Chapter 6

LU factorization

Of the three factorization methods (Cholesky, QR, and LU) presented here, LU factorization is the oldest. As a factorization method, it factors a matrix A into the product LU, where L is lower triangular and U is upper triangular. The historical method for dense matrices is a right-looking one (Gaussian elimination); both it and a left-looking method are presented here. The latter is used in CSparse, since it leads to a much simpler implementation for the sparse case.

6.1 Upper bound on fill-in

Theorem 4.1, which describes the filled graph of the Cholesky factor, also holds for the directed graph of $L + U$ if no pivoting occurs and A is assumed to be square. However, a more useful analysis accounts for partial pivoting with row interchanges, based on an important relationship between the LU and QR factorizations of a matrix. Consider both $LU = PA$ and $QR = A$, where P is determined by partial pivoting.

Theorem 6.1 (George and Ng [97], Gilbert [101], and Gilbert and Ng [106]). *If the matrix A is strong Hall, R is an upper bound on the nonzero pattern of U. More precisely, u_{ij} can be nonzero if and only if $r_{ij} \neq 0$.*

This upper bound is tight in a one-at-a-time sense; for any $r_{ij} \neq 0$, there exists an assignment of numerical values to entries in the pattern of A that makes $u_{ij} \neq 0$. The outline of the proof can be seen by comparing Gaussian elimination with Householder reflections. Both eliminate entries below the diagonal. For a Householder reflection, the nonzero pattern of all rows affected by the transformation takes on a nonzero pattern that is the union of all of these rows (Theorem 5.2). With partial pivoting and row interchanges, these rows are candidate pivot rows (all the rows i for which $a_{ik}^{[k-1]} \neq 0$ or $i \in A_{*k}^{[k-1]}$). Only one of them is selected as the pivot row. Every other candidate pivot row is modified by adding to it a scaled copy of the pivot row. An upper bound on the pivot row pattern is the union of

all candidate pivot rows. This proof also establishes a bound on L, namely, the nonzero pattern of V.

Theorem 6.2 (Gilbert [101] and Gilbert and Ng [106]). *If the matrix A is strong Hall, and assuming $a_{kk}^{[k-1]} \neq 0$ for all k, the Householder matrix V is an upper bound on the nonzero pattern of L obtained with partial pivoting. More precisely, l_{ij} can be nonzero if and only if $v_{ij} \neq 0$.*

With this relationship, a symbolic QR ordering and analysis becomes one possible method for ordering a matrix for LU factorization. It is also possible to statically preallocate space for L and U. The bound can be loose, however. In particular, if the matrix is diagonally dominant, then no pivoting is needed to maintain numerical accuracy.[10] If it also has a symmetric nonzero pattern (or if all entries in the pattern of $A + A^T$ are considered to be "nonzero"), then the nonzero patterns of L and U are identical to the patterns of the Cholesky factors L and L^T, respectively, of a symmetric positive definite matrix with the same nonzero pattern as $A + A^T$. In this case, a symmetric fill-reducing ordering of $A + A^T$ is appropriate. Alternatively, the permutation matrix Q can be selected to reduce the worst case fill-in for $PAQ = LU$ for any P, and then the permutation P can be selected solely on the basis of partial pivoting with no regard for sparsity. Thus, the `cs_sqr` function provides four basic strategies for finding a fill-reducing permutation Q.

- `order=0`: No column permutation is used; $LU = PA$. This is useful if A is already known to have a good column ordering.
- `order=1`: The column permutation Q is found from a fill-reducing ordering of $A + A^T$. During factorization, an attempt is made to ensure $P = Q^T$ (preference for selecting the pivot is given to the diagonal entry of Q^TAQ). This strategy is well suited to the many unsymmetric matrices arising in practice that have a roughly symmetric nonzero structure and reasonably large entries on the diagonal.
- `order=2`: Q is obtained from a fill-reducing ordering of S^TS, where $S = A$ except that all entries in "dense" rows of S are removed. A row in A with more entries than a heuristic threshold is considered dense. With partial pivoting, the (optimistic) hope is that these rows will not be selected as pivot rows until very late in the factorization.
- `order=3`: Q is obtained from the ordering of A^TA with no dense rows dropped. In this case, the ordering tries to reduce the QR upper bounds on L and U given in Theorems 6.1 and 6.2.

[10] This is called *static pivoting*; it can be used even if the matrix is not quite diagonally dominant, if iterative refinement is used after the solution has been found.

