SIAM J. MATRIX ANAL. APPL.
Vol. 26, No. 3, pp. 621–639

ROW MODIFICATIONS OF A SPARSE CHOLESKY FACTORIZATION*

TIMOTHY A. DAVIS† AND WILLIAM W. HAGER‡

Abstract. Given a sparse, symmetric positive definite matrix $\mathbf{C}$ and an associated sparse Cholesky factorization $\mathbf{LDL}^\mathsf{T}$, we develop sparse techniques for updating the factorization after a symmetric modification of a row and column of $\mathbf{C}$. We show how the modification in the Cholesky factorization associated with this rank-2 modification of $\mathbf{C}$ can be computed efficiently using a sparse rank-1 technique developed in [T. A. Davis and W. W. Hager, *SIAM J. Matrix Anal. Appl.*, 20 (1999), pp. 606–627]. We also determine how the solution of a linear system $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ changes after changing a row and column of $\mathbf{C}$ or after a rank-r change in $\mathbf{C}$.

Key words. numerical linear algebra, direct methods, Cholesky factorization, sparse matrices, mathematical software, matrix updates

AMS subject classifications. 65F05, 65F50, 65Y20

DOI. 10.1137/S089547980343641X

1. Introduction.

The problem of updating a Cholesky factorization after a small rank change in the matrix is a fundamental problem with many applications, including optimization algorithms, least-squares problems in statistics, the analysis of electrical circuits and power systems, structural mechanics, boundary condition changes in partial differential equations, domain decomposition methods, and boundary element methods (see [12]). Some specific examples follow.

1. A linear programming problem has the form

$$\min\ \boldsymbol{c}^\mathsf{T}\boldsymbol{x}\ \text{ subject to }\ \mathbf{A}\boldsymbol{x} = \boldsymbol{b},\quad \boldsymbol{x} \geq \mathbf{0}, \tag{1.1}$$

where $\mathbf{A}$ is m-by-n, typically n is much larger than m, and all vectors are of compatible size. In this formulation, the vector $\boldsymbol{x}$ is called the primal variable. The dual approach utilizes a multiplier $\boldsymbol{\lambda}$ corresponding to the linear equation $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$. In each iteration of the linear programming dual active set algorithm (LPDASA) (see [5, 13, 14, 15, 16, 17]), we solve a symmetric linear system of the form

$$\mathbf{C}\boldsymbol{\lambda} = \boldsymbol{f},\quad \mathbf{C} = \mathbf{A}_F\mathbf{A}_F^\mathsf{T} + \sigma\mathbf{I},$$

where $\sigma > 0$ is a small parameter, $F \subset \{1, 2, \ldots, n\}$ are the indices associated with "free variables" (strictly positive primal variables), $\mathbf{A}_F$ is a submatrix of $\mathbf{A}$ associated with column indices in F, and $\boldsymbol{f}$ is a function of $\boldsymbol{b}$ and $\boldsymbol{c}$. As the dual iterates converge to optimality, the set F changes as the primal variables either reach their bound or become free. Since $\mathbf{C}$ can be expressed as

$$\mathbf{C} = \sum_{j\in F} \mathbf{A}_{*j}\mathbf{A}_{*j}^\mathsf{T} + \sigma\mathbf{I},$$

*Received by the editors October 21, 2003; accepted for publication (in revised form) by E. Ng April 9, 2004; published electronically March 3, 2005. This material is based upon work supported by the National Science Foundation under grant CCR-0203270.
http://www.siam.org/journals/simax/26-3/43641.html

†Department of Computer and Information Science and Engineering, University of Florida, P.O. Box 116120, Gainesville, FL 32611-6120 (davis@cise.ufl.edu, http://www.cise.ufl.edu/∼davis).

‡Department of Mathematics, University of Florida, P.O. Box 118105, Gainesville, FL 32611-8105 (hager@math.ufl.edu, http://www.math.ufl.edu/∼hager).

where $\mathbf{A}_{*j}$ denotes the jth column of $\mathbf{A}$, it follows that a small change in F leads to a small rank change in $\mathbf{C}$; hence, we solve a sequence of linear systems where each matrix is a small rank modification of the previous matrix.

2. Consider a network of resistors connecting nodes $\{1, 2, \ldots, n\}$ in a graph. Let $\mathcal{A}_i$ denote the set of nodes adjacent to i in the graph, let R_{ij} be the resistance between i and j, and let V_j be the potential at node j (some of the nodes may be held at a fixed potential by a battery). By Kirchhoff's first law, the sum of the currents entering each node is zero:

$$\sum_{j \in \mathcal{A}_i} \frac{V_j - V_i}{R_{ij}} = 0.$$

If the resistance on an arc (k, l) is changed from R_{kl} to $\bar{R}_{kl}$, then there is a rank-1 change in the matrix given by

$$\left(\frac{1}{R_{kl}} - \frac{1}{\bar{R}_{kl}} \right) \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}, \quad \boldsymbol{w} = \boldsymbol{e}_k - \boldsymbol{e}_l,$$

where $\boldsymbol{e}_i$ is the ith column of the identity matrix. In other words, the only change in the coefficient matrix occurs in rows k and l and in columns k and l. Changing the resistance on r arcs in the network corresponds to a rank-r change in the matrix. Additional illustrations can be found in [12].

A variety of techniques for modifying a dense Cholesky factorization are given in the classic reference [11]. Recently in [3, 4] we considered a sparse Cholesky factorization $\mathbf{LDL}^{\mathsf{T}}$ of a symmetric, positive definite matrix $\mathbf{C}$, and the modification associated with a rank-r change of the form $\overline{\mathbf{C}} = \mathbf{C} \pm \mathbf{W}\mathbf{W}^{\mathsf{T}}$, where $\mathbf{W}$ is n-by-r with r typically much less than n. In a rank-1 update of the form $\overline{\mathbf{C}} = \mathbf{C} + \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}$, the columns that change in $\mathbf{L}$ correspond to a path in the elimination tree of the modified factor $\overline{\mathbf{L}}$. The path starts at the node corresponding to the row index of the first nonzero entry in $\boldsymbol{w}$. The total work of the rank-1 update is proportional to the number of entries in $\mathbf{L}$ that change, so our algorithm is optimal. A downdate is analogous; it follows a path in the original elimination tree, which becomes a subtree in the new elimination tree.

A rank-r update of the form $\overline{\mathbf{C}} = \mathbf{C} + \mathbf{W}\mathbf{W}^{\mathsf{T}}$, where $\mathbf{W}$ has r columns, can be cast as a sequence of r rank-1 updates. In [4], we show that a rank-r update can be done more efficiently in a single pass. Rather than following a single path in the tree, multiple paths are followed. When paths merge, multiple updates are performed to the corresponding columns of $\mathbf{L}$. Our rank-r algorithm is also optimal.

Figure 1.1 shows an example of a sparse rank-2 update (see Figure 4.1 in [4]). Entries that change in $\overline{\mathbf{C}}$ are shown as a plus. It is not shown in the figure, but the updates follow two paths in the tree, one with nodes $\{1, 2, 6, 8\}$ and the second one with nodes $\{3, 4, 5, 6, 7, 8\}$.

In this paper, we consider a special, but important, rank-2 change corresponding to a symmetric modification of a row and column of $\mathbf{C}$. Although we could, in principle, use our previous methodology to update the factorization, we observe that this rank-2 approach is much less efficient than the streamlined approach we develop here. In fact, the rank-r approach with $r = 2$ could result in a completely dense modification of the factorization, where nonzero entries are first introduced and then canceled out. Figure 1.2 shows a sparse modification to row 4 and column 4 of the matrix $\mathbf{C}$ from Figure 1.1.

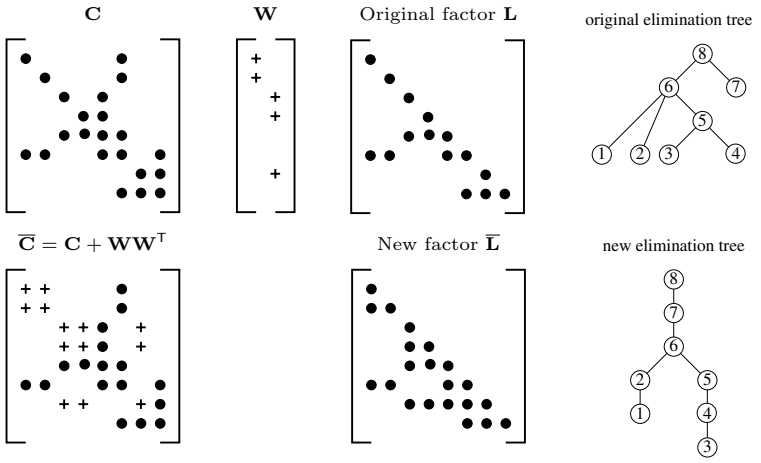

FIG. 1.1. *Rank-2 update,* $\overline{\mathbf{C}} = \mathbf{C} + \mathbf{W}\mathbf{W}^{\mathsf{T}}$.

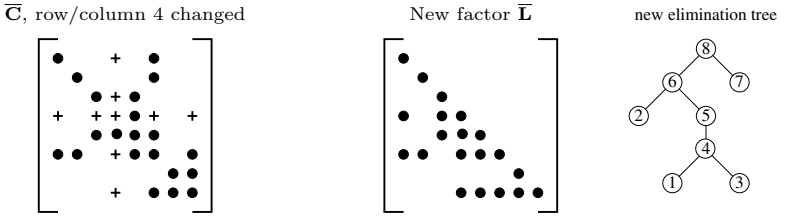

FIG. 1.2. *Modification to a row and column of* $\mathbf{C}$.

