Preconditioning of linear least squares by robust incomplete factorization for implicitly held normal equations

Jennifer Scott and Miroslav Tůma

Published version information:

Citation: Scott J & Tuma M. "Preconditioning of linear least squares by robust incomplete factorization for implicitly held normal equations." SIAM Journal on Scientific Computing, vol. 38, no. 6 (2016): C603-C623.

doi: 10.1137/16M105890X

First Published in SIAM Journal on Scientific Computing in 2016, published by the Society for Industrial and Applied Mathematics (SIAM).

SIAM J. SCI. COMPUT.
Vol. 38, No. 6, pp. C603–C623

PRECONDITIONING OF LINEAR LEAST SQUARES BY ROBUST INCOMPLETE FACTORIZATION FOR IMPLICITLY HELD NORMAL EQUATIONS*

JENNIFER SCOTT† AND MIROSLAV TŮMA‡

Abstract. The efficient solution of the normal equations corresponding to a large sparse linear least squares problem can be extremely challenging. Robust incomplete factorization (RIF) preconditioners represent one approach that has the important feature of computing an incomplete LL^T factorization of the normal equations matrix without having to form the normal matrix itself. The right-looking implementation of Benzi and Tůma has been used in a number of studies but experience has shown that in some cases it can be computationally slow and its memory requirements are not known a priori. Here a new left-looking variant is presented that employs a symbolic preprocessing step to replace the potentially expensive searching through entries of the normal matrix. This involves a directed acyclic graph (DAG) that is computed as the computation proceeds. An inexpensive but effective pruning algorithm is proposed to limit the number of edges in the DAG. Problems arising from practical applications are used to compare the performance of the right-looking approach with a left-looking implementation that computes the normal matrix explicitly and our new implicit DAG-based left-looking variant.

Key words. sparse matrices, sparse linear systems, indefinite symmetric systems, iterative solvers, preconditioning, incomplete factorizations

AMS subject classifications. Primary, 65F08, 65F20, 65F50; Secondary, 15A06, 15A23

DOI. 10.1137/16M105890X

1. Introduction.

Linear least squares (LS) problems arise in a wide variety of practical applications. Let us consider the algebraic problem of linear LS in the following form:

$$\min_x \|b - Ax\|_2, \tag{1.1}$$

where $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ is a large sparse matrix with full column rank and $b \in \mathbb{R}^m$ is given. Solving (1.1) is mathematically equivalent to solving the $n\times n$ *normal equations*

$$Cx = A^T b, \qquad C = A^T A, \tag{1.2}$$

where, since A has full column rank, the *normal matrix* C is symmetric positive definite. To solve very large LS problems, an iterative method may be the method of choice because it can require much less storage and fewer operations than direct counterparts. However, iterative methods do not offer the same level of reliability and their successful application often needs a good preconditioner to achieve acceptable convergence rates (or, indeed, to obtain convergence at all).

*Submitted to the journal's Software and High-Performance Computing section January 28, 2016; accepted for publication (in revised form) August 23, 2016; published electronically November 1, 2016.

http://www.siam.org/journals/sisc/38-6/M105890.html

Funding: The first author's work was supported by EPSRC grants EP/I013067/1 and EP/M025179/1. The second author's work was supported by project 13-06684S of the Grant Agency of the Czech Republic and by ERC-CZ project MORE LL1202 financed by MŠMT of the Czech Republic.

†Scientific Computing Department, Rutherford Appleton Laboratory, Harwell Campus, Oxfordshire, OX11 0QX, UK (jennifer.scott@stfc.ac.uk).

‡Department of Numerical Mathematics, Faculty of Mathematics and Physics, Charles University, Sokolovská 83, 186 75 Prague, Czech Republic (mirektuma@karlin.mff.cuni.cz).

In recent years, a number of techniques for preconditioning LS problems have been proposed. In particular, significant attention has been devoted to the development of algorithms based on incomplete orthogonal factorizations of A [30, 38, 40, 43]. Most recently, there is the multilevel incomplete QR (MIQR) factorization of Li and Saad [33]. An alternative is the LU-based strategy that was first introduced as a direct method in 1961 in [32]; see its further development as the Peters–Wilkinson method [39] (and later in [11, 14]). For large-scale LS problems, methods based on randomized algorithms have been proposed, for example, in [3, 35].

While these and other approaches are very useful in some circumstances, here we focus on the most traditional approach that is based on the normal equations and an incomplete factorization of the symmetric positive-definite matrix C. A direct variant of this approach dates back to 1924 [4]. One problem connected to the normal equations is that they may not be very sparse, for example, if A contains some dense (or close to dense) rows. A possible way to overcome this is to treat the dense rows separately. An alternative approach is to avoid explicitly forming C (that is, to work only with A and A^T) and to compute its factorization implicitly. Working with C implicitly is also important for very large problems for which computing C may be too costly (in terms of both time and memory). Moreover, forming the normal equations may lead to severe loss of information in highly ill-conditioned cases. The recent limited memory incomplete Cholesky factorization code `HSL_MI35` of Scott and Tůma [29, 41, 42] is designed for normal equations. It offers an option to input either the matrix C or A. In the latter case, a single column of C is computed at each stage of the incomplete factorization process, thus avoiding the need to store C explicitly but not the work needed to form the product A^TA [10]. Recent results by Gould and Scott [26, 27] illustrate that `HSL_MI35` can perform well on a range of problems but that constructing the incomplete factorization can be expensive when A has dense rows.

An implicit factorization scheme that uses a Schur complement-based approach is the robust incomplete factorization (known as the RIF algorithm) of Benzi and Tůma [9]. RIF is based on C-orthogonalization and works entirely with A and A^T and can be derived as the "dual" of the SAINV (stabilized approximate inverse) preconditioner [9]. The preconditioner is guaranteed to be positive definite and, in exact arithmetic, the incomplete factorization process is breakdown free. A right-looking implementation of RIF for LS problems is available as part of the SPARSLAB software collection of Benzi and Tůma (see http://www.karlin.mff.cuni.cz/~mirektuma) and has been used in a number of studies on LS preconditioners (including [2, 36]). Important weaknesses of this code are that it can be computationally slow (see the results of [26]) and the amount of memory needed is not known a priori (although for the experiments in [9] the total storage is estimated to be approximately 25% more than the storage needed for the final incomplete factor but this itself is not known and will depend on the dropping strategy). The aim of this paper is to present a new potentially more computationally efficient left-looking RIF algorithm that avoids the need to compute *any* entries of C. A key contribution is to propose and implement the use of a symbolic step to replace the searching through C that is needed in previous implementations. Symbolic preprocessing is a standard tool in sparse direct methods for linear systems (see, for example, the early symbolic decomposition for symmetric and positive-definite matrices in [20], symbolic evaluations in LU decomposition [24], and the overview of the theory of nonsymmetric elimination trees given in [19]). The proposed symbolic step involves a search using a directed acyclic graph (DAG) that we construct as the computation proceeds. When used for RIF applied to LS problems, it has the potentially attractive property of offering the possibility of replacing the

update step in the left-looking algorithm by a parallelizable sparse saxpy operation, although the design and development of parallel implementations lie beyond the scope of the current paper.

The rest of the paper is organized as follows. Section 2 recalls both the right- and left-looking RIF algorithms and summarizes existing strategies for exploiting sparsity in the left-looking approach. Then, in section 3, our new symbolic preprocessing is introduced. The use of RIF for solving LS problems is described in section 4. Numerical experiments in section 5 demonstrate the efficiency and robustness of the new approach using a range of problems from real-world linear systems and LS applications. Finally, section 6 presents some concluding remarks.

Notation. We end this section by introducing the notation that is employed in the rest of the paper. We let the number of nonzero entries of a matrix A be $nz(A)$ and we let these entries be $a_{i,j}$. Furthermore, we denote the jth column of A by a_j, and e_j denotes the jth unit basis vector. For any column a_j, we define its structure (sparsity pattern) to be the set $Struct(a_j) = \{i \mid a_{i,j} \neq 0\}$. Section notation is used to denote part of the matrix so that $A_k \equiv A_{1:k,1:k}$ is the leading submatrix of order k and $A_{1:k,j}$ is the first k entries of column j.

For a symmetric positive-definite matrix C, we define the C-inner product to be

$$\langle x, y\rangle_C = y^T C x \quad \forall\, x, y \in \mathbb{R}^n \tag{1.3}$$

and associated C-norm

$$\|x\|_C = \sqrt{x^T C x}.$$

In the LS case,

$$\langle x, y\rangle_C = (y^T A^T)(Ax) \quad \forall\, x, y \in \mathbb{R}^n.$$

We also recall some standard notation related to sparse matrices and their graphs; see, e.g., [23]. We define $G(A) = (V, E)$ to be the directed graph of the (nonsymmetric) matrix $A \in \mathbb{R}^{n\times n}$ with nonzero diagonal entries as follows: the vertex set is $V = \{1, \ldots, n\}$ and for $1 \leq i \neq j \leq n$ there is an edge $(i, j) \in E$ from i to j if and only if $a_{i,j} \neq 0$. There is a directed path from vertex $i \in V$ to vertex $k \in V$ if there exists a sequence of vertices $i = i_0, i_1, i_2, ..., i_{k-1}, i_k = k$ belonging to V such that each edge (i_{j-1}, i_j) is in E. This path is denoted by $i \Rightarrow k$ and k is said to be reachable from i. The set of vertices that are reachable from i is denoted by $Reach(i)$.

