Fundamentals *of* Algorithms

Direct Methods for Sparse Linear Systems

Timothy A. Davis

siam

Direct Methods for Sparse Linear Systems

Fundamentals of Algorithms

Editor-in-Chief: Nicholas J. Higham, University of Manchester

The SIAM series on Fundamentals of Algorithms publishes monographs on state-of-the-art numerical methods to provide the reader with sufficient knowledge to choose the appropriate method for a given application and to aid the reader in understanding the limitations of each method. The monographs focus on numerical methods and algorithms to solve specific classes of problems and are written for researchers, practitioners, and students.

The goal of the series is to produce a collection of short books, written by experts on numerical methods, that include an explanation of each method and a summary of theoretical background. What distinguishes a book in this series is both its emphasis on explaining how to best choose a method, algorithm, or software package to solve a specific type of problem and its descriptions of when a given algorithm or method succeeds or fails.

Series Volumes

Davis, T. A. *Direct Methods for Sparse Linear Systems*

Kelley, C. T. *Solving Nonlinear Equations with Newton's Method*

Timothy A. Davis
University of Florida
Gainesville, Florida

Direct Methods for Sparse Linear Systems

siam
Society for Industrial and Applied Mathematics
Philadelphia

10 9 8 7 6 5 4 3 2 1

The algorithms presented in this book were developed with support from various sources, including Sandia National Laboratory, the National Science Foundation (ASC-9111263, DMS-9223088, DMS-9504974, DMS-9803599, and CCR-0203720), The MathWorks, Inc., and the University of Florida.

Library of Congress Cataloging-in-Publication Data

Davis, Timothy A.
Direct methods for sparse linear systems / Timothy A. Davis.
p. cm. — (Fundamentals of algorithms)
Includes bibliographical references and index.
ISBN-13: 978-0-898716-13-9 (pbk.)
ISBN-10: 0-89871-613-6 (pbk.)
1. Sparse matrices. 2. Linear systems. I. Title.

QA188.D386 2006
512.9'434—dc22

2006044387

ISBN-13: 978-0-898716-13-9
ISBN-10: 0-89871-613-6

For Connie, Emily, and Timothy J. ("TJ")

This page intentionally left blank

Contents

This page intentionally left blank

Preface

This book presents the fundamentals of sparse matrix algorithms, from theory to algorithms and data structures to working code. The focus is on direct methods for solving systems of linear equations; iterative methods and solvers for eigenvalue problems are beyond the scope of this book.

The goal is to impart a working knowledge of the underlying theory and practice of sparse matrix algorithms, so that you will have the foundation to understand more complex (but faster) algorithms. Methods that operate on dense submatrices of a larger sparse matrix (multifrontal and supernodal methods) are much faster, but a complete sparse matrix package based on these methods can be tens of thousands of lines long. The sparse LU, Cholesky, and QR factorization codes in MATLAB®, for example, total about 100,000 lines of code. Trying to understand the sparse matrix technique by starting with such huge codes is a daunting task. To overcome this obstacle, a sparse matrix package, CSparse,[1] has been written specifically for this book.[2] It can solve $Ax = b$ when A is unsymmetric, symmetric positive definite, or rectangular, using about 2,200 lines of code. Although simple and concise, it is based on recently developed methods and theory. All of CSparse is printed in this book. Take your time to read and understand these codes; do not gloss over them. You will find them much easier to comprehend and learn from than their larger (yet faster) cousins. The larger packages you may use in practice are based on much of the theory and some of the algorithms presented more concisely and simply in CSparse. For example, the MATLAB statement `x=A\b` relies on the theory and algorithms from almost every section of this book. Parallel sparse matrix algorithms are excluded, yet they too rely on the theory discussed here.

For the computational scientist with a problem to solve using sparse matrix methods, these larger packages may be faster, but you need to understand how they work to use them effectively. They might not have every function needed to interface them into your application. You may need to write some code of your own to manipulate your matrix prior to or after using a large sparse matrix package. One of the goals of this book is to equip you for these tasks. The same question applies to MATLAB. You might ask, "*What is the most efficient way of solving my sparse matrix problem in MATLAB?*" The short answer is to always operate on whole matrices, large submatrices, or column vectors in MATLAB and to not rely

[1] CSparse: a Concise Sparse matrix package.

[2] The index gives page numbers in bold that contain CSparse and related software.

heavily on accessing the rows or individual entries of a sparse matrix. The long answer to this question is to read this book. MATLAB and the C programming language are a strong emphasis of this book. In particular, one goal of the book is to explain how MATLAB performs its sparse matrix computations.