With the new approach, the work connected with *removing* a nonzero row and column is comparable to the work associated with a sparse rank-1 *update*, while the work associated with *adding* a new nonzero row and column is comparable to the work associated with a sparse rank-1 *downdate*. This connection between the modification of the matrix and the modification of the factorization is nonintuitive: When we remove elements from the matrix, we update the factorization; when we add elements to the matrix, we downdate the factorization.

As a byproduct of our analysis, we show how the solution to a triangular system $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ changes when both $\mathbf{L}$ and $\boldsymbol{b}$ change as a result of the row and column modification problem discussed in this paper, or as a result of a rank-r change to $\mathbf{C}$ [3, 4].

One specific application for the techniques developed in this paper is LPDASA. An inequality $\boldsymbol{a}^{\mathsf{T}}\boldsymbol{x} \leq b$ in a primal linear program is converted to an equality, when the problem is written in the standard form (1.1), by introducing a primal slack variable: $\boldsymbol{a}^{\mathsf{T}}\boldsymbol{x} + y = b$, where $y \geq 0$ is the slack variable. If the index j corresponds to a primal slack variable in equation i, and if $j \in F$, then it can be shown that $\lambda_i = c_j$. In essence, we can eliminate the ith dual variable and the ith equation: The ith equality is satisfied by simply solving for the value of the slack variable, and the ith dual variable is $\lambda_i = c_j$. Thus, in this dual approach to linear programming, inactive inequalities are identified dynamically, during the solution process; dropping these inactive inequalities amounts to removing a row and a column from a Cholesky

factorized matrix. In the same way, when a dropped inequality later becomes active, a new row and column must be inserted in the matrix, and the resulting modification in the Cholesky factorization evaluated. In general, the techniques developed in this paper are useful in any setting where a system of equations is solved repeatedly with equations added or dropped before each solve.

A brief overview of our paper follows. In section 2 we consider the *row addition* problem, in which a row and column, originally zero except for the diagonal element, are modified in a matrix. The *row deletion* problem is the opposite of row addition and is discussed in section 3. Section 4 describes modifications to a row and column of sparse or dense $\mathbf{C}$. We show that arbitrary modifications can be efficiently implemented as a row deletion followed by a row addition. In contrast, we also show that if sparse modifications to a sparse row of $\mathbf{C}$ are made, some improvement can be obtained over a row deletion followed by a row addition. The efficient methods presented in sections 2 through 4 are contrasted with performing the modifications as a rank-2 outer product modification in section 5, which is shown to be costly, particularly in the sparse case. Section 6 shows how to efficiently modify the solution to $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ when $\mathbf{L}$ and $\boldsymbol{b}$ change. A brief presentation of the experimental performance of these methods in the context of matrices arising in linear programming is given in section 7.

We use the notation $\overline{\mathbf{C}}$ to denote the matrix $\mathbf{C}$ after it has been modified. Bold uppercase $\mathbf{A}$ refers to a matrix. Bold lowercase italic $\boldsymbol{a}$ is a column vector; thus, $\boldsymbol{a}^{\mathsf{T}}$ always refers to a row vector. Plain lowercase letters (such as a and α) are scalars. We use $|\mathbf{A}|$ to denote the number of nonzero entries in the sparse matrix $\mathbf{A}$. Without parentheses, the notation $\mathbf{A}_i$ or $\mathbf{A}_{ij}$ refers to submatrices of a matrix $\mathbf{A}$ (sometimes 1-by-1 submatrices). We use parentheses $(\mathbf{A})_{ij}$ to refer to the entry in row i and column j of the matrix $\mathbf{A}$, $(\mathbf{A})_{*j}$ to refer to column j of $\mathbf{A}$, and $(\mathbf{A})_{i*}$ to refer to row i of $\mathbf{A}$. When counting floating-point operations (flops), we count one flop for any arithmetic operation including $*$, $/$, $+$, $-$, and $\sqrt{}$.

2. Adding a row and column to C. If we have a rectangular n-by-m matrix $\mathbf{A}$, and $\mathbf{C} = \alpha\mathbf{I} + \mathbf{A}\mathbf{A}^{\mathsf{T}}$, then modifying row k of $\mathbf{A}$ leads to changes in the kth row and column of $\mathbf{C}$. In this section, we consider the special case where row k is initially zero and becomes nonzero in $\overline{\mathbf{A}}$ (the *row addition* case). Equivalently, the kth row and column of $\mathbf{C}$ is initially a multiple of the kth row and column of the identity matrix and changes to some other value.

We first discuss the linear algebra that applies whether $\mathbf{C}$ is dense or sparse. Specific issues for the dense and sparse case are discussed in sections 2.1 and 2.2.

Let $\overline{\boldsymbol{a}}_2^{\mathsf{T}}$ be the new nonzero kth row of $\overline{\mathbf{A}}$,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{0}^{\mathsf{T}} \\ \mathbf{A}_3 \end{bmatrix}, \quad \overline{\mathbf{A}} = \begin{bmatrix} \mathbf{A}_1 \\ \overline{\boldsymbol{a}}_2^{\mathsf{T}} \\ \mathbf{A}_3 \end{bmatrix},$$

where $\mathbf{0}^{\mathsf{T}}$ is a row vector whose entries are all 0. This leads to a modification to row and column k of $\mathbf{C}$,

$$\mathbf{C} = \begin{bmatrix} \alpha\mathbf{I} + \mathbf{A}_1\mathbf{A}_1^{\mathsf{T}} & \mathbf{0} & \mathbf{A}_1\mathbf{A}_3^{\mathsf{T}} \\ \mathbf{0}^{\mathsf{T}} & \alpha & \mathbf{0}^{\mathsf{T}} \\ \mathbf{A}_3\mathbf{A}_1^{\mathsf{T}} & \mathbf{0} & \alpha\mathbf{I} + \mathbf{A}_3\mathbf{A}_3^{\mathsf{T}} \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{0} & \mathbf{C}_{31}^{\mathsf{T}} \\ \mathbf{0}^{\mathsf{T}} & c_{22} & \mathbf{0}^{\mathsf{T}} \\ \mathbf{C}_{31} & \mathbf{0} & \mathbf{C}_{33} \end{bmatrix},$$

where $c_{22} = \alpha$. We can let α be zero; even though $\mathbf{C}$ would no longer be positive definite, the submatrix excluding row and column k could still be positive definite.

The linear system $\mathbf{C}\boldsymbol{x} = \boldsymbol{b}$ is well defined in this case, except for $\boldsymbol{x}_k$. The new matrix $\overline{\mathbf{C}}$ is given as

$$\overline{\mathbf{C}} = \begin{bmatrix} \alpha\mathbf{I} + \mathbf{A}_1\mathbf{A}_1^\mathsf{T} & \mathbf{A}_1\overline{\boldsymbol{a}}_2 & \mathbf{A}_1\mathbf{A}_3^\mathsf{T} \\ \overline{\boldsymbol{a}}_2^\mathsf{T}\mathbf{A}_1^\mathsf{T} & \alpha + \overline{\boldsymbol{a}}_2^\mathsf{T}\overline{\boldsymbol{a}}_2 & \overline{\boldsymbol{a}}_2^\mathsf{T}\mathbf{A}_3^\mathsf{T} \\ \mathbf{A}_3\mathbf{A}_1^\mathsf{T} & \mathbf{A}_3\overline{\boldsymbol{a}}_2 & \alpha\mathbf{I} + \mathbf{A}_3\mathbf{A}_3^\mathsf{T} \end{bmatrix} = \begin{bmatrix} \mathbf{C}_{11} & \overline{\boldsymbol{c}}_{12} & \mathbf{C}_{31}^\mathsf{T} \\ \overline{\boldsymbol{c}}_{12}^\mathsf{T} & \overline{c}_{22} & \overline{\boldsymbol{c}}_{32}^\mathsf{T} \\ \mathbf{C}_{31} & \overline{\boldsymbol{c}}_{32} & \mathbf{C}_{33} \end{bmatrix}.$$

Thus, adding a row k to $\mathbf{A}$ is equivalent to adding a row and column k to $\mathbf{C}$. Note that changing row and column k of $\mathbf{C}$ from zero (except for the diagonal entry) to a nonzero value also can be viewed as increasing the dimension of the $(n-1)$-by-$(n-1)$ matrix

$$\begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{31}^\mathsf{T} \\ \mathbf{C}_{31} & \mathbf{C}_{33} \end{bmatrix}.$$

The original factorization of the n-by-n matrix $\mathbf{C}$ may be written as

$$\begin{aligned} \mathbf{L}\mathbf{D}\mathbf{L}^\mathsf{T} &= \begin{bmatrix} \mathbf{L}_{11} & & \\ \mathbf{0}^\mathsf{T} & 1 & \\ \mathbf{L}_{31} & \mathbf{0} & \mathbf{L}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{D}_{11} & & \\ & d_{22} & \\ & & \mathbf{D}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^\mathsf{T} & \mathbf{0} & \mathbf{L}_{31}^\mathsf{T} \\ & 1 & \mathbf{0}^\mathsf{T} \\ & & \mathbf{L}_{33}^\mathsf{T} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{C}_{11} & \mathbf{0} & \mathbf{C}_{31}^\mathsf{T} \\ \mathbf{0}^\mathsf{T} & \alpha & \mathbf{0}^\mathsf{T} \\ \mathbf{C}_{31} & \mathbf{0} & \mathbf{C}_{33} \end{bmatrix}, \end{aligned}$$

which leads to the four equations

$$\begin{aligned} \mathbf{L}_{11}\mathbf{D}_{11}\mathbf{L}_{11}^\mathsf{T} &= \mathbf{C}_{11}, \\ d_{22} &= \alpha, \\ \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{11}^\mathsf{T} &= \mathbf{C}_{31}, \\ \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^\mathsf{T} + \mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^\mathsf{T} &= \mathbf{C}_{33}. \end{aligned} \tag{2.1}$$

