total.

3. The ILUT preconditioner terminates normally but the incomplete factorization preconditioner which is computed is *unstable*.

An unstable ILU factorization is one for which $M^{-1} = U^{-1}L^{-1}$ has a very large norm leading to poor convergence or divergence of the outer iteration. The case (1) can be overcome to a certain degree by assigning an arbitrary nonzero value to a zero diagonal element that is

encountered. Clearly, this is not a satisfactory remedy because of the loss in accuracy in the preconditioner. The ideal solution in this case is to use pivoting. However, a form of pivoting is desired which leads to an algorithm with similar cost and complexity to ILUT. Because of the data structure used in ILUT, row pivoting is not practical. Instead, column pivoting can be implemented rather easily.

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

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

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

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

Table 10.5: A test run of GMRES with ILUTP preconditioning.

Example 10.5. Table 10.5 shows the results of applying the GMRES algorithm with ILUTP$(1, 10^{-4})$ preconditioning to the five test problems described in Section 3.7. The *permtol* parameter is set to 1.0 in this case. See Example 6.1 for the meaning of the column headers in the table. The results are identical with those of ILUT$(1, 10^{-4})$ shown in Table 10.3, for the first four problems, but there is an improvement for the fifth problem. □

10.4.5 The ILUS Approach

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

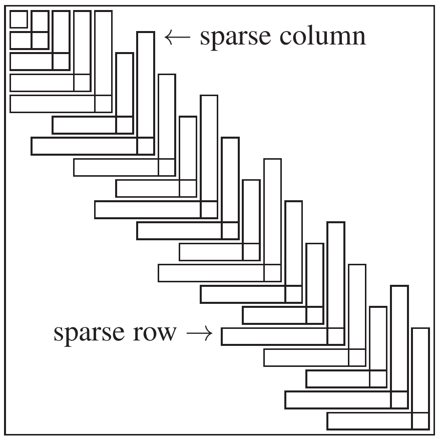

Figure 10.14: Illustration of the sparse skyline format.

In this format, the matrix A is decomposed as

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

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

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

Consider the sequence of matrices

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

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

$$A_k = L_k D_k U_k$$

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

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

in which

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

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

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

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

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

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

ALGORITHM **10.7** *ILUS(ϵ, p)*

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

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

Also note that a simple iterative procedure such as MR or GMRES(m) can be used to solve the triangular systems in sparse-sparse mode. Similar techniques will be seen in Section 10.5. Experience shows that these alternatives are not much better than the Neumann series approach [79].

10.4.6 The Crout ILU Approach

A notable disadvantage of the standard delayed-update IKJ factorization is that it requires access to the entries in the k-th row of L in sorted order of columns. This is further complicated by the fact that the working row (denoted by w in Algorithm 10.6), is dynamically modified by fill-in as the elimination proceeds. Searching for the leftmost entry in the k-th row of L is usually not a problem when the fill-in allowed is small. Otherwise, when an accurate factorization is sought, it can become a significant burden and may ultimately even dominate the cost of the factorization. Sparse direct solution methods that are based on the IKJ form of Gaussian elimination obviate this difficulty by a technique known as the Gilbert-Peierls method [146]. Because of dropping, this technique cannot, however, be used as is. Another possible option is to reduce the cost of the searches through the use of clever data structures and algorithms, such as binary search trees or heaps [90].

The Crout formulation provides the most elegant solution to the problem. In fact the Crout version of Gaussian elimination has other advantages which make it one of the most appealing ways of implementing incomplete LU factorizations.

The Crout form of Gaussian elimination consists of computing, at step k, the entries $a_{k+1:n,k}$ (in the unit lower triangular factor, L) and $a_{k,k:n}$ (in the upper triangular factor, U). This is done by post-poning the rank-one update in a way similar to the IKJ variant. In Figure 10.15 the parts of the factors being computed at the k-th step are shown in black and those being accessed are in the shaded areas. At the k-th step, all the updates of the previous steps are applied to the entries $a_{k+1:n,k}$ and $a_{k,k:n}$ and it is therefore convenient to store L by columns and U by rows.

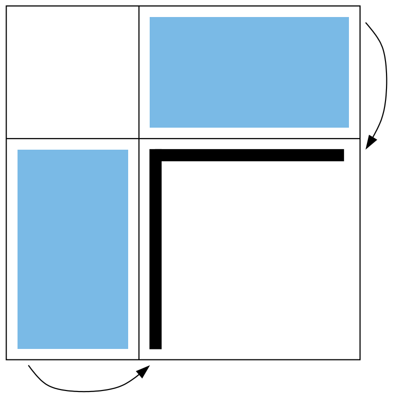

Figure 10.15: Computational pattern of the Crout algorithm.