If the `qr` parameter of `cs_sqr` is true, the QR upper bound is found for the permuted matrix AQ (here, Q is the column permutation, not the orthogonal factor Q). In this case, LU factorization can proceed using a statically allocated memory space. This bound can be quite high, however (a comparison between the upper bound and the actual $|L|$ and $|U|$ is left as an exercise). It is sometimes better just to make a guess at the final $|L|$ and $|U|$ or to guess that no partial pivoting will be needed and to use a symbolic Cholesky analysis to determine a guess for $|L|$ and $|U|$ (this is left as an exercise). Sometimes a good guess is available from the LU factorization of a similar matrix in the same application. If `qr` is false, `cs_sqr` makes an optimistic guess that $|L| = |U| = 4|A| + n$. This guess is suitable for some matrices but too low for others. After calling `cs_sqr`, the guess `S->lnz` and `S->unz` can be easily modified as desired. The only penalty for making a wrong guess is that the memory space for $|L|$ or $|U|$ must be reallocated if the guess is too low, or memory may run out if the guess is too high.

6.2 Left-looking LU

The *left-looking* LU factorization algorithm computes L and U one column at a time. At the kth step, it accesses columns 1 to $k-1$ of L and column k of A. If partial pivoting is ignored, it can be derived from the following 3-by-3 block matrix expression, which is very similar to (4.6) for the left-looking Cholesky factorization algorithm. The matrix L is assumed to have a unit diagonal.

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{32} & A_{33} \end{bmatrix}. \tag{6.1}$$

The middle row and column of each matrix is the kth row and column of L, U, and A, respectively. If the first $k-1$ columns of L and U are known, three equations can be used to derive the kth columns of L and U: $L_{11}u_{12} = a_{12}$ is a triangular system that can be solved for u_{12} (the kth column of U), $l_{21}u_{12} + u_{22} = a_{22}$ can be solved for the pivot entry u_{22}, and $L_{31}u_{12} + l_{32}u_{22} = a_{32}$ can then be solved for l_{32} (the kth column of L). However, these three equations can be rearranged so that nearly all of them are given by the solution to a single triangular system:

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix}. \tag{6.2}$$

The solution to this system gives $u_{12} = x_1$, $u_{22} = x_2$, and $l_{32} = x_3/u_{22}$. The algorithm is expressed in the MATLAB function `lu_left`, except that partial pivoting with row interchanges has been added. It returns `L`, `U`, and `P` so that `L*U = P*A`. It does not exploit sparsity.

```
function [L,U,P] = lu_left (A)
n = size (A,1) ;
P = eye (n) ;
L = zeros (n) ;
U = zeros (n) ;
for k = 1:n
    x = [ L(:,1:k-1) [ zeros(k-1,n-k+1) ; eye(n-k+1) ]] \ (P * A (:,k)) ;
    U (1:k-1,k) = x (1:k-1) ;           % the column of U
    [a i] = max (abs (x (k:n))) ;       % find the pivot row i
    i = i + k - 1 ;
    L ([i k],:) = L ([k i], :) ;        % swap rows i and k of L, P, and x
    P ([i k],:) = P ([k i], :) ;
    x ([i k]) = x ([k i]) ;
    U (k,k) = x (k) ;
    L (k,k) = 1 ;
    L (k+1:n,k) = x (k+1:n) / x (k) ;   % divide the pivot column by U(k,k)
end
```

The derivation of how partial pivoting works in a left-looking algorithm is not included. A proof of the correctness of applying the row permutations to the rows of L is given in the next section in a right-looking context.

A direct implementation of a sparse version of `lu_left` would be difficult, since it swaps rows i and k of L at the kth step. Access to the rows of L is not trivial. Rather than swapping rows of L, the `cs_lu` function leaves the row indices in L in their original order. That is, a row index i in L corresponds to the same row in the original unpermuted matrix A. The sparse triangular solve `cs_spsolve` is used to solve (6.2) at each step. It uses the inverse row permutation, `pinv`, to perform a permuted triangular solve (the columns of L are in their final ordering, but the rows of L are unpermuted). Then, when the factorization is complete, all row indices of L can be updated to reflect the final row permutation.