After adding row and column k to obtain $\overline{\mathbf{C}}$, we have the factorization

$$\begin{aligned} \overline{\mathbf{L}\mathbf{D}\mathbf{L}}^\mathsf{T} &= \begin{bmatrix} \mathbf{L}_{11} & & \\ \overline{\boldsymbol{l}}_{12}^\mathsf{T} & 1 & \\ \mathbf{L}_{31} & \overline{\boldsymbol{l}}_{32} & \overline{\mathbf{L}}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{D}_{11} & & \\ & \overline{d}_{22} & \\ & & \overline{\mathbf{D}}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^\mathsf{T} & \overline{\boldsymbol{l}}_{12} & \mathbf{L}_{31}^\mathsf{T} \\ & 1 & \overline{\boldsymbol{l}}_{32}^\mathsf{T} \\ & & \overline{\mathbf{L}}_{33}^\mathsf{T} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{C}_{11} & \overline{\boldsymbol{c}}_{12} & \mathbf{C}_{31}^\mathsf{T} \\ \overline{\boldsymbol{c}}_{12}^\mathsf{T} & \overline{c}_{22} & \overline{\boldsymbol{c}}_{32}^\mathsf{T} \\ \mathbf{C}_{31} & \overline{\boldsymbol{c}}_{32} & \mathbf{C}_{33} \end{bmatrix}. \end{aligned} \tag{2.2}$$

Note that $\mathbf{L}_{11}$ and $\mathbf{L}_{31}$ do not change as a result of modifying row and column k of $\mathbf{C}$. From (2.2), the relevant equations are

$$\mathbf{L}_{11}\mathbf{D}_{11}\overline{\boldsymbol{l}}_{12} = \overline{\boldsymbol{c}}_{12}, \tag{2.3}$$

$$\begin{aligned} \overline{\boldsymbol{l}}_{12}^\mathsf{T}\mathbf{D}_{11}\overline{\boldsymbol{l}}_{12} + \overline{d}_{22} &= \overline{c}_{22}, \\ \mathbf{L}_{31}\mathbf{D}_{11}\overline{\boldsymbol{l}}_{12} + \overline{\boldsymbol{l}}_{32}\overline{d}_{22} &= \overline{\boldsymbol{c}}_{32}, \end{aligned}$$

$$\mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^\mathsf{T} + \overline{\boldsymbol{l}}_{32}\overline{d}_{22}\overline{\boldsymbol{l}}_{32}^\mathsf{T} + \overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^\mathsf{T} = \mathbf{C}_{33}. \tag{2.4}$$

Let $\boldsymbol{w} = \bar{\boldsymbol{l}}_{32}\sqrt{\bar{d}_{22}}$. Combining (2.4) with the original equation (2.1), we obtain

$$\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = (\mathbf{C}_{33} - \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^{\mathsf{T}}) - \bar{\boldsymbol{l}}_{32}\bar{d}_{22}\bar{\boldsymbol{l}}_{32}^{\mathsf{T}} = \mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} - \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}.$$

The factorization of $\mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} - \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}$ can be computed as a rank-1 downdate of the original factorization $\mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}}$ (see [3]). This derivation leads to Algorithm 1 for computing the modified factorization $\overline{\mathbf{L}\mathbf{D}\mathbf{L}}^{\mathsf{T}}$, which is applicable in both the dense and sparse cases.

ALGORITHM 1 (ROW ADDITION).

1. Solve the lower triangular system $\mathbf{L}_{11}\mathbf{D}_{11}\bar{\boldsymbol{l}}_{12} = \bar{\boldsymbol{c}}_{12}$ for $\bar{\boldsymbol{l}}_{12}$.
2. $\bar{d}_{22} = \bar{c}_{22} - \bar{\boldsymbol{l}}_{12}^{\mathsf{T}}\mathbf{D}_{11}\bar{\boldsymbol{l}}_{12}$
3. $\bar{\boldsymbol{l}}_{32} = (\bar{\boldsymbol{c}}_{32} - \mathbf{L}_{31}\mathbf{D}_{11}\bar{\boldsymbol{l}}_{12})/\bar{d}_{22}$
4. $\boldsymbol{w} = \bar{\boldsymbol{l}}_{32}\sqrt{\bar{d}_{22}}$
5. Perform the rank-1 downdate $\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = \mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} - \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}$.

end Algorithm 1

2.1. Dense row addition. Consider the case when $\mathbf{C}$ is dense.

1. Step (1) of Algorithm 1 requires the solution of a unit lower triangular system $\mathbf{L}_{11}\boldsymbol{y} = \bar{\boldsymbol{c}}_{12}$ of order $k-1$. The computation of $\boldsymbol{y}$ takes $(k-1)^2 - (k-1)$ flops, and computing $\bar{\boldsymbol{l}}_{12} = \mathbf{D}_{11}^{-1}\boldsymbol{y}$ takes another $k-1$ flops.
2. Step (2) requires $2(k-1)$ work, using $\boldsymbol{y}$.
3. Step (3) is the matrix-vector multiply $\mathbf{L}_{31}\boldsymbol{y}$, where $\mathbf{L}_{31}$ is $(n-k)$-by-$(k-1)$ and thus takes $2(n-k)(k-1)$ operations and $k-1$ more to divide by $\bar{d}_{22}$.
4. Step (4) requires one square root operation and $n-k$ multiplications.
5. Finally, the rank-1 downdate of step (5) takes $2(n-k)^2 + 4(n-k)$ operations using method C1 of [11] (see [4]).

For the dense case, the total number of flops performed by Algorithm 1 is $2n^2 + 3n + k^2 - (2nk + 2k + 1)$. This is roughly $2n^2$ when $k = 1$, n^2 when $k = n$, and $(5/4)n^2$ when $k = n/2$.

2.2. Sparse row addition. If $\mathbf{C}$ is sparse, each step of Algorithm 1 must operate on sparse matrices. The graph algorithms and data structures must efficiently support each step. We will assume that $\mathbf{L}$ is stored in a compressed column vector form, where the row indices in each column are sorted in ascending order. This is the same data structure used in [3, 4], except that the algorithms presented there do not require sorted row indices, but they do require the integer *multiplicity* of each nonzero entry of $\mathbf{L}$ to support an efficient symbolic downdate operation. The algorithm discussed below will not require the multiplicities.

Maintaining the row indices in sorted order requires a merge operation for the set union computation to determine the new nonzero patterns of the columns of $\mathbf{L}$, rather than a simpler unsorted set union used in [3, 4]. It has no effect on asymptotic complexity and little effect on the run time. Although more work is required to maintain the row indices in sorted order, time is gained elsewhere in the algorithm. Operating on columns in sorted order in the forward solve of $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$, for example, is faster than operating on a matrix with jumbled columns. No additional space is required to keep the columns sorted.

Step (1) of Algorithm 1 solves the lower triangular system $\mathbf{L}_{11}\boldsymbol{y} = \bar{\boldsymbol{c}}_{12}$, where all three terms in this system are sparse. Gilbert and Peierls have shown how to solve this system optimally, in time proportional to the number of flops required [9, 10]. We review their method here.

Consider the n-by-n system $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$, where $\mathbf{L}$ is lower triangular and both $\mathbf{L}$ and $\boldsymbol{b}$ are sparse. The solution will be sparse, and the total work may be less than $O(n)$. We cannot use a conventional algorithm that iterates over each column of $\mathbf{L}$ and skips those for which $\boldsymbol{x}$ is zero since the work involved will then be at least n. Instead, the nonzero pattern of $\boldsymbol{x}$ must first be computed, and then the corresponding columns of $\mathbf{L}$ can be used to compute $\boldsymbol{x}$ in time proportional to the floating-point work required.

Let G_L be a graph with n nodes and with a directed edge from node j to node i if and only if l_{ij} is nonzero. Gilbert and Peierls show that x_j is nonzero (ignoring numerical cancellation) if and only if there is a path of length zero or more from some node i, where $b_i \neq 0$, to node j in the graph G_L. Computing the pattern of $\boldsymbol{x}$ requires a graph traversal, starting from the nodes corresponding to the nonzero pattern of $\boldsymbol{b}$. It can be done in time proportional to the number of edges traversed. Each of these edges is a nonzero in $\mathbf{L}$ that takes part in the subsequent numerical computation.

The above result holds for any lower triangular matrix $\mathbf{L}$. In our case, $\mathbf{L}$ arises from a Cholesky factorization and has an *elimination tree* [18, 19]. The elimination tree of $\mathbf{L}$ has n nodes. The parent of node j in the elimination tree is the smallest index $i > j$ for which $l_{ij} \neq 0$; node j is a root if there is no such i. Since the nonzero pattern of $(\mathbf{L})_{*j}$ is a subset of its path to the root of the elimination tree [20], all the nodes in G_L that can be reached from node j correspond to the path from node j to the root of the elimination tree. Traversing the paths in the tree, starting at nodes corresponding to nonzero entries in $\boldsymbol{b}$, takes time proportional to the number of nonzero entries in $\boldsymbol{x}$. A general graph traversal of G_L is not required. Step (1) of Algorithm 1 takes

$$O\left(\sum_{(\bar{\boldsymbol{l}}_{12})_j \neq 0} |(\mathbf{L}_{11})_{*j}|\right)$$

time to compute the nonzero pattern and numerical values of both $\boldsymbol{y}$ and $\bar{\boldsymbol{l}}_{12}$. The insertion of the nonzero entries of $\bar{\boldsymbol{l}}_{12}$ into the data structure of $\mathbf{L}$ is performed in conjunction with step (3).