ALGORITHM **10.8** *Crout LU Factorization*

```
1.  For $k = 1 : n$ Do :
2.      For $i = 1 : k-1$ and if $a_{ki} \neq 0$ Do :
3.          $a_{k,k:n} = a_{k,k:n} - a_{ki} * a_{i,k:n}$
4.      EndDo
5.      For $i = 1 : k-1$ and if $a_{ik} \neq 0$ Do :
6.          $a_{k+1:n.k} = a_{k+1:n,k} - a_{ik} * a_{k+1:n,i}$
7.      EndDo
8.      $a_{ik} = a_{ik}/a_{kk}$ for $i = k+1, ..., n$
9.  EndDo
```

The k-th step of the algorithm generates the k-th row of U and the k-th column of L. This step is schematically represented in Figure 10.16. The above Algorithm will now be adapted to the sparse case. Sparsity is taken into account and a dropping strategy is included, resulting in the following Crout version of ILU (termed ILUC).

ALGORITHM **10.9** *ILUC - Crout version of ILU*

```
1.  For $k = 1 : n$ Do :
2.      Initialize row $z$: $z_{1:k-1} = 0, \quad z_{k:n} = a_{k,k:n}$
3.      For $\{i \mid 1 \leq i \leq k-1 \text{ and } l_{ki} \neq 0\}$ Do :
4.          $z_{k:n} = z_{k:n} - l_{ki} * u_{i,k:n}$
5.      EndDo
6.      Initialize column $w$: $w_{1:k} = 0, \quad w_{k+1:n} = a_{k+1:n,k}$
7.      For $\{i \mid 1 \leq i \leq k-1 \text{ and } u_{ik} \neq 0\}$ Do :
8.          $w_{k+1:n} = w_{k+1:n} - u_{ik} * l_{k+1:n,i}$
9.      EndDo
10.     Apply a dropping rule to row $z$
11.     Apply a dropping rule to column $w$
```

```
12.            $u_{k,:} = z$
13.            $l_{:,k} = w/u_{kk}, \quad l_{kk} = 1$
14.        Enddo
```

Two potential sources of difficulty will require a careful and somewhat complex implementation. First, looking at Lines 4 and 8, only the section $(k : n)$ of the i-th row of U is required, and similarly, only the section $(k+1 : n)$ of the i-th column of L is needed. Second, Line 3 requires access to the k-th row of L which is stored by columns while Line 7 requires access to the k-th column of U which is accessed by rows.

The first issue can be easily handled by keeping pointers that indicate where the relevant part of each row of U (resp. column of L) starts. An array `Ufirst` is used to store for each row i of U the index of the first column that will used next. If k is the current step number, this means that `Ufirst`(i) holds the first column index $> k$ of all nonzero entries in the the i-th row of U. These pointers are easily updated after each elimination step, assuming that column indices (resp. column indices for L) are in increasing order.

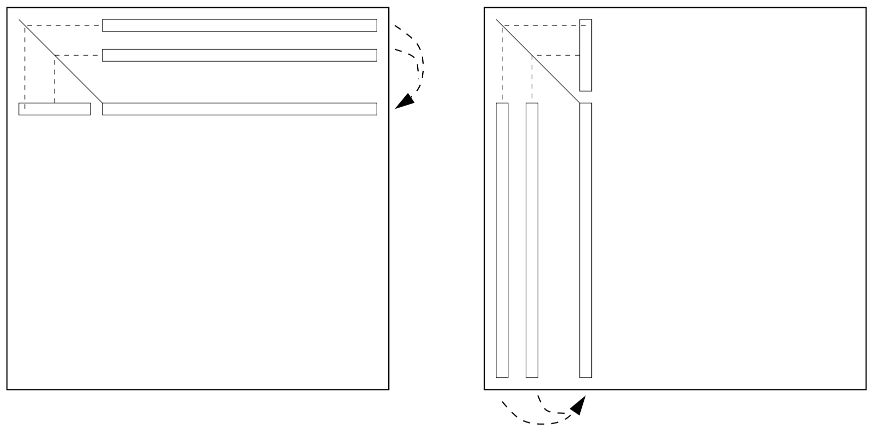

Figure 10.16: Computing the k-th row of U (left side) and the k-column of L (right side).