Given a fill-reducing column ordering q, `cs_lu` computes L, U, and `pinv` so that `L*U = A(p,q)` (where p is the inverse of `pinv`). The identity matrix in (6.2) is implicitly maintained. For a nonpivotal row index i, `jnew=pinv[i]=-1`, and this column `jnew` is skipped when performing the sparse triangular solve.

```
csn *cs_lu (const cs *A, const css *S, double tol)
{
    cs *L, *U ;
    csn *N ;
    double pivot, *Lx, *Ux, *x,  a, t ;
    int *Lp, *Li, *Up, *Ui, *pinv, *xi, *q, n, ipiv, k, top, p, i, col, lnz,unz;
    if (!CS_CSC (A) || !S) return (NULL) ;          /* check inputs */
    n = A->n ;
    q = S->q ; lnz = S->lnz ; unz = S->unz ;
    x = cs_malloc (n, sizeof (double)) ;            /* get double workspace */
    xi = cs_malloc (2*n, sizeof (int)) ;            /* get int workspace */
    N = cs_calloc (1, sizeof (csn)) ;               /* allocate result */
    if (!x || !xi || !N) return (cs_ndone (N, NULL, xi, x, 0)) ;
    N->L = L = cs_spalloc (n, n, lnz, 1, 0) ;       /* allocate result L */
    N->U = U = cs_spalloc (n, n, unz, 1, 0) ;       /* allocate result U */
    N->pinv = pinv = cs_malloc (n, sizeof (int)) ;  /* allocate result pinv */
    if (!L || !U || !pinv) return (cs_ndone (N, NULL, xi, x, 0)) ;
    Lp = L->p ; Up = U->p ;
    for (i = 0 ; i < n ; i++) x [i] = 0 ;           /* clear workspace */
```



```
    for (i = 0 ; i < n ; i++) pinv [i] = -1 ;       /* no rows pivotal yet */
    for (k = 0 ; k <= n ; k++) Lp [k] = 0 ;         /* no cols of L yet */
    lnz = unz = 0 ;
    for (k = 0 ; k < n ; k++)       /* compute L(:,k) and U(:,k) */
    {
        /* --- Triangular solve ------------------------------------------- */
        Lp [k] = lnz ;              /* L(:,k) starts here */
        Up [k] = unz ;              /* U(:,k) starts here */
        if ((lnz + n > L->nzmax && !cs_sprealloc (L, 2*L->nzmax + n)) ||
            (unz + n > U->nzmax && !cs_sprealloc (U, 2*U->nzmax + n)))
        {
            return (cs_ndone (N, NULL, xi, x, 0)) ;
        }
        Li = L->i ; Lx = L->x ; Ui = U->i ; Ux = U->x ;
        col = q ? (q [k]) : k ;
        top = cs_spsolve (L, A, col, xi, x, pinv, 1) ;  /* x = L\A(:,col) */
        /* --- Find pivot ------------------------------------------------- */
        ipiv = -1 ;
        a = -1 ;
        for (p = top ; p < n ; p++)
        {
            i = xi [p] ;            /* x(i) is nonzero */
            if (pinv [i] < 0)       /* row i is not yet pivotal */
            {
                if ((t = fabs (x [i])) > a)
                {
                    a = t ;         /* largest pivot candidate so far */
                    ipiv = i ;
                }
            }
            else                    /* x(i) is the entry U(pinv[i],k) */
            {
                Ui [unz] = pinv [i] ;
                Ux [unz++] = x [i] ;
            }
        }
        if (ipiv == -1 || a <= 0) return (cs_ndone (N, NULL, xi, x, 0)) ;
        if (pinv [col] < 0 && fabs (x [col]) >= a*tol) ipiv = col ;
        /* --- Divide by pivot -------------------------------------------- */
        pivot = x [ipiv] ;          /* the chosen pivot */
        Ui [unz] = k ;              /* last entry in U(:,k) is U(k,k) */
        Ux [unz++] = pivot ;
        pinv [ipiv] = k ;           /* ipiv is the kth pivot row */
        Li [lnz] = ipiv ;           /* first entry in L(:,k) is L(k,k) = 1 */
        Lx [lnz++] = 1 ;
        for (p = top ; p < n ; p++) /* L(k+1:n,k) = x / pivot */
        {
            i = xi [p] ;
            if (pinv [i] < 0)       /* x(i) is an entry in L(:,k) */
            {
                Li [lnz] = i ;      /* save unpermuted row in L */
                Lx [lnz++] = x [i] / pivot ;    /* scale pivot column */
            }
            x [i] = 0 ;             /* x [0:n-1] = 0 for next k */
        }
    }
    /* --- Finalize L and U ------------------------------------------------ */
```

```
    Lp [n] = lnz ;
    Up [n] = unz ;
    Li = L->i ;                         /* fix row indices of L for final pinv */
    for (p = 0 ; p < lnz ; p++) Li [p] = pinv [Li [p]] ;
    cs_sprealloc (L, 0) ;               /* remove extra space from L and U */
    cs_sprealloc (U, 0) ;
    return (cs_ndone (N, NULL, xi, x, 1)) ;     /* success */
}
```

The first part of the `cs_lu` function allocates workspace and obtains the information from the symbolic ordering and analysis. The number of nonzeros in `L` and `U` is not known; `S->lnz` and `S->unz` are either upper bounds computed from a symbolic QR factorization or simply a guess.

Triangular solve: The kth iteration of the `for k` loop first records the start of the kth columns of `L` and `U` and then reallocates these two matrices if space might not be sufficient. Next, the triangular system (6.2) is solved for `x`. No post-permutation is required for `U`, since `pinv[i]` is well defined.

Find pivot: The largest nonpivotal entry in the pivot column is found. An entry `x[i]` corresponding to a row `i` that is already pivotal is copied directly into `U`. If no nonpivotal row index `i` is found (`ipiv` is `-1`), the matrix is structurally rank deficient. If the largest entry in nonpivotal rows is numerically zero (`a` is zero), the matrix is numerically rank deficient. The diagonal entry (`x[col]`, where `col` is the kth column of AQ and Q is the fill-reducing column ordering), is selected if it is large enough compared with the partial pivoting choice (`x[ipiv]`).