Step (2) is a scaled dot product operation and can be computed in time proportional to the number of nonzero entries in $\bar{\boldsymbol{l}}_{12}$.

Step (3) is a matrix-vector multiply operation. It accesses the same columns of $\mathbf{L}$ used by the sparse lower triangular solve, namely, each column j for which the jth entry in $\bar{\boldsymbol{l}}_{12}$ is nonzero. These same columns need to be modified by shifting entries in $\mathbf{L}_{31}$ down by one and inserting the new entries in $\bar{\boldsymbol{l}}_{12}$, the kth row of $\overline{\mathbf{L}}$. No other columns in the range 1 to $k-1$ need to be accessed or modified by steps (1) through (3). When step (3) completes, the new column k of $\overline{\mathbf{L}}$ needs to be inserted into the data structure. This can be done in one of two ways. In the general case, we can store the columns themselves in a noncontiguous manner and simply allocate new space for this column. A similar strategy can be used for any columns 1 through $k-1$ of $\overline{\mathbf{L}}$ that outgrow their originally allocated space with no increase in asymptotic run time. Alternatively, we may know an a priori upper bound on the size of each column of $\mathbf{L}$ after all row additions have been performed. In this case, a simpler static allocation strategy is possible. This latter case occurs in our use of the row addition algorithm in LPDASA, our target application [5]. In either case, the time to insert $\bar{\boldsymbol{l}}_{12}$ into the data structure and to compute $\bar{\boldsymbol{l}}_{32}$ in step (3) is

$$O\left(\sum_{(\bar{\boldsymbol{l}}_{12})_j \neq 0} |(\mathbf{L}_{31})_{*j}|\right).$$

Step (4) is a simple scalar-times-vector operation. The total time for steps (1) through (4) is

$$O\left(\sum_{(\bar{\boldsymbol{l}}_{12})_j \neq 0} |(\mathbf{L})_{*j}|\right).$$

Step (5) almost fits the specifications of the sparse rank-1 modification in [3], but with one interesting twist. The original kth row and column of $\mathbf{C}$ are zero, except for the placeholder diagonal entry, α. The new row and column only add entries to $\mathbf{C}$, and thus the nonzero pattern of the original factor $\mathbf{L}$ is a subset of the nonzero pattern of $\overline{\mathbf{L}}$ (ignoring numerical cancellation). The rank-1 modification in Algorithm 1 is a symbolic update (new nonzero entries are added, not removed) and a numeric downdate $\mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} - \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}$. Since the multiplicities used in [3] are needed only for a subsequent symbolic downdate, they are not required by the row addition algorithm. They would be required by a row deletion algorithm that maintains a strict nonzero pattern of $\mathbf{L}$; this issue is addressed in section 3.2.

The rank-1 modification to obtain the factorization $\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}}$ takes time proportional to the number of nonzero entries in $\overline{\mathbf{L}}_{33}$ that change. The columns that change correspond to the path from node k to the root of the elimination tree of $\overline{\mathbf{L}}$. This path is denoted $\overline{\mathcal{P}}$ in [3]. At each node j along the path $\overline{\mathcal{P}}$, at most four flops are performed for each nonzero entry in $(\overline{\mathbf{L}})_{*j}$.

With our choice of data structures, exploitation of the elimination tree, and the rank-1 modification from [3], the total time taken by Algorithm 1 is proportional to the total number of nonzero entries in columns corresponding to nonzero entries in $\bar{\boldsymbol{l}}_{12}$, to compute steps (1) through (4), plus the time required for the rank-1 modification in step (5). The total time is

$$O\left(\sum_{(\bar{\boldsymbol{l}}_{12})_j \neq 0} |(\mathbf{L})_{*j}| + \sum_{j \in \overline{\mathcal{P}}} |(\overline{\mathbf{L}})_{*j}|\right).$$

This time includes all data structure manipulations, sparsity pattern computation, and graph algorithms required to implement the algorithm. It is identical to the total number of flops required, and thus Algorithm 1 is optimal. In the sparse case, if every column j takes part in the computation, the time is $O(|\overline{\mathbf{L}}|)$. Normally, not all columns will be affected by the sparse row addition. If the new row and column of $\overline{\mathbf{L}}$ are very sparse, only a few columns take part in the computation.

3. Row deletion. By *deleting* a row and column k from the matrix $\mathbf{C}$, we mean setting the entire row and column to zero, except for the diagonal entry $(\mathbf{C})_{kk}$ which is set to α. This is the opposite of row addition. Here, we present an algorithm that applies whether $\mathbf{C}$ is sparse or dense. Specific issues in the dense case are considered in section 3.1, and the sparse case is discussed in section 3.2.

Prior to deleting row and column k, we have the original factorization

$$\begin{aligned}\mathbf{L}\mathbf{D}\mathbf{L}^{\mathsf{T}} &= \begin{bmatrix} \mathbf{L}_{11} & & \\ \boldsymbol{l}_{12}^{\mathsf{T}} & 1 & \\ \mathbf{L}_{31} & \boldsymbol{l}_{32} & \mathbf{L}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{D}_{11} & & \\ & d_{22} & \\ & & \mathbf{D}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^{\mathsf{T}} & \boldsymbol{l}_{12} & \mathbf{L}_{31}^{\mathsf{T}} \\ & 1 & \boldsymbol{l}_{32}^{\mathsf{T}} \\ & & \mathbf{L}_{33}^{\mathsf{T}} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{C}_{11} & \boldsymbol{c}_{12} & \mathbf{C}_{31}^{\mathsf{T}} \\ \boldsymbol{c}_{12}^{\mathsf{T}} & c_{22} & \boldsymbol{c}_{32}^{\mathsf{T}} \\ \mathbf{C}_{31} & \boldsymbol{c}_{32} & \mathbf{C}_{33} \end{bmatrix}.\end{aligned}$$

After deleting row and column k, we have

$$\overline{\mathbf{L}}\overline{\mathbf{D}}\overline{\mathbf{L}}^{\mathsf{T}} = \begin{bmatrix} \mathbf{L}_{11} & & \\ \mathbf{0}^{\mathsf{T}} & 1 & \\ \mathbf{L}_{31} & \mathbf{0} & \overline{\mathbf{L}}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{D}_{11} & & \\ & \alpha & \\ & & \overline{\mathbf{D}}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{L}_{11}^{\mathsf{T}} & \mathbf{0} & \mathbf{L}_{31}^{\mathsf{T}} \\ & 1 & \mathbf{0}^{\mathsf{T}} \\ & & \overline{\mathbf{L}}_{33}^{\mathsf{T}} \end{bmatrix}$$
$$= \begin{bmatrix} \mathbf{C}_{11} & \mathbf{0} & \mathbf{C}_{31}^{\mathsf{T}} \\ \mathbf{0}^{\mathsf{T}} & \alpha & \mathbf{0}^{\mathsf{T}} \\ \mathbf{C}_{31} & \mathbf{0} & \mathbf{C}_{33} \end{bmatrix}.$$

Thus we only need to set row and column k of $\overline{\mathbf{L}}$ to zero, set the diagonal entry to α, and compute $\overline{\mathbf{L}}_{33}$ and $\overline{\mathbf{D}}_{33}$. The original factorization is

$$\mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} = \mathbf{C}_{33} - \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^{\mathsf{T}} - \boldsymbol{l}_{32}d_{22}\boldsymbol{l}_{32}^{\mathsf{T}},$$

while the new factorization is given as

$$\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = \mathbf{C}_{33} - \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^{\mathsf{T}}.$$

Combining these two equations, we have a numeric rank-1 update,

$$\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = \mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} + \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}, \tag{3.1}$$

where $\boldsymbol{w} = \boldsymbol{l}_{32}\sqrt{d_{22}}$. Algorithm 2 gives the complete row deletion algorithm, which is applicable in both the dense and sparse cases.

ALGORITHM 2 (ROW DELETION).

1. $\overline{\boldsymbol{l}}_{12} = \mathbf{0}$
2. $\overline{d}_{22} = \alpha$
3. $\overline{\boldsymbol{l}}_{32} = \mathbf{0}$
4. $\boldsymbol{w} = \boldsymbol{l}_{32}\sqrt{d_{22}}$
5. Perform the rank-1 update $\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = \mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} + \boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}$.

end Algorithm 2

3.1. Dense row deletion. When $\mathbf{C}$ is dense, the number of flops performed by Algorithm 2 is $2(n-k)^2+5(n-k)+1$ (the same as steps (4) and (5) of Algorithm 1). This is roughly $2n^2$ when $k = 1$, and $(1/2)n^2$ when $k = n/2$. No work is required when $k = n$.

3.2. Sparse row deletion. When row and column k of a sparse $\mathbf{C}$ are deleted to obtain $\overline{\mathbf{C}}$, no new nonzero terms will appear in $\overline{\mathbf{L}}$, and some nonzero entries in $\mathbf{L}$ may become zero. We refer to the deletion of entries in $\mathbf{L}$ as a symbolic downdate [3]. The symbolic downdate is combined with a numeric rank-1 update because of the addition of $\boldsymbol{w}\boldsymbol{w}^{\mathsf{T}}$ in (3.1).

We cannot simply delete entries from $\overline{\mathbf{L}}$ that become numerically zero. An entry in $\overline{\mathbf{L}}$ can be removed only if it becomes *symbolically zero* (that is, its value is zero regardless of the assignment of numerical values to the nonzero pattern of $\mathbf{C}$). If entries are zero because of exact numerical cancellation and are dropped from the data structure of $\mathbf{L}$, then the elimination tree no longer characterizes the structure of the matrix. If the elimination tree is no longer valid, subsequent updates and downdates will not be able to determine which columns of $\mathbf{L}$ must be modified.