For the second issue, consider the situation with the U factor. The problem is that the k-th column of U is required for the update of L, but U is stored row-wise. An elegant solution to this problem is known since the pioneering days of sparse direct methods [115, 144]. Before discussing this idea, consider the simpler solution of including a linked list for each column of U. These linked lists would be easy to update because the rows of U are computed one at a time. Each time a new row is computed, the nonzero entries of this row are queued to the linked lists of their corresponding columns. However, this scheme would entail nonnegligible additional storage. A clever alternative is to exploit the array `Ufirst` mentioned above to form incomplete linked lists of each column. Every time k is incremented the `Ufirst` array is updated. When `Ufirst`(i) is updated to point to a new nonzero with column index j, then the row index i is added to the linked list for column i. What is interesting is that though the columns structures constructed in this manner are incomplete, they *become complete as soon as they are needed.* A similar technique is used for the rows of the L factor.

In addition to avoiding searches, the Crout version of ILU has another important advantage. It enables some new dropping strategies which may be viewed as more rigorous than the standard ones seen so far. The straightforward dropping rules used in ILUT can be easily adapted for ILUC. In addition, the data structure of ILUC allows options which are based on estimating the norms of the inverses of L and U.

For ILU preconditioners, the error made in the inverses of the factors is more important to control than the errors in the factors themselves. This is because when $A = LU$, and

$$\tilde{L}^{-1} = L^{-1} + X \qquad \tilde{U}^{-1} = U^{-1} + Y,$$

then the preconditioned matrix is given by

$$\tilde{L}^{-1}A\tilde{U}^{-1} = (L^{-1} + X)A(U^{-1} + Y) = I + AY + XA + XY.$$

If the errors X and Y in the inverses of L and U are small, then the preconditioned matrix will be close to the identity matrix. On the other hand, small errors in the factors themselves may yield arbitrarily large errors in the preconditioned matrix.

Let L_k denote the matrix composed of the first k rows of L and the last $n - k$ rows of the identity matrix. Consider a term l_{jk} with $j > k$ that is dropped at step k. Then, the resulting perturbed matrix $\tilde{L}_k$ differs from L_k by $l_{jk}e_je_k^T$. Noticing that $L_ke_j = e_j$ then,

$$\tilde{L}_k = L_k - l_{jk}e_je_k^T = L_k(I - l_{jk}e_je_k^T)$$

from which this relation between the inverses follows:

$$\tilde{L}_k^{-1} = (I - l_{jk}e_je_k^T)^{-1}L_k^{-1} = L_k^{-1} + l_{jk}e_je_k^TL_k^{-1}.$$

Therefore, the inverse of L_k will be perturbed by l_{jk} times the k-th row of L_k^{-1}. This perturbation will affect the j-th row of L_k^{-1}. Hence, using the infinity norm for example, it is important to limit the norm of this perturbing row which is $\|l_{jk}e_je_k^TL_k^{-1}\|_\infty$. It follows that it is a good strategy to drop a term in L when

$$|l_{jk}|\ \|e_k^TL_k^{-1}\|_\infty < \epsilon.$$

A similar criterion can be used for the upper triangular factor U.

This strategy is not complete because the matrix L^{-1} is not available. However, standard techniques used for estimating condition numbers [149] can be adapted for estimating the norm of the k-th row of L^{-1} (resp. k-th column of U^{-1}). The idea is to construct a vector b one component at a time, by following a greedy strategy to make $L^{-1}b$ large at each step. This is possible because the first $k - 1$ columns of L are available at the k-th step. The simplest method constructs a vector b of components $\beta_k = \pm 1$ at each step k, in such a way as to maximize the norm of the

k-th component of $L^{-1}b$. Since the first $k-1$ columns of L are available at step k, the k-th component of the solution x is given by

$$\xi_k = \beta_k - e_k^T L_{k-1} x_{k-1} \, .$$

This makes the choice clear: if ξ_k is to be large in modulus, then the sign of β_k should be opposite that of $e_k^T L_{k-1} x_{k-1}$. If b is the current right-hand side at step k, then $\|e_k^T L^{-1}\|_\infty$ can be estimated by the k-th component of the solution x of the system $Lx = b$:

$$\|e_k^{\mathrm{T}} L^{-1}\|_\infty \approx \frac{|e_k^{\mathrm{T}} L^{-1} b|}{\|b\|_\infty} \, .$$

Details, along with other strategies for dynamically building b, may be found in [202].

10.5 Approximate Inverse Preconditioners

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

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

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

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

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

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