A column approximate minimum degree ordering algorithm [*]

Timothy A. Davis [†]   John R. Gilbert [‡]
Stefan I. Larimore [§]   Esmond G. Ng [¶]

October 16, 2000

Abstract

Sparse Gaussian elimination with partial pivoting computes the factorization $\mathbf{PAQ} = \mathbf{LU}$ of a sparse matrix $\mathbf{A}$, where the row ordering $\mathbf{P}$ is selected during factorization using standard partial pivoting with row interchanges. The goal is to select a column preordering, $\mathbf{Q}$, based solely on the nonzero pattern of $\mathbf{A}$ such that the factorization remains as sparse as possible, regardless of the subsequent choice of $\mathbf{P}$. The choice of $\mathbf{Q}$ can have a dramatic impact on the number of nonzeros in $\mathbf{L}$ and $\mathbf{U}$. One scheme for determining a good column ordering for $\mathbf{A}$ is to compute a symmetric ordering that reduces fill-in in the Cholesky factorization of $\mathbf{A}^\mathsf{T}\mathbf{A}$. This approach, which requires the sparsity structure of $\mathbf{A}^\mathsf{T}\mathbf{A}$ to be computed, can be expensive both in

[†]Computer and Info. Sci. and Eng. Dept., University of Florida, Gainesville, FL, USA. email: davis@cise.ufl.edu. http://www.cise.ufl.edu/~davis.

[‡]Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304-1314. email: gilbert@parc.xerox.com.

[§]Microsoft, Inc. email: slarimor@microsoft.com.

[¶]Lawrence Berkeley National Laboratory, One Cyclotron Road, Mail Stop 50F, Berkeley, CA 94720. email: EGNg@lbl.gov.

1

terms of space and time since $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ may be much denser than $\mathbf{A}$. An alternative is to compute $\mathbf{Q}$ directly from the sparsity structure of $\mathbf{A}$; this strategy is is used by Matlab's `colmmd` preordering algorithm. A new ordering algorithm, `colamd`, is presented. It is based on the same strategy but uses a better ordering heuristic. `Colamd` is faster and computes better orderings, with fewer nonzeros in the factors of the matrix.

1 Introduction

Sparse Gaussian elimination with partial pivoting computes the factorization $\mathbf{PAQ} = \mathbf{LU}$ for the sparse nonsymmetric matrix $\mathbf{A}$, where $\mathbf{P}$ and $\mathbf{Q}$ are permutation matrices, $\mathbf{L}$ is a lower triangular matrix, and $\mathbf{U}$ is an upper triangular matrix. Gilbert and Peierls [30] have shown that sparse partial pivoting can be implemented in time proportional to the number of floating-point operations required. The method is used by Matlab when solving a system of linear equations $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is sparse [27]. An improved implementation is in a sparse matrix package, SuperLU [11]. The solution process starts by finding a sparsity-preserving permutation $\mathbf{Q}$. Next, the permutation $\mathbf{P}$ is selected during numerical factorization using standard partial pivoting with row interchanges. The permutation $\mathbf{P}$ is selected without regard to sparsity. Our goal is to compute a sparsity-preserving permutation $\mathbf{Q}$ solely from the pattern of $\mathbf{A}$ such that the LU factorization $\mathbf{PAQ} = \mathbf{LU}$ remains as sparse as possible, regardless of the subsequent choice of $\mathbf{P}$. Our resulting code has been incorporated in SuperLU and in Matlab Version 6.

Section 2 provides the theoretical background for our algorithm. In Section 3, we describe symbolic LU factorization (with no column interchanges) as a precursor to the `colamd` ordering algorithm presented in Section 4. Section 4 also describes the various metrics for selecting columns that we evaluated in the design of the code. Experimental results for square nonsymmetric matrices, rectangular matrices, and symmetric matrices are presented in Section 5. Section 6 presents our conclusions, and describes how to obtain the `colamd` and `symamd` codes.

Our notation is as follows. Set subtraction is denoted by the "\" operator. We use $|...|$ to denote either the absolute value of a scalar, the number of nonzero entries in a matrix, or the size of a set. The usage will be clear in context. The structure of a matrix $\mathbf{A}$, denoted by $\texttt{Struct}(\mathbf{A})$, is a set that contains the locations of the nonzero entries in $\mathbf{A}$; that is, $\texttt{Struct}(\mathbf{A}) = \{(i,j) : \mathbf{A}_{ij} \neq 0\}$. Throughout this paper, and in the algorithms, we assume that exact numerical cancellations do not occur. Some entries in the matrix may happen to become zero during factorization (because of accidental cancellation); we still refer to these as "nonzero" entries.

2 $\mathbf{A}^\top\mathbf{A}$ Orderings

Let $\mathbf{A}$ be a given nonsymmetric matrix and assume that it is nonsingular. It follows from Duff [14] that the rows (or the columns) of $\mathbf{A}$ can be permuted so that the diagonal entries of the permuted $\mathbf{A}$ are all nonzero. We therefore assume throughout this paper that the given matrix $\mathbf{A}$ has been permuted accordingly so that it has a zero-free diagonal.

Suppose that $\mathbf{L}$ and $\mathbf{U}$ are the triangular factors obtained when Gaussian elimination with partial pivoting is applied to $\mathbf{A}$. That is, $\mathbf{PA} = \mathbf{LU}$, for some row permutation $\mathbf{P}$. Consider the symmetric positive definite matrix $\mathbf{A}^\top\mathbf{A}$, which has a Cholesky factorization $\mathbf{A}^\top\mathbf{A} = \mathbf{L}_C\mathbf{L}_C^\top$ with $\mathbf{L}_C$ being lower triangular. George and Ng [25] showed that if $\mathbf{A}$ has a zero-free diagonal, then the pattern of $\mathbf{L}_C + \mathbf{L}_C^\top$ includes the patterns of $\mathbf{L}$ and $\mathbf{U}$, regardless of the row permutation used in partial pivoting. We summarize the result in the following theorem.

Theorem 2.1 (George and Ng [25]) *Let $\mathbf{A}$ be a nonsingular and nonsymmetric matrix that has a zero-free diagonal. Let $\mathbf{L}_C$ denote the Cholesky factor of $\mathbf{A}^\top\mathbf{A}$. Let $\mathbf{L}$ and $\mathbf{U}$ be the LU factors of $\mathbf{A}$ obtained by partial pivoting. Then* $\mathtt{Struct}(\mathbf{U}) \subseteq \mathtt{Struct}(\mathbf{L}_C^\top)$*, and the entries within each column of $\mathbf{L}$ can be rearranged to yield a triangular matrix $\hat{\mathbf{L}}$ with* $\mathtt{Struct}(\hat{\mathbf{L}}) \subseteq \mathtt{Struct}(\mathbf{L}_C)$*.*

Gilbert and Ng [28] also showed that the bound on $\mathbf{U}$ is tight when $\mathbf{A}$ is a strong Hall matrix.[1]

Theorem 2.2 (Gilbert and Ng [28]) *Let $\mathbf{A}$ be a nonsingular and nonsymmetric matrix that has a zero-free diagonal. Assume that $\mathbf{A}$ is strong Hall. Let $\mathbf{L}_C$ denote the Cholesky factor of $\mathbf{A}^\top\mathbf{A}$. Let $\mathbf{L}$ and $\mathbf{U}$ be the LU factors of $\mathbf{A}$ obtained by partial pivoting. For any choice of $(i,j) \in \mathtt{Struct}(\mathbf{L}_C^\top)$, there exists an assignment of numerical values to the nonzero entries of $\mathbf{A}$ such that $\mathbf{U}_{ij} \neq 0$.*