Steps (1) through (3) of Algorithm 2 require no numerical work, but they do require some data structure modifications. All of the entries in row and column k

of $\overline{\mathbf{L}}$ become symbolically zero and can be removed. If $\mathbf{L}$ is stored by columns, it is trivial in step (3) to immediately delete all entries in column $\boldsymbol{l}_{32}$. On the other hand, the statement $\bar{\boldsymbol{l}}_{12} = \mathbf{0}$ in step (1) is less obvious. Each nonzero element in row k lies in a different column. We must either set these values to zero, delete them from the data structure, or flag row k as zero and require any subsequent algorithm that accesses the matrix $\mathbf{L}$ to ignored flagged rows. The latter option would lead to a sparse row deletion algorithm with optimal run time but complicates all other algorithms in our application and increases their run time. We choose to search for the row k entries in each column in which they appear and delete them from the data structure.

To set $\boldsymbol{l}_{12}$ to zero and delete the entries from the data structure for $\mathbf{L}$ requires a scan of all the columns j of $\mathbf{L}$ for which the jth entry of $\boldsymbol{l}_{12}$ is nonzero. The time taken for this operation is asymptotically bounded by the time taken for steps (1) through (3) of sparse row addition (Algorithm 1), but the bound is not tight. The nonzero pattern of row k of $\mathbf{L}$ can easily be found from the elimination tree. Finding the row index k in the columns takes less time than step (1) of Algorithm 1 since a binary search can be used. Deleting the entries takes time equivalent to step (3) of Algorithm 1 since we maintain each column with sorted row indices.

The immediate removal of entries in $\overline{\mathbf{L}}_{33}$ can be done using the symbolic rank-1 downdate presented in [3]. However, this requires an additional array of multiplicities, which is one additional integer value for each nonzero in the matrix. Instead, we can allow these entries to become numerically zero (or very small values due to numerical roundoff) and not remove them immediately. Since they become numerically zero (or tiny), they can simply remain in the data structure for the matrix and have no effect on subsequent operations that use the matrix $\mathbf{L}$. The entries can be pruned later on by a complete symbolic factorization, taking $O(|\mathbf{L}|)$ time [6, 7, 8]. If this is done rarely, the overall run time of the application that uses the sparse row deletion algorithm will not be affected adversely.

The asymptotic run time of our sparse row deletion algorithm is the same as sparse row addition (or less, because of the binary search), even though sparse row addition requires more numerical work. This is nonoptimal but no worse than sparse row addition, whose run time is optimal.

4. Row modification. It is possible to generalize our row deletion and addition algorithms to handle the case where the kth row and column of $\mathbf{C}$ is neither originally zero (the row addition case) nor set to zero (the row deletion case), but is changed arbitrarily. Any change of this form can be handled as a row deletion followed by a row addition, but the question may remain as to whether or not it can be done faster as a single step. Here, we show that an arbitrary row modification can be efficiently implemented as a row deletion followed by a row addition. If the changes to the row are sparse, however, some work can be saved by combining the two steps.

4.1. Arbitrary row modification. In this section we show that no flops are saved in a single-pass row modification algorithm, as compared to the row deletion + row addition approach, if the change in the kth row and column is arbitrary. This is true whether $\mathbf{C}$ is sparse or dense.

The original matrix $\mathbf{C}$ and the new matrix $\overline{\mathbf{C}}$ are

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \boldsymbol{c}_{12} & \mathbf{C}_{31}^{\mathsf{T}} \\ \boldsymbol{c}_{12}^{\mathsf{T}} & c_{22} & \boldsymbol{c}_{32}^{\mathsf{T}} \\ \mathbf{C}_{31} & \boldsymbol{c}_{32} & \mathbf{C}_{33} \end{bmatrix} \quad \text{and} \quad \overline{\mathbf{C}} = \begin{bmatrix} \mathbf{C}_{11} & \bar{\boldsymbol{c}}_{12} & \mathbf{C}_{31}^{\mathsf{T}} \\ \bar{\boldsymbol{c}}_{12}^{\mathsf{T}} & \bar{c}_{22} & \bar{\boldsymbol{c}}_{32}^{\mathsf{T}} \\ \mathbf{C}_{31} & \bar{\boldsymbol{c}}_{32} & \mathbf{C}_{33} \end{bmatrix}. \tag{4.1}$$

Computing $\bar{\boldsymbol{l}}_{12}$, $\bar{d}_{22}$, and $\bar{\boldsymbol{l}}_{32}$ is the same as the row addition algorithm and takes exactly the same number of flops. The original factorization of the (33)-block is

$$\mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} = \mathbf{C}_{33} - \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^{\mathsf{T}} - \boldsymbol{l}_{32}d_{22}\boldsymbol{l}_{32}^{\mathsf{T}},$$

while the new factorization is

$$\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = \mathbf{C}_{33} - \mathbf{L}_{31}\mathbf{D}_{11}\mathbf{L}_{31}^{\mathsf{T}} - \bar{\boldsymbol{l}}_{32}\bar{d}_{22}\bar{\boldsymbol{l}}_{32}^{\mathsf{T}}.$$

These can be combined into a rank-1 update and rank-1 downdate

$$\overline{\mathbf{L}}_{33}\overline{\mathbf{D}}_{33}\overline{\mathbf{L}}_{33}^{\mathsf{T}} = \mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}} - \boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}, \tag{4.2}$$

where $\boldsymbol{w}_1 = \boldsymbol{l}_{32}\sqrt{d_{22}}$ and $\boldsymbol{w}_2 = \bar{\boldsymbol{l}}_{32}\sqrt{\bar{d}_{22}}$. The multiple rank update/downdate presented in [4] cannot perform a simultaneous update and downdate, but if the sparse downdate (removal of entries that become symbolically zero) is not performed, it would be possible to compute the simultaneous update and downdate (4.2) in a single pass. This may result in some time savings since the data structure for $\mathbf{L}$ is scanned once, not twice. No floating-point work would be saved, however. The total flop count of the row modification algorithm is identical to a row deletion followed by a row addition.

4.2. Sparse row modification. Suppose we modify some, but not all, of the elements of the kth row and column of $\mathbf{C}$. In this case, we can reduce the total amount of floating-point work required to compute the modified factorization $\overline{\mathbf{L}\mathbf{D}\mathbf{L}}^{\mathsf{T}}$, as shown below.

More precisely, consider the case where only a few entries in row and column k of $\mathbf{C}$ are changed. Let

$$\begin{aligned}\Delta\boldsymbol{c}_{12} &= \bar{\boldsymbol{c}}_{12} - \boldsymbol{c}_{12},\\ \Delta c_{22} &= \bar{c}_{22} - c_{22},\\ \Delta\boldsymbol{c}_{32} &= \bar{\boldsymbol{c}}_{32} - \boldsymbol{c}_{32},\end{aligned}$$

and assume that the change in the kth row and column is much sparser than in the original kth row and column of $\mathbf{C}$ (for example, $|\Delta\boldsymbol{c}_{12}| \ll |\bar{\boldsymbol{c}}_{12}|$).

If we consider (2.3) and its analog for the original matrix $\mathbf{C}$, we have

$$\mathbf{L}_{11}\mathbf{D}_{11}\boldsymbol{l}_{12} = \boldsymbol{c}_{12}, \tag{4.3}$$

$$\mathbf{L}_{11}\mathbf{D}_{11}\bar{\boldsymbol{l}}_{12} = \bar{\boldsymbol{c}}_{12}. \tag{4.4}$$

Combining these two equations gives

$$\mathbf{L}_{11}\mathbf{D}_{11}\Delta\boldsymbol{l}_{12} = \Delta\boldsymbol{c}_{12},$$

where $\Delta\boldsymbol{l}_{12} = \bar{\boldsymbol{l}}_{12} - \boldsymbol{l}_{12}$. Since $\Delta\boldsymbol{c}_{12}$ is sparse, the solution of this lower triangular system will be sparse in general. It can be solved in time proportional to the time required to multiply $\mathbf{L}_{11}$ times $\Delta\boldsymbol{l}_{12}$, or

$$O\left(\sum_{(\Delta\boldsymbol{l}_{12})_j \neq 0} |(\mathbf{L}_{11})_{*j}|\right)$$

[9, 10]. We can then compute $\bar{\boldsymbol{l}}_{12} = \boldsymbol{l}_{12} + \Delta\boldsymbol{l}_{12}$. This approach for computing $\bar{\boldsymbol{l}}_{12}$ can be much faster than solving (4.4) directly, which would take

$$O\left(\sum_{(\bar{\boldsymbol{l}}_{12})_j \neq 0} |(\mathbf{L}_{11})_{*j}|\right)$$

time. Computing $\bar{d}_{22}$ can be done in time proportional to $|\Delta\boldsymbol{l}_{12}|$, using the following formula that modifies the dot product computation in Algorithm 1:

$$\bar{d}_{22} = d_{22} + \Delta d_{22} = d_{22} + \Delta c_{22} - \sum_{(\Delta\boldsymbol{l}_{12})_j \neq 0} (\Delta\boldsymbol{l}_{12})_j ((\boldsymbol{l}_{12})_j + (\bar{\boldsymbol{l}}_{12})_j)(\mathbf{D}_{11})_{jj}.$$

Similarly, the kth column of $\overline{\mathbf{L}}$ can be computed as

$$\bar{\boldsymbol{l}}_{32} = \left(\Delta\mathbf{C}_{32} + \boldsymbol{l}_{32} d_{22} - \mathbf{L}_{31}\mathbf{D}_{11}\Delta\boldsymbol{l}_{12}\right)/\bar{d}_{22}.$$

The key component in this computation is the sparse matrix-vector multiplication $\mathbf{L}_{31}\mathbf{D}_{11}\Delta\boldsymbol{l}_{12}$, which takes less time to compute than the corresponding computation $\mathbf{L}_{31}\mathbf{D}_{11}\boldsymbol{l}_{12}$ in step (2) of Algorithm 1.

The remaining work for the rank-1 update/downdate of $\mathbf{L}_{33}\mathbf{D}_{33}\mathbf{L}_{33}^{\mathsf{T}}$ is identical to the arbitrary row modification (4.2). If k is small, it is likely that this update/downdate computation will take much more time than the computation of $\bar{\boldsymbol{l}}_{12}$, d_{22}, and $\bar{\boldsymbol{l}}_{32}$. If k is large, however, a significant reduction in the total amount of work can be obtained by exploiting sparsity in the change in the row and column of $\mathbf{C}$.

5. Row modifications as a rank-2 outer-product. Modifying row and column k of a symmetric matrix $\mathbf{C}$ can be written as a rank-2 modification $\overline{\mathbf{C}} = \mathbf{C} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}} - \boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}$. Suppose $\mathbf{C}$ and $\overline{\mathbf{C}}$ are as given in (4.1). Let

$$\boldsymbol{d} = \begin{bmatrix} \bar{\boldsymbol{c}}_{12} - \boldsymbol{c}_{12} \\ (\bar{c}_{22} - c_{22})/2 \\ \bar{\boldsymbol{c}}_{32} - \boldsymbol{c}_{32} \end{bmatrix}.$$

Let $\boldsymbol{e}_k$ be the kth column of the identity. Then $\overline{\mathbf{C}} = \mathbf{C} + \boldsymbol{d}\boldsymbol{e}_k^{\mathsf{T}} + \boldsymbol{e}_k\boldsymbol{d}^{\mathsf{T}}$. This can be put into the form $\overline{\mathbf{C}} = \mathbf{C} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}} - \boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}$ using the following relationship (note that $\boldsymbol{e}$, below, is an arbitrary column vector, not necessarily $\boldsymbol{e}_k$). Given $\boldsymbol{d}$ and $\boldsymbol{e} \in \mathbb{R}^n$, define

$$\boldsymbol{p} = \frac{\boldsymbol{d}}{\|\boldsymbol{d}\|} + \frac{\boldsymbol{e}}{\|\boldsymbol{e}\|} \quad \text{and} \quad \boldsymbol{q} = \frac{\boldsymbol{d}}{\|\boldsymbol{d}\|} - \frac{\boldsymbol{e}}{\|\boldsymbol{e}\|}.$$

Then we have

$$\boldsymbol{d}\boldsymbol{e}^{\mathsf{T}} + \boldsymbol{e}\boldsymbol{d}^{\mathsf{T}} = \frac{\|\boldsymbol{d}\|\|\boldsymbol{e}\|}{2}\left(\boldsymbol{p}\boldsymbol{p}^{\mathsf{T}} - \boldsymbol{q}\boldsymbol{q}^{\mathsf{T}}\right). \tag{5.1}$$

In our case, $\boldsymbol{e} = \boldsymbol{e}_k$ and $||\boldsymbol{e}|| = 1$. Defining

$$\boldsymbol{w}_1 = \sqrt{\frac{\|\boldsymbol{d}\|}{2}}\left(\frac{\boldsymbol{d}}{\|\boldsymbol{d}\|} + \boldsymbol{e}_k\right) \quad \text{and} \quad \boldsymbol{w}_2 = \sqrt{\frac{\|\boldsymbol{d}\|}{2}}\left(\frac{\boldsymbol{d}}{\|\boldsymbol{d}\|} - \boldsymbol{e}_k\right),$$

it follows from (5.1) that

$$\overline{\mathbf{C}} = \mathbf{C} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}} - \boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}. \tag{5.2}$$

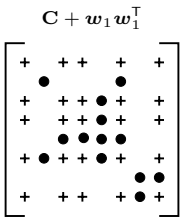

FIG. 5.1. *Modifying* $\mathbf{C}$ *after the first outer product.*

In the dense case, computing (5.2) using a rank-1 update and rank-1 downdate takes $4n^2 + 11n$ flops, independent of k (including the work to compute $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$). If we use Algorithms 1and 2 to make an arbitrary change in the kth row and column of $\mathbf{C}$, the total work is $4n^2 + 8n + 3k^2 - (6nk + 7k)$, which is roughly $4n^2$ when $k = 1$, $n^2 + n$ when $k = n$, and $(7/4)n^2$ when $k = n/2$. Using the rank-2 update/downdate method to modify row and column k of $\mathbf{C}$ can thus take up to four times the work compared to the row addition/deletion presented here.

If the modification requires only the addition or deletion of row and column k, then only Algorithm 1 or 2 needs to be used, but the entire rank-2 update/downdate method presented in this section is still required (about $4n^2$ work, independent of k). Considering only the quadratic terms, Algorithm 1 performs $2n^2 + k^2 - 2nk$ operations, and Algorithm 2 performs $2(n-k)^2$ operations. The row modification methods require much less work, particularly for the row deletion case when $k \approx n$, in which the work drops from $4n^2$ for the rank-2 update/downdate method to nearly no work at all.

In the sparse case, the differences in work and memory usage can be extreme. Both the rank-1 update and downdate could affect the entire matrix $\mathbf{L}$ and could cause catastrophic intermediate fill-in. Consider the row addition case when $k \approx n$ and the new row and column of $\mathbf{C}$ is dense. The factorization of $\overline{\mathbf{C}}$ will still be fairly sparse, since the large submatrix $\mathbf{C}_{11}$ does not change, and remains very sparse. The factor $\mathbf{L}_{11}$ can have as few as $O(n)$ entries. However, after one rank-1 update, the (11)-block of $\mathbf{C} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}}$ is completely dense, since $\boldsymbol{w}_1$ is a dense column vector. After the rank-1 downdate (with $-\boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}$), massive cancellation occurs, and the factorization of $\mathbf{C}_{11}$ is restored to its original sparse nonzero pattern. But the damage has been done, since we require $O(n^2)$ memory to hold the intermediate factorization. This is infeasible in a sparse matrix algorithm. The memory problem could be solved if a single-pass rank-2 update/downdate algorithm were used, but even then, the total work required would be $O(n^2)$, which is much more than the $O(|\overline{\mathbf{L}}|)$ time required for Algorithm 1 in this case. The same problem occurs if the downdate with $-\boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}$ is applied first.

Figure 5.1 illustrates the change in $\mathbf{C}$ after the first rank-1 update, if the row modification of Figure 1.2 is performed as the rank-2 modification $\overline{\mathbf{C}} = \mathbf{C} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}} - \boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}$. The vectors $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ have the same nonzero pattern as the change in row k of $\mathbf{C}$. The graph of the matrix $\boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}}$ is a single clique; its entries are shown as pluses in Figure 5.1. If column 8 of this matrix were modified to become completely dense, then Figure 5.1 would be a full matrix of pluses.

6. Modifying a lower triangular system. We now consider a related operation that can be performed efficiently at the same time that we modify the Cholesky

factorization. Suppose we have a linear system $\mathbf{C}\boldsymbol{x} = \boldsymbol{b}$ and the system is modified, either by changing a row and column of $\mathbf{C}$ as discussed above, or due to a low-rank change $\overline{\mathbf{C}} = \mathbf{C} \pm \mathbf{W}\mathbf{W}^{\mathsf{T}}$ as discussed in [3, 4]. After the factorization is modified, the new factorization $\overline{\mathbf{L}}\overline{\mathbf{D}}\overline{\mathbf{L}}^{\mathsf{T}} = \overline{\mathbf{C}}$ will normally be used to solve a modified linear system $\overline{\mathbf{C}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$. The complete solution $\overline{\boldsymbol{x}}$ will likely be different in every component because of the backsolve, but a significant reduction of work can be obtained in the forward solve of the lower triangular system $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$. We thus focus only on this lower triangular system, not the complete system.

First, consider the simpler case of a low-rank change of a dense matrix. As shown below, it takes double the work to modify the solution instead of computing it from scratch, but the dense matrix method can be used for submatrices in the sparse case, resulting in a significant reduction in work. Suppose we have the system $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$, including its solution $\boldsymbol{x}$, and the new linear system is $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$. Combining these two equations gives

$$\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}} - \boldsymbol{b} + \mathbf{L}\boldsymbol{x} = \Delta\boldsymbol{b} + \mathbf{L}\boldsymbol{x},$$

where $\Delta\boldsymbol{b} = \overline{\boldsymbol{b}} - \boldsymbol{b}$. Suppose we are given $\mathbf{L}$, $\boldsymbol{x}$, and $\Delta\boldsymbol{b}$. The matrix $\overline{\mathbf{L}}$ is computed column-by-column by either the low-rank update/downdate algorithm in [3, 4] or the row and column modification algorithm discussed in this paper. We can combine the modification of $\mathbf{L}$ with the computation of $\overline{\boldsymbol{x}}$, as shown in Algorithm 3.

ALGORITHM 3 (DENSE MODIFICATION OF $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ TO SOLVE $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$).

$\overline{\boldsymbol{x}} = \Delta\boldsymbol{b}$
for $j = 1$ to n **do**
  $\overline{\boldsymbol{x}} = \overline{\boldsymbol{x}} + (\mathbf{L})_{*j} x_j$
  compute the new column $(\overline{\mathbf{L}})_{*j}$
  $(\overline{\boldsymbol{x}})_{j+1\ldots n} = (\overline{\boldsymbol{x}})_{j+1\ldots n} - (\overline{\mathbf{L}})_{j+1\ldots n,j}\overline{x}_j$
end for
end Algorithm 3

The total work to modify the solution to $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$ is roughly $2n^2$, as compared to n^2 work to solve $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$ from scratch. One would never use this method if $\mathbf{L}$ and $\boldsymbol{b}$ are dense.