Algorithms are presented in a mixture of pseudocode, MATLAB, and C, so knowledge of these is assumed. Also required is a basic knowledge of linear algebra, graph theory, algorithms, and data structures. A short review of these topics is provided. Each chapter includes a set of exercises to reinforce the topic.[3]

CSparse is written in C, using a spartan coding style. Using C instead of (say) Java or C++ allows for concise exposition, full disclosure of time and memory complexity, efficiency, and portability. CSparse can be downloaded from SIAM at www.siam.org/books/fa02. MATLAB 7.2 (R2006a) was used for this book. CSparse handles only real matrices and `int` integers. CXSparse is an extended version that includes support for real and complex matrices and `int` and `long` integers and can also be downloaded from www.siam.org/books/fa02.

The genesis of this book was a collection of lecture notes for a course on sparse matrix algorithms I taught at Stanford in 2003. I would like to thank Gene Golub, Esmond Ng, and Horst Simon for enabling me to spend a sabbatical at Stanford and Lawrence Berkeley National Laboratory for the 2002–2003 academic year. Several extended visits to Sandia National Laboratory at Mike Heroux's invitation enabled me to develop my versions of the left-looking sparse LU factorization algorithm and the Dulmage–Mendelsohn decomposition for use in Sandia's circuit simulation efforts. The algorithms presented here were developed with support from various sources, including Sandia National Laboratory, the National Science Foundation (ASC-9111263, DMS-9223088, DMS-9504974, DMS-9803599, and CCR-0203720), The MathWorks, Inc., and the University of Florida. I would like to thank David Bateman for adding support for complex matrices and `long` integers to CXSparse.

Nick Higham, Cleve Moler, and the other members of the Editorial Board of the SIAM Fundamentals of Algorithms book series encouraged me to turn these lecture notes and codes into the printed page before you by inviting me to write this book for the series. Finally, I would like to thank David Day, John Gilbert, Chen Greif, Nick Higham, Sara Murphy, Pat Quillen, David Riegelhaupt, Ken Stanley, Linda Thiel, and my Spring 2006 sparse matrix class (Suranjit Adhikari, Pawan Aurora, Okiemute Brume, Yanqing "Morris" Chen, Eric Dattoli, Bing Jian, Nick Lord, Siva Rajamanickam, and Ozlem Subakan), who provided helpful feedback on the content and presentation of the book.

Tim Davis
University of Florida, Gainesville, Florida
www.cise.ufl.edu/~davis
April 2006

[3]Instructors: please do not post solutions on the web where they are publicly readable. Use a password-protected web page instead.

Chapter 1
Introduction

This book presents the fundamentals of sparse matrix algorithms for the direct solution of sparse linear systems, from theory to algorithms and data structures to working code. The algorithms presented here have been chosen with these goals in mind: they must embody much of the theory behind sparse matrix algorithms; they must be either asymptotically optimal in their run time and memory usage or be fast in practice; they must be concise so as to be easy to understand and short enough to print in their entirety in this book; they must cover a wide spectrum of matrix operations; and they must be accurate and robust.

Algorithms are presented in a mixture of pseudocode, MATLAB, and C, so knowledge of these is assumed. Also required is a basic knowledge of linear algebra, graph theory, algorithms, and data structures. This background is reviewed below and in an appendix on the C programming language.

Chapter 2 presents basic data structures and algorithms, including matrix multiplication, addition, transpose, and data structure manipulations. Chapter 3 considers the solution of triangular systems of equations. Chapters 4 through 6 present the three most commonly used decompositions: Cholesky, QR, and LU. Factorization methods specifically for symmetric indefinite matrices are not discussed. Section 4.10 presents a method for updating and downdating a sparse Cholesky factorization after a low-rank change. Chapter 7 discusses ordering methods that reduce work and memory requirements. Chapter 8 draws on the theory and algorithms presented in Chapters 1 through 7 to solve a sparse linear system $Ax = b$, where A can be symmetric positive definite, unsymmetric, or rectangular, just like the backslash operator in MATLAB, `x=A\b`, when `A` is sparse and `b` is a dense column vector. Chapter 9 is a summary of the CSparse sparse matrix package. Finally, Chapter 10 explains how to use sparse matrices in MATLAB.

To avoid breaking the flow of discussion, few citations appear in the body of each chapter. They are discussed at the end of each chapter instead in a "Further reading" section, which gives an overview of software, books, and papers related to that chapter. Notable exceptions to this rule are the theorems stated in the book. The final section in each chapter is a set of exercises for that chapter.