It is well known that the sparsity of $\mathbf{L}_C$ depends drastically on the way in which the rows and columns of $\mathbf{A}^\top\mathbf{A}$ are permuted [23]. Thus, Theorems 2.1

---

[1]A matrix $\mathbf{A}$ is strong Hall if every set of k columns of $\mathbf{A}$, $1 \leq k \leq n-1$, contain at least $k+1$ nonzero rows. A strong Hall matrix is irreducible. See Coleman et al. [6] for details.

and 2.2 suggest a way to reduce the amount of fill in $\mathbf{L}$ and $\mathbf{U}$ [25, 26]. Given a matrix $\mathbf{A}$, we first form the pattern of $\mathbf{A}^\mathsf{T}\mathbf{A}$. Then we compute a symmetric ordering $\mathbf{Q}$ of $\mathbf{A}^\mathsf{T}\mathbf{A}$ to reduce the amount of fill in $\mathbf{L}_C$. Finally, we apply the permutation $\mathbf{Q}$ to the columns of $\mathbf{A}$.

The main disadvantage with the approach above is the cost of forming $\mathbf{A}^\mathsf{T}\mathbf{A}$. Even if $\mathbf{A}$ is sparse, the product $\mathbf{A}^\mathsf{T}\mathbf{A}$ may be dense. Consequently, the time and storage required to form the product may be high. The primary goal of this paper is to describe ordering algorithms that compute $\mathbf{Q}$ from the pattern of $\mathbf{A}$ without explicitly forming $\mathbf{A}^\mathsf{T}\mathbf{A}$.

3 Symbolic LU factorization

Our column ordering technique is based on a symbolic analysis of the conventional outer-product formulation of Gaussian elimination of the n-by-n matrix $\mathbf{A}$ with row interchanges to maintain numerical stability. Let $\mathbf{A}^{(0)} = \mathbf{A}$. For $k = 1, 2, \ldots, n - 1$, let $\mathbf{A}^{(k)}$ denote the bottom right $(n - k)$-by-$(n - k)$ submatrix of $\mathbf{A}^{(k-1)}$ *after* Gaussian elimination is performed. We assume that the rows and columns of $\mathbf{A}^{(k)}$ are labeled from $k + 1$ to n.

At the k-th step of Gaussian elimination, one finds the entry with the largest magnitude in column k of $\mathbf{A}^{(k-1)}$ and swaps rows to place this entry on the diagonal. Column k of $\mathbf{L}$ is then a scaled version of column k of $\mathbf{A}^{(k-1)}$, and row k of $\mathbf{U}$ is row k of $\mathbf{A}^{(k-1)}$. The outer product of the column k of $\mathbf{L}$ and the row k of $\mathbf{U}$ is subtracted from $\mathbf{A}^{(k-1)}$ to give $\mathbf{A}^{(k)}$. The result is the factorization $\mathbf{P}\mathbf{A} = \mathbf{L}\mathbf{U}$, where $\mathbf{P}$ is the permutation matrix determined by the row interchanges. When $\mathbf{A}$ is sparse, the update using the outer product may turn some of the zero entries into nonzero. These new nonzero entries are referred to as *fill-in*. The amount of fill-in that occurs depends on the order in which the columns of $\mathbf{A}$ are eliminated [23]. Our goal, therefore, is to find a column ordering $\mathbf{Q}$ of the matrix $\mathbf{A}$ prior to numerical factorization, so that the LU factorization of the column-permuted matrix $\mathbf{A}\mathbf{Q}$ is sparser than the LU factorization of the original matrix $\mathbf{A}$, regardless of how $\mathbf{P}$ is chosen. Another application of column orderings is sparse QR factorization [33].

In order to control the fill-in that occurs when $\mathbf{A}$ is sparse, we need to know the nonzero patterns of the matrices $\mathbf{A}^{(k)}$, for each k from 0 to $n - 1$. From this, we can determine a column ordering $\mathbf{Q}$ that attempts to keep the factorization of $\mathbf{A}\mathbf{Q}$ sparse as the factorization progresses.

We must compute symbolic patterns $\mathcal{L}$, $\mathcal{U}$, and $\mathcal{A}^{(k)}$ that account for all

possible row interchanges that may occur during numerical factorization; the actual structures of $\mathbf{L}$, $\mathbf{U}$, and $\mathbf{A}^{(k)}$ during the factorization of $\mathbf{PA}$ with any specific row permutation $\mathbf{P}$ will be subsets of these patterns.

Let $\mathcal{L}_k$ and $\mathcal{U}_k$ denote the nonzero patterns of column k of $\mathcal{L}$ and row k of $\mathcal{U}$, respectively. For $i > k$, let $\mathcal{R}_i^{(k)}$ denote the set of column indices of nonzero entries in row i of $\mathcal{A}^{(k)}$,

$$\mathcal{R}_i^{(k)} = \{j > k : a_{ij}^{(k)} \neq 0\}.$$

Similarly, for $j > k$, let $\mathcal{C}_j^{(k)}$ denote the set of row indices of nonzero entries in column j of $\mathcal{A}^{(k)}$,

$$\mathcal{C}_j^{(k)} = \{i > k : a_{ij}^{(k)} \neq 0\}.$$

Note that these definitions imply

$$j \in \mathcal{R}_i^{(k)} \Leftrightarrow i \in \mathcal{C}_j^{(k)} \tag{1}$$

for all k, i, and j.

We now describe how to compute $\mathcal{L}_k$, $\mathcal{U}_k$, and $\mathcal{A}_k$ in order of increasing k. We begin with $\mathcal{A}^{(0)} = \texttt{Struct}(\mathbf{A}^{(0)})$, and with $\mathbf{A}^{(0)} = \mathbf{A}$.

Using the assumption that $\mathbf{A}$ has a zero-free diagonal, it is easy to see that the pattern of the k-th column of $\mathbf{L}$ is simply column k of $\mathbf{A}^{(k-1)}$, so we may take

$$\mathcal{L}_k = \mathcal{C}_k^{(k-1)},$$

regardless of how the nonzero numerical values are interchanged. Consider the possible nonzero pattern of the k-th row of $\mathbf{U}$ after partial pivoting. Any row i can be selected as the pivot row for which $a_{ik}^{(k-1)}$ is nonzero. This means the potential candidate pivot rows correspond to the set $\mathcal{L}_k$. The pattern of the k-th pivot row is bounded by the union of all candidate pivot rows [26], so we may take

$$\mathcal{U}_k = \bigcup_{i \in \mathcal{L}_k} \mathcal{R}_i^{(k-1)}.$$

If an arbitrary row $r \in \mathcal{L}_k$ is selected as the pivot row, then $\mathcal{R}_i^{(k-1)} \subseteq \mathcal{U}_k$ for all $i \in \mathcal{L}_k$, so that $\mathcal{U}_k$ can accommodate any row interchanges due to numerical partial pivoting.