Now consider the sparse case. Suppose we have a low-rank sparse update of $\mathbf{L}$. The columns that change in $\overline{\mathbf{L}}$ correspond to a single path $\overline{\mathcal{P}}$ in the elimination tree for a rank-1 update. Every entry in these specific columns is modified. For a rank-r update, where $r > 1$, the columns that change correspond to a set of paths in the elimination tree, which we also will refer to as $\overline{\mathcal{P}}$ in this more general case. In both cases, the nonzero pattern of each column j in the path $\overline{\mathcal{P}}$ is a subset of the path [20]. We can thus partition the matrices $\mathbf{L}$ and $\overline{\mathbf{L}}$ into two parts, according to the set $\overline{\mathcal{P}}$. The original system is

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{bmatrix},$$

where the matrix $\mathbf{L}_{22}$ consists of all the rows and columns corresponding to nodes in the path $\overline{\mathcal{P}}$. If the changes in the right-hand side $\overline{\boldsymbol{b}}$ are also constrained to the set $\overline{\mathcal{P}}$, the new linear system is

$$\begin{bmatrix} \mathbf{L}_{11} & \mathbf{0} \\ \mathbf{L}_{12} & \overline{\mathbf{L}}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \overline{\boldsymbol{x}}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \overline{\boldsymbol{b}}_2 \end{bmatrix}.$$

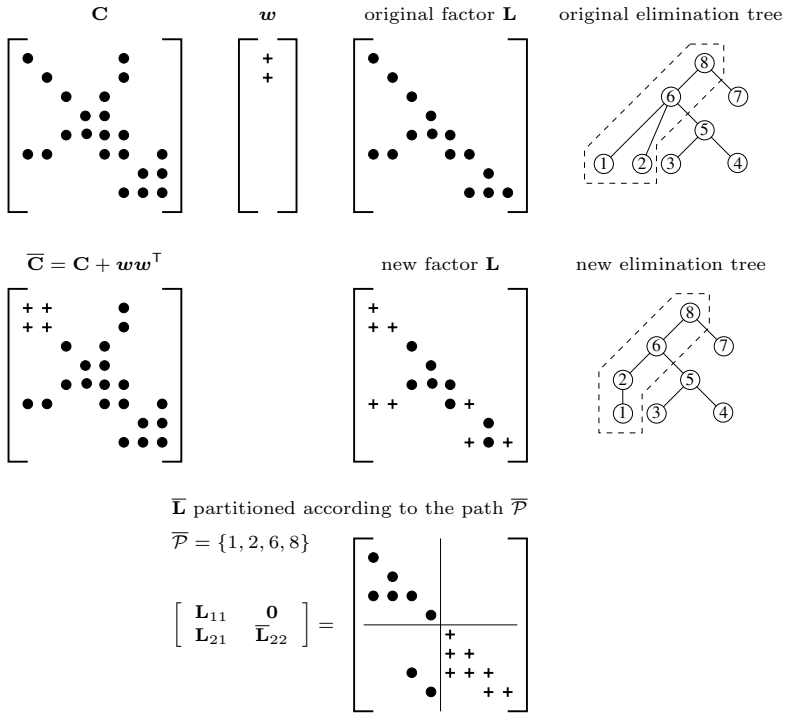

FIG. 6.1. *Modifying* $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ *after a rank-*1 *update.*

Note that $\mathbf{L}_{11}$, $\mathbf{L}_{12}$, and $\boldsymbol{b}_1$ do not change, and thus $\boldsymbol{x}_1$ does not change. If the change in the right-hand side is arbitrary (not constrained to $\overline{\mathcal{P}}$, for example), then $\boldsymbol{x}_1$ changes and no work is saved over solving $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$ from scratch.

We have made a sparse change to both the matrix $\mathbf{L}$ and the right-hand side. The solution to the subsystem $\mathbf{L}_{11}\boldsymbol{x}_1 = \boldsymbol{b}_1$ does not change. We have

$$\mathbf{L}_{21}\boldsymbol{x}_1 + \mathbf{L}_{22}\boldsymbol{x}_2 = \boldsymbol{b}_2,$$
$$\mathbf{L}_{21}\boldsymbol{x}_1 + \overline{\mathbf{L}}_{22}\overline{\boldsymbol{x}}_2 = \overline{\boldsymbol{b}}_2.$$

We can apply Algorithm 3 to the subsystem

$$\overline{\mathbf{L}}_{22}\overline{\boldsymbol{x}}_2 = \Delta\boldsymbol{b}_2 + \mathbf{L}_{22}\boldsymbol{x}_{22},$$

where $\Delta\boldsymbol{b}_2 = \overline{\boldsymbol{b}}_2 - \boldsymbol{b}_2$, to obtain the new solution $\overline{\boldsymbol{x}}_2$. Algorithm 3 takes $4|\overline{\mathbf{L}}_{22}| + O(\overline{\mathcal{P}})$ flops to compute $\overline{\boldsymbol{x}}_2$. An additional $4r|\overline{\mathbf{L}}_{22}| + O(\overline{\mathcal{P}})$ flops are required to update $\overline{\mathbf{L}}_{22}$. Solving $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$ after $\overline{\mathbf{L}}$ is updated takes $2|\overline{\mathbf{L}}|$ flops. If the path $\overline{\mathcal{P}}$ is short, making a sparse modification to the old solution $\boldsymbol{x}$ to obtain the new solution $\overline{\boldsymbol{x}}$ during the update of $\overline{\mathbf{L}}$ takes much less work than solving the new linear system after $\overline{\mathbf{L}}$ is updated.

Figure 6.1 shows a matrix $\mathbf{C}$, its factorization, and its elimination tree after a rank-1 update, using just the first column of $\mathbf{W}$ from the example shown in Figure 1.1. As in our other figures, a plus denotes an entry that changes, or an entry of $\boldsymbol{w}$. The subtree of the original elimination tree consisting of nodes $\{1, 2, 6, 8\}$ becomes a single path $\overline{\mathcal{P}} = \{1, 2, 6, 8\}$ in the new elimination tree. The rows and columns of $\overline{\mathbf{L}}$ can be

partitioned according to the path $\overline{\mathcal{P}}$, as shown in Figure 6.1. We do not perform this permutation, of course, but only illustrate it here. The algorithm that updates $\mathbf{L}$ and $\boldsymbol{x}$ accesses only columns $\{1, 2, 6, 8\}$ of $\mathbf{L}$ (the submatrix $\mathbf{L}_{22}$) and the same rows of $\boldsymbol{x}$ and $\boldsymbol{b}$.

Finally, consider the case discussed in this paper of modifying row and column k of $\mathbf{C}$. If we add row k, the original lower triangular system is

$$\begin{bmatrix} \mathbf{L}_{11} & & \\ \mathbf{0}^{\mathsf{T}} & 1 & \\ \mathbf{L}_{31} & \mathbf{0} & \mathbf{L}_{33} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ x_2 \\ \boldsymbol{x}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ b_2 \\ \boldsymbol{b}_3 \end{bmatrix}. \tag{6.1}$$

The new lower triangular system is

$$\begin{bmatrix} \mathbf{L}_{11} & & \\ \bar{\boldsymbol{l}}_{12}^{\mathsf{T}} & 1 & \\ \mathbf{L}_{31} & \bar{\boldsymbol{l}}_{32} & \overline{\mathbf{L}}_{33} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \overline{x}_2 \\ \overline{\boldsymbol{x}}_3 \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_1 \\ \bar{b}_2 \\ \overline{\boldsymbol{b}}_3 \end{bmatrix}, \tag{6.2}$$

where we assume $\boldsymbol{b}_1$ does not change, and the entries that change in $\boldsymbol{b}_3$ are a subset of the path $\overline{\mathcal{P}}$. Let $\Delta\boldsymbol{b}_3 = \overline{\boldsymbol{b}}_3 - \boldsymbol{b}_3$. The term $\overline{x}_2$ can be computed as

$$\overline{x}_2 = \bar{b}_2 - \bar{\boldsymbol{l}}_{12}^{\mathsf{T}} \boldsymbol{x}_1.$$

The (33)-blocks of (6.1) and (6.2) give the equations

$$\begin{aligned} \mathbf{L}_{33}\boldsymbol{x}_3 &= \boldsymbol{b}_3 - \mathbf{L}_{31}\boldsymbol{x}_1, \\ \overline{\mathbf{L}}_{33}\overline{\boldsymbol{x}}_3 &= (\boldsymbol{b}_3 - \mathbf{L}_{31}\boldsymbol{x}_1) + (\Delta\boldsymbol{b}_3 - \bar{\boldsymbol{l}}_{32}\overline{x}_2). \end{aligned} \tag{6.3}$$

The change in the right-hand side of this system is $\Delta\boldsymbol{b}_3 - \bar{\boldsymbol{l}}_{32}\overline{\boldsymbol{x}}_2$. Since the nonzero pattern of $\bar{\boldsymbol{l}}_{32}$ is a subset of $\overline{\mathcal{P}}$, this computation fits the same requirements for the sparse change in $\boldsymbol{x}$ due to a sparse low-rank change in $\mathbf{L}$ and $\boldsymbol{b}$, discussed above. The row deletion case is analogous.