We let $L = \{l_{i,j}\}$ and $Z = \{z_{i,j}\}$ denote a lower and an upper triangular matrix, respectively. A nonzero off-diagonal entry $l_{i,j}$ must have $i > j$, so any edge in the directed graph of L must satisfy $i > j$. Thus the directed graph of a triangular matrix has no loops and belongs to the class of DAGs. We denote the DAG of L by $DAG(L)$.

In the following, we assume some basic knowledge of the concept of an elimination tree and its role in sparse factorizations; this is described, for example, in the survey paper by Liu [34].

2. Robust incomplete factorization and symbolic decomposition.

We start by considering general sparse symmetric positive-definite matrices that are not necessarily normal matrices.

2.1. Left- and right-looking approaches.

Consider the factorization of a general sparse symmetric positive-definite matrix C into the product of two triangular factors of the form

$$C = LL^T. \tag{2.1}$$

Algorithm 1. Left-looking Gram–Schmidt process (classical CGS, modified MGS, and mixed AINV variants).

Input: Symmetric positive-definite matrix $C \in R^{n\times n}$.
Output: Factors Z and L satisfying (2.2).

1. **for** $k = 1 : n$ **do**
2. Set $z_k^{(0)} = e_k$
3. **for** $j = 1 : k-1$ **do**
4. **if** MGS
5. Set $l_{k,j} = \langle z_k^{(j-1)}, z_j\rangle_C$
6. **else if** CGS
7. Set $l_{k,j} = \langle z_k^{(0)}, z_j\rangle_C$
8. **else if** AINV
9. Set $l_{k,j} = \langle z_k^{(j-1)}, e_j\rangle_C$
10. **end if**
11. Set $z_k^{(j)} = z_k^{(j-1)} - l_{k,j} z_j$
12. **end do**
13. Set $l_{k,k} = \|z_k^{(k-1)}\|_C$
14. Set $z_k = z_k^{(k-1)}/l_{k,k}$
15. **end do**

This factorization is unique (up to the sign of the diagonal entries of L) but there are a number of approaches to computing it. For example, we can consider different computational variants of the Cholesky decomposition or the Gram–Schmidt process with the C-inner product. While the former is well-known, our focus is on the latter and, in particular, the RIF algorithm [8, 9]. Given n linearly independent vectors, the Gram–Schmidt process builds a C-orthogonal set of vectors $z_1, z_2, \ldots, z_n$. This can be written as

$$Z^T C Z = I, \qquad I = L^T Z, \tag{2.2}$$

where $Z = [z_1, z_2, \ldots, z_n]$ is upper triangular with positive diagonal entries. In exact arithmetic, L^T is the transposed Cholesky factor of C and Z is its inverse. The left-looking Gram–Schmidt process is outlined in Algorithm 1; the right-looking process is outlined in Algorithm 2. Three computational options are included that correspond to the classical, modified, and mixed variants (denoted by CGS, MGS, and AINV, respectively); for details, see [31]. The relationship between L and Z can be found, for example, in the 1952 seminal paper on the conjugate gradient (CG) method by Hestenes and Stiefel [28].

For large sparse matrices, we need to consider incomplete (approximate) factorizations. Here and elsewhere, we let $\tilde{L} = \{\tilde{l}_{i,j}\}$ and $\tilde{Z} = \{\tilde{z}_{i,j}\}$ denote incomplete factors such that

$$\tilde{Z}^T C \tilde{Z} \approx I, \qquad I \approx \tilde{L}^T \tilde{Z}. \tag{2.3}$$

These incomplete factors may be used as preconditioners for the CG method. Two different types of preconditioner can be obtained by carrying out the C-orthogonalization process incompletely. The first approach drops small entries from the computed vectors as the C-orthogonalization proceeds, that is, after line 11 of Algorithm 1 entries that are smaller in absolute value than some chosen drop tolerance are discarded.

Algorithm 2. Right-looking Gram–Schmidt process (classical CGS, modified MGS, and mixed AINV variants).

Input: Symmetric positive-definite matrix $C \in R^{n\times n}$.
Output: Factors Z and L satisfying (2.2).

1. **for** $k = 1 : n$ **do**
2. Set $z_k^{(0)} = e_k$
3. **end do**
4. **for** $k = 1 : n$ **do**
5. Set $l_{k,k} = \|z_k^{(k-1)}\|_C$
6. Set $z_k = z_k^{(k-1)}/l_{k,k}$
7. **for** $j = k + 1 : n$ **do**
8. **if** MGS
9. Set $l_{k,j} = \langle z_j^{(k-1)}, z_k\rangle_C$
10. **else if** CGS
11. Set $l_{k,j} = \langle z_j^{(0)}, z_k\rangle_C$
12. **else if** AINV
13. Set $l_{k,j} = \langle z_j^{(k-1)}, e_k\rangle_C/l_{k,k}$
14. **end if**
15. Set $z_j^{(j)} = z_j^{(j-1)} - l_{k,j}z_k$
16. **end do**
17. **end do**

Alternatively, a relative drop tolerance can be used. Whatever dropping strategy is used, the result is an incomplete inverse factorization of the form

$$C^{-1} \approx \tilde{Z}\tilde{Z}^T.$$

This is a factored sparse approximate inverse and is known as the SAINV preconditioner. The diagonal entries are positive and so the preconditioner is guaranteed to be positive definite.

The second approach (the RIF preconditioner) is obtained by saving the multipliers $\tilde{l}_{k,j}$. Again, those that are smaller than a chosen drop tolerance can be discarded after they are computed in the inner loop of Algorithm 1. In some special cases (for example, banded C) it is possible to discard some computed and sparsified columns $\tilde{z}_k$ of $\tilde{Z}$ before the end of the factorization. However, this involves both more complicated data structures and more complex implementation details and since an objective in this paper is to simplify these, we do not seek to exploit particular structures. The RIF approach computes an incomplete Cholesky factorization

$$C \approx \tilde{L}\tilde{L}^T.$$

Again, the preconditioner is guaranteed to be positive definite. Benzi and Tůma [9] report that, for LS problems, the RIF preconditioner is generally more effective at reducing the number of CG iterations than the SAINV preconditioner and thus it is the one we consider.

2.2. Implementing left- and right-looking approaches. We now consider the basic data structures and techniques used to implement the left- and right-looking approaches. The left-looking algorithm (Algorithm 1) computes the (sparse) columns

of Z one by one. This requires knowledge of which indices j give nonzero C-inner products $l_{k,j}$; this is discussed in sections 2.3 and 3. The columnwise computation allows an approximation $\tilde{Z}$ to be computed by limiting the number of nonzeros that are retained in each column or by discarding entries with respect to a drop tolerance. To compute the $l_{k,j}$, C is accessed by columns. Note that the AINV variant requires a single column at a time. The actual computation of $l_{k,j}$ is straightforward and is based on standard sparse matrix-vector products and sparse dot products. These multipliers form the RIF factor stored by rows.

For the right-looking algorithm (Algorithm 2), determining the nonzero $l_{k,j}$ in the loop starting at line 7 is straightforward. Even in the most involved MGS variant, it reduces to a sparse product of the vector z_k with the matrix C. The result is then compared with the matrix whose columns comprise the vectors $z_j^{(k-1)}$ for $j = k+1$ to n. For efficiency, this matrix needs to be held as a sparse matrix. The amount of fill-in is typically modest and the sparse rank-one updates can often be fast. But a data structure that allows fast dynamic operations is needed. The Benzi and Tůma right-looking implementation [8, 9] employs the data structure described in [37, 44] (and is used in the early sparse direct solvers `MA28` [17] and Y12M [44]). While developments in sparse matrix technologies (see, for example, [18, 24]), as well as the advent of block methods, mean that such dynamic schemes have long been superseded in sparse direct solver software, the same is not true for incomplete factorization algorithms. This is at least partly because their memory requirements keep right-looking schemes viable. At the start of the computation, three arrays are allocated. The compressed row indices, column indices, and matrix values in individual columns are held as sections within these arrays with some additional spare space between each row and column. This is gradually filled and, at each step j of the factorization, for each array the memory used is recorded to be the first unused entry in the array plus the size of the j computed columns of the lower triangular factor. At some step, the size of an array may be found to be insufficient. In this case, it is reallocated to be larger and the data in the old array that is still required (which may have become fragmented) is copied to the front of the new array.

2.3. Existing strategies for exploiting sparsity. The practical success of solvers based on factorizing large sparse matrices crucially depends on exploiting sparsity. Over the last 40 or more years, many sophisticated techniques have been developed for sparse direct solvers. For incomplete factorizations, far less has been done. This may be because of the relative simplicity of many incomplete factorization schemes and they offer far fewer opportunities for the employment of block algorithms. However, the RIF approach is not straightforward to implement and the computational schemes in the original papers [5, 6, 8] do not discuss the exploitation of sparsity in depth.

To take advantage of sparsity in the left-looking approach, we first observe that the inner products at lines 5, 7, and 9 of Algorithm 1 involve matrix-vector products and, if C is sparse, most of these inner products are zero and so the corresponding update operation at line 11 of Algorithm 1 can be skipped. The most crucial step from the point of view of exploiting sparsity is thus the determination of which inner products are nonzero. Once known, it is straightforward to use sparsity in the other steps of the algorithm.

In an early left-looking implementation, determining the nonzero inner products was interleaved with the actual numerical updates and was based on an efficient search of columns of C. For LS problems, C may not be available explicitly and, in this case,

such searching is unacceptably slow. Consequently, a right-looking approach was used by Benzi and Tůma [9] but, as already observed, this has the major disadvantage of not being memory limited. Thus we want to develop a left-looking limited memory approach that is able to exploit sparsity when C is stored implicitly.

There are two basic graph-based strategies to determine the nonzero inner products in Algorithm 1 (with no dropping). The most straightforward (which we will call Strategy I) is based on the sparsity pattern of Z, which is known theoretically. The following result was shown in [7, 12] (and is a consequence of [22]).

LEMMA 2.1. *Assume there is no cancellation in the factorization process. Then* $z_{k,j} \neq 0$ *if and only if* j *is an ancestor of* k *in the elimination tree of* C.

Consequently, for each k, the sparsity pattern of column z_k can be found using the elimination tree. Once the pattern is known, which inner products should be evaluated can be determined; that is, the required inner products are flagged by a search through a submatrix of C that is determined by the symbolic structure of z_k. A bound on the complexity of the corresponding symbolic procedure for the mixed Gram–Schmidt method (AINV), for example, is given by the following straightforward result.

LEMMA 2.2. *For Strategy* I *and a given* $k \geq 2$, *the complexity to determine all indices* $j < k$ *of the AINV nonzero inner products at line* 9 *of Algorithm* 1 *is bounded by* $O(nz(\sum_{\{j|j\in anc(k)\}} C_{1:k-1,j})$, *where* $j \in anc(k)$ *denotes* j *is an ancestor of* k *in the elimination tree of* C.

However, Strategy I has a serious disadvantage: it typically results in many more inner products being evaluated than is necessary, even in exact arithmetic. This is illustrated by Bridson and Tang [13]. The reason for this overdetermination is that the structure of z_k is just the final structure of $z_k^{(k-1)}$ and this is generally a superset of the patterns that are considered in the minor steps of the algorithm. Therefore, many of the inner products determined using Strategy I are still zero.

A significant enhancement can be achieved by using the relation between L and Z given in [28] and which has been considered in [13]; we will refer to it here as Strategy II. It is based on the fact that the inner products used to update column $z_k^{(j-1)}$ are the entries of the kth row of L. Assuming no dropping, we can formulate this observation in terms of the elimination tree and its row subtrees [34], which can be cheaply computed on the fly.

LEMMA 2.3. *The inner product* $\langle z_k^{(j-1)}, z_j\rangle_C \equiv \langle z_k^{(j-1)}, e_j\rangle_C$ *is nonzero if and only if* j *belongs to the* k*th row subtree* $T_r(k)$ *of the elimination tree of* L.

Without going into the theory of sparse matrix decompositions, let us explain why this characterization implies significantly fewer dot products. The total number of entries from the row subtrees is equal to the number of nonzeros in L because the row subtrees describe the sparsity of the rows of L. This is the number of inner products given by this lemma. Moreover, all the ancestors in Lemma 2.1 contain as many nonzeros as L^{-1}, which is clearly often much greater than the nonzeros in L. Based on this characterization, we have the following complexity result for the AINV Gram–Schmidt process (but note that, in the case of exact arithmetic, the MGS, CGS, and AINV variants are equivalent).

LEMMA 2.4. *Suppose there is no cancellation in the factorization process. Then for a given* $k \geq 2$, *the complexity to determine all indices* $j < k$ *of the AINV nonzero inner products at line* 9 *of Algorithm* 1 *is bounded by* $O(nz(\sum_{\{j|j\in T_r(k)\}} C_{1:k-1,j})$.

3. Exploiting sparsity for implicit C and incomplete factorizations. In this section, we introduce our new symbolic preprocessing to exploit sparsity in C when it is available only implicitly. In the previous section, we assumed there was no dropping. In an incomplete factorization, there is dropping and as row subtrees are no longer useful, an alternative approach is needed. Consider the following approximate bordering scheme for a symmetric positive-definite matrix C_{k-1} that is extended symmetrically by appending one row and column c_k:

$$\begin{pmatrix} C_{k-1} & c_k \\ c_k^T & \gamma_k \end{pmatrix} \approx \begin{pmatrix} \tilde{L}_{k-1} & \\ \tilde{l}_k^T & \tilde{\lambda}_k \end{pmatrix} \begin{pmatrix} \tilde{L}_{k-1}^T & \tilde{l}_k \\ & \tilde{\lambda}_k \end{pmatrix}. \tag{3.1}$$

The off-diagonal entries $\tilde{l}_k$ of the new column of $\tilde{L}^T$ satisfy $\tilde{l}_k = \tilde{L}_{k-1}^{-1} c_k$. To obtain the sparsity pattern of $\tilde{l}_k$, recall the following lemma of Gilbert [22] that holds for any triangular matrix (and thus in particular for the incomplete factor $\tilde{L}_{k-1}$).

LEMMA 3.1. *The sparsity structure of $\tilde{l}_k$ is equal to the subset of vertices of the DAG of $\tilde{L}_{k-1}$ that is reachable by directed paths from the nonzeros of c_k (that is, $Reach_{k-1}(Struct(c_k))$).*

Employing this result in Algorithm 1, the sparsity pattern of row k of $\tilde{L}$ determines the nonzero inner products that should be evaluated by the Gram–Schmidt process and used to obtain the numerical values of the entries in the row. There are two main problems connected with this but, as we now show, they can both be overcome. First, we do not have $DAG(\tilde{L})$ readily available since Algorithm 1 computes $\tilde{L}$ by rows but the efficient computation of the pattern of $\tilde{L}$ needs access by columns. To deal with this, $DAG(\tilde{L})$ is computed as the computation proceeds. At each step, a new row is added into $DAG(\tilde{L})$. Therefore, it is computed only once. To store $DAG(\tilde{L})$ by columns, we use a set of linked lists, one for each column, held using a single array that is reallocated as necessary. If $DAG(\tilde{L})$ has nz_d nonzeros, then we need $2 * nz_d$ memory positions to store the indices (not including the diagonal ones) plus n positions to point to the location of the first entry in each column. The pruning algorithms that we describe below are such that they can be implemented efficiently using this linked list data structure. Note that in the case of a complete factorization, computing a DAG on the fly is unnecessary since the elimination tree is available and no other graph structure is needed.

The second potential problem is that $DAG(\tilde{L})$ may have more edges than are needed to generate the row structure. The graph with the smallest number of entries needed to generate the row structure is the transitive reduction [1] and for a DAG it is unique. In general, finding the transitive reduction is expensive; instead, a cheap preprocessing called pruning is used. Pruning aims to remove edges that are both cheap to find and redundant in preserving the set of paths in $DAG(\tilde{L})$. We note that a similar mechanism for pruning based on structural symmetry in LU factorizations was proposed by Eisenstat and Liu [21].

The first simple pruning approach that we consider is based on the following lemma.

LEMMA 3.2. *Let $\tilde{L}$ be a lower triangular matrix with graph $DAG(\tilde{L})$. Assume that for some $i < j < k$, $\tilde{l}_{k,i} \neq 0$ and $\tilde{l}_{k,j} \neq 0$, $\tilde{l}_{j,i} \neq 0$. Then the set of paths in $DAG(\tilde{L})$ stays the same if $\tilde{l}_{k,i}$ is set to* 0.

Proof. Because of the edges (k, j) and (j, i), we have the path $k \Rightarrow i$ and so the edge (k, i) can be pruned. ∎

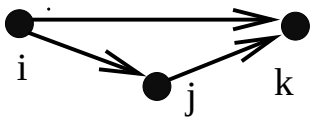

FIG. 1. *Simple illustration of Lemma* 3.2. *The edge* (i, k) *can be pruned.*

Algorithm 3. Simple pruning algorithm.

Input: Lower triangular matrix $\tilde{L} \in R^{n\times n}$.
Output: Pruned graph $DAG_p(\tilde{L})$ with vertex set V and edge set E.

1. Set $V = \{1\}$, $E = \emptyset$
2. Set $PREV(i) = i$, $1 \le i \ldots \le n$
3. **for** $k = 2 : n$
4. Set $V = V \cup \{k\}$
5. Obtain the sparsity structure $J_k = \{j_1, \ldots, j_p\}$ of $\tilde{L}_{k,1:k-1}$ with $1 \le j_1 < \ldots < j_p \le k-1$
6. Set $E' = (k, j_1) \cup \ldots \cup (k, j_p)$
7. **for** $i = 1 : p$
8. **if** $PREV(j_i) = i$ **or** $(k, PREV(j_i)) \notin E'$ **then**
9. Set $E = E \cup (k, j_i)$
10. $PREV(j_i) = k$
11. **end if**
12. **end for**
13. **end for**

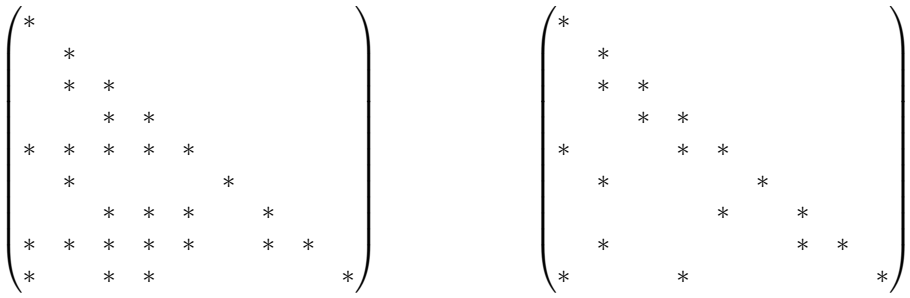

FIG. 2. *The structure of* $\tilde{L}$ *is on the left and the matrix corresponding to the pruned graph* $DAG_p(\tilde{L})$ *is on the right.*

Figure 1 provides a simple illustration of Lemma 3.2. Its assumptions imply the edge (i, k) can be pruned from the DAG.

Algorithm 3 applies the test in Lemma 3.2 successively to vertices of $DAG(\tilde{L})$ to get the pruned $DAG_p(\tilde{L})$. The algorithm is illustrated using the example given in Figure 2.

It is easy to see that the complexity of Algorithm 3 is linear, that is, its iteration count is of the order $O(nz(\tilde{L}))$. However, in some cases it is not able to prune $DAG(\tilde{L})$ sufficiently. Consider a lower bidiagonal matrix that has additional nonzero entries at positions (i, j) such that $i = 2 * l, j = 1, \ldots, (i-4)/2$ for $l = 2, \ldots, \ulcorner n/2 \urcorner$. This

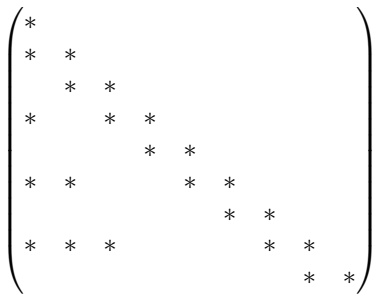

FIG. 3. *The structure of a matrix $\tilde{L}$ that Algorithm 3 is unable to prune.*

Algorithm 4. More powerful pruning algorithm.

Input: Lower triangular matrix $\tilde{L} \in R^{n \times n}$.
Output: Pruned graph $DAG_p(\tilde{L})$ with vertex set V and edge set E.

```
1.  Set V = {1}, E = ∅
3.  for k = 2 : n
4.     Set V = V ∪ {k}
5.     Obtain the sparsity structure J_k = {j_1, ..., j_p} of L̃_{k,1:k-1} with
       1 ≤ j_1 < ... < j_p ≤ k − 1
7.     for i = 1 : p
8.     found = false
15.       for kk such that {(kk, j_i)} ∈ E in the increasing order
16.          if (k, kk) ∈ E then
17.             found =true
18.          end if
19.       end for
20.       if found = false
21.          Set E = E ∪ (k, j_i)
23.       end if
24.    end for
25. end for
```

sparsity pattern with $n = 9$ is depicted in Figure 3. The number of edges in the corresponding DAG is of the order n^2. For this example, Algorithm 3 does not prune any edges. This suggests we need a more powerful approach to pruning. For each index $j_i \in J_k$, the simple approach used by Algorithm 3 applies a test based on just one nonzero (line 8). It seems reasonable to do more searches, provided the number of tests is limited. A straightforward approach is presented in Algorithm 4.

It is straightforward to see that after applying Algorithm 4 to the lower triangular matrix given in Figure 3, $DAG_p(\tilde{L})$ has $O(n)$ nonzero entries but that it is still not transitively reduced since the worst-case complexity of this reduction based on the appropriate reachability sets is, in general, $O(n^2)$. The following result describes the complexity of Algorithm 4.

LEMMA 3.3. *Assume that the number of nonzero entries in each column of $\tilde{L}$ or in each row of $\tilde{L}$ is bounded by lz. Then the number of comparisons in Algorithm 4 is bounded by $O(n * lz^2)$.*

Proof. The comparisons in the algorithm imply that for each nonzero entry $\tilde{l}_{i,j}$, all other row indices i_k such that $l_{i_k,j} \neq 0$ and $j < i_k < i$ should be tested. If each column of $\tilde{L}$ has at most lz entries, then $nz(\tilde{L}) \leq n * lz$, and for each entry at most lz comparisons are performed. If each row of $\tilde{L}$ has at most lz entries, then for each of its n rows we need to compare only entries in those rows that correspond to its nonzero entries. The maximum number of such rows is lz and the result follows. ∎

Note that the above bound is asymptotically the same as the number of operations needed by the exact Cholesky factorization with the same upper bound for the number of entries in its columns. Our case of the incomplete factorization based on inner products and different data structures is not exactly comparable but it is clear that the complexity of pruning specified in Lemma 3.3 is affordable. Further additional dropping in $\tilde{L}$ based on the magnitudes of the computed inner products can be applied, typically for each k once the structure of column $\tilde{l}_k$ has been computed.

4. Implicit left-looking RIF for LS problems. While the previous section looked at exploiting sparsity in the left-looking RIF algorithm for general symmetric positive-definite matrices, here we focus on its application to the solution of LS problems. In Algorithm 5, we present an outline of our implicit left-looking RIF algorithm for LS problems that avoids computing the normal matrix $C = A^T A$ explicitly. Algorithm 5 includes scaling and the optional use of a nonzero shift α so that the RIF factorization of $S(C + \alpha I)S$ is computed where the diagonal matrix S is an $n \times n$ column scaling matrix.

Observe that since the algorithm treats $C = A^T A$ implicitly, the shift is performed within the main loop and the scaling is applied to the shifted entries. In some applications in optimization, such as the Levenberg–Marquardt method for solving nonlinear LS, a nonzero shift is used (see, e.g., [16]). However, if $\alpha = 0$, the algorithm to compute the RIF factorization can be significantly accelerated, as we state in the following proposition.

PROPOSITION 4.1. *If the shift α is equal to zero, then the computed v_j do not need to be stored. Instead, if $v_j = \tilde{z}_j$ (MGS variant,) the matrix-vector products $p_j = Av_j$ can be precomputed and stored (that is, once $\tilde{z}_k$ is computed at line* 26*, $A\tilde{z}_k$ may be computed and stored). Note that the case $v_j = e_j$ (CGS and AINV variants) trivially uses a column of A, that is, $p_j = a_j$.*

5. Numerical experiments.

5.1. Test environment. Most of the test problems used in our experiments are taken from the University of Florida Sparse Matrix Collection [15]. The exceptions are a Laplacian test example, problem IPROB (which is part of the CUTEst linear programming set [25]) and problem PIGS_large1 from a pig breeding application (see [2] for details). In our experiments involving symmetric positive-definite linear systems $Cx = b$, the right-hand-side vector b is computed so that the exact solution is $x = 1$, and the stopping criteria used for preconditioned conjugate gradients (PCG) is

$$\|Cx - b\|_2 \leq 10^{-6}\|b\|_2. \tag{5.1}$$

We define the *density ratio* of the computed incomplete factor $\tilde{L}$ to be

$$\rho = nz(\tilde{L})/nz(C).$$

For the LS tests, we use PCGs for the normal equations (CGNE). We employ the following stopping rules that are taken from [26]:

Algorithm 5. Left-looking RIF algorithm for LS with $C = A^TA$ held implicitly.

Input: $A \in R^{m\times n}$ with full column rank, a shift $\alpha \geq 0$, and drop tolerance $\tau > 0$.
Output: Incomplete RIF factor $\tilde{L}$ (stored by rows).

Compute a column scaling S and scale: $A \leftarrow AS$
Set $\tilde{L}_{1,:} = (1+\alpha)\sqrt{a_1^Ta_1}$
Set $\tilde{z}_1 = e_1$
for $k = 2 : n$ **do**
  Set $\tilde{z}_k^{(0)} = e_k$
  Let $c_k = A^TA_{1:k-1,k}$
  Compute the sparsity structure J_k of $\tilde{L}_{k-1}^{-1}c_k$ as $Reach_{k-1}(Struct(c_k))$.
  Prune J_k using Algorithm 3 or 4 to get $J_k = \{j_1 < \cdots < j_p\}$ with $p \leq k-1$; set $j_0 = 0$
  for $s = 1 : p$ **do**
    $j = j_s$
    if MGS
      Set $v_j = \tilde{z}_j$ and $u_k = \tilde{z}_k^{(j_{s-1})}$
    else if CGS
      Set $v_j = e_j$ and $u_k = \tilde{z}_k^{(0)}$
    else if AINV
      Set $v_j = e_j$ and $u_k = \tilde{z}_k^{(j_{s-1})}$
    end if
    Compute $p_j = Av_j, \quad q_k = Au_k \quad$ and, $\quad$ if $\alpha > 0, \quad \beta_k = v_j^TS^2u_k$
    if $p_j^Tq_k + \alpha\beta_k > \tau$ **do**
      Set $\tilde{l}_{k,j} = p_j^Tq_k + \alpha\beta_k$
      Set $\tilde{z}_k^{(j)} = \tilde{z}_k^{(j_{s-1})} - \tilde{l}_{k,j}\tilde{z}_j$
      Discard all components of $z_k^{(j)}$ less than τ in absolute value
    end if
  end do
  Set $\tilde{l}_{k,k} = \sqrt{(A\tilde{z}_k^{(j_p)})^T(A\tilde{z}_k^{(j_p)})}$
  Set $\tilde{z}_k = \tilde{z}_k^{(j_p)}/\tilde{l}_{k,k}$
end do

C1. Stop if $\|r\|_2 < \delta_1$, or
C2. Stop if

$$\frac{\|A^Tr\|_2}{\|r\|_2} < \frac{\|A^Tr_0\|_2}{\|r_0\|_2} * \delta_2,$$

where $r = Ax - b$ is the residual, r_0 is the initial residual, and δ_1 and δ_2 are convergence tolerances that we set to 10^{-8} and 10^{-6}, respectively. In all our experiments, we take the initial solution guess to be $x_0 = 0$ and in this case C2 reduces to

$$\frac{\|A^Tr\|_2}{\|r\|_2} < \frac{\|A^Tb\|_2}{\|b\|_2} * \delta_2.$$

We define the density ratio of the computed incomplete factor $\tilde{L}$ to be

$$\rho = nz(\tilde{L})/nz(A).$$

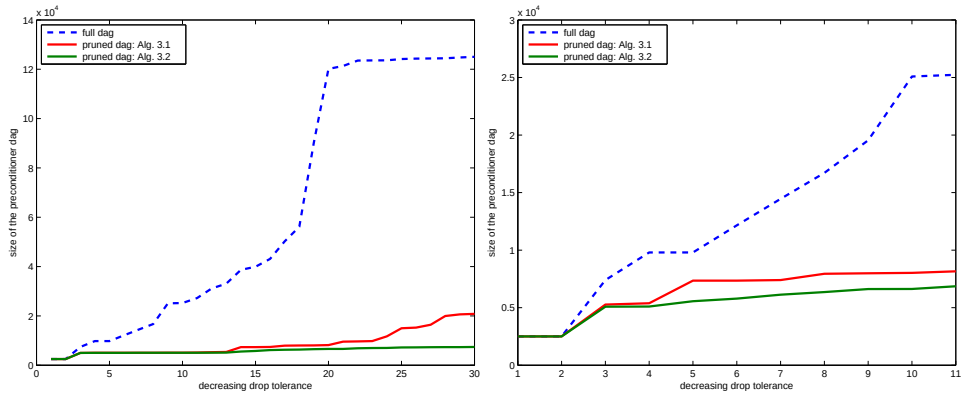

FIG. 4. *Effect of the two pruning approaches (Algorithms* 3 *and* 4*) on the two-dimensional Laplacian matrix (left) and on matrix Pothen/bodyy4 (right). We depict here the dependence of the number of edges in the auxiliary DAG on the drop tolerance.*

All the reported experiments use the MGS variant of the left- and right-looking RIF algorithms. The experiments are performed on an Intel Core i5-4590 CPU running at 3.30 GHz with 12 GB of internal memory. The codes are written in Fortran and the Visual Fortran Intel 64 XE compiler (version 14.0.3.202) is used.

5.2. The case for pruning. Our first experiment, which we report on in Figure 4, demonstrates the need for pruning. Results are given for two examples: a two-dimensional Laplacian matrix of dimension 25,000 and problem Pothen/bodyy4 (n = 17,546, $nz(C)$ = 121,550). In each case, we repeatedly increase the density ratio ρ by decreasing the drop tolerance τ used in the incomplete factorization (30 different tolerances are used for the Laplacian and 11 for Pothen/bodyy4). We see that Algorithm 3 is highly effective in limiting the growth in the number of edges in $DAG_p(\tilde{L})$ but that some further reductions are possible using the more sophisticated Algorithm 4, particularly as the number of entries in $\tilde{L}$ increases. Note that we get exactly the same preconditioner (that is, the same nonzero entries) with pruning as without pruning.

5.3. Memory management for left- and right-looking approaches. Our second experiment is designed to demonstrate the principal differences in the memory management of the left- and right-looking approaches. Here we employ symmetric positive-definite test matrices while the experiments in the next subsection are for RIF applied to the normal equations ($C = A^T A$). Figure 5 depicts the reported memory for three preconditioners of different densities for problem Nasa/nasa1824 (n =1,824, $nz(C)$ = 39,208). The density ratios are $\rho = 1.2$, 1.6, and 2.8, respectively. The PCG iteration counts are 26, 18, and 8, respectively. To understand the results, we return to how we measure memory consumption during the factorization. For the left-looking approach, the reported memory is the size of the approximate inverse factor $\tilde{Z}$ plus that of the preconditioner $\tilde{L}$ plus the memory for the DAG. As described in section 2.2, for the right-looking approach the rows and columns of $\tilde{Z}$ are stored as contiguous parts of large arrays with empty locations between each of them. In our implementation, we allow $lsize = 5$ such locations and the arrays are initially allocated to be of size $n * lsize$ plus an additional $2 * nz(A)$ free locations. In general, we have found that the run time is not very sensitive to the choice of $lsize$ and the

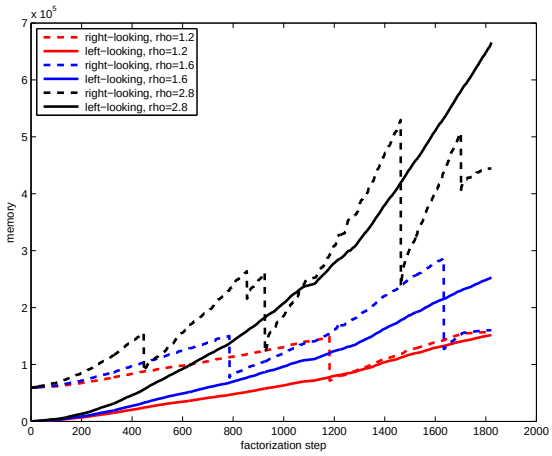

FIG. 5. *Memory comparison for the left- and right-looking RIF approaches for Nasa/nasa*1824 *for three different density ratios (denoted here by rho).*

initial array size; the cost is mainly driven by the amount of fill-in and the overhead incurred from the data structures. As the computation proceeds, the *lsize* empty locations between the contiguous segments in the large arrays become filled. Once there is no space to extend a row or a column, it is copied into the free locations at the end of the array. If there is insufficient space for this, the array is reallocated (in all our experiments, when reallocating we double the previous size) and (fragmented) data in the old array that is still needed is copied to the front of the new array, again with each row and column that is not yet computed separated by *lsize* empty locations, and the memory pointer is reset. Locations from the first to the last nonzero position form the *active part* of the array. In reporting the memory consumption, we show just the size of the active part, which explains the sharp drop in the reported memory for the right-looking approach when an array is found to be too small. We observe that the reallocation process may be repeated several times as the memory required is not known a priori. Note that if the original array is sufficiently large there will be no drops in the memory usage as the factorization proceeds but because of fragmentation, using such a large array will potentially result in significantly more memory being used than is necessary.

We see from Figure 5 that as the density ratio ρ increases, the influence of the memory to store $\tilde{Z}$ becomes more significant. But, in general, this occurs when the preconditioner is too large to be practical. Similar results for problems Nasa/nassarb and GHS_psdef/hood are given in Figure 6. For Nasa/nassarb, ρ is 0.8, 1.2, and 2.1 and the PCG iteration count decreases from 194 to 43. For GHS_psdef/hood, the densities are 0.5 and 1.5 and the corresponding PCG iteration counts are 224 and 51. GHS_psdef/hood illustrates large but not atypical differences between the right- and left-looking memory demands.

5.4. Least-squares problems. We now explore some differences between the left- and right-looking approaches when used for solving the normal equations (1.2). Our first example is a small square matrix IPROB of order $n = 3001$ for which A^TA is dense ($nz(A^TA) \approx 9 \times 10^6$). The RIF preconditioner is of size $nz(\tilde{L}) \approx 2.5 \times 10^6$ and $nz(\tilde{Z}) \approx 8.6 \times 10^4$; three CGNE iterations are required for convergence. The plot on the left in Figure 7 shows that both the left- and right-looking algorithms

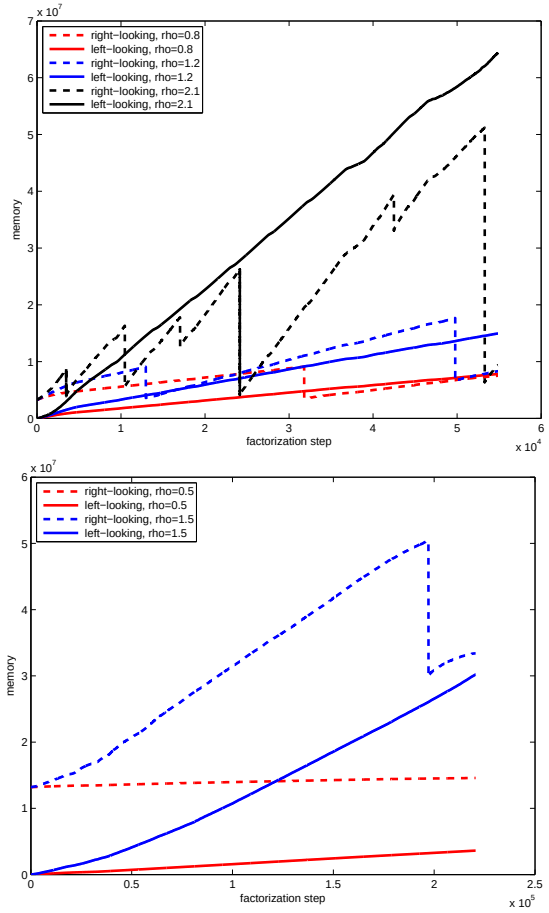

FIG. 6. *Memory comparison for the left- and right-looking RIF approaches for problem Nasa/nasasrb for three different density ratios (left) and GHS_psdef/hood for two different density ratios (right) (denoted here by rho).*

have similar memory requirements if the size of A^TA (left-looking approach) or of A (right-looking approach) is not taken into account. However, in the plot on the right we include the size of A^TA for the left-looking approach and of A and A^T for the right-looking approach (A must be held by both rows and columns) and we now see that the explicit computation of the normal matrix A^TA can result in an unacceptable overhead for the left-looking algorithm.

In many applications, m is significantly larger than n and consequently $nz(A^TA)$ can be smaller than $nz(A) + nz(A^T)$. For problem LPnetlib/lp_osa_30 ($m = 104{,}374$, $n = 4{,}350$, and $nz(A) = 604{,}488$), Figure 8 shows the RIF memory requirements for a sparse preconditioner (here the density ratio ρ is approximately 0.5). The corresponding comparison for an increased density ratio of 0.8 is given in Figure 9. Note again the sharp drops in the memory for the right-looking approach indicate memory reallocations are needed. We also observe that for this example the factors fill in significantly toward the end of the factorization, resulting in a sudden increase in the memory for the left-looking RIF where the memory is dominated by $\tilde{L}$ (for the right-looking approach the total memory dominates that of $\tilde{L}$ so the increase from the fill-in

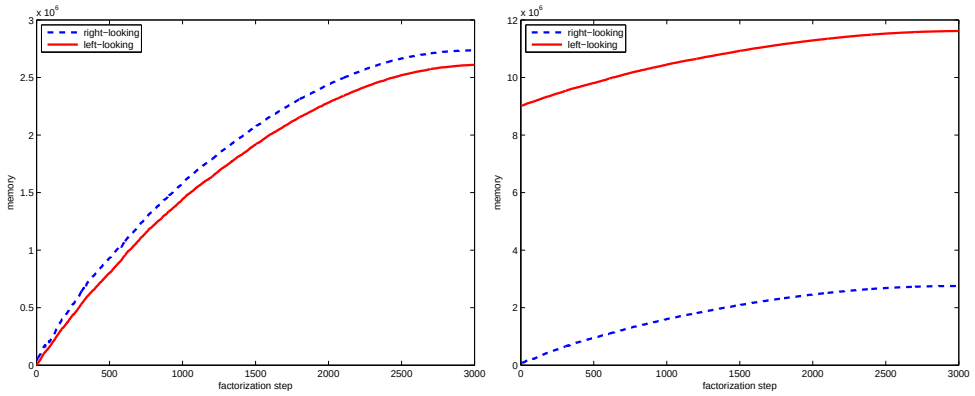

FIG. 7. *Memory comparison for the left- and right-looking RIF approaches for problem IPROB for which the normal equations $C = A^TA$ are dense. The plot on the left does not take into account the memory for input matrix C. The plot on the right adds to the explicit left-looking approach the size of A^TA and adds $nz(A) + nz(A^T)$ to the right-looking approach.*

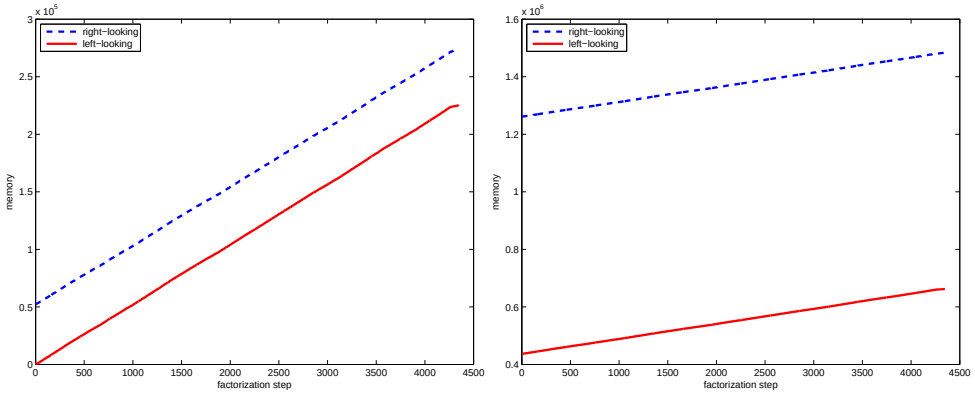

FIG. 8. *Memory comparison for the left- and right-looking RIF approaches applied to the normal equations for problem LPnetlib/lp_osa_30 with $\rho = 0.5$. The plot on the left does not take into account the input matrix A. The plot on the right adds to the explicit left-looking approach the size of A^TA and adds to the right-looking approach the sizes of A and A^T.*

is not seen). In Figure 10, we plot memory usage for problem Yoshiyasu/mesh_deform ($m = 234{,}023$, $n = 9393$, and $nz(A) = 853{,}829$); the density ratio is approximately 4. Again, for this example with $m \gg n$, the left-looking approach is considerably more memory efficient than the right-looking one.

5.5. Explicit versus implicit left-looking approaches. We next consider the differences between the left-looking algorithm that explicitly forms the normal matrix $C = A^TA$ and the new implicit DAG-based variant that avoids forming C. Note that both result in preconditioners of a similar quality and lead to essentially the same CGNE iteration counts. In some practical applications it may not be possible to form C, in which case the only computational possibility with limited memory is the DAG-based approach. But this is also the method of choice if $nz(A^TA) \gg nz(A)$, as illustrated in Table 1 by our reported results for problems IPROB and Bydder/mri2. Note that for these examples, pruning (using Algorithm 4) leads to a significant

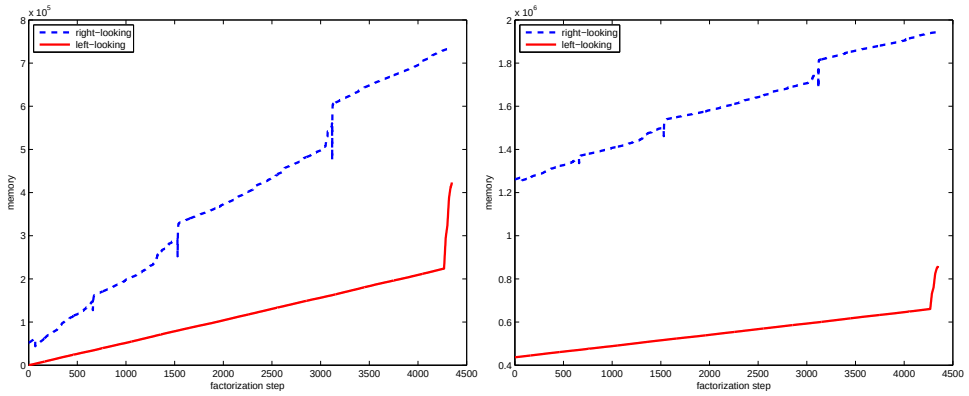

FIG. 9. *Memory comparison for the left- and right-looking RIF approaches applied to the normal equations for problem LPnetlib/lp_osa_30 with* $\rho = 0.8$. *The plot on the left does not take into account the input matrix* A. *The plot on the right adds to the explicit left-looking approach the size of* A^TA *and adds to the right-looking approach the sizes of* A *and* A^T.

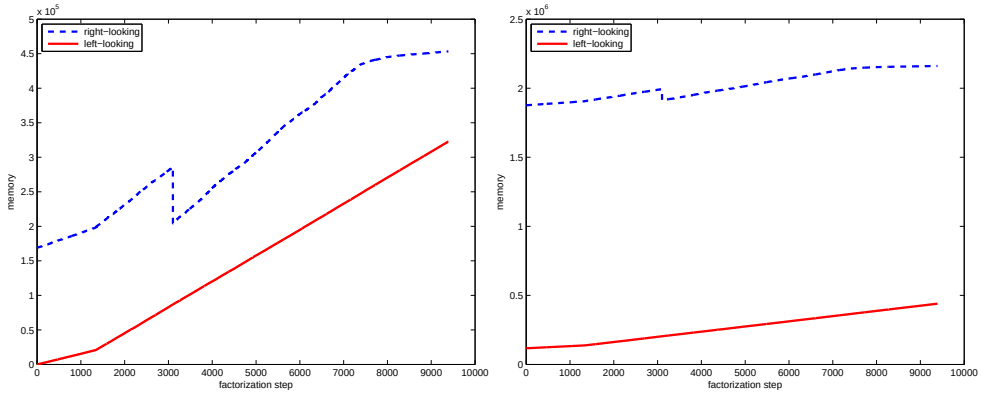

FIG. 10. *Memory comparison for the left- and right-looking RIF approaches applied to the normal equations for problem Yoshiyasu/mesh_deform with* $\rho = 4$. *The plot on the left does not take into account the input matrix* A. *The plot on the right adds to the explicit left-looking approach the size of* A^TA *and adds to the right-looking approach the sizes of* A *and* A^T.

reduction in the number of edges in the DAG. However, if C can be formed and stored and is sufficiently sparse, the explicit algorithm can be much faster than the implicit one. This happens, in particular, if $\tilde{L}$ is not very sparse. This is demonstrated by problems PIGS_large1 and Pereyra/landmark. The final three problems in Table 1 are known to be challenging (see [26]) and for these relaxed stopping tolerances $\delta_1 = 10^{-5}$ and $\delta_2 = 10^{-3}$ are used together with a shift $\alpha = 0.1 * \|A^TA\|_F$.

Figure 11 (left-hand plot) for example LPnetlib/lp_osa30 illustrates that the explicit algorithm can be as fast as the implicit left-looking algorithm but, in general, which is faster depends on the preconditioner size. For problem Kemelmacher/Kemelmacher (right-hand plot in Figure 11), we see that as ρ increases, the explicitly computed C does not prevent the computational time from steadily increasing. Thus, summarizing our experience, we conclude that both the explicit and implicit algorithms can be useful, and even when C is available, deciding which approach

TABLE 1

A comparison of the explicit left-looking (Explicit LL), implicit left-looking (Implicit LL), and implicit right-looking (Implicit RL) RIF approaches when used to precondition CGNE. Time is the time (in seconds) to compute the preconditioner; Iters is the number of CGNE iterations; size_p denotes the number of entries in the preconditioner (for Implicit LL this is the number of edges in the DAG, and before and after indicate the number of edges before and after pruning). $*$ denotes stopping tolerances $\delta_1 = 10^{-5}$ and $\delta_2 = 10^{-3}$ are used together with a shift $\alpha = 0.1 * \|A^T A\|_F$.

Identifier	$nnz(A)$	$nnz(A^T A)$	Explicit LL Time	Explicit LL *size_p*	Implicit LL Time	Implicit LL *size_p* before	Implicit LL *size_p* after	Implicit RL Time	Implicit RL *size_p*	Iters
Pereyra/ landmark	1.15×10^6	1.20×10^5	0.22	2.59×10^4	2.62	2.59×10^4	7.84×10^3	1.35	2.69×10^4	50
PIGS_large1	7.50×10^4	1.29×10^5	0.08	4.83×10^4	0.42	4.83×10^4	3.94×10^4	0.08	4.74×10^4	94
LPnetlib/ lp_ken_18	3.58×10^5	6.09×10^5	3.02	3.99×10^5	1.46	4.10×10^5	2.98×10^5	21.6	3.62×10^5	103
LPnetlib/ lp_osa30	6.04×10^5	4.37×10^5	11.5	2.21×10^5	14.8	2.20×10^5	1.05×10^5	17.9	2.20×10^5	155
JGD_Groebner/ f855_mat9*	1.71×10^5	4.49×10^6	33.3	2.22×10^6	6.03	2.23×10^6	2.00×10^4	13.6	2.42×10^6	799
JGD_Groebner/ c8_mat11*	2.46×10^6	1.69×10^7	277.	7.90×10^6	146.	8.14×10^6	4.64×10^4	120.	7.83×10^6	961
Bydder/mri2*	5.69×10^5	3.13×10^7	569.	7.42×10^6	129.	1.03×10^7	1.64×10^5	83.5	1.13×10^7	671

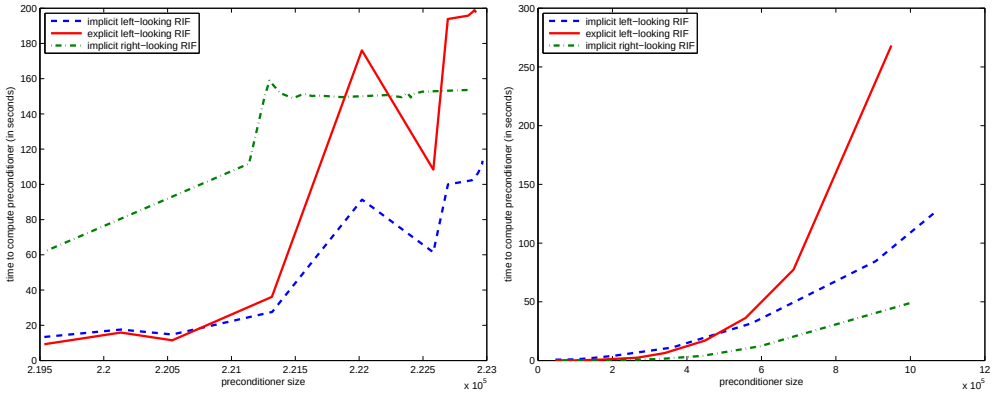

FIG. 11. *Dependence of the time to compute the explicit and implicit left-looking RIF on the size of the preconditioner $\tilde{L}$ for problems LPnetlib/lp_osa30 (left) and Kemelmacher/Kemelmacher (right).*

will be the computationally most efficient is not a clear choice. Construction of the preconditioner using the auxiliary DAG can be considered as a way to a parallel implementation since the implicit algorithm enables the classical generalized Gram–Schmidt algorithm to be used to compute the factor $\tilde{Z}$. This offers the potential to significantly enhance the exploitation of parallelism, but a more detailed discussion lies outside the scope of the current study. We also note in this context that we could prescribe the structure for $\tilde{L}$. This introduces another parameter into the construction of the preconditioner.

Another interesting question is how the RIF approach compares with other preconditioning techniques for LS problems, in particular, those based on orthogonal decompositions. Comparisons with preconditioners based on incomplete Gram–Schmidt decompositions and with the Givens rotation-based strategies for solving LS problems

are given in [9]. These illustrate that, in terms of total solution time, RIF is often the winner. Very recently, Gould and Scott [26, 27] presented the most comprehensive numerical evaluation yet of preconditioned iterative methods for solving LS problems; RIF was included in this study along with the MIQR factorization of Li and Saad [33]. Gould and Scott show that, while generally slower than MIQR, in some instances RIF can be the faster approach, with the difference in their run times being highly problem dependent. For example, for problem JGD_Groebner/f855_mat9 (see Table 1), Gould and Scott report RIF is three times faster than MIQR (total solution time of 78 versus 220 seconds). Although [26] finds the best preconditioner to be our limited-memory incomplete Cholesky factorization [41, 42], because LS problems are so diverse and can be very tough to solve, the development of complementary approaches is important.

6. Concluding remarks. In this paper, we have proposed a new left-looking variant of the RIF approach for computing an incomplete LL^T factorization of a sparse positive-definite matrix and, in particular, the normal equations matrix $C = A^T A$. The practical success of solvers for the solution of large sparse problems crucially depends on the efficient and effective exploitation of sparsity. In the case of direct methods, the importance of sparsity is well understood and, over many years, sophisticated techniques have been developed to take advantage of sparsity throughout the factorization. Much less has been done for incomplete factorizations. While this may partly be because of the relative simplicity of many such algorithms, this is definitely not the case for sophisticated schemes such as RIF that need complicated data structures and combine techniques from Gaussian elimination and orthogonalization. For the left-looking RIF algorithm, we have introduced a global symbolic preprocessing step that constructs a directed acyclic graph $DAG(\tilde{L})$ that determines the sparsity pattern of the preconditioner factor $\tilde{L}$, without the need to explicitly construct the matrix C. An efficient pruning algorithm has been proposed to limit the number of edges in $DAG(\tilde{L})$. Numerical experiments have shown this pruning algorithm to be highly effective.

A fundamental difference between the left- and right-looking RIF approaches is their memory management. For the latter, the memory required is not known a priori and so it can be necessary to increase the memory available during the factorization and this reallocation may have to be done more than once. Our results have illustrated that, for linear systems, which approach is most memory efficient not only is problem dependent but also depends on the density of the computed $\tilde{L}$ (that is, on the choice of the dropping parameter). For LS problems, we additionally need to take into account the number of entries in A and A^T or, in the case of the explicit left-looking algorithm, the number of entries in $C = A^T A$. We have found that the performances of the different variants can vary significantly but for a given problem, without some prior knowledge of the problem and its characteristics, it is not obvious which approach will be most efficient. Clearly, if a series of similar problems must be solved and it is possible to construct and store C, it may be worthwhile to try both the explicit and implicit approaches and select the most efficient. If it is not possible to explicitly compute C, then the new implicit variant offers a viable alternative.

While this study was carried out with prototype codes, in the future we plan to develop library-quality implementations that will be made available as part of the HSL software library [29]. We anticipate that these implementations will include options for ordering and scaling of the problem, which can potentially significantly enhance the performance.

Acknowledgment. We are grateful to two anonymous reviewers for their detailed and constructive comments.

REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman, *The transitive reduction of a directed graph*, SIAM J. Comput., 1 (1972), pp. 131–137.
[2] M. Arioli and I. S. Duff, *Preconditioning linear least-sqaures problems by identifying a basis matrix*, SIAM J. Sci. Comput., 37 (2015), pp. S544–S561.
[3] H. Avron, P. Maymounkov, and S. Toledo, *Blendenpik: Supercharging LAPACK's least squares solver*, SIAM J. Sci. Comput., 32 (2010), pp. 1217–1236.
[4] C. Benoit, *Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des moindres carrés a un systeme d'équations linéaires en nombre inférieur a celui des inconnues. application de la méthode a la résolution d'un systeme défini d'équations linéaires*, Bull. Géodésique, 2 (1924), pp. 5–77.
[5] M. Benzi, J. K. Cullum, and M. Tůma, *Robust approximate inverse preconditioning for the conjugate gradient method*, SIAM J. Sci. Comput., 22 (2000), pp. 1318–1332.
[6] M. Benzi, C. D. Meyer, and M. Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM J. Sci. Comput., 17 (1996), pp. 1135–1149.
[7] M. Benzi and M. Tůma, *Orderings for factorized sparse approximate inverse preconditioners*, SIAM J. Sci. Comput., 21 (2000), pp. 1851–1868.
[8] M. Benzi and M. Tůma, *A robust incomplete factorization preconditioner for positive definite matrices*, Numer. Linear Algebra Appl., 10 (2003), pp. 385–400.
[9] M. Benzi and M. Tůma, *A robust preconditioner with low memory requirements for large sparse least squares problems*, SIAM J. Sci. Comput., 25 (2003), pp. 499–512.
[10] Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.
[11] A. Björck and J. Y. Yuan, *Preconditioners for least squares problems by LU factorization*, Electron. Trans. Numer. Anal., 8 (1999), pp. 26–35.
[12] R. Bridson and W.-P. Tang, *Ordering, anisotropy, and factored sparse approximate inverses*, SIAM J. Sci. Comput., 21 (1999), pp. 867–882.
[13] R. Bridson and W.-P. Tang, *Refining an approximate inverse*, J. Comput. Appl. Math., 123 (2000), pp. 293–306.
[14] R. Bru, J. Marín, J. Mas, and M. Tůma, *Preconditioned iterative methods for solving linear least squares problems*, SIAM J. Sci. Comput., 36 (2014), pp. A2002–A2022.
[15] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1:1–1:25.
[16] J. E. Dennis, Jr. and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Classics in Appl. Math. 16, SIAM, Philadelphia, 1996.
[17] I. S. Duff, *MA28—A Set of Fortran Subroutines for Sparse Unsymmetric Linear Equations*, Harwell Report UK AERE-R.8730, Harwell Laboratories, 1980.
[18] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325.
[19] S. Eisenstat and J.-W. H. Liu, *The theory of elimination trees for sparse unsymmetric matrices*, SIAM J. Matrix Anal. Appl., 26 (2005), pp. 686–705.
[20] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *The Yale Sparse Matrix Package (YSMP)—I: The symmetric codes*, Internat. J. Numer. Methods Engrg., 18 (1982), pp. 1145–1151.
[21] S. C. Eisenstat and J.-W. H. Liu, *Exploiting structural symmetry in unsymmetric sparse symbolic factorization*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 202–211.
[22] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 62–79.
[23] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 334–352.
[24] 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.
[25] N. I. M. Gould, D. Orban, and P. L. Toint, *CUTEst: A constrained and unconstrained testing environment with safe threads for mathematical optimization*, Comput. Optim. Appl., 60 (2015), pp. 545–557.
[26] N. I. M. Gould and J. A. Scott, *The State-of-the-Art of Preconditioners for Sparse Linear Least Squares Problems*, Technical Report RAL-P-2015-010, Rutherford Appleton Laboratory, 2015.

[27] N. I. M. Gould and J. A. Scott, *The State-of-the-Art of Preconditioners for Sparse Linear Least Squares Problems: The Complete Results*, Technical Report RAL-TR-2015-009 (revision 1), Rutherford Appleton Laboratory, 2016.
[28] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. National Bureau of Standards, 49 (1952), pp. 409–435.
[29] HSL, *A Collection of Fortran Codes for Large-Scale Scientific Computation*, http://www.hsl.rl.ac.uk (2016).
[30] A. Jennings and M. A. Ajiz, *Incomplete methods for solving $A^T Ax = b$*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 978–987.
[31] J. Kopal, M. Rozložník, A. Smoktunowicz, and M. Tůma, *Rounding error analysis of orthogonalization with a non-standard inner product*, BIT, 52 (2012), pp. 1035–1058.
[32] P. Läuchli, *Jordan-Elimination und Ausgleichung nach kleinsten Quadraten*, Numer. Math., 3 (1961), pp. 226–240.
[33] N. Li and Y. Saad, *MIQR: A multilevel incomplete QR preconditioner for large sparse least-squares problems*, SIAM J. Matrix Anal. Appl., 28 (2006), pp. 524–550.
[34] J. W. H. Liu, *The role of elimination trees in sparse factorizations*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172.
[35] X. Meng, M. A. Saunders, and M. W. Mahoney, *LSRN: A parallel iterative solver for strongly over- or underdetermined systems*, SIAM J. Sci. Comput., 36 (2014), pp. C95–C118.
[36] K. Morikuni and K. Hayami, *Inner-iteration Krylov subspace methods for least squares problems*, SIAM J. Matrix Anal. Appl., 34 (2013), pp. 1–22.
[37] O. Østerby and Z. Zlatev, *Direct Methods for Sparse Matrices*, Lecture Notes in Comput. Sci. 157, Springer-Verlag, Berlin, 1983.
[38] A. T. Papadopoulus, I. S. Duff, and A. J. Wathen, *A class of incomplete orthogonal factorization methods.* II*: Implementation and results*, BIT, 45 (2005), pp. 159–179.
[39] G. Peters and J. H. Wilkinson, *The least squares problem and pseudo-inverse*, Comput. J., 131 (1970), pp. 309–316.
[40] Y. Saad, *Preconditioning techniques for nonsymmetric and indefinite linear systems*, J. Comput. Appl. Math., 24 (1988), pp. 89–105.
[41] J. A. Scott and M. Tůma, `HSL_MI28`*: An efficient and robust limited-memory incomplete Cholesky factorization code*, ACM Trans. Math. Software, 40 (2014), pp. 24:1–24:19.
[42] J. A. Scott and M. Tůma, *On positive semidefinite modification schemes for incomplete Cholesky factorization*, SIAM J. Sci. Comput., 36 (2014), pp. A609–A633.
[43] X. Wang, K. A. Gallivan, and R. Bramley, *CIMGS: An incomplete orthogonal factorization preconditioner*, SIAM J. Sci. Comput., 18 (1997), pp. 516–536.
[44] Z. Zlatev, *Computational Methods for General Sparse Matrices*, Kluwer, Dordrecht, the Netherlands, 1991.