In the outer product step, multiples of the pivot row are added to each row i in the pivot column. Since $\mathcal{R}_i^{(k-1)} \subseteq \mathcal{U}_k$, the pattern of each row i in $\mathcal{L}_k \setminus \{k\}$ of the matrix in $\mathbf{A}^{(k)}$ after this outer product step must be contained

in the set $\mathcal{U}_k \setminus \{k\}$. Since we do not know which row is the pivot row, we can bound the pattern of row i of $\mathbf{A}^{(k)}$ with

$$\mathcal{R}_i^{(k)} = \mathcal{U}_k \setminus \{k\}.$$

Note that all rows i in $\mathcal{L}_k \setminus \{k\}$ now have the same pattern, and we can save time and space by not storing all of them. Any step that modifies one of these rows will modify all of them, and the rows will again have the same nonzero pattern. Let $p = \min \mathcal{U}_k \setminus \{k\}$. At step p, $\mathcal{L}_k \setminus \{k\} \subseteq \mathcal{L}_p$, and so $\mathcal{R}_i^{(p)} = \mathcal{U}_p \setminus \{p\}$ for all i in $\mathcal{L}_k \setminus \{k\}$. Thus, at step k, we can replace all rows i in $\mathcal{L}_k \setminus \{k\}$ with a single *super-row* $\mathcal{R}_r^{(k)} = \mathcal{U}_k \setminus \{k\}$ that represents the pattern all of the rows i in the set $\mathcal{L}_k \setminus \{k\}$. The integer r is an arbitrary place-holder; in our symbolic factorization algorithm we choose an arbitrary row index from the set $\mathcal{C}_k^{(k-1)}$. We define $[r]$ as the set of rows (with size $|[r]|$) represented by the super-row r. When the symbolic factorization starts, every row belongs to exactly one set; i.e., $[r] = \{r\}$. To compute the symbolic update, we need only place the single representative, r, into the set $\mathcal{C}_j^{(k)}$, and we can remove the discarded rows $\mathcal{L}_k \setminus \{r\}$. Sherman [43] introduced the concept of super-columns in the context of sparse Cholesky factorization.

With these simplifications, at step k the sets $\mathcal{R}$ and $\mathcal{C}$ represent a bound on the pattern of $\mathbf{A}^{(k)}$ for LU factorization. They are also a quotient graph representation [17, 21] of the matrix obtained at the k-th step of the Cholesky factorization of $\mathbf{A}^\mathsf{T}\mathbf{A}$. Our column preordering method does not require the patterns of $\mathcal{L}$ and $\mathcal{U}$, so these can be discarded. As a result, the algorithm shown in Figure 1 requires only $O(|\mathbf{A}|)$ space. Superscripts on $\mathcal{R}$ and $\mathcal{C}$ are dropped for clarity.

The initial storage required is $O(|\mathbf{A}|)$. At step k,

$$|\mathcal{R}_r| < \sum_{i \in C_k} |\mathcal{R}_i|$$

because $\mathcal{R}_r$ is computed as

$$\mathcal{R}_r = \left(\bigcup_{i \in \mathcal{C}_k} \mathcal{R}_i \right) \setminus \{k\}. \tag{2}$$

The sets $\mathcal{R}_i$ for all i in $\mathcal{C}_k \setminus \{r\}$ are then discarded. When the outer product is formed, each column $\mathcal{C}_j$ (where $j \in \mathcal{R}_r$) reduces in size or stays the same size. If $j \in \mathcal{R}_r$, then from (2) this implies $\exists i \in \mathcal{C}_k$ such that $j \in \mathcal{R}_i$. From

7

Symbolic LU factorization algorithm

Let $\mathcal{R}_i = \{j : a_{ij} \neq 0\}$ for all i

Let $\mathcal{C}_j = \{i : a_{ij} \neq 0\}$ for all j

for $k = 1$ **to** n **do**

   determine pattern for pivot row and column:

   Let r be an arbitrary index in $\mathcal{C}_k$

   $\mathcal{R}_r = \left(\bigcup_{i \in \mathcal{C}_k} \mathcal{R}_i \right) \setminus \{k\}$

   symbolic outer product update:

   for $i \in \mathcal{C}_k \setminus \{r\}$ **do**

      $\mathcal{R}_i = \emptyset$

   end for

   for $j \in \mathcal{R}_r$ **do**

      $\mathcal{C}_j = (\mathcal{C}_j \setminus \mathcal{C}_k) \cup \{r\}$

   end for

   $\mathcal{C}_k = \emptyset$

end for

Figure 1: Symbolic LU factorization algorithm

(1), $j \in \mathcal{R}_i$ implies that $i \in \mathcal{C}_j$. Thus, $\exists i \in \mathcal{C}_k \cap \mathcal{C}_j$, which implies that $|\mathcal{C}_j \setminus \mathcal{C}_k| < |\mathcal{C}_j|$. Thus, each step strictly reduces the total storage required for the pattern of $\mathbf{A}^{(k)}$.

The total time taken is dominated by the symbolic update. Computing the pivot rows $\mathcal{R}_r$ for the entire algorithm takes a total of $O(|\mathcal{U}|)$ time, since $\mathcal{R}_r = \mathcal{U}_k \setminus \{k\}$ is the pattern of the off-diagonal part of row k of $\mathcal{U}$, and since $\mathcal{R}_r$ is discarded when it is included in a set union for a subsequent pivot row. Modifying a single column $\mathcal{C}_j$ in the symbolic update takes no more than $O(|\mathbf{A}_{*j}|)$ time, since $|\mathcal{C}_j| \leq |\mathbf{A}_{*j}|$. Column j is modified at most once for each row $\mathcal{U}_k$ that contains column index j. Thus, column j is modified at most $|\mathcal{U}_{*j}|$ times during the entire algorithm. Here $\mathcal{U}_{*j}$ is the upper bound pattern on column j of the matrix $\mathbf{U}$ for any row permutation $\mathbf{P}$ due to partial pivoting. Thus, the total time taken is

$$O\left(\sum_{j=1}^{n} |\mathbf{A}_{*j}||\mathcal{U}_{*j}|\right),$$

where the nonzero pattern of $\mathcal{U}$ can accommodate any row permutation $\mathbf{P}$ due to partial pivoting. This is the same time required to compute the sparse matrix product $\mathbf{A}\mathcal{U}^\mathsf{T}$, and is typically much less than the time required for numerical factorization, which is equal to the time required to compute the sparse matrix product $\mathbf{L}$ times $\mathbf{U}$ [31],

$$O\left(\sum_{k=1}^{n} |\mathbf{L}_{*k}||\mathbf{U}_{k*}|\right).$$

Faster methods exist to compute this upper bound on symbolic factorization with partial pivoting. George and Ng's method [26] for computing the patterns of $\mathcal{L}$ and $\mathcal{U}$ takes only $O(|\mathcal{L}| + |\mathcal{U}|)$ time if the column ordering is fixed. The method does not produce the nonzero patterns of $\mathcal{A}^{(k)}$, however, so it cannot be used as a basis for the `colamd` algorithm described in the next section.

4 The `colamd` ordering algorithm

We now present a column ordering method that is based on the symbolic LU factorization algorithm presented in the last section and that has the same time complexity and storage requirements. At step k, we select a pivot

column c to minimize some metric on all of the candidate pivot columns as an attempt to reduce fill-in. Columns c and k are then exchanged.

The presence of dense (or nearly dense) rows or columns in $\mathbf{A}$ can greatly affect both the ordering quality and run time. A single dense row in $\mathbf{A}$ renders all our bounds useless; the bound on the nonzero pattern of $\mathbf{A}^{(1)}$ is a completely dense matrix. We can hope that this dense row will not be selected as the first pivot row during numerical factorization, and withhold the row from the ordering algorithm. Dense columns do not affect the ordering quality, but they do increase the ordering time. A single dense column increases the ordering time by $O(n^2)$. Dense columns are withheld from the symbolic factorization and ordering algorithm, and placed last in the column ordering $\mathbf{Q}$. Determining how dense a row or column should be for it to be ignored is problem dependent. We used the same default threshold used by Matlab, 50%, which is probably too high for most matrices.

Taking advantage of super-columns can greatly reduce the ordering time. In the symbolic update step, we look at all columns j in the set $\mathcal{R}_r$. If any two or more columns have the same pattern (tested via a hash function [2]), they are merged into a single super-column. This saves both time and storage in subsequent steps. Selecting a super-column c allows us to *mass-eliminate* [24] all columns $[c]$ represented by the super-column c, and we skip ahead to step $k + |[c]|$ of symbolic factorization. If the pattern of column or super-column $j \in \mathcal{R}_r$ becomes the same as the pivot column pattern ($\mathcal{C}_j = \{r\}$) after the symbolic update, it can be eliminated immediately. Since the columns $[c]$ represented by a super-column keep the same nonzero pattern until they are eliminated, we only need to maintain the degree, or other column selection metric, for the representative column c.

With super-rows, the size of the set $\mathcal{C}_j$ differs from the sum of the numbers of rows represented by the super-rows i in the set $\mathcal{C}_j$. Similarly, $|\mathcal{R}_i|$ is the number of super-columns in row i, which is not the same as the number of columns represented by the super-columns in $\mathcal{R}_i$. Let $\|\mathcal{R}_i\|$ denote the sum of the numbers of rows represented by the super-columns j in $\mathcal{R}_i$,

$$\|\mathcal{R}_i\| = \sum_{j \in \mathcal{R}_i} |[j]|.$$

Similarly, we define $\|\mathcal{C}_j\|$ as

$$\|\mathcal{C}_j\| = \sum_{i \in \mathcal{R}_j} |[i]|.$$

Without super-columns, $|\mathcal{R}_i|$ and $\|\mathcal{R}_i\|$ are the same. Similarly, $|\mathcal{C}_j| = \|\mathcal{C}_j\|$ when there are no super-rows.

By necessity, the choice of the pivot column c is a heuristic, since obtaining an ordering with minimum fill-in is an NP-complete problem [45]. Several strategies are possible. Each selects a column c that minimizes some metric, described below. For most of these methods, we need to compute the initial metric for each column prior to symbolic factorization and ordering, and then recompute the metric for each column j in the pivot row pattern $\mathcal{R}_r$ at step k.

1. **Exact external row degree:**

   Select c to minimize the size of the resulting pivot row,

   $$\|\mathcal{R}_r\| = \left\| \left(\bigcup_{i \in \mathcal{C}_c} \mathcal{R}_i \right) \setminus \{c\} \right\|.$$

   (We exclude the pivot column itself, thus computing an *external* row degree [37].) The exact degree is expensive to compute (its time can dominate the symbolic factorization time), and our experience with symmetric orderings is that exact degrees do not provide better orderings than good approximate degrees (such as those from the ordering methods `amdbar` and `mc47bd`) [1]. We thus did not test this method.

2. **Matlab approximate external row degree:**

   The `colmmd` column ordering algorithm in Matlab is based on the symbolic LU factorization algorithm presented in Section 3 [27]. `Colmmd` selects as pivot the column c that minimizes a loose upper bound on the external row degree,

   $$\|\mathcal{R}_r\| \le \sum_{i \in \mathcal{C}_c} (\|\mathcal{R}_i \setminus \{c\}\|).$$

   Note that $c \in \mathcal{R}_i$, so $\|\mathcal{R}_i \setminus \{c\}\| = \|\mathcal{R}_i\| - |[c]|$. Using this bound for the symbolic update does not increase the asymptotic time complexity of the ordering algorithm above that of the symbolic LU factorization algorithm. Computing the Matlab metric for the initial columns of $\mathbf{A}$ is very fast, taking only $O(|\mathbf{A}|)$ time.

11

3. **AMD approximate external row degree:**

   The AMD algorithm for ordering symmetric matrices prior to a Cholesky factorization is based on a bound on the external row degree that is tighter than the Matlab bound [1]. It was first used in the nonsymmetric-pattern multifrontal method (UMFPACK), to compute the ordering during numerical factorization [8, 9]. In the context of a column ordering algorithm to bound the fill-in for the sparse partial pivoting method, the bound on $\|\mathcal{R}_r\|$ is

   $$\|\mathcal{R}_r\| \leq \|\mathcal{R}_s \setminus \{c\}\| + \sum_{i \in \mathcal{C}_c \setminus \{s\}} (\|\mathcal{R}_i \setminus \mathcal{R}_s\|), \tag{3}$$

   where $\mathcal{R}_s$ is the most recent pivot row that modified $\mathcal{C}_c$ in the symbolic update, and thus $s \in \mathcal{C}_c$. To compute the AMD metric on the initial columns of $\mathbf{A}$, we select an arbitrary $s \in \mathcal{C}_c$ to compute the bound for column c. Computing the initial AMD metric takes the same amount of time as it takes to compute the sparse matrix product $\mathbf{A}\mathbf{A}^\mathsf{T}$. Although this is costly, it does not increase the asymptotic time of our algorithm. The time to compute this bound during the symbolic update phase is asymptotically the same as computing the Matlab bound.

4. **Size of the Householder update:**

   The symbolic LU factorization also computes the pattern of $\mathbf{R}$ for a QR factorization of $\mathbf{A}$ [6, 32]. At step k, the size of the Householder update is $\|\mathcal{C}_k\|$-by-$\|\mathcal{R}_r\|$. The term $\|\mathcal{C}_k\|$ is known exactly; the $\|\mathcal{R}_r\|$ term can be computed or approximated using any of the above methods. The method we tested selected c to minimize the product of $\|\mathcal{C}_c\|$ and the AMD approximation of $\|\mathcal{R}_r\|$ in (3). We tested and then discarded this method, since it gave much worse orderings than the other methods.

5. **Approximate Markowitz criterion:**

   The Markowitz heuristic selects as pivot the entry a_{ij} that minimizes the product of the degrees of row i and column j, which is the amount of work required in the subsequent outer product step [38]. The criterion assumes that the first $k-1$ pivot rows and columns have been selected. In our case, we do not know the exact pivot row ordering yet, so we do not know the true row or column degrees. In our tests, we selected c to minimize $\|\mathcal{C}_c\|$ times $\max_{i \in \mathcal{C}_c} \|\mathcal{R}_i\|$. This is a tighter upper bound on the

actual pivot row degree if column c is selected as the k-th pivot column and row i is selected as the k-th pivot row. Although $\mathcal{R}_i$ does not bound the pattern $\mathcal{R}_r$ of the k-th pivot row for arbitrary row partial pivoting, $\|\mathcal{R}_i\|$ does bound the size of the actual pivot row once the row ordering is selected. Since $\|\mathcal{C}_c\|$ and $\|\mathcal{R}_i\|$ are known exactly, no approximations need to be used. We tested and then discarded this method, since it gave much worse orderings than the other methods.

6. **Approximate deficiency:**

   The *deficiency* of a pivot is the number of new nonzero entries that would be created if that pivot is selected; selecting the pivot with least deficiency leads to a minimum deficiency ordering algorithm. Exact deficiency is very costly to compute. Approximate minimum deficiency ordering algorithms have been successfully used for symmetric matrices, in the context of Cholesky factorization [40, 42]. In an nonsymmetric context, the deficiency of column c can be bounded by

   $$\|\mathcal{C}_c\|\|\mathcal{R}_r\| - \left(\sum_{i \in \mathcal{C}_c} |[i]|(\|\mathcal{R}_i\| - |[c]|) \right).$$

   Any new nonzeros are limited to the $\|\mathcal{C}_c\|$-by-$\|\mathcal{R}_r\|$ Householder update. Each super-row $i \in \mathcal{C}_c$, however, is contained in this submatrix and thus reduces the possible fill-in by the number of nonzeros it represents. The $\|\mathcal{C}_c\|$ and $\|\mathcal{R}_i\|$ terms are known exactly; we used the AMD approximation for $\|\mathcal{R}_r\|$, from (3).

   We tested one variant of approximate deficiency [35, 36]. The initialization phase computes the approximate deficiency of all columns, and the approximate deficiency is recomputed any time a column is modified. No aggressive row absorption is used (discussed below), since it worsens the approximation. The experimental results were mixed. Compared to our final `colamd` variant, approximate deficiency was better for about 60% of the matrices in our large test suite. When it was worse than `colamd` it was sometimes much worse, and vice versa.

As a by-product of the AMD row degree computation, we compute the sizes of the set differences $\|\mathcal{R}_i \setminus \mathcal{R}_r\|$ when super-row r is the bound on the pivot row at step k, for all rows i remaining in the matrix. If we find that this set difference is zero, $\mathcal{R}_i$ is a subset of row $\mathcal{R}_r$. The presence of row i does

not affect the exact row degree, although it does affect the Matlab and AMD approximations. Row i can be deleted, and absorbed into $[r]$. We refer to the deletion of $\mathcal{R}_i$ when $j \notin \mathcal{R}_r$ as *aggressive row absorption*. It is similar to element absorption, introduced by Duff and Reid [17]. If AMD row degrees are computed as the initial metric, then *initial* aggressive row absorption can occur in that phase as well. Aggressive row absorption reduces the run time, since it costs almost nothing to detect this condition when the AMD metric is used, and results in fewer rows to consider in subsequent steps.

We tested sixteen variants of `colamd`, based on all possible combinations of the following design decisions:

1. Initial metric: Matlab or AMD approximation.

2. Metric computed during the symbolic outer product update: Matlab or AMD approximation.

3. With or without initial aggressive row absorption.

4. With or without aggressive row absorption during the symbolic factorization.

Note that some of the combinations are somewhat costly, since aggressive row absorption is easy when using the AMD row degrees, but difficult when using the Matlab approximation.

Since the AMD metric was shown to be superior to the Matlab approximation in the context of minimum degree orderings for sparse Cholesky factorization [1], we expected the four variants based solely on the AMD metric to be superior, so much so that we did not intend to test all sixteen methods. To our surprise, we found that better orderings were obtained with an initial Matlab metric and an AMD metric during the symbolic update. We discovered this by accident. A bug in our initial computation of the AMD metric resulted in the Matlab approximation being computed instead. This "bug" gave us better orderings, so we kept it. Using an initial Matlab metric gave, on average, orderings with 8% fewer floating-point operations in the subsequent numerical factorization than using an initial AMD metric. The initial Matlab metric is also faster to compute.

The version of our ordering algorithm that we recommend, which we now simply call `colamd`, uses the initial Matlab metric, the AMD metric for the symbolic update, no initial aggressive row absorption, and aggressive row absorption during the symbolic update. Since the AMD metric is incompatible

with both incomplete degree update [18, 19] and multiple elimination [37], it uses neither strategy. In multiple elimination, a set of pivotal columns is selected such that the symbolic update of any column in the set does not affect the other columns in the set. The selection of this set reduces the number of symbolic updates that must be performed.

Matlab's `colmmd` ordering algorithm uses the Matlab metric throughout, multiple elimination with a relaxed threshold, super-rows and super-columns, mass elimination, and aggressive row absorption. Aggressive row absorption is not free, as it is in `colamd`. Instead, an explicit test is made every three stages of multiple elimination. Super-columns are searched for every three stages, by default. Rows more than 50% dense are ignored. Since the Matlab metric can lead to lower quality orderings than an exact metric, options are provided for selecting an exact external row degree metric, modifying the multiple elimination threshold, changing the frequency of super-column detection and aggressive row absorption, and changing the dense row threshold. Selecting `spparms ('tight')` in Matlab improves the ordering computed by `colmmd`, but at high cost in ordering time.

5   Experimental results

We tested our `colamd` ordering algorithm with three sets of matrices: square nonsymmetric matrices, rectangular matrices, and symmetric positive definite matrices. Our test set is the entire University of Florida sparse matrix collection [7], which includes the Harwell/Boeing test set [15, 16], the linear programming problems in Netlib at `http://www.netlib.org` [13], as well as many other matrices. We exclude complex matrices, nonsymmetric matrices for which only the pattern was provided, and unassembled finite element matrices. Some matrices include explicit zero entries in the description of their pattern. Since we ignore numerical cancellation, we included these entries when finding the ordering and determining the resulting factorization. Each method is compared by ordering time and quality (number of nonzeros in the factors, and number of floating-point operations to compute the factorization). Although we tested all matrices in the collection, we present here a summary of only some of the larger problems (those for which the best ordering resulted in a factorization requiring 10^7 operations or more to compute). Complete results are presented in Larimore's thesis [36], available as a technical report at `http://www.cise.ufl.edu/`. We performed these exper-

iments on a Sun Ultra Enterprise 4000/5000 with 2 GB of main memory and eight 248 Mhz UltraSparc-II processors (only one processor was used). The code was written in ANSI/ISO C and compiled using the Solaris C compiler (via Mathwork's `mex` shell script for interfacing Matlab to software written in C), with strict ANSI/ISO compliance.

5.1 Square nonsymmetric matrices

For square nonsymmetric matrices, we used `colamd` to find a column pre-ordering $\mathbf{Q}$ for sparse partial pivoting. These results are compared with `colmmd` with default option settings, and with `amdbar` [1] applied to the pattern of $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ (ignoring numerical cancellation, and withholding the same dense rows and columns from $\mathbf{A}$ that were ignored by `colamd` and `colmmd`). The `amdbar` routine is the same as `mc47bd` in the Harwell Subroutine Library, except that it does not perform aggressive row absorption. Both `amdbar` and `mc47bd` use the AMD-style approximate degree. For sparse partial pivoting with the matrices in our test set, `amdbar` provides slightly better orderings than `mc47bd`. After finding a column ordering, we factorized the matrix $\mathbf{AQ}$ with the SuperLU package [11]. This package was chosen since `colamd` was written to replace the column ordering in the current version of SuperLU. SuperLU is based on the BLAS [12].[2]

We excluded matrices that can be permuted to upper block triangular form [14] with a substantial improvement in factorization time, for two reasons: (1) our bounds are not tight if the matrix $\mathbf{A}$ is not strong Hall, and (2) SuperLU does not take advantage of reducibility to block upper triangular form. Such matrices should be factorized by ordering and factorizing each irreducible diagonal submatrix, after permuting the matrix to block triangular form. Our resulting test set had 106 matrices, all requiring more than 10^7 or more operations to factorize.

A representative sample is shown in in Table 1, sorted according to `colamd` versus `colmmd` ordering quality.[3] Table 2 reports the ordering time of `colamd`,

---

[2]We used the BLAS routines provided in SuperLU because factorization time is a secondary measure, and because we had difficulty using the Sun-optimized BLAS from a Matlab-callable driver.

[3]To obtain the sample, we sorted the 106 matrices according to the relative floating-point operation count for the `colamd` ordering and the `colmmd` ordering, and then selected every 8th matrix. We also included TWOTONE, the matrix with the largest dimension in the test set.

Table 1: Unsymmetric matrices

matrix	n	nnz	description
GOODWIN	7320	324784	finite-element, fluid dynamics, Navier-Stokes and elliptic mesh (R. Goodwin)
RAEFSKY2	3242	294276	incompressible flow in pressure-driven pipe (A. Raefsky)
TWOTONE	120750	1224224	frequency-domain analysis of nonlinear analog circuit [20]
RMA10	46835	2374001	3D computational fluid dynamics (CFD), Charleston Harbor (Steve Bova)
LHR17C	17576	381975	light hydrocarbon recovery problem [47]
AF23560	23560	484256	airfoil [3]
EPB2	25228	175027	plate-fin heat exchanger with flow redistribution (D. Averous)
RDIST3A	2398	61896	chemical process separation [46]
GARON2	13535	390607	2D finite-element, Navier-Stokes (A. Garon)
GRAHAM1	9035	335504	Galerkin finite-element, Navier-Stokes, two-phase fluid flow (D. Graham)
CAVITY25	4562	138187	driven cavity, finite-element (A. Chapman)
EX20	2203	69981	2D attenuation of a surface disturbance, nonlinear CFD (Y. Saad)
WANG1	2903	19093	electron continuity, 3D diode [39]
EX14	3251	66775	2D isothermal seepage flow (Y. Saad)
EX40	7740	458012	3D die swell problem on a square die, computational fluid dynamics (Y. Saad)

colmmd, and amdbar (the time to compute the pattern of $\mathbf{A}^\mathsf{T}\mathbf{A}$ is included in the amdbar time). For comparison, the last column reports the SuperLU factorization time, using the colamd ordering. Table 3 gives the resulting number of nonzeros in $\mathbf{L} + \mathbf{U}$, and the floating-point operations required to factorize the permuted matrix, for each of the ordering methods. The median results reported in this paper are for just the representative samples, not the full test sets. The results for the samples and the full test sets are similar.

Colamd is superior to the other methods for these matrices. Colamd was typically 3.9 times faster than colmmd and 2.1 times faster than amdbar. For two matrices, colmmd took more time to order the matrix than SuperLU took to factorize it. The orderings found by colmmd result in a median increase of

Table 2: Ordering and factorization time, in seconds

matrix	colamd	colmmd	amdbar	SuperLU
GOODWIN	0.31	0.42	1.51	44.65
RAEFSKY2	1.11	1.58	2.11	62.36
TWOTONE	5.88	70.82	14.15	306.85
RMA10	2.37	25.71	12.07	120.67
LHR17C	1.62	15.22	3.08	5.80
AF23560	6.64	115.99	1.72	108.35
EPB2	0.44	2.62	0.61	16.32
RDIST3A	0.03	0.07	0.21	0.44
GARON2	0.61	1.19	1.29	25.12
GRAHAM1	0.43	0.58	1.50	31.63
CAVITY25	0.12	0.21	0.46	2.95
EX20	0.08	0.17	0.24	1.13
WANG1	0.07	0.27	0.08	2.65
EX14	0.10	0.53	0.20	2.47
EX40	2.34	13.58	2.61	27.21
Median time relative to colamd	-	3.9	2.1	37.1

Table 3: Ordering quality, as factorized by SuperLU

| matrix | Nonzeros in $\mathbf{L} + \mathbf{U}$ (10^3) | | | Flop. count (10^6) | | |
	colamd	colmmd	amdbar	colamd	colmmd	amdbar
GOODWIN	5634	3103	2734	1909	665	498
RAEFSKY2	3684	3236	2877	2374	2012	1682
TWOTONE	21030	19624	21280	8848	8923	8578
RMA10	18540	17544	15624	5064	5303	4041
LHR17C	1715	1767	1717	111	120	112
AF23560	12110	13333	13131	4515	5350	5052
EPB2	3186	3547	3135	646	839	620
RDIST3A	233	264	231	11	15	11
GARON2	5179	5120	5128	1061	1563	1505
GRAHAM1	4920	5344	4518	1333	2115	1414
CAVITY25	1146	1347	1210	130	219	178
EX20	483	589	464	48	86	50
WANG1	632	868	752	120	242	174
EX14	711	1012	885	91	214	157
EX40	4315	8537	6072	1075	5369	2606
Median result relative to colamd	-	1.10	0.99	-	1.36	1.05

10% in nonzeros in the LU factors and 36% in floating-point operations, as compared to `colamd`. The ordering quality of `colamd` and `amdbar` are similar, although there are large variations in both directions in a small number of matrices. For a few matrices (in our larger test set, not in Table 1) `amdbar` requires more space to store $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ than SuperLU requires to factorize the permuted matrix.

5.2   Rectangular matrices

For m-by-n rectangular matrices with $m > n$, we found a column ordering $\mathbf{Q}$ for the Cholesky factorization of $(\mathbf{AQ})^{\mathsf{T}}(\mathbf{AQ})$, which is one method for solving a least squares problem. If $m < n$, we found a row ordering $\mathbf{P}$ for the Cholesky factorization of $(\mathbf{PA})\mathbf{D}^2(\mathbf{PA})^{\mathsf{T}}$, which arises in interior point methods for solving linear programming problems [34, 44]. Here, $\mathbf{D}$ is a diagonal matrix. In the latter case, `colamd` and `colmmd` found a column ordering of $\mathbf{A}^{\mathsf{T}}$ and we used that as the row ordering $\mathbf{P}$. We compared these two methods with `amdbar` on the corresponding matrix, $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ (if $m > n$) or $\mathbf{AA}^{\mathsf{T}}$ (if $m < n$).

Our test set was limited. It included only 37 matrices requiring more than 10^7 operations; all but three of these were Netlib linear programming problems (with $m < n$). A representative selection[4] is shown in Table 4.

We compared the three methods based on their ordering time, number of nonzeros in the factor $\mathbf{L}$, and floating-point operation count required for the Cholesky factorization. These metrics were obtained from `symbfact` in Matlab, a fast symbolic factorization [22, 23, 29]. We did not perform the numerical Cholesky factorization. The time to construct $\mathbf{A}^{\mathsf{T}}\mathbf{A}$ or $\mathbf{AA}^{\mathsf{T}}$ is included in the ordering time for `amdbar`. The results are shown in Tables 5 and 6.

For these matrices, `colamd` was twice as fast as `colmmd` and slightly faster than `amdbar`. All three produced comparable orderings. Each method has a few matrices it does well on, and a few that it does poorly on. `Colamd` and `colmmd` both typically require less storage than `amdbar`, sometimes by several orders of magnitude, because of the need to compute the pattern of $\mathbf{AA}^{\mathsf{T}}$ prior to ordering it with `amdbar`. The size of the Cholesky factors always dominates the size of $\mathbf{AA}^{\mathsf{T}}$, however. The primary advantage of `colamd` and

---

[4]Every 5th matrix from the 37 large matrices, in increasing order of `colamd` versus `colmmd` ordering quality, was chosen.

Table 4: Rectangular matrices (all linear programming problems)

matrix	m	n	nnz	nnz($\mathbf{AA}^\mathsf{T}$)	description
GRAN	2629	2525	20111	60511	British Petroleum operations
NUG08	912	1632	7296	28816	LP lower bound for a quadratic assignment problem [41]
FIT1P	627	1677	9868	274963	fitting linear inequalities to data, min. sum of piecewise-linear penalties (R. Fourer)
KLEIN2	477	531	5062	139985	(E. Klotz)
PILOT87	2030	6680	74949	238624	(J. Stone)
PDS_20	33874	108175	232647	320196	military airlift operations [5]
D2Q06C	2171	5831	33081	56153	(J. Tomlin)

Table 5: Ordering time, in seconds

matrix	colamd	colmmd	amdbar
GRAN	0.04	0.07	0.06
NUG08	0.05	0.06	0.04
FIT1P	0.01	0.01	0.08
KLEIN2	0.01	0.02	0.07
PILOT87	0.56	2.60	0.52
PDS_20	3.86	10.53	3.92
D2Q06C	0.07	0.44	0.08
Median time relative to `colamd`	-	2.00	1.14

Table 6: Ordering quality

matrix	Nonzeros in $\mathbf{L}$ (10^3)			Flop. count (10^6)		
	colamd	colmmd	amdbar	colamd	colmmd	amdbar
GRAN	191	158	143	48	33	27
NUG08	210	202	230	78	70	92
FIT1P	197	197	197	82	82	82
KLEIN2	81	82	80	17	18	17
PILOT87	423	475	406	169	195	164
PDS_20	6827	8159	6929	8598	12640	8812
D2Q06C	175	271	147	35	97	27
Median result relative to `colamd`	-	1.00	0.99	-	1.02	0.98

`colmmd` over `amdbar` in this context is the ability to analyze the Cholesky factorization by finding the ordering and size of the resulting factor $\mathbf{L}$ in space proportional to the number of nonzeros in the matrix $\mathbf{A}$, rather than proportional to the size of the matrix to be factorized ($\mathbf{A}^\mathsf{T}\mathbf{A}$ or $\mathbf{A}\mathbf{A}^\mathsf{T}$).

5.3 Symmetric matrices

For symmetric matrices, Matlab's `symmmd` routine constructs a matrix $\mathbf{M}$ such that the pattern of $\mathbf{M}^\mathsf{T}\mathbf{M}$ is the same as $\mathbf{A}$, and then finds a column ordering of $\mathbf{M}$ using `colmmd`. There is one row in $\mathbf{M}$ for each entry a_{ij} below the diagonal of $\mathbf{A}$, with nonzero entries in column i and j. This method gives a reasonable ordering for the Cholesky factorization of $\mathbf{A}$. We implemented an analogous `symamd` routine that is based on `colamd`.

Our test set included 50 matrices requiring 10^7 or more operations to factorize. The largest was a computational fluid dynamics problem from Ed Rothberg requiring 136 billion operations to factorize (`cfd2`). Our sample test matrices[5] are shown in Table 7. Results from `symamd`, `symmmd`, `amdbar`, and `mc47bd` are shown in Tables 8 and 9. The time to construct $\mathbf{M}$ is included in the ordering time for `symamd` and `symmmd`.

For these 9 matrices, `symamd` was over six times faster than `symmmd`, on

---

[5]Every 7th matrix, in order of increasing ratio of `symamd` to `symmmd` ordering quality, was chosen.

Table 7: Symmetric matrices

matrix	n	nnz	description
PWT	36519	326107	pressurized wind tunnel (R. Grimes)
MSC00726	726	34518	symmetric test matrix from MSC/NASTRAN
BCSSTK38	8032	355460	stiffness matrix, airplane engine component (R. Grimes)
NASA2146	2146	72250	structural engineering problem (NASA Langley)
CFD2	123440	3087898	computational fluid dynamics (E. Rothberg)
3DTUBE	45330	3213618	3D pressure tube (E. Rothberg)
GEARBOX	153746	9080404	ZF aircraft flap actuator (E. Rothberg)
CRYSTM02	13965	322905	crystal, free vibration mass matrix (R. Grimes)
FINAN512	74752	596992	portfolio optimization [4]

Table 8: Ordering time, in seconds

matrix	symamd	symmmd	amdbar	mc47bd
PWT	0.78	3.53	0.45	0.44
MSC00726	0.05	0.24	0.02	0.02
BCSSTK38	0.53	22.45	0.12	0.11
NASA2146	0.08	0.49	0.02	0.02
CFD2	7.36	48.68	3.90	3.71
3DTUBE	6.09	77.75	0.95	0.92
GEARBOX	16.58	454.49	2.94	2.89
CRYSTM02	0.71	1.36	0.28	0.28
FINAN512	1.60	2.61	0.92	0.95
Median time relative to symamd	-	6.13	0.39	0.39

Table 9: Ordering quality

matrix	Nonzeros in $\mathbf{L}$ (10^3)				Cholesky flop. count (10^6)			
	symamd	symmmd	amdbar	mc47bd	symamd	symmmd	amdbar	mc47bd
PWT	1497	1425	1592	1556	156	140	173	162
MSC00726	103	108	111	111	20	22	23	23
BCSSTK38	735	803	752	737	118	143	127	119
NASA2146	135	157	140	140	11	15	12	12
CFD2	74832	94444	75008	75008	137470	211328	136476	136476
3DTUBE	26128	33213	26355	26355	29969	45578	30053	30053
GEARBOX	49268	63940	48556	48555	49849	88616	47068	47067
CRYSTM02	2025	3159	2286	2286	546	1240	719	719
FINAN512	2028	8223	2851	2838	249	12356	644	633
Median result relative to colamd	-	1.26	1.03	1.03	-	1.52	1.08	1.04

average. It nearly always produced significantly better orderings. In contrast, `symamd` was always slower than `amdbar` and `mc47bd`, although it found orderings of similar quality (the FINAN512 is a notable exception, but none of these methods finds as good an ordering as a tree dissection method [4]).

An ordering algorithm designed for symmetric matrices (`amdbar` or `mc47bd`) is thus superior to one based on a column ordering of $\mathbf{M}$. There may be at least one important exception, however. In some applications, a better matrix $\mathbf{M}$ is available. Consider an n-by-n finite element matrix constructed as the summation of e finite elements, where normally $e < n$. Each finite element matrix is a dense symmetric submatrix. Suppose element k is nonzero for all entries a_{ij} for which both i and j are in the set $\mathcal{E}_k$. We can construct a e-by-n matrix $\mathbf{M}$, where row k has the pattern $\mathcal{E}_k$. Since $\mathbf{M}^\mathsf{T}\mathbf{M}$ has the same pattern as $\mathbf{A}$, we can compute a column ordering of $\mathbf{M}$ and use it for the Cholesky factorization of $\mathbf{A}$. `Colamd` would be faster than when using an $\mathbf{M}$ constructed without the knowledge of the finite element structure. The space to perform the ordering and symbolic analysis is less as well, since $\mathbf{M}$ has fewer nonzeros than $\mathbf{A}$. Although many of the matrices in our test set arise from finite element problems, only a few small ones are available in unassembled form, as a collection of finite elements. Thus, we are not able to evaluate this strategy on large, realistic test matrices.

24

6 Summary

Two new ordering routines, `colamd` and `symamd`, have been presented. For square nonsymmetric matrices, `colamd` is much faster and provides better orderings than Matlab's `colmmd` routine. It is also faster than symmetric-based ordering methods (such as `amdbar`), and uses less storage. For rectangular matrices (such as those arising in least squares problems and interior point methods for linear programming), `colamd` is faster than `colmmd` and `amdbar` and finds orderings of comparable quality. We presented a symmetric ordering method `symamd` based on `colamd`; although it produces orderings as good as a truly symmetric ordering algorithm (`amdbar`), it is slower than `amdbar`.

The `colamd` and `symamd` routines are written in ANSI/ISO C, with Matlab-callable interfaces. Version 2.0 of the code is freely available from the following sources:

1. University of Florida, `http://www.cise.ufl.edu/research/sparse`.

2. Netlib, `http://www.netlib.org/linalg/colamd/`.

3. The MathWorks, Inc., for user-contributed contributions to Matlab, `http://www.mathworks.com`. Colamd and `symamd` are built-in functions in Matlab Version 6.0.

4. The collected algorithms of the ACM, as Algorithm 8xx, described in [10].

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.

[2] C. Ashcraft. Compressed graphs and the minimum degree algorithm. Technical Report BCSTECH-93-024, Boeing Computer Services, Seattle, Washington, 1993.

[3] Z. Bai, D. Day, J. Demmel, and J. Dongarra. Test matrix collection (non-Hermitian eigenvalue problems), Release 1. Technical report, University of Kentucky, September 1996. Available at ftp://ftp.ms.uky.edu/pub/misc/bai/Collection.

[4] A. J. Berger, J. M. Mulvey, E. Rothberg, and R. J. Vanderbei. Solving multistage stochastic programs using tree dissection. Technical Report SOR-95-07, Dept. of Civil Eng. and Operations Research, Princeton Univ., Princeton, NJ, June 1995.

[5] W. J. Carolan, J. E. Hill, J. L. Kennington, S. Niemi, and S. J. Wichmann. An empirical evaluation of the korbx algorithms for military airlift applications. *Operations Research*, 38(2):240–248, 1990.

[6] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert. Predicting fill for sparse orthogonal factorization. *J. Assoc. Comput. Mach.*, 33:517–532, 1986.

[7] T. A. Davis. Univ. of Florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse.

[8] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Applic.*, 18(1):140–158, 1997.

[9] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw.*, 25(1):1–19, 1999.

[10] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 8xx: Colamd, a column approximate minimum degree ordering algorithm. Technical Report TR-00-006, Univ. of Florida, CISE Department, Gainesville, FL, October 2000. (submitted to ACM Trans. on Mathematical Software).

[11] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Applic.*, 20(3):720–755, 1999.

[12] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[13] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Comm. ACM*, 30:403–407, 1987.

[14] I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw.*, 7:315–330, 1981.

[15] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15:1–14, 1989.

[16] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release 1). Technical Report RAL-92-086, Rutherford Appleton Laboratory, Didcot, Oxon, England, Dec. 1992.

[17] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Softw.*, 9(3):302–325, 1983.

[18] 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 Eng.*, 18:1145–1151, 1982.

[19] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Algorithms and data structures for sparse symmetric Gaussian elimination. *SIAM J. Sci. Statist. Comput.*, 2:225–237, 1981.

[20] P. Feldmann, R. Melville, and D. Long. Efficient frequency domain analysis of large nonlinear analog circuits. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, Santa Clara, CA, 1996.

[21] A. George and J. W. H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Trans. Math. Softw.*, 6(3):337–358, 1980.

[22] A. George and J. W. H. Liu. An optimal algorithm for symbolic factorization of symmetric matrices. *SIAM Journal on Computing*, 9:583–593, 1980.

[23] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[24] A. George and D. R. McIntyre. On the application of the minimum degree algorithm to finite element systems. *SIAM J. Numer. Anal.*, 15:90–111, 1978.

[25] A. George and E. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Statist. Comput.*, 6(2):390–409, 1985.

[26] A. George and E. Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Statist. Comput.*, 8(6):877–898, 1987.

[27] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Applic.*, 13(1):333–356, 1992.

[28] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In A. George, J. R. Gilbert, and J. W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, Volume 56 of the IMA Volumes in Mathematics and its Applications, pages 107–139. Springer-Verlag, 1993.

[29] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 15(4):1075–1091, 1994.

[30] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9:862–874, 1988.

[31] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[32] D. R. Hare, C. R. Johnson, D. D. Olesky, and P. Van Den Driessche. Sparsity analysis of the QR factorization. *SIAM J. Matrix Anal. Applic.*, 14(3):665–669, 1993.

[33] P. Heggernes and P. Matstoms. Finding good column orderings for sparse QR factorization. Technical report, Dept. of Informatics, Univ. of Bergen, Bergen, Norway, 1994.

[34] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4:373–395, 1978.

[35] J. L. Kern. Approximate deficiency for ordering the columns of a matrix. Technical report, Univ. of Florida, Gainesville, FL, 1999. (senior thesis, see http://www.cise.ufl.edu/colamd/kern.ps).

[36] S. I. Larimore. An approximate minimum degree column ordering algorithm. Technical Report TR-98-016, Univ. of Florida, Gainesville, FL, 1998. (Master's thesis, see http://www.cise.ufl.edu/tech-reports/).

[37] J. W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Softw.*, 11(2):141–153, 1985.

[38] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, 1957.

[39] J. J. H. Miller and Wang S. An exponentially fitted finite element method for a stationary convection-diffusion problem. In J. J. H. Miller, editor, *Computatioal methods for boundary and interior layers in several dimensions*, pages 120–137. Boole Press, Dublin, 1991.

[40] E. G. Ng and P. Raghavan. Performance of greedy ordering heuristics for sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 20(4):902–914, 1999.

[41] Resende Ramakrishnan and Drezner. Computing lower bounds for the quadratic assignment problem with an interior point algorithm for linear programming. *Operations Research*, 43:781–791, 1995.

[42] E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix orderings. *SIAM J. Matrix Anal. Applic.*, 19(3):682–695, 1998.

[43] A. H. Sherman. On the efficient solution of sparse systems of linear and nonlinear equations. Technical Report 46, Yale Univ. Dept. of Computer Science, Dec. 1975.

[44] S. J. Wright. *Primal-dual interior-point methods.* SIAM Publications, 1996.

[45] M. Yannakakis. Computing the minimum fill-in is np-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

[46] S. E. Zitney. Sparse matrix methods for chemical process separation calculations on supercomputers. In *Proc. Supercomputing '92*, pages 414–423, Minneapolis, MN, Nov. 1992. IEEE Computer Society Press.

[47] S. E. Zitney, J. Mallya, T. A. Davis, and M. A. Stadtherr. Multifrontal vs. frontal techniques for chemical process simulation on supercomputers. *Comput. Chem. Eng.*, 20(6/7):641–646, 1996.