The number of flops in Algorithm 3 to modify the solution to $\overline{\mathbf{L}}\overline{\boldsymbol{x}} = \overline{\boldsymbol{b}}$ is roughly the same as a rank-1 update to compute the modified $\overline{\mathbf{L}}$. However, the run time is not doubled compared to a stand-alone rank-1 update. At step j, for each j in the path $\overline{\mathcal{P}}$, the rank-1 update must read column j of $\mathbf{L}$, modify it, and then write the jth column of $\overline{\mathbf{L}}$ back into memory. No extra memory traffic to access $\overline{\mathbf{L}}$ is required to compute the solution to the lower triangular system using (6.3). With current technology, memory traffic can consume more time than flops. Thus, when modifying the kth row and column of $\mathbf{C}$, or performing a rank-r update/downdate to $\mathbf{C}$, we can update the solution to the lower triangular system at almost no extra cost. In our target application [5], solving the linear system $\mathbf{C}\boldsymbol{x} = \boldsymbol{b}$, given its $\mathbf{LDL}^{\mathsf{T}}$ factorization, can often be the dominant step. The method presented here can cut this time almost in half since the forward solve time is virtually eliminated, leaving us with the time for the upper triangular backsolve.

7. Experimental results. To illustrate the performance of the algorithms developed in this paper in a specific application, Table 7.1 gives the flops associated with the solution of the four largest problems in the Netlib linear programming test set using the LPDASA [5]. The problems passed to the solver were first simplified using the ILOG CPLEX [2] version 7.0 presolve routine. An LP presolver [1] preprocesses the problem by removing redundant constraints and variables whose values can be

TABLE 7.1
Experimental results.

Problem	Performance	Forward solve	Column updates	Row deletions
DFL001	number:		2629	87
n: 3881	avg. mod. rank		1.2	1
	avg. mod. flops		2.43×10^6	1.64×10^6
	avg. solve flops	1.46×10^6	2.03×10^6	1.64×10^6
PDS20	number:		1736	65
n: 10214	avg. mod. rank		3.5	1
	avg. mod. flops		3.37×10^6	2.13×10^6
	avg. solve flops	1.11×10^6	1.21×10^6	2.12×10^6
KEN18	number:		2799	23
n: 39856	avg. mod. rank		15.6	1
	avg. mod. flops		61.7×10^3	2242
	avg. solve flops	195.8×10^3	7155	2075
OSA60	number:		71	277
n: 10209	avg. mod. rank		58.9	1
	avg. mod. flops		4415	62
	avg. solve flops	11.0×10^3	270	37

easily determined. Hence, the number of rows n of the problems listed in the first column of Table 7.1 is much smaller than the number of rows in the original Netlib problems.

The third column (labeled "Forward solve") gives the average number of flops that would have been required if forward solves were done in a conventional way, by a forward elimination process. This number is simply twice the average number of nonzeros in $\mathbf{L}$. As the LP is solved, the sparsity of $\mathbf{L}$ changes slightly due to changes in the current basis. Hence, we give the average flops needed for a conventional forward solve, which can be compared with the average flops in modifying the solution using the technique developed in this paper. The problems are sorted according to this column.

The fourth column of the table, entitled "Column updates," lists the number of rank-r column updates performed (of the form $\mathbf{C} + \mathbf{W}\mathbf{W}^{\mathsf{T}}$), the average rank r of those updates, the flops required to modify $\mathbf{L}$, and the flops required to modify the solution to the forward solve when $\mathbf{L}$ changes as a result of a column update. Since we have developed [4] a multiple-rank approach for performing column updates, the average ranks listed are all greater than 1. They are near 1 for the densest problem DFL001 and near 60 for the sparsest problem OSA60. Recall that the column update requires about $4r$ flops per entry in $\mathbf{L}$ that change. Modifying the forward solve takes 4 flops per entry in $\mathbf{L}$ that change. The average flops associated with the modification of $\mathbf{L}$ is thus always greater than the flops associated with the forward solve update, especially for multiple rank column updates. The conventional forward solve requires 2 flops for each nonzero entry in $\mathbf{L}$, whether they change or not. In the worst case, when all of the entries in $\mathbf{L}$ change, modifying the forward solve takes no more than twice the work of the conventional forward solve, and a column update takes no more than $2r$ times the work of a conventional forward solve.

The last column of the table, entitled "Row deletions," lists the number of deletions of a row (and column) from $\mathbf{C}$, the average number of flops required to modify $\mathbf{L}$ after deleting a row from $\mathbf{C}$, and the average number of flops required to modify the solution to the forward solve when $\mathbf{L}$ changes as a result of a row deletion. Since

the row deletion algorithm developed in this paper is a single-rank process, the ranks listed are all 1.

When the matrix $\mathbf{L}$ is very sparse (such as OSA60), both the column update and row deletion methods modify only a small portion of the matrix. The elimination tree tends to be short and wide, with short paths from any node to the root. Thus, for very sparse problems the conventional forward solve (which accesses all of $\mathbf{L}$) takes much more work than either the column update or the row deletion. For the OSA60 matrix, the conventional forward solve takes about 40 times the work compared to modifying the forward solve during a column update.

For a matrix $\mathbf{L}$ that is fairly dense (such as DFL001), the elimination tree tends to be tall and thin, and the column updates or row deletions modify most entries in $\mathbf{L}$. In this case, the work required to modify the forward solve is more on average than that of a full forward solve, but it is never more than twice the work. A combined update of the factorization and forward solve cuts the memory traffic at least in half compared with an update of $\mathbf{L}$ followed by a conventional forward solve. The update accesses only the parts of $\mathbf{L}$ that change, whereas the conventional forward solve accesses all of $\mathbf{L}$. The performance of most modern computers is substantially affected by the amount of memory transfers between cache and main memory, so cutting memory traffic at least in half at the cost of at most doubling the flop count will normally lead to an overall improvement in performance, even when the matrix is fairly dense.

8. Summary. We have presented a method for modifying the sparse Cholesky factorization of a symmetric positive definite matrix $\mathbf{C}$ after a row and column of $\mathbf{C}$ have been modified. One algorithm, the sparse row addition, is optimal. The corresponding row deletion algorithm is not optimal but takes no more time than the row addition. Although changing a row and column of $\mathbf{C}$ can be cast as rank-2 change of the form $\mathbf{C} + \boldsymbol{w}_1\boldsymbol{w}_1^{\mathsf{T}} - \boldsymbol{w}_2\boldsymbol{w}_2^{\mathsf{T}}$, the latter is impractical in a sparse context. We have shown how to modify the solution to a lower triangular system $\mathbf{L}\boldsymbol{x} = \boldsymbol{b}$ when the matrix $\mathbf{L}$ changes as a result of either the row addition/deletion operation discussed here or an update/downdate of the form $\mathbf{C} \pm \mathbf{W}\mathbf{W}^{\mathsf{T}}$ described in our previous papers [3, 4]. By postponing the symbolic downdate, the memory usage has been reduced by 25% (assuming 8-byte floating-point values and 4-byte integers), compared with the column update/downdate methods described in [3, 4], which also store the multiplicities. Together, the row addition/deletion algorithms and the column update/downdate algorithms form a useful suite of tools for modifying a sparse Cholesky factorization and for solving a sparse system of linear equations. Using these algorithms, the linear programming solver LPDASA is able to achieve an overall performance that rivals, and sometimes exceeds, the performance of current commercially available solvers [5].

REFERENCES

[1] E. D. Andersen and K. D. Andersen, *Presolving in linear programming*, Math. Program., 71 (1995), pp. 221–245.

[2] R. E. Bixby, *Progress in linear programming*, ORSA J. Comput., 6 (1994), pp. 15–22.

[3] T. A. Davis and W. W. Hager, *Modifying a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 606–627.

[4] T. A. Davis and W. W. Hager, *Multiple-rank modifications of a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 997–1013.

[5] T. A. Davis and W. W. Hager, *A sparse proximal implementation of the LP dual active set algorithm*, Math. Program., submitted; also available online from http://www.math.ufl.edu/∼hager.

[6] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale sparse matrix package* I: *The symmetric codes*, Internat. J. Numer. Methods Engrg., 18 (1982), pp. 1145–1151.
[7] A. George and J. W. H. Liu, *An optimal algorithm for symbolic factorization of symmetric matrices*, SIAM J. Comput., 9 (1980), pp. 583–593.
[8] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, NJ, 1981.
[9] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 62–79.
[10] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 862–874.
[11] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders, *Methods for modifying matrix factorizations*, Math. Comp., 28 (1974), pp. 505–535.
[12] W. W. Hager, *Updating the inverse of a matrix*, SIAM Rev., 31 (1989), pp. 221–239.
[13] W. W. Hager, *The dual active set algorithm*, in Advances in Optimization and Parallel Computing, P. M. Pardalos, ed., North–Holland, Amsterdam, 1992, pp. 137–142.
[14] W. W. Hager, *The LP dual active set algorithm*, in High Performance Algorithms and Software in Nonlinear Optimization, R. D. Leone, A. Murli, P. M. Pardalos, and G. Toraldo, eds., Kluwer Academic Publishers, Norwell, MA, 1998, pp. 243–254.
[15] W. W. Hager, *The dual active set algorithm and its application to linear programming*, Comput. Optim. Appl., 21 (2002), pp. 263–275.
[16] W. W. Hager, *The dual active set algorithm and the iterative solution of linear programs*, in Novel Approaches to Hard Discrete Optimization, P. M. Pardalos and H. Wolkowicz, eds., Kluwer Academic Publishers, Norwell, MA, 2003, pp. 95–107.
[17] W. W. Hager and D. W. Hearn, *Application of the dual active set algorithm to quadratic network optimization*, Comput. Optim. Appl., 1 (1993), pp. 349–373.
[18] J. W. H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. Math. Software, 12 (1986), pp. 127–148.
[19] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.
[20] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276.