Divide by pivot: The pivot entry is saved as `U(k,k)`, the last entry in `U(:,k)`, as required by `cs_usolve`. A unit diagonal entry is stored as the first entry in `L(:,k)`, as required by `cs_lsolve`. Note that `ipiv` corresponds to a row index of `A`, not PA.

Finalize `L` and `U`: The last column pointers for `L` and `U` are recorded, the row indices of `L` are fixed to refer to their permuted ordering, and any extra space is removed from `L` and `U`.

The algorithm takes $O(n+|A|+f)$ time, where f is the number of floating-point operations performed. This is essentially $O(f)$, except when `A` is diagonal (for example). MATLAB uses the algorithm above for the `[L,U,P]=lu(A)` syntax (GPLU). It uses a right-looking multifrontal method (UMFPACK) for `[L,U,P,Q]=lu(A)` and `x=A\b` when `A` is sparse, square, and not symmetric positive definite.

6.3 Right-looking and multifrontal LU

Gaussian elimination is a *right-looking* variant of LU factorization. The method is not used in CSparse, but it is presented here for two reasons: (1) it leads to a simpler constructive proof of the existence of the $LU = PA$ factorization, and (2) it forms the basis of UMFPACK, the multifrontal method for sparse LU factorization used in MATLAB.

At each step, an outer product of the pivot column and the pivot row is subtracted from the lower right submatrix of A. The derivation of the method (ignoring pivoting) starts with an equation very similar to (4.7) for the right-looking

Cholesky factorization,

$$\begin{bmatrix} l_{11} & \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ & U_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{bmatrix}, \tag{6.3}$$

where $l_{11} = 1$ is a scalar, and all three matrices are square and partitioned identically. Other choices for l_{11} are possible; this choice leads to a unit lower triangular L and the four equations

$$u_{11} = a_{11}, \tag{6.4}$$

$$u_{12} = a_{12}, \tag{6.5}$$

$$l_{21}u_{11} = a_{21}, \tag{6.6}$$

$$l_{21}u_{12} + L_{22}U_{22} = A_{22}. \tag{6.7}$$

Solving each equation in turn leads to the recursive `lu_rightr`, written in MATLAB below. This function is meant as a working description of the algorithm, not an efficient implementation.

```
function [L,U] = lu_rightr (A)
n = size (A,1)
if (n == 1)
    L = 1 ;
    U = A ;
else
    u11 = A (1,1) ;                                          % (6.4)
    u12 = A (1,2:n) ;                                        % (6.5)
    l21 = A (2:n,1) / u11 ;                                  % (6.6)
    [L22,U22] = lu_rightr (A (2:n,2:n) - l21*u12) ;          % (6.7)
    L = [ 1 zeros(1,n-1) ; l21 L22 ] ;
    U = [ u11 u12 ; zeros(n-1,1) U22 ] ;
end
```

The `lu_rightr` function uses *tail recursion*, where the recursive call is the very last step (the last two lines of the loop do not do any work; they just define the contents of `L` and `U` computed via (6.4) through (6.7)). Tail recursion can easily be converted into an iterative algorithm, as shown by the `lu_right` function. This is how a right-looking LU factorization algorithm would normally be written, except that in the dense case, `A` is normally overwritten with `L` and `U`.

```
function [L,U] = lu_right (A)
n = size (A,1)
L = eye (n) ;
U = zeros (n) ;
for k = 1:n
    U (k,k:n) = A (k,k:n) ;                                  % (6.4) and (6.5)
    L (k+1:n,k) = A (k+1:n,k) / U (k,k) ;                    % (6.6)
    A (k+1:n,k+1:n) = A (k+1:n,k+1:n) - L (k+1:n,k) * U (k,k+1:n) ;   % (6.7)
end
```

The derivation above is an inductive proof of the existence of the $LU = A$ factorization with base case (6.4) and the inductive hypothesis from (6.7),

$$L_{22}U_{22} = A_{22} - l_{21}u_{12}. \tag{6.8}$$

The factorization $LU = A$ exists only if each diagonal entry u_{kk} is nonzero.