1.1 Linear algebra

Definitions and notation for linear algebra are briefly described below. An m-by-n *matrix* is denoted with a capital letter, $A \in \mathbb{R}^{m \times n}$, where $\mathbb{R}^{m \times n}$ denotes the set of m-by-n matrices with real elements. An ij subscript on the corresponding lower case letter a_{ij} denotes an entry in row i and column j of the matrix A. It can also denote a row vector, column vector, or block submatrix of A (described below), but in this case i and j are typically small constants (a_{12}, for example). L is a lower triangular matrix, U and R are upper triangular, and I denotes the *identity matrix* (which is zero except for $(I)_{ii} = 1$). A matrix A is *lower triangular* if $a_{ij} = 0$ when $i < j$, and *upper triangular* if $a_{ij} = 0$ when $i > j$. Lower case letters denote vectors (b, x, y, z), except for letters i through n, which always denote integer scalars. Row vectors are shown with a superscript, x^T. With a subscript, x_i can denote either a scalar or a vector, depending on the context. Lower case Greek letters (α, β) represent scalars. The jth column of the matrix A is denoted A_{*j}, or `A(:,j)` in MATLAB. Likewise, A_{i*} denotes row i of A, or `A(i,:)` in MATLAB.

The *transpose* A^T of the real matrix A is defined by $(A^T)_{ij} = a_{ji}$. The matrix *addition* $C = A + B$ is defined by $c_{ij} = a_{ij} + b_{ij}$. A scalar can be multiplied by a matrix; $C = \alpha A$ is defined by $c_{ij} = \alpha a_{ij}$. Matrix *multiplication*, $C = AB$, is

$$c_{ij} = \sum_{k=1}^{s} a_{ik} b_{kj}, \tag{1.1}$$

where $C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times s}$, and $B \in \mathbb{R}^{s \times n}$. Matrix multiplication requires the number of columns of A to be equal to the number of rows of B. The *dot product* of two vectors x and y is the scalar $x^T y$. The *outer product* of two vectors x and y is the matrix xy^T; the computation of $A = A + \alpha xy^T$ is called a rank-1 *update*. If X and Y are matrices with k columns, XY^T is referred to as a *rank-k outer product* (this nomenclature is commonly used even if X and Y do not have a numerical rank of k). The computation $A = A + \alpha XY^T$ is called a rank-k update.

A *block matrix* is a matrix where each entry can be a matrix, vector, or scalar. Suppose the rows of an m-by-n matrix A are partitioned into r subsets, and the columns are partitioned into c subsets; A can be written as a block r-by-c matrix

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1c} \\ \vdots & & \vdots \\ A_{r1} & \cdots & A_{rc} \end{bmatrix},$$

where A_{ij} is m_i-by-n_j, if m_i is the size of the ith row subset and n_j is the size of the jth column subset. Two block matrices can be added if they are partitioned identically. Two block matrices can be multiplied, $C = AB$, if the columns of A are partitioned identically to the rows of B; the rows of C and A are partitioned identically, as are the columns of C and B. If c is the number of partitions of the columns of A and rows of B, (1.1) becomes

$$C_{ij} = \sum_{k=1}^{c} A_{ik} B_{kj}.$$

A set of vectors $a_1, a_2, \ldots, a_n$ is *linearly independent* if $\sum \alpha_j a_j = 0$ implies α_j is zero for all j. The *span* of a set of vectors is the set of vectors that can be written as a linear combination of vectors in the set; $\text{span}(a_1, a_2, \ldots, a_n) = \{\sum \alpha_j a_j\}$. The *range* of a matrix A is the span of its column vectors. The *rank* of a matrix A is the maximal size of the subsets of the columns of A that are linearly independent. An n-by-n matrix is *singular* if its rank is less than n. An m-by-n matrix is *rank deficient* if its rank is less than $\min(m, n)$; it has *full rank* otherwise.

The 1-*norm* of a column vector x or row vector x^T is $\|x\|_1 = \sum |x_i|$, its 2-norm is $\|x\|_2 = \sqrt{\sum x_i^2}$, and its ∞-norm is $\|x\|_\infty = \max |x_i|$. The 1-norm of a matrix is the largest 1-norm of its column vectors. The ∞-norm of a matrix is the largest 1-norm of its row vectors.

The *inverse* of a matrix A is A^{-1}, where $AA^{-1} = A^{-1}A = I$. It exists only if A is square and nonsingular. Two vectors x and y are *orthogonal* if $x^T y = 0$. A matrix Q is *orthonormal* if $Q^T Q = I$. A real square orthonormal Q matrix is called orthogonal, in which case $Q^T Q = QQ^T = I$ (that is, $Q^T = Q^{-1}$ if Q is orthogonal). The 2-norm of a vector x and the product Qx are identical if Q is orthogonal.

The kth *diagonal* of an m-by-n matrix A is a vector d consisting of the set of entries $\{a_{ij}\}$, