SPARSE MATRICES IN MATLAB: DESIGN AND IMPLEMENTATION

JOHN R. GILBERT*, CLEVE MOLER†, AND ROBERT SCHREIBER‡

Dedicated to Gene Golub on the occasion of his 60th birthday.

Abstract. We have extended the matrix computation language and environment MATLAB to include sparse matrix storage and operations. The only change to the outward appearance of the MATLAB language is a pair of commands to create full or sparse matrices. Nearly all the operations of MATLAB now apply equally to full or sparse matrices, without any explicit action by the user. The sparse data structure represents a matrix in space proportional to the number of nonzero entries, and most of the operations compute sparse results in time proportional to the number of arithmetic operations on nonzeros.

Key words. MATLAB, mathematical software, matrix computation, sparse matrix algorithms.

AMS subject classifications. 65–04, 65F05, 65F20, 65F50, 68N15, 68R10.

1. Introduction. MATLAB is an interactive environment and programming language for numeric scientific computation [18]. One of its distinguishing features is the use of matrices as the only data type. In MATLAB, a matrix is a rectangular array of real or complex numbers. All quantities, even loop variables and character strings, are represented as matrices, although matrices with only one row, or one column, or one element are sometimes treated specially.

The part of MATLAB that involves computational linear algebra on dense matrices is based on direct adaptations of subroutines from LINPACK and EISPACK [5, 23]. An $m \times n$ real matrix is stored as a full array of mn floating point numbers. The computational complexity of basic operations such as addition or transposition is proportional to mn. The complexity of more complicated operations such as triangular factorization is proportional to mn^2. This has limited the applicability of MATLAB to problems involving matrices of order a few hundred on contemporary workstations and perhaps a few thousand on contemporary supercomputers.

We have now added sparse matrix storage and operations to MATLAB. This report describes our design and implementation.

Sparse matrices are widely used in scientific computation, especially in large-scale optimization, structural and circuit analysis, computational fluid dynamics, and, generally, the numerical solution of partial differential equations. Several effective Fortran subroutine packages for solving sparse linear systems are available, including SPARSPAK [11], the Yale Sparse Matrix Package [9], and some of the routines in the Harwell Subroutine Library [25].

Our work was facilitated by our knowledge of the techniques used in the Fortran sparse matrix packages, but we have not directly adapted any of their code. MATLAB

---

* Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

† The MathWorks, 325 Linfield Place, Menlo Park, California 94025.

‡ Research Institute for Advanced Computer Science, MS T045-1, NASA Ames Research Center, Moffett Field, CA 94035. This author's work was supported by the NAS Systems Division and DARPA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA).

TABLE 1
Operations with the 4096 by 4096 discrete Laplacian.

	Sparse	Full
Memory	0.25 megabyte	128 megabytes
Compute Dx	0.2 seconds	30 seconds
Solve $Dx = b$	10 seconds	$>$ 12 hours

is written in C and we wished to take advantage of the data structures and other programming features of C that would not be used in a simple translation of Fortran code. We also wanted to implement the full range of matrix operations that MATLAB provides; the Fortran packages do not generally have routines for simply adding or transposing sparse matrices, for example. And, finally, we wanted to incorporate some recent algorithmic ideas that are not used in the Fortran packages.

J. H. Wilkinson's informal working definition of a sparse matrix was "any matrix with enough zeros that it pays to take advantage of them." So sparsity is an economic issue. By avoiding arithmetic operations on zero elements, sparse matrix algorithms require less computer time. And, perhaps more importantly, by not storing many zero elements, sparse matrix data structures require less computer memory. In a sense, we have not added any new functionality to MATLAB; we've merely made some existing functionality more efficient in terms of both time and storage.

An important descriptive parameter of a sparse matrix S is $\mathrm{nnz}(S)$, the number of nonzero elements in S. Computer storage requirements are proportional to nnz. The computational complexity of simple array operations should also be should be proportional to nnz, and perhaps also depend linearly on m or n, but be independent of the product mn. The complexity of more complicated operations involves such factors as ordering and fill-in, but an objective of a good sparse matrix algorithm should be:

> The time required for a sparse matrix operation should be proportional to number of arithmetic operations on nonzero quantities.

We call this the "time is proportional to flops" rule; it is a fundamental tenet of our design.

With sparse techniques, it is practical to handle matrices involving tens of thousands of nonzero elements on contemporary workstations. As one example, let D be the matrix representation of the discrete 5-point Laplacian on a square 64×64 grid with a nested dissection ordering. This is a 4096×4096 matrix with 20224 nonzeros. Table 1 gives the memory requirements for storing D as a MATLAB sparse matrix and as a traditional Fortran or MATLAB full matrix, as well as the execution time on a Sun SPARCstation-1 workstation for computing a matrix-vector product and solving a linear system of equations by elimination.

Band matrices are special cases of sparse matrices whose nonzero elements all happen to be near the diagonal. It would be somewhat more efficient, in both time and storage, to provide a third data structure and collection of operations for band matrices. We have decided against doing this because of the added complexity, particular in cases involving mixtures of full, sparse and band matrices. We suspect that solving linear systems with matrices which are dense within a narrow band might be twice as fast with band storage as it is with sparse matrix storage, but that linear systems with matrices that are sparse within the band (such as those obtained from two-dimensional grids) are more efficiently solved with general sparse matrix technology. However, we have not investigated these tradeoffs in any detail.

In this paper, we concentrate on elementary sparse matrix operations, such as addition and multiplication, and on direct methods for solving sparse linear systems of equations. These operations are now included in the "core" of MATLAB. Except for a few short examples, we will not discuss higher level sparse matrix operations, such as iterative methods for linear systems. We intend to implement such operations as interpreted programs in the MATLAB language, so-called "m-files," outside the MATLAB core.

2. The user's view of sparse MATLAB.

2.1. Sparse matrix storage. We wish to emphasize the distinction between a matrix and what we call its *storage class*. A given matrix can conceivably be stored in many different ways—fixed point or floating point, by rows or by columns, real or complex, full or sparse—but all the different ways represent the same matrix. We now have two matrix storage classes in MATLAB, full and sparse.

Two MATLAB variables, `A` and `B`, can have different storage classes but still represent the same matrix. They occupy different amounts of computer memory, but in most other respects they are the same. Their elements are equal, their determinants and their eigenvalues are equal, and so on. The crucial question of which storage class to choose for a given matrix is the topic of Section 2.5.

Even though MATLAB is written in C, it follows its LINPACK and Fortran predecessors and stores full matrices by columns [5, 19]. This organization has been carried over to sparse matrices. A sparse matrix is stored as the concatenation of the sparse vectors representing its columns. Each sparse vector consists of a floating point array of nonzero entries (or two such arrays for complex matrices), together with an integer array of row indices. A second integer array gives the locations in the other arrays of the first element in each column. Consequently, the storage requirement for an $m \times n$ real sparse matrix with nnz nonzero entries is nnz reals and $nnz + n$ integers. On typical machines with 8-byte reals and 4-byte integers, this is $12nnz + 4n$ bytes. Complex matrices use a second array of nnz reals. Notice that m, the number of rows, is almost irrelevant. It is not involved in the storage requirements, nor in the operation counts for most operations. Its primary use is in error checks for subscript ranges. Similar storage schemes, with either row or column orientation, are used in the Fortran sparse packages.

2.2. Converting between full and sparse storage. Initially, we contemplated schemes for automatic conversion between sparse and full storage. There is a MATLAB precedent for such an approach. Matrices are either real or complex and the conversion between the two is automatic. Computations such as square roots and logarithms of negative numbers and eigenvalues of nonsymmetric matrices generate complex results from real data. MATLAB automatically expands the data structure by adding an array for the imaginary parts.

Moreover, several of MATLAB's functions for building matrices produce results that might effectively be stored in the sparse organization. The function `zeros(m,n)`, which generates an $m \times n$ matrix of all zeros, is the most obvious candidate. The functions `eye(n)` and `diag(v)`, which generate the $n \times n$ identity matrix and a diagonal matrix with the entries of vector v on the main diagonal, are also possibilities. Even `tril(A)` and `triu(A)`, which take the lower and upper triangular parts of a matrix A, might be considered. But this short list begins to demonstrate a difficulty—how far

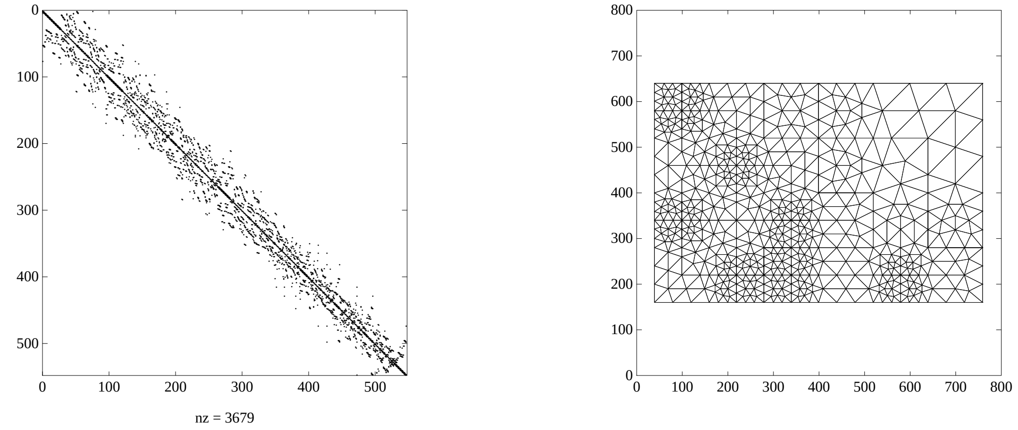

FIG. 1. *The Eppstein mesh as plotted by* `spy(A)` *and* `gplot(A,xy)`.

should "automatic sparsification" be carried? Is there some threshold value of sparsity where the conversion should be done? Should the user provide the value for such a sparsification parameter? We don't know the answers to these questions, so we decided to take another approach, which we have since found to be quite satisfactory.

No sparse matrices are created without some overt direction from the user. Thus, the changes we have made to MATLAB do not affect the user who has no need for sparsity. Operations on full matrices continue to produce full matrices. But once initiated, sparsity propagates. Operations on sparse matrices produce sparse matrices. And an operation on a mixture of sparse and full matrices produces a sparse result unless the operator ordinarily destroys sparsity. (Matrix addition is an example; more on this later.)

There are two new built-in functions, `full` and `sparse`. For any matrix A, `full(A)` returns A stored as a full matrix. If A is already full, then A is returned unchanged. If A is sparse, then zeros are inserted at the appropriate locations to fill out the storage. Conversely, `sparse(A)` removes any zero elements and returns A stored as a sparse matrix, regardless of how sparse A actually is.

2.3. Displaying sparse matrices. Sparse and full matrices print differently. The statement

```
A = [0 0 11; 22 0 0; 0 33 0]
```

produces a conventional MATLAB full matrix that prints as

```
A =
          0      0     11
         22      0      0
          0     33      0
```

The statement `S = sparse(A)` converts A to sparse storage, and prints

```
S =
       (2,1)        22
       (3,2)        33
       (1,3)        11
```

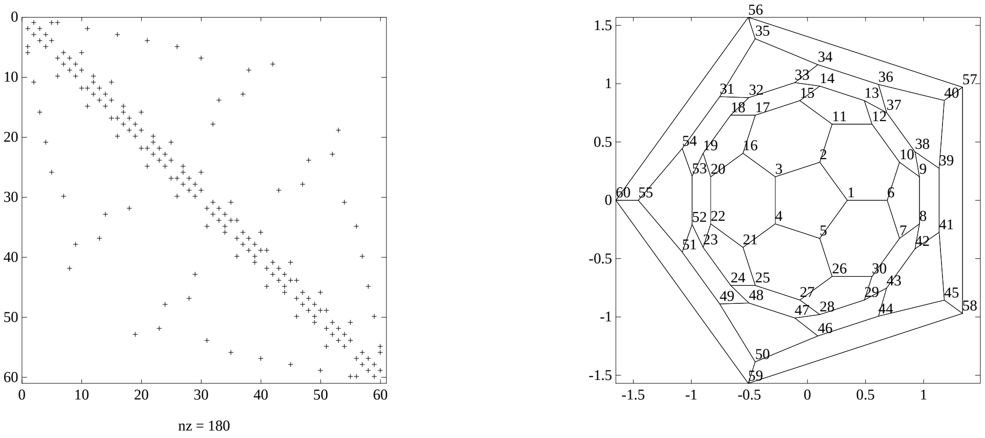

FIG. 2. *The buckyball as rendered by* `spy` *and* `gplot`.

As this illustrates, sparse matrices are printed as a list of their nonzero elements (with indices), in column major order.

The function `nnz(A)` returns the number of nonzero elements of A. It is implemented by scanning full matrices, and by access to the internal data structure for sparse matrices. The function `nzmax(A)` returns the number of storage locations for nonzeros allocated for A.

Graphic visualization of the structure of a sparse matrix is often a useful tool. The function `spy(A)` plots a silhouette of the nonzero structure of A. Figure 1 illustrates such a plot for a matrix that comes from a finite element mesh due to David Eppstein. A picture of the graph of a matrix is another way to visualize its structure. Laying out an arbitrary graph for display is a hard problem that we do not address. However, some sparse matrices (from finite element applications, for example) have spatial coordinates associated with their rows or columns. If xy contains such coordinates for matrix A, the function `gplot(A,xy)` draws its graph. The second plot in Figure 1 shows the graph of the sample matrix, which in this case is just the same as the finite element mesh. Figure 2 is another example: The `spy` plot is the 60×60 adjacency matrix of the graph of a Buckminster Fuller geodesic dome, a soccer ball, and a C_{60} molecule, and the `gplot` shows the graph itself.

Section 3.3.4 describes another function for visualizing the elimination tree of a matrix.

2.4. Creating sparse matrices. Usually one wants to create a sparse matrix directly, without first having a full matrix A and then converting it with `S = sparse(A)`. One way to do this is by simply supplying a list of nonzero entries and their indices. Several alternate forms of `sparse` (with more than one argument) allow this. The most general is

```
S = sparse(i,j,s,m,n,nzmax)
```

Ordinarily, i and j are vectors of integer indices, s is a vector of real or complex entries, and m, n, and $nzmax$ are integer scalars. This call generates an $m \times n$ sparse matrix, having one nonzero for each entry in the vectors i, j, and s, with $S(i(k), j(k)) = s(k)$, and with enough space allocated for S to have $nzmax$ nonzeros. The indices in i and j need not be given in any particular order.

If a pair of indices occurs more than once in i and j, `sparse` adds the corresponding values of s together. Then the sparse matrix S is created with one nonzero for each nonzero in this modified vector s. The argument s and one of the arguments i and j may be scalars, in which case they are expanded so that the first three arguments all have the same length.

There are several simplifications of the full six-argument call to `sparse`.

`S = sparse(i,j,s,m,n)` uses $nzmax = \text{length}(s)$.

`S = sparse(i,j,s)` uses $m = \max(i)$ and $n = \max(j)$.

`S = sparse(m,n)` is the same as `S = sparse([],[],[],m,n)`, where `[]` is MATLAB's empty matrix. It produces the ultimate sparse matrix, an $m \times n$ matrix of all zeros.

Thus for example

```
S = sparse([1 2 3], [3 1 2], [11 22 33])
```

produces the sparse matrix S from the example in Section 2.3, but does not generate any full 3 by 3 matrix during the process.

MATLAB's function `k = find(A)` returns a list of the positions of the nonzeros of A, counting in column-major order. For sparse MATLAB we extended the definition of `find` to extract the nonzero elements together with their indices. For any matrix A, full or sparse, `[i,j,s] = find(A)` returns the indices and values of the nonzeros. (The square bracket notation on the left side of an assignment indicates that the function being called can return more than one value. In this case, `find` returns three values, which are assigned to the three separate variables i, j, and s.) For example, this dissects and then reassembles a sparse matrix:

```
[i,j,s] = find(S);
[m,n] = size(S);
S = sparse(i,j,s,m,n);
```

So does this, if the last row and column have nonzero entries:

```
[i,j,s] = find(S);
S = sparse(i,j,s);
```

Another common way to create a sparse matrix, particularly for finite difference computations, is to give the values of some of its diagonals. Two functions `diags` and `blockdiags` can create sparse matrices with specified diagonal or block diagonal structure.

There are several ways to read and write sparse matrices. The MATLAB `save` and `load` commands, which save the current workspace or load a saved workspace, have been extended to accept sparse matrices and save them efficiently. We have written a Fortran utility routine that converts a file containing a sparse matrix in the Harwell-Boeing format [6] into a file that MATLAB can load.

2.5. The results of sparse operations. What is the result of a MATLAB operation on sparse matrices? This is really two fundamental questions: what is the value of the result, and what is its storage class? In this section we discuss the answers that we settled on for those questions.

A function or subroutine written in MATLAB is called an *m-file*. We want it to be possible to write m-files that produce the same results for sparse and for full inputs. Of course, one could ensure this by converting all inputs to full, but that would defeat the goal of efficiency. A better idea, we decided, is to postulate that

> The value of the result of an operation does not depend on the storage class of the operands, although the storage class of the result may.

The only exception is a function to inquire about the storage class of an object: `issparse(A)` returns 1 if A is sparse, 0 otherwise.

Some intriguing notions were ruled out by our postulate. We thought, for a while, that in cases such as `A ./ S` (which denotes the pointwise quotient of A and S) we ought not to divide by zero where S is zero, since that would not produce anything useful; instead we thought to implement this as if it returned $A(i,j)/S(i,j)$ wherever $S(i,j) \neq 0$, leaving A unchanged elsewhere. All such ideas, however, were dropped in the interest of observing the rule that the result does not depend on storage class.

The second fundamental question is how to determine the storage class of the result of an operation. Our decision here is based on three ideas. First, the storage class of the result of an operation should depend only on the storage classes of the operands, not on their values or sizes. (Reason: it's too risky to make a heuristic decision about when to sparsify a matrix without knowing how it will be used.) Second, sparsity should not be introduced into a computation unless the user explicitly asks for it. (Reason: the full matrix user shouldn't have sparsity appear unexpectedly, because of the performance penalty in doing sparse operations on mostly nonzero matrices.) Third, once a sparse matrix is created, sparsity should propagate through matrix and vector operations, concatenation, and so forth. (Reason: most m-files should be able to do sparse operations for sparse input or full operations for full input without modification.)

Thus full inputs always give full outputs, except for functions like `sparse` whose purpose is to create sparse matrices. Sparse inputs, or mixed sparse and full inputs, follow these rules (where S is sparse and F is full):

- Functions from matrices to scalars or fixed-size vectors, like `size` or `nnz`, always return full results.
- Functions from scalars or fixed-size vectors to matrices, like `zeros`, `ones`, and `eye`, generally return full results. Having `zeros(m,n)` and `eye(m,n)` return full results is necessary to avoid introducing sparsity into a full user's computation; there are also functions `spzeros` and `speye` that return sparse zero and identity matrices.
- The remaining unary functions from matrices to matrices or vectors generally return a result of the same storage class as the operand (the main exceptions are `sparse` and `full`). Thus, `chol(S)` returns a sparse Cholesky factor, and `diag(S)` returns a sparse vector (a sparse $m \times 1$ matrix). The vectors returned by `max(S)`, `sum(S)`, and their relatives (that is, the vectors of column maxima and column sums respectively) are sparse, even though they may well be all nonzero.
- Binary operators yield sparse results if both operands are sparse, and full results if both are full. In the mixed case, the result's storage class depends on the operator. For example, `S + F` and `F \ S` (which solves the linear system $SX = F$) are full; `S .* F` (the pointwise product) and `S & F` are sparse.
- A block matrix formed by concatenating smaller matrices, like

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix},$$

> is written as `[A B ; C D]` in MATLAB. If all the inputs are full, the result is full, but a concatenation that contains any sparse matrix is sparse. Submatrix indexing on the right counts as a unary operator; `A = S(i,j)` produces a sparse result (for sparse S) whether i and j are scalars or vectors. Submatrix indexing on the left, as in `A(i,j) = S`, does not change the storage class of the matrix being modified.

These decisions gave us some difficulty. Cases like `~S` and `S >= T`, where the result has many ones when the operands are sparse, made us consider adding more exceptions to the rules. We discussed the possibility of "sparse" matrices in which all the values not explicitly stored would be some scalar (like 1) rather than zero. We rejected these ideas in the interest of simplicity.

3. Implementation.

This section describes the algorithms for the sparse operations in MATLAB in some detail. We begin with a discussion of fundamental data structures and design decisions.

3.1. Fundamentals.

3.1.1. Data structure. A most important implementation decision is the choice of a data structure. The internal representation of a sparse matrix must be flexible enough to implement all the MATLAB operations. For simplicity, we ruled out the use of different data structures for different operations. The data structure should be compact, storing only nonzero elements, with a minimum of overhead storage for integers or pointers. Wherever possible, it should support matrix operations in time proportional to flops. Since MATLAB is an interpreted, high-level matrix language, efficiency is more important in matrix arithmetic and matrix-vector operations than in accessing single elements of matrices.

These goals are met by a simple column-oriented scheme that has been widely used in sparse matrix computation. A sparse matrix is a C record structure with the following constituents. The nonzero elements are stored in a one-dimensional array of double-precision reals, in column major order. (If the matrix is complex, the imaginary parts are stored in another such array.) A second array of integers stores the row indices. A third array of $n+1$ integers stores the index into the first two arrays of the leading entry in each of the n columns, and a terminating index whose value is nnz. Thus a real $m \times n$ sparse matrix with nnz nonzeros uses nnz reals and $nnz + n + 1$ integers.

This scheme is not efficient for manipulating matrices one element at a time: access to a single element takes time at least proportional to the logarithm of the length of its column; inserting or removing a nonzero may require extensive data movement. However, element-by-element manipulation is rare in MATLAB (and is expensive even in full MATLAB). Its most common application would be to create a sparse matrix, but this is more efficiently done by building a list $[i, j, s]$ of matrix elements in arbitrary order and then using `sparse(i,j,s)` to create the matrix.

The sparse data structure is allowed to have unused elements after the end of the last column of the matrix. Thus an algorithm that builds up a matrix one column at a time can be implemented efficiently by allocating enough space for all the expected nonzeros at the outset.

3.1.2. Storage allocation. Storage allocation is one of the thorniest parts of building portable systems. MATLAB handles storage allocation for the user, invisibly allocating and deallocating storage as matrices appear, disappear, and change size.

Sometimes the user can gain efficiency by preallocating storage for the result of a computation. One does this in full MATLAB by allocating a matrix of zeros and filling it in incrementally. Similarly, in sparse MATLAB one can preallocate a matrix (using `sparse`) with room for a specified number of nonzeros. Filling in the sparse matrix a column at a time requires no copying or reallocation.

Within MATLAB, simple "allocate" and "free" procedures handle storage allocation. (We will not discuss how MATLAB handles its free storage and interfaces to the operating system to provide these procedures.) There is no provision for doing storage allocation within a single matrix; a matrix is allocated as a single block of storage, and if it must expand beyond that block it is copied into a newly allocated larger block.

MATLAB must allocate space to hold the results of operations. For a full result, MATLAB allocates mn elements at the start of the computation. This strategy could be disastrous for sparse matrices. Thus, sparse MATLAB attempts to make a reasonable choice of how much space to allocate for a sparse result.

Some sparse matrix operations, like Cholesky factorization, can predict in advance the exact amount of storage the result will require. These operations simply allocate a block of the right size before the computation begins. Other operations, like matrix multiplication and LU factorization, have results of unpredictable size. These operations are all implemented by algorithms that compute one column at a time. Such an algorithm first makes a guess at the size of the result. If more space is needed at some point, it allocates a new block that is larger by a constant factor (typically 1.5) than the current block, copies the columns already computed into the new block, and frees the old block.

Most of the other operations compute a simple upper bound on the storage required by the result to decide how much space to allocate—for example, the pointwise product `S .* T` uses the smaller of $\mathrm{nnz}(S)$ and $\mathrm{nnz}(T)$, and `S + T` uses the smaller of $\mathrm{nnz}(S)+\mathrm{nnz}(T)$ and mn.

3.1.3. The sparse accumulator. Many sparse matrix algorithms use a dense working vector to allow random access to the currently "active" column or row of a matrix. The sparse MATLAB implementation formalizes this idea by defining an abstract data type called the sparse accumulator, or SPA. The SPA consists of a dense vector of real (or complex) values, a dense vector of true/false "occupied" flags, and an unordered list of the indices whose occupied flags are true.

The SPA represents a column vector whose "unoccupied" positions are zero and whose "occupied" positions have values (zero or nonzero) specified by the dense real or complex vector. It allows random access to a single element in constant time, as well as sequencing through the occupied positions in constant time per element. Most matrix operations allocate the SPA (with appropriate dimension) at their beginning and free it at their end. Allocating the SPA takes time proportional to its dimension (to turn off all the occupied flags), but subsequent operations take only constant time per nonzero.

In a sense the SPA is a register and an instruction set in an abstract machine architecture for sparse matrix computation. MATLAB manipulates the SPA through some thirty-odd access procedures. About half of these are operations between the SPA and a sparse or dense vector, from a "spaxpy" that implements SPA $:=$ SPA$+ax$ (where a is a scalar and x is a column of a sparse matrix) to a "spaeq" that tests elementwise equality. Other routines load and store the SPA, permute it, and access individual elements. The most complicated SPA operation is a depth-first search on an acyclic graph, which marks as "occupied" a topologically ordered list of reachable vertices; this is used in the sparse triangular solve described in Section 3.4.2.

The SPA simplifies data structure manipulation, because all fill occurs in the SPA; that is, only in the SPA can a zero become nonzero. The "spastore" routine does not store exact zeros, and in fact the sparse matrix data structure never contains any explicit zeros. Almost all real arithmetic operations occur in SPA routines, too, which simplifies MATLAB's tally of flops. (The main exceptions are in certain scalar-matrix operations like `2*A`, which are implemented without the SPA for efficiency.)

3.1.4. Asymptotic complexity analysis. A strong philosophical principle in the sparse MATLAB implementation is that it should be possible to analyze the complexity of the various operations, and that they should be efficient in the asymptotic sense as well as in practice. This section discusses this principle, in terms of both theoretical ideals and engineering compromises.

Ideally all the matrix operations would use time proportional to flops, that is, their running time would be proportional to the number of nonzero real arithmetic operations performed. This goal cannot always be met: for example, `[0 1] + [1 0]` does no nonzero arithmetic. A more accurate statement is that time should be proportional to flops or data size, whichever is larger. Here "data size" means the size of the output and that part of the input that is used nontrivially; for example, in `A*b` only those columns of A corresponding to nonzeros in b participate nontrivially.

This more accurate ideal can be realized in almost all of MATLAB. The exceptions are some operations that do no arithmetic and cannot be implemented in time proportional to data size. The algorithms to compute most of the reordering permutations described in Section 3.3 are efficient in practice but not linear in the worst case. Submatrix indexing is another example: if i and j are vectors of row and column indices, `B = A(i,j)` may examine all the nonzeros in the columns $A(:,j)$, and `B(i,j) = A` can at worst take time linear in the total size of B.

The MATLAB implementation actually violates the "time proportional to flops" philosophy in one systematic way. The list of occupied row indices in the SPA is not maintained in numerical order, but the sparse matrix data structure does require row indices to be ordered. Sorting the row indices when storing the SPA would theoretically imply an extra factor of $O(\log n)$ in the worst-case running times of many of the matrix operations. All our algorithms could avoid this factor—usually by storing the matrix with unordered row indices, then using a linear-time transposition sort to reorder all the rows of the final result at once—but for simplicity of programming we included the sort in "spastore".

The idea that running time should be susceptible to analysis helps the user who writes programs in MATLAB to choose among alternative algorithms, gives guidance in scaling up running times from small examples to larger problems, and, in a general-purpose system like MATLAB, gives some insurance against an unexpected worst-case instance arising in practice. Of course complete *a priori* analysis is impossible—the work in sparse LU factorization depends on numerical pivoting choices, and the efficacy of a heuristic reordering such as minimum degree is unpredictable—but we feel it is worthwhile to stay as close to the principle as we can.

In a technical report [14] we present some experimental evidence that sparse MATLAB operations require time proportional to flops and data size in practice.

3.2. Factorizations. The LU and Cholesky factorizations of a sparse matrix yield sparse results. MATLAB does not yet have a sparse QR factorization. Section 3.6 includes some remarks on sparse eigenvalue computation in MATLAB.

3.2.1. LU Factorization.

If A is a sparse matrix, `[L,U,P] = lu(A)` returns three sparse matrices such that $PA = LU$, as obtained by Gaussian elimination with partial pivoting. The permutation matrix P uses only $O(n)$ storage in sparse format. As in dense MATLAB, `[L,U] = lu(A)` returns a permuted unit lower triangular and an upper triangular matrix whose product is A.

Since sparse LU must behave like MATLAB's full LU, it does not pivot for sparsity. A user who happens to know a good column permutation Q for sparsity can, of course, ask for `lu(A*Q')`, or `lu(A(:,q))` where q is an integer permutation vector. Section 3.3 describes a few ways to find such a permutation. The matrix division operators `\` and `/` do pivot for sparsity by default; see Section 3.4.

We use a version of the `GPLU` algorithm [15] to compute the LU factorization. This computes one column of L and U at a time by solving a sparse triangular system with the already-finished columns of L. Section 3.4.2 describes the sparse triangular solver that does most of the work. The total time for the factorization is proportional to the number of nonzero arithmetic operations (plus the size of the result), as desired.

The column-oriented data structure for the factors is created as the factorization progresses, never using any more storage for a column than it requires. However, the total size of L or U cannot be predicted in advance. Thus the factorization routine makes an initial guess at the required storage, and expands that storage (by a factor of 1.5) whenever necessary.

3.2.2. Cholesky factorization.

As in full MATLAB, `R = chol(A)` returns the upper triangular Cholesky factor of a Hermitian positive definite matrix A. Pivoting for sparsity is not automatic, but minimum degree and profile-limiting permutations can be computed as described in Section 3.3.

Our current implementation of Cholesky factorization emphasizes simplicity and compatibility with the rest of sparse MATLAB; thus it does not use some of the more sophisticated techniques such as the compressed index storage scheme [11, Sec. 5.4.2], or supernodal methods to take advantage of the clique structure of the chordal graph of the factor [2]. It does, however, run in time proportional to arithmetic operations with little overhead for data structure manipulation.

We use a slightly simplified version of an algorithm from the Yale Sparse Matrix Package [9], which is described in detail by George and Liu [11]. We begin with a combinatorial step that determines the number of nonzeros in the Cholesky factor (assuming no exact cancellation) and allocates a large enough block of storage. We then compute the lower triangular factor R^T one column at a time. Unlike YSMP and SPARSPAK, we do not begin with a symbolic factorization; instead, we create the sparse data structure column by column as we compute the factor. The only reason for the initial combinatorial step is to determine how much storage to allocate for the result.

3.3. Permutations.

A permutation of the rows or columns of a sparse matrix A can be represented in two ways. A permutation matrix P acts on the rows of A as `P*A` or on the columns as `A*P'`. A permutation vector p, which is a full vector of length n containing a permutation of `1:n`, acts on the rows of A as `A(p,:)` or on the columns as `A(:,p)`. Here p could be either a row vector or a column vector.

Both representations use $O(n)$ storage, and both can be applied to A in time proportional to $\mathrm{nnz}(A)$. The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return vectors—full row vectors, to be precise—with the exception of the pivoting permutation in LU factorization.

Converting between the representations is almost never necessary, but it is simple. If I is a sparse identity matrix of the appropriate size, then P is `I(p,:)` and P^T is `I(:,p)`. Also p is `(P*(1:n)')'` or `(1:n)*P'`. (We leave to the reader the puzzle of using `find` to obtain p from P without doing any arithmetic.) The inverse of P is `P'`; the inverse r of p can be computed by the "vectorized" statement `r(p) = 1:n`.

3.3.1. Permutations for sparsity: Asymmetric matrices. Reordering the columns of a matrix can often make its LU or QR factors sparser. The simplest such reordering is to sort the columns by increasing nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.

The MATLAB function `p = colperm(A)` computes this column-count permutation. It is implemented as a two-line m-file:

```
[i,j] = find(A);
[ignore,p] = sort(diff(find(diff([0 j' inf]))));
```

The vector j is the column indices of all the nonzeros in A, in column major order. The inner `diff` computes first differences of j to give a vector with ones at the starts of columns and zeros elsewhere; the `find` converts this to a vector of column-start indices; the outer `diff` gives the vector of column lengths; and the second output argument from `sort` is the permutation that sorts this vector.

The symmetric reverse Cuthill-McKee ordering described in Section 3.3.2 can be used for asymmetric matrices as well; the function `symrcm(A)` actually operates on the nonzero structure of $A + A^T$. This is sometimes a good ordering for matrices that come from one-dimensional problems or problems that are in some sense long and thin.

Minimum degree is an ordering that often performs better than `colperm` or `symrcm`. The sparse MATLAB function `p = colmmd(A)` computes a minimum degree ordering for the columns of A. This column ordering is the same as a symmetric minimum degree ordering for the matrix A^TA, though we do not actually form A^TA to compute it.

George and Liu [10] survey the extensive development of efficient and effective versions of symmetric minimum degree, most of which is reflected in the symmetric minimum degree codes in SPARSPAK, YSMP, and the Harwell Subroutine Library. The MATLAB version of minimum degree uses many of these ideas, as well as some ideas from a parallel symmetric minimum degree algorithm by Gilbert, Lewis, and Schreiber [13]. We sketch the algorithm briefly to show how these ideas are expressed in the framework of column minimum degree. The reader who is not interested in all the details can skip to Section 3.3.2.

Although most column minimum degree codes for asymmetric matrices are based on a symmetric minimum degree code, our organization is the other way around: MATLAB's symmetric minimum degree code (described in Section 3.3.2) is based on its column minimum degree code. This is because the best way to represent a symmetric matrix (for the purposes of minimum degree) is as a union of cliques, or full submatrices. When we begin with an asymmetric matrix A, we wish to reorder its columns by using a minimum degree order on the symmetric matrix A^TA—but each row of A induces a clique in A^TA, so we can simply use A itself to represent A^TA instead of forming the product explictly. Speelpenning [24] called such a clique representation of a symmetric graph the "generalized element" representation; George and

Liu [10] call it the "quotient graph model." Ours is the first column minimum degree implementation that we know of whose data structures are based directly on A, and which does not need to spend the time and storage to form the structure of A^TA. The idea for such a code is not new, however—George and Liu [10] suggest it, and our implementation owes a great deal to discussions between the first author and Esmond Ng and Barry Peyton of Oak Ridge National Laboratories.

We simulate symmetric Gaussian elimination on A^TA, using a data structure that represents A as a set of vertices and a set of cliques whose union is the graph of A^TA. Initially, each column of A is a vertex and each row is a clique. Elimination of a vertex j induces fill among all the (so far uneliminated) vertices adjacent to j. This means that all the vertices in cliques containing j become adjacent to one another. Thus all the cliques containing vertex j merge into one clique. In other words, all the rows of A with nonzeros in column j disappear, to be replaced by a single row whose nonzero structure is their union. Even though fill is implicitly being added to A^TA, the data structure for A gets smaller as the rows merge, so no extra storage is required during the elimination.

Minimum degree chooses a vertex of lowest degree (the sparsest remaining column of A^TA, or the column of A having nonzero rows in common with the fewest other columns), eliminates that vertex, and updates the remainder of A by adding fill (i.e. merging rows). This whole process is called a "stage"; after n stages the columns are all eliminated and the permutation is complete. In practice, updating the data structure after each elimination is too slow, so several devices are used to perform many eliminations in a single stage before doing the update for the stage.

First, instead of finding a single minimum-degree vertex, we find an entire "independent set" of minimum-degree vertices with no common nonzero rows. Eliminating one such vertex has no effect on the others, so we can eliminate them all at the same stage and do a single update. George and Liu call this strategy "multiple elimination". (They point out that the resulting permutation may not be a strict minimum degree order, but the difference is generally insignificant.)

Second, we use what George and Liu call "mass elimination": After a vertex j is eliminated, its neighbors in A^TA form a clique (a single row in A). Any of those neighbors whose own neighbors all lie within that same clique will be a candidate for elimination at the next stage. Thus, we may as well eliminate such a neighbor during the same stage as j, immediately after j, delaying the update until afterward. This often saves a tremendous number of stages because of the large cliques that form late in the elimination. (The number of stages is reduced from the height of the elimination tree to approximately the height of the clique tree; for many two-dimensional finite element problems, for example, this reduces the number of stages from about $\sqrt{n}$ to about $\log n$.) Mass elimination is particularly simple to implement in the column data structure: after all rows with nonzeros in column j are merged into one row, the columns to be eliminated with j are those whose only remaining nonzero is in that new row.

Third, we note that any two columns with the same nonzero structure will be eliminated in the same stage by mass elimination. Thus we allow the option of combining such columns into "supernodes" (or, as George and Liu call them, "indistinguishable nodes"). This speeds up the ordering by making the data structure for A smaller. The degree computation must account for the sizes of supernodes, but this turns out to be an advantage for two reasons. The quality of the ordering actually improves slightly if the degree computation does not count neighbors within the same supernode. (George and Liu observe this phenomenon and call the number of neighbors outside a vertex's supernode its "external degree.") Also, supernodes improve the approximate degree computation described below. Amalgamating columns into supernodes is fairly slow (though it takes time only proportional to the size of A). Supernodes can be amalgamated at every stage, periodically, or never; the current default is every third stage.

Fourth, we note that the structure of A^TA is not changed by dropping any row of A whose nonzero structure is a subset of that of another row. This row reduction speeds up the ordering by making the data structure smaller. More significantly, it allows mass elimination to recognize larger cliques, which decreases the number of stages dramatically. Duff and Reid [8] call this strategy "element absorption." Row reduction takes time proportional to multiplying AA^T in the worst case (though the worst case is rarely realized and the constant of proportionality is very small). By default, we reduce at every third stage; again the user can change this.

Fifth, to achieve larger independent sets and hence fewer stages, we relax the minimum degree requirement and allow elimination of any vertex of degree at most $\alpha d+\beta$, where d is the minimum degree at this stage and α and β are parameters. The choice of threshold can be used to trade off ordering time for quality of the resulting ordering. For problems that are very large, have many right-hand sides, or factor many matrices with the same nonzero structure, ordering time is insignificant and the tightest threshold is appropriate. For one-off problems of moderate size, looser thresholds like $1.5d+2$ or even $2d+10$ may be appropriate. The threshold can be set by the user; its default is $1.2d+1$.

Sixth and last, our code has the option of using an "approximate degree" instead of computing the actual vertex degrees. Recall that a vertex is a column of A, and its degree is the number of other columns with which it shares some nonzero row. Computing all the vertex degrees in A^TA takes time proportional to actually computing A^TA, though the constant is quite small and no extra space is needed. Still, the exact degree computation can be the slowest part of a stage. If column j is a supernode containing $n(j)$ original columns, we define its approximate degree as

$$d(j) = \sum_{a_{ij}\neq 0} (\mathrm{nnz}(A(i,:)) - n(j)).$$

This can be interpreted as the sum of the sizes of the cliques containing j, except that j and the other columns in its supernode are not counted. This is a fairly good approximation in practice; it errs only by overcounting vertices that are members of at least three cliques containing j. George and Liu call such vertices "outmatched nodes," and observe that they tend to be rare in the symmetric algorithm. Computing approximate degrees takes only time proportional to the size of A.

Column minimum degree sometimes performs poorly if the matrix A has a few very dense rows, because then the structure of A^TA consists mostly of the cliques induced by those rows. Thus `colmmd` will withhold from consideration any row containing more than a fixed proportion (by default, 50%) of nonzeros.

All these options for minimum degree are under the user's control, though the casual user of MATLAB never needs to change the defaults. The default settings use approximate degrees, row reduction and supernode amalgamation every third stage, and a degree threshold of $1.2d+1$, and withhold rows that are at least 50% dense.

3.3.2. Permutations for sparsity: Symmetric matrices. Preorderings for Cholesky factorization apply symmetrically to the rows and columns of a symmetric positive definite matrix. Sparse MATLAB includes two symmetric preordering permutation functions. The `colperm` permutation can also be used as a symmetric ordering, but it is usually not the best choice.

Bandwidth-limiting and profile-limiting orderings are useful for matrices whose structure is "one-dimensional" in a sense that is hard to make precise. The reverse Cuthill-McKee ordering is an effective and inexpensive profile-limiting permutation. MATLAB function `p = symrcm(A)` returns a reverse Cuthill-McKee permutation for symmetric matrix A. The algorithm first finds a "pseudo-peripheral" vertex of the graph of A, then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudo-peripheral vertex. Our implementation is based closely on the SPARSPAK implementation as described in the book by George and Liu [11].

Profile methods like reverse Cuthill-McKee are not the best choice for most large matrices arising from problems with two or more dimensions, or problems without much geometric structure, because such matrices typically do not have reorderings with low profile. The most generally useful symmetric preordering in MATLAB is minimum degree, obtained by the function `p = symmmd(A)`. Our symmetric minimum degree implementation is based on the column minimum degree described in Section 3.3.1. In fact, `symmmd` just creates a nonzero structure K with a column for each column of A and a row for each above-diagonal nonzero in A, such that K^TK has the same nonzero structure as A; it then calls the column minimum degree code on K.

3.3.3. Nonzero diagonals and block triangular form.

A square nonsingular matrix A always has a row permutation p such that $A(p,:)$ has nonzeros on its main diagonal. The MATLAB function `p = dmperm(A)` computes such a permutation. With two output arguments, the function `[p,q] = dmperm(A)` gives both row and column permutations that put A into block upper triangular form; that is, $A(p,q)$ has a nonzero main diagonal and a block triangular structure with the largest possible number of blocks. Notice that the permutations p returned by these two calls are likely to be different.

The most common application of block triangular form is to solve a reducible system of linear equations by block back substitution, factoring only the diagonal blocks of the matrix. Figure 9 is an m-file that implements this algorithm. The m-file illustrates the call `[p,q,r] = dmperm(A)`, which returns p and q as before, and also a vector r giving the boundaries of the blocks of the block upper triangular form. To be precise, if there are b blocks in each direction, then r has length $b+1$, and the i-th diagonal block of $A(p,q)$ consists of rows and columns with indices from $r(i)$ through $r(i+1)-1$.

Any matrix, whether square or not, has a form called the "Dulmage-Mendelsohn decomposition" [4, 20], which is the same as ordinary block upper triangular form if the matrix is square and nonsingular. The most general form of the decomposition, for arbitrary rectangular A, is `[p,q,r,s] = dmperm(A)`. The first two outputs are permutations that put $A(p,q)$ into block form. Then r describes the row boundaries of the blocks and s the column boundaries: the i-th diagonal block of $A(p,q)$ has rows $r(i)$ through $r(i+1)-1$ and columns $s(i)$ through $s(i+1)-1$. The first diagonal block may have more columns than rows, the last diagonal block may have more rows than columns, and all the other diagonal blocks are square. The subdiagonal blocks are all zero. The square diagonal blocks have nonzero diagonal elements. All the diagonal blocks are irreducible; for the non-square blocks, this means that they have the "strong Hall property" [4]. This block form can be used to solve least squares

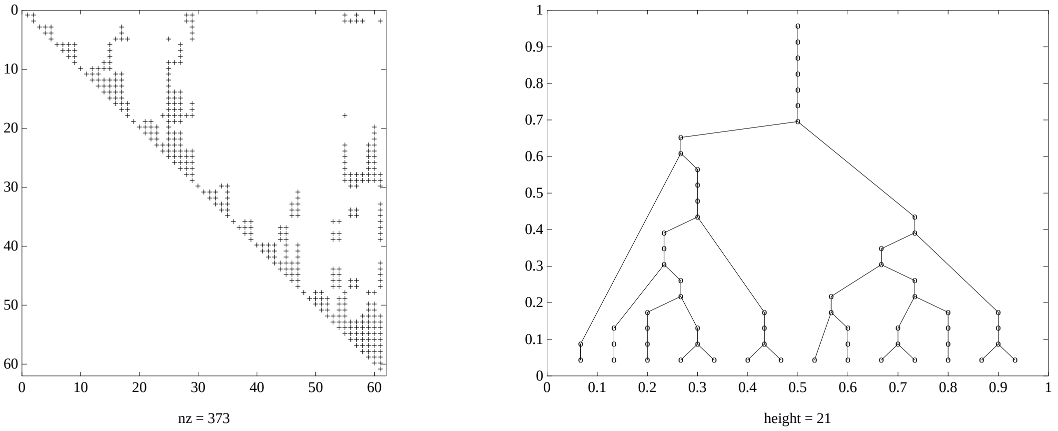

FIG. 3. *The Cholesky factor of a matrix and its elimination tree.*

problems by a method analogous to block back-substitution; see the references for more details.

3.3.4. Elimination trees. The elimination tree [21] of a symmetric positive definite matrix describes the dependences among rows or columns in Cholesky factorization. Liu [16] surveys applications of the elimination tree in sparse factorization. The nodes of the tree are the integers 1 through n, representing the rows of the matrix and of its upper triangular Cholesky factor. The parent of row i is the smallest $j > i$ such that the (i, j) element of the upper triangular Cholesky factor of the matrix is nonzero; if row i of the factor is zero after the diagonal, then i is a root. If the matrix is irreducible then its only root is node n.

Liu describes an algorithm to find the elimination tree without forming the Cholesky factorization, in time almost linear in the size of the matrix. That algorithm is implemented as the MATLAB function `[t,q] = etree(A)`. The resulting tree is represented by a row vector t of parent pointers: $t(i)$ is the parent of node i, or zero if i is a root.

The optional second output q is a permutation vector which gives a postorder permutation of the tree, or of the rows and columns of A. This permutation reorders the tree vertices so that every subtree is numbered consecutively, with the subtree's root last. This is an "equivalent reordering" of A, to use Liu's terminology: the Cholesky factorization of $A(q, q)$ has the same fill, operation count, and elimination tree as that of A. The permutation brings together the "fundamental supernodes" of A, which are full blocks in the Cholesky factor whose structure can be exploited in vectorized or parallel supernodal factorization [2, 17].

The postorder permutation can also be used to lay out the vertices for a picture of the elimination tree. The function `tspy(A)` plots a picture of the elimination tree of A, as shown in Figure 3.

3.4. Matrix division. The usual way to solve systems of linear equations in MATLAB is not by calling `lu` or `chol`, but with the matrix division operators / and \\. If A is square, the result of `X = A\B` is the solution to the linear system $AX = B$; if A is not square then a least squares solution is computed. The result of `X = A/B` is the solution to $A = XB$, which is `(B'\A')'`. Full MATLAB computes $A\backslash B$ by LU

factorization with partial pivoting if A is square, or by QR factorization with column pivoting if not.

3.4.1. The sparse linear equation solver. Like full MATLAB, sparse MATLAB uses direct factorization methods to solve linear systems. The philosophy behind this is that iterative linear system solvers are best implemented as MATLAB m-files, which can use the sparse matrix data structures and operations in the core of MATLAB.

If A is sparse, MATLAB chooses among a sparse triangular solve, sparse Cholesky factorization, and sparse LU factorization, with optional preordering by minimum degree in the last two cases. The result returned has the same storage class as B. The outline of sparse $A\backslash B$ is as follows.

- If A is not square, solve the least squares problem.
- Otherwise, if A is triangular, perform a sparse triangular solve for each column of B.
- Otherwise, if A is a permutation of a triangular matrix, permute it and then perform a sparse triangular solve for each column of B.
- Otherwise, if A is Hermitian and has positive real diagonal elements, find a symmetric minimum degree order p and attempt to compute the Cholesky factorization of $A(p,p)$. If successful, finish with two sparse triangular solves for each column of B.
- Otherwise (if A is not Hermitian with positive diagonal or if Cholesky factorization fails), find a column minimum degree order p, compute the LU factorization with partial pivoting of $A(:,p)$, and perform two sparse triangular solves for each column of B.

Section 3.5 describes the sparse least squares method we currently use.

For a square matrix, the four possibilities are tried in order of increasing cost. Thus, the cost of checking alternatives is a small fraction of the total cost. The test for triangular A takes only $O(n)$ time if A is n by n; it just examines the first and last row indices in each column. (Notice that a test for triangularity would take $O(n^2)$ time for a full matrix.) The test for a "morally triangular" matrix, which is a row and column permutation of a nonsingular triangular matrix, takes time proportional to the number of nonzeros in the matrix and is in practice very fast. (A Dulmage-Mendelsohn decomposition would also detect moral triangularity, but would be slower.) These tests mean that, for example, the MATLAB sequence

```
[L,U] = lu(A);
y = L\b;
x = U\y;
```

will use triangular solves for both matrix divisions, since L is morally triangular and U is triangular.

The test for Hermitian positive diagonal is an inexpensive guess at when to use Cholesky factorization. Cholesky is quite a bit faster than LU, both because it does half as many operations and because storage management is simpler. (The time to look at every element of A in the test is insignificant.) Of course it is possible to construct examples in which Cholesky fails only at the last column of the reordered matrix, wasting significant time, but we have not seen this happen in practice.

The function `spparms` can be used to turn the minimum degree preordering off if the user knows how to compute a better preorder for the particular matrix in question.

MATLAB's matrix division does not have a block-triangular preordering built in, unlike (for example) the Harwell `MA28` code. Block triangular preordering and solution can be implemented easily as an m-file using the `dmperm` function; see Section 4.3.

Full MATLAB uses the LINPACK condition estimator and gives a warning if the denominator in matrix division is nearly singular. Sparse MATLAB should do the same, but the current version does not yet implement it.

3.4.2. Sparse triangular systems.

The triangular linear system solver, which is also the main step of LU factorization, is based on an algorithm of Gilbert and Peierls [15]. When A is triangular and b is a sparse vector, $x = A\backslash b$ is computed in two steps. First, the nonzero structures of A and b are used (as described below) to make a list of the nonzero indices of x. This list is also the list of columns of A that participate nontrivially in the triangular solution. Second, the actual values of x are computed by using each column on the list to update the sparse accumulator with a "spaxpy" operation (Section 3.1.3). The list is generated in a "topological" order, which is one that guarantees that x_i is computed before column i of A is used in a spaxpy. Increasing order is one topological order of a lower triangular matrix, but any topological order will serve.

It remains to describe how to generate the topologically ordered list of indices efficiently. Consider the directed graph whose vertices are the columns of A, with an edge from j to i if $a_{ij} \neq 0$. (No extra data structure is needed to represent this graph—it is just an interpretation of the standard column data structure for A.) Each nonzero index of b corresponds to a vertex of the graph. The set of nonzero indices of x corresponds to the set of all vertices of b, plus all vertices that can be reached from vertices of b via directed paths in the graph of A. (This is true even if A is not triangular [12].) Any graph-searching algorithm could be used to identify those vertices and find the nonzero indices of x. A depth-first search has the advantage that a topological order for the list can be generated during the search. We add each vertex to the list at the time the depth-first search backtracks from that vertex. This creates the list in the reverse of a topological order; the numerical solution step then processes the list backwards, in topological order.

The reason to use this "reverse postorder" as the topological order is that there seems to be no way to generate the list in increasing or decreasing order, and the time wasted in sorting it would often be more than the number of arithmetic operations. However, the depth-first search examines just once each nonzero of A that participates nontrivially in the solve. Thus generating the list takes time proportional to the number of nonzero arithmetic operations in the numerical solve. This means that LU factorization can run in time proportional to arithmetic operations.

3.5. Least squares and the augmented system.

We have not yet written a sparse QR factorization for the core of MATLAB. Instead, linear least squares problems of the form

$$\min \|b - Ax\|$$

are solved via the augmented system of equations

$$\begin{aligned} r + Ax &= b \\ A^T r &= 0. \end{aligned}$$

Introducing a residual scaling parameter α this can be written

$$\begin{pmatrix} \alpha I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} r/\alpha \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}.$$

The augmented matrix, which inherits any sparsity in A, is symmetric, but clearly not positive definite. We ignore the symmetry and solve the linear system with a general sparse LU factorization, although a symmetric, indefinite factorization might be twice as fast.

A recent note by Björck [3] analyzes the choice of the parameter α by bounding the effect of roundoff errors on the error in the computed solution x. The value of α which minimizes the bound involves two quantities, $||r||$ and the smallest singular value of A, which are too expensive to compute. Instead, we use an apparently satisfactory substitute,

$$\alpha = \max |a_{ij}|/1000.$$

This approach has been used by several other authors, including Arioli et al. [1], who do use a symmetric factorization and a similar heuristic for choosing α.

It is not clear whether augmented matrices, orthogonal factorizations, or iterative methods are preferable for least squares problems, from either an efficiency or an accuracy point of view. We have chosen the augmented matrix approach because it is competitive with the other approaches, and because we could use exisiting code.

3.6. Eigenvalues of sparse matrices. We expect that most eigenvalue computations involving sparse matrices will be done with iterative methods of Lanczos and Arnoldi type, implemented outside the core of MATLAB as m-files. The most time-consuming portion will be the computation of Ax for sparse A and dense x, which can be done efficiently using our core operations.

However, we do provide one almost-direct technique for computing all the eigenvalues (but not the eigenvectors) of a real symmetric or complex Hermitian sparse matrix. The reverse Cuthill-McKee algorithm is first used to provide a permutation which reduces the bandwidth. Then an algorithm of Schwartz [22] provides a sequence of plane rotations which further reduces the bandwidth to tridiagonal. Finally, the symmetric tridiagonal QR algorithm from dense MATLAB yields all the eigenvalues.

4. Examples.

This section gives the flavor of sparse MATLAB by presenting several examples. First, we show the effect of reorderings for sparse factorization by illustrating a Cholesky factorization with several different permutations. Then we give two examples of m-files, which are programs written in the MATLAB language to provide functionality that is not implemented in the "core" of MATLAB. These sample m-files are simplified somewhat for the purposes of presentation. They omit some of the error-checking that would be present in real implementations, and they could be written to contain more flexible options than they do.

4.1. Effect of permutations on Cholesky factors. This sequence of examples illustrates the effect of reorderings on the computation of the Cholesky factorization of one symmetric test matrix. The matrix is $S = WW^T$ where W is the Harwell-Boeing matrix `WEST0479` [6], a model due to Westerberg of an eight-stage chemical distillation column.

There are four figures. Each figure shows two `spy` plots, first a particular symmetric permutation of S and then the Cholesky factor of the permuted matrix. The

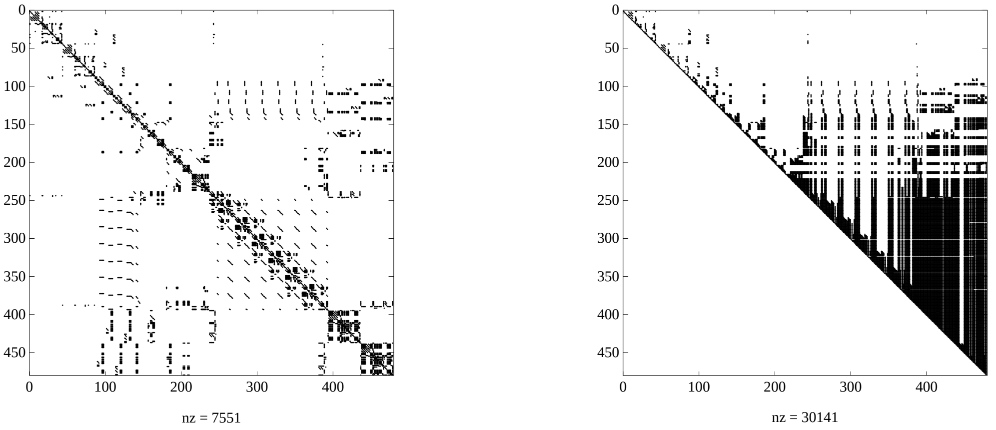

FIG. 4. *The structure of S and its Cholesky factor.*

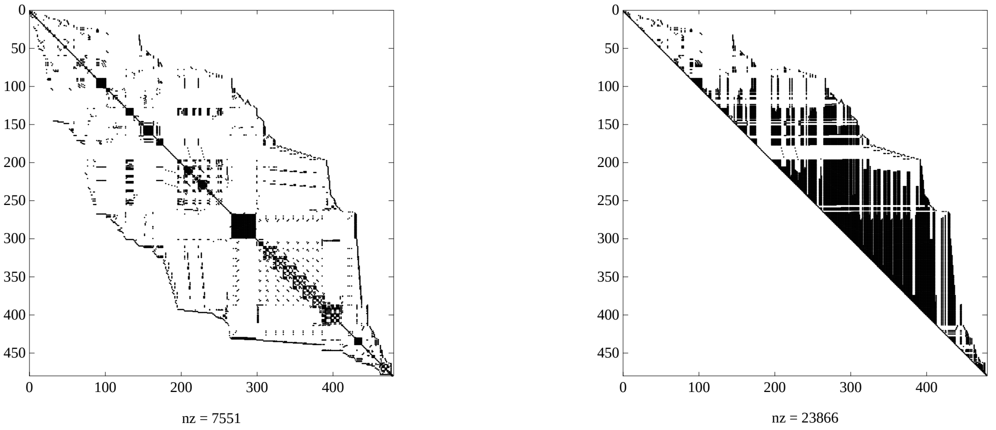

FIG. 5. *Matrix S and its Cholesky factor after reverse Cuthill-McKee reordering.*