Partial pivoting leads to a more stable variant, $LU = PA$, where P is a permutation matrix. With partial pivoting, the rows of `A` are interchanged so that $|u_{kk}|$ is maximized at each step. This swap can be determined for the first column of A, and the recursion will construct the remaining permutation of A. Let $P_1 \in \mathbb{R}^{n \times n}$ be the permutation matrix that interchanges two rows of A such that

$$P_1 A = \overline{A} = \begin{bmatrix} \overline{a}_{11} & \overline{a}_{12} \\ \overline{a}_{21} & \overline{A}_{22} \end{bmatrix}$$

and $|\overline{a}_{11}| \geq \max |\overline{a}_{21}|$. If (6.3) and its equivalent form in (6.4) through (6.7) are used directly on $\overline{A}$, the inductive hypothesis (6.8) cannot be used. If $LU = PA$ is the statement being proven, the inductive hypothesis must be applied to a matrix of smaller dimension but with the same form; (6.8) does not have a permutation matrix. The inductive hypothesis

$$L_{22} U_{22} = P_2 (\overline{A}_{22} - l_{21} u_{12}) \tag{6.9}$$

must be used instead, where P_2 is a permutation matrix. To make use of this, the expression (6.9) can be incorporated into a 2-by-2 block matrix expression by applying (6.4) through (6.7) to $\overline{A}$ and multiplying both sides of (6.6) by P_2, resulting in

$$u_{11} = \overline{a}_{11}, \tag{6.10}$$

$$u_{12} = \overline{a}_{12}, \tag{6.11}$$

$$P_2 l_{21} u_{11} = P_2 \overline{a}_{21}, \tag{6.12}$$

$$P_2 l_{21} u_{12} + L_{22} U_{22} = P_2 \overline{A}_{22}. \tag{6.13}$$

These four equations can be written as the 2-by-2 matrix expression

$$\begin{aligned} \begin{bmatrix} 1 & \\ P_2 l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ & U_{22} \end{bmatrix} &= \begin{bmatrix} \overline{a}_{11} & \overline{a}_{12} \\ P_2 \overline{a}_{21} & P_2 \overline{A}_{22} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix} \begin{bmatrix} \overline{a}_{11} & \overline{a}_{12} \\ \overline{a}_{21} & \overline{A}_{22} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix} P_1 A. \end{aligned} \tag{6.14}$$

Equation (6.14) is in the desired form $LU = PA$, where

$$L = \begin{bmatrix} 1 & \\ P_2 l_{21} & L_{22} \end{bmatrix}, \; U = \begin{bmatrix} u_{11} & u_{12} \\ & U_{22} \end{bmatrix}, \text{ and } P = \begin{bmatrix} 1 & 0 \\ 0 & P_2 \end{bmatrix} P_1.$$

This inductive derivation is demonstrated by the `lu_rightpr` function below. It is not a tail-recursive procedure, since P_2 must be applied to l_{21} after the recursive call completes, and P must be constructed as well. These permutations can be postponed and applied when the factorization is complete; this is what the `cs_lu` function does. For a dense matrix factorization, access to the rows of `L` is much simpler, and the permutations can be applied immediately, as done by `lu_left`. Either method leads to the same $LU = PA$ factorization. After replacing the recursion in `lu_rightpr` with its nonrecursive implementation and allowing `A` to be overwritten with its LU factorization, the conventional outer-product form of Gaussian elimination is obtained, as demonstrated by the `lu_rightp` function, shown below.


```
function [L,U,P] = lu_rightpr (A)
n = size (A,1)
if (n == 1)
    P = 1 ;
    L = 1 ;
    U = A ;
else
    [x,i] = max (abs (A (1:n,1))) ;                          % partial pivoting
    P1 = eye (n) ;
    P1 ([1 i],:) = P1 ([i 1], :) ;
    A = P1*A ;
    u11 = A (1,1) ;                                          % (6.10)
    u12 = A (1,2:n) ;                                        % (6.11)
    l21 = A (2:n,1) / u11 ;                                  % (6.12)
    [L22,U22,P2] = lu_rightpr (A (2:n,2:n) - l21*u12) ;      % (6.9) or (6.13)
    o = zeros(1,n-1) ;
    L = [ 1 o ; P2*l21 L22 ] ;                               % (6.14)
    U = [ u11 u12 ; o' U22 ] ;
    P = [ 1 o ; o' P2] * P1 ;
end
```

```
function [L,U,P] = lu_rightp (A)
n = size (A,1)
P = eye (n) ;
for k = 1:n
    [x,i] = max (abs (A (k:n,k))) ;                          % partial pivoting
    i = i+k-1 ;
    P ([k i],:) = P ([i k], :) ;
    A ([k i],:) = A ([i k], :) ;                             % (6.10), (6.11)
    A (k+1:n,k) = A (k+1:n,k) / A (k,k) ;                                % (6.12)
    A (k+1:n,k+1:n) = A (k+1:n,k+1:n) - A (k+1:n,k) * A (k,k+1:n) ;      % (6.9)
end
L = tril (A,-1) + eye (n) ;
U = triu (A) ;
```

A right-looking sparse LU factorization is significantly more complicated than the left-looking algorithm. It forms the basis of the multifrontal method for sparse LU factorization. The simpler case where the nonzero pattern of A is symmetric is considered first. Consider an unsymmetric matrix with the same symmetric nonzero pattern as the matrix shown in Figures 4.2 and 6.1 with the L and U factors shown as a single matrix. Suppose no numerical pivoting occurs. Each node in the elimination tree corresponds to one *frontal matrix*, which holds one rank-1 outer product. The frontal matrix for node k is an $|\mathcal{L}_k|$-by-$|\mathcal{L}_k|$ dense matrix. If the parent p and its single child c have the same nonzero pattern ($\mathcal{L}_p = \mathcal{L}_c \setminus \{c\}$), they can be combined (*amalgamated*) into a larger frontal matrix that represents both of them.

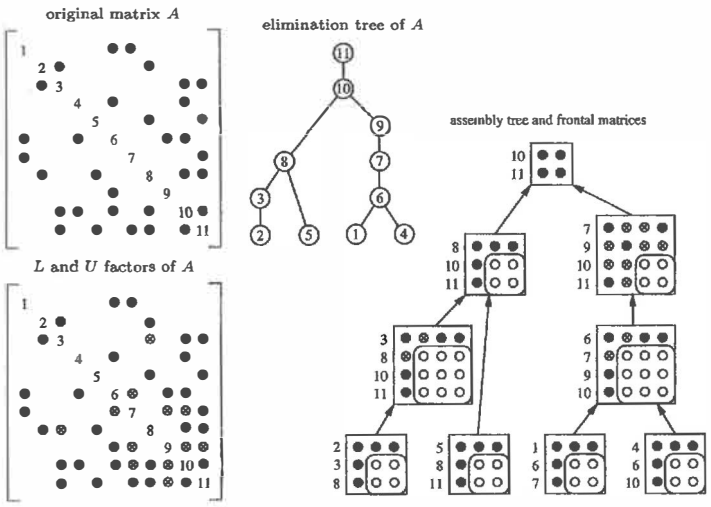

Figure 6.1. *Multifrontal example*

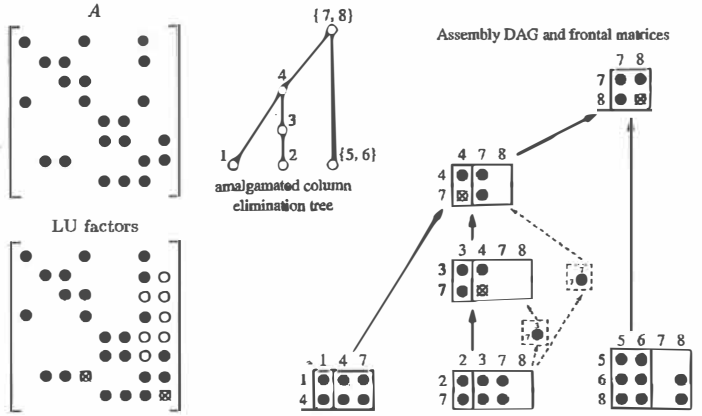

Figure 6.2. *Unsymmetric-pattern multifrontal example*

The frontal matrices are related to one another via the *assembly tree*, which is a coarser version of the elimination tree (some nodes having been merged together via amalgamation). To factorize a frontal matrix, the original entries of A are added, along with a summation of the *contribution blocks* of its children (called the *assembly*). One or more steps of dense LU factorization are performed within the frontal matrix, leaving behind its contribution block (the Schur complement of its pivot rows and columns). A high level of performance can be obtained using dense matrix kernels (the BLAS). The contribution block is placed on a stack, and deleted when it is assembled into its parent.

An example is shown in Figure 6.1. Black circles represent the original entries of A. Circled x's represent fill-in entries. White circles represent entries in the contribution block of each frontal matrix. The arrows between the frontal matrices represent both the data flow and the parent/child relationship of the assembly tree.

A symbolic analysis phase determines the elimination tree and the amalgamated assembly tree. During numerical factorization, numerical pivoting may be required. In this case it may be possible to pivot within the fully assembled rows and columns of the frontal matrix. For example, consider the frontal matrix holding diagonal elements a_{77} and a_{99} in Figure 6.1. If a_{77} is numerically unacceptable, it may be possible to select a_{79} and a_{97} as the next two pivot entries instead. If this is not possible, the contribution block of frontal matrix 7 will be larger than expected. This larger frontal matrix is assembled into its parent, causing the parent frontal matrix to be larger than expected. Within the parent, all pivots originally assigned to the parent and all failed pivots from the children (or any descendants) comprise the set of pivot candidates. If all of these are numerically acceptable, the parent contribution block is the same size as expected by the symbolic analysis.

If the nonzero pattern of A is unsymmetric, the frontal matrices become rectangular. They are related either by the column elimination tree (the elimination tree of A^TA) or by a directed acyclic graph. An example is shown in Figure 6.2.

This is the same matrix used for the QR factorization example in Figure 5.1. Using a column elimination tree, arbitrary partial pivoting can be accommodated without any change to the tree. The size of each frontal matrix is bounded by the size of the Householder update for the QR factorization of A (the kth frontal matrix is at most $|\mathcal{V}_k|$-by-$|\mathcal{R}_{k*}|$ in size), regardless of any partial pivoting. In the LU factors in Figure 6.2, original entries of A are shown as black circles. Fill-in entries when no partial pivoting occurs are shown as circled x's. White circles represent entries that could become fill-in because of partial pivoting. In this small example, they all happen to be in U, but in general they can appear in both L and U. Amalgamation can be done, just as in the symmetric-pattern case; in Figure 6.2, nodes 5 and 6, and nodes 7 and 8, have been merged together. The upper bound on the size of each frontal matrix is large enough to hold all candidate pivot rows, but this space does not normally need to be allocated.

In Figure 6.2, the assembly tree has been expanded to illustrate each frontal matrix. The tree represents the relationship between the frontal matrices but not the data flow. The assembly of contribution blocks can occur not just between parent and child but between ancestor and descendant. For example, the contribution to a_{77} made by frontal matrix 2 could be included into its parent 3, but this would

require one additional column to be added to frontal matrix 3. The upper bound of the size of this frontal matrix is 2-by-4, but only a 2-by-2 frontal matrix needs to be allocated if no partial pivoting occurs. Instead of augmenting frontal matrix 3 to include the a_{77} entry, the entry is assembled into the ancestor frontal matrix 4. The data flow between frontal matrices is thus represented by a directed acyclic graph.

One advantage of the right-looking method over left-looking sparse LU factorization is that it can select a sparse pivot row. The left-looking method does not keep track of the nonzero pattern of the $A^{[k]}$ submatrix, and thus cannot determine the number of nonzeros in its pivot rows. The disadvantage of the right-looking method is that it is significantly more difficult to implement.

MATLAB uses the unsymmetric-pattern multifrontal method (UMFPACK) in `x=A\b` when `A` is sparse and either unsymmetric or symmetric but not positive definite. It is also used in `[L,U,P,Q]=lu(A)`. For the `[L,U,P]=lu(A)` syntax when `A` is sparse, MATLAB uses GPLU, a left-looking sparse LU factorization much like `cs_lu`.

6.4 Further reading

Rose and Tarjan [174] and Rose, Tarjan, and Lueker [175] describe the filled graph of $L+U$ with no pivoting. MA28 by Duff and Reid [61] is an early right-looking sparse LU factorization method. In [97] George and Ng show that the nonzero pattern for LU factorization is bounded by the Cholesky factorization of A^TA. The row-merge model of symbolic LU factorization is the topic of a subsequent paper [98], which gives a tighter bound. The left-looking algorithm (GPLU) used in `cs_lu` is due to Gilbert and Peierls [109]. The book by Duff, Erisman, and Reid [53] delves into great detail on sparse LU factorization. Duff and Reid [62, 63] present the multifrontal method for unsymmetric matrices with symmetric pattern and symmetric indefinite matrices. The methods predate the unsymmetric-pattern multifrontal method of Davis [27, 28], Davis and Duff [31, 32], and GPLU. Liu [152] summarizes the multifrontal method, including LU factorization. Gilbert and Liu [103] introduce elimination DAGs for LU factorization. Hadfield [122] discusses the use of the elimination DAG in an unsymmetric-pattern multifrontal method. Eisenstat and Liu [74, 75] show how to reduce the amount of work in the depth-first search for left-looking LU factorization, via symmetric pruning, and provide a theory of elimination trees for sparse LU factorization. Gilbert and Ng [106] survey methods for determining the nonzero patterns of LU and QR factorizations. Many software packages are available for computing a sparse LU factorization. They are summarized in Section 8.6.

Some sparse LU factorization packages provide a combined row and column prescaling and permutation option, such as the method described by Duff and Koster [57, 58]. This increases the magnitude of the diagonal entries and increases the likelihood of computing an accurate factorization with no partial pivoting at all. Li and Demmel [147] show how static pivoting is particularly useful in a parallel sparse LU algorithm [4, 5, 7, 118, 119, 147, 179, 180].

Exercises

6.1. Use `cs_lu`, `cs_ltsolve`, and `cs_utsolve` to solve $A^Tx = b$ without forming A^T. See Problem 6.15 for an example application.

6.2. Reduce the size of the workspace `xi` in `cs_lu`. Note that `L` and `U` both contain at least `n` unused space after the call to `cs_sprealloc`. This space could be used in a modified `cs_spsolve` for `pstack`. Also note that `L` is unit diagonal, which simplifies `cs_spsolve`.

6.3. Implement column pivoting in `cs_lu`. If no pivot is found in a column or if the largest pivot candidate is below a given tolerance, permute it to the end of the matrix and try the next column in its place. Do not modify `S->q`.

6.4. Write a function with prototype `void cs_relu (cs *A, csn *N, css *S)` that computes the LU factorization of A. It should assume that the nonzero patterns of `L` and `U` have already been computed in a prior call to `cs_lu`. The nonzero pattern of `A` should be the same as in the prior call to `cs_lu`. Use the same pivot permutation.

6.5. Modify `cs_lu` so that it can factorize both numerically and structurally rank-deficient matrices.

6.6. Modify `cs_lu` so that it can factorize rectangular matrices.

6.7. Derive an LU factorization algorithm that computes the kth column of L and the kth row of U at the kth step of factorization (Crout's method). Write a MATLAB prototype and then a C function that implements this factorization for a sparse matrix A. Optionally include partial pivoting.

6.8. Derive an LU factorization algorithm that computes the kth row of L and the kth column of U at the kth step of factorization Why is it difficult to add partial pivoting to this algorithm?

6.9. The MATLAB interface for `cs_lu` sorts both `L` and `U` with a double transpose. Modify it so that it requires only one transpose for `L` and another for `U`. Hint: see Problem 6.1.

6.10. Create `cs_slu`, identical to `cs_sqr` except for one additional option: a symbolic Cholesky analysis, used for the case when `order=1`. Use this as a guess for `S->lnz` and `S->unz`.

6.11. If `cs_sprealloc` fails in `cs_lu`, the function simply halts and reports that it is out of memory. The requested memory space is far more than what might be needed, however. Implement a scheme where $2|L|+n$ is attempted (for $|L|$, as in the current `cs_lu`). If this fails, reduce the request slowly until the request succeeds or until requesting the bare minimum $(|L|+n-k)$ fails. The bare minimum for U is $|U|+k+1$. This feature cannot be tested via a MATLAB mexFunction, because `mxRealloc` terminates a mexFunction if it fails.

6.12. Write a version of `lu_rightpr` that uses a permutation vector `p` instead of a permutation matrix.

6.13. An *incomplete LU factorization* computes approximations of L and U with

fewer nonzeros. It is useful as a preconditioner for iterative methods. One method for computing it is to drop small entries from L and U as they are computed. Another is to use a fixed sparsity pattern, such as the nonzero pattern of A. Write an incomplete LU factorization based on `cs_lu`. The simplest way to do this is where the entries in `x` are copied into `L` and `U` and `lnz` and `unz` are incremented. If the entry is small (`x[i]` for `U` or `x[i]/pivot` for `L`), do not store it and do not increment the corresponding `unz` or `lnz` counter. To drop entries that do not appear in A, scatter the pattern of the kth column of A into the integer work vector, `w`. When storing entries into `U` or `L`, store the value only if `w[i]` is equal to `amark`. If a numerically or structurally zero pivot is encountered, replace it with an arbitrary value (1, say) and select as the pivot row an arbitrary nonpivotal row (preferably the diagonal). See also the MATLAB `luinc` function. Saad [178] provides a detailed look at incomplete Cholesky and LU factorizations for iterative methods.

6.14. *Symmetric pruning* is a technique that can reduce the time to compute Reach($\mathcal{B}$) for the sparse triangular solve. If both $l_{ij} \neq 0$ and $u_{ji} \neq 0$, then any row index $k > i$ in column j of L is not required when computing Reach($\mathcal{B}$). Modify `cs_lu` to exploit symmetric pruning. If A has a symmetric pattern and no partial pivoting occurs, the result is the elimination tree of A.

6.15. Implement a 1-norm condition number estimator in C, using Hager's method in [123] below. Also see Higham's implementation [134] and its generalization [136]. This problem is one example where the solution to $A^T x = b$ is required after factorizing $PAQ = LU$ (Problem 6.1). See also `condest` and `normest` in MATLAB.

```
function c = condest (A)    % estimate of 1-norm condition number of A
[m n] = size (A) ;
if (m ~= n || ~isreal (A))
    error ('A must be square and real') ;
end
if isempty(A)
    c = 0 ;
    return ;
end
[L,U,P,Q] = lu (A) ;
if (~isempty (find (abs (diag (U)) == 0)))
    c = Inf ;
else
    c = norm (A,1) * normest (L,U,P,Q) ;
end

function est = normest (L,U,P,Q)    % 1-norm estimate of inv(A)
n = size (L,1) ;
for k = 1:5
    if (k == 1)
        est = 0 ;
        x = ones (n,1) / n ;
        jold = -1 ;
    else
```



```
        j = min (find (abs (x) == norm (x,inf))) ;
        if (j == jold) break, end ;
        x = zeros (n,1) ;
        x (j) = 1 ;
        jold = j ;
    end
    x = Q * (U \ (L \ (P*x))) ;
    est_old = est ;
    est = norm (x,1) ;
    if (k > 1 && est <= est_old) break, end ;
    s = ones (n,1) ;
    s (find (x < 0)) = -1 ;
    x = P' * (L' \ (U' \ (Q'*s))) ;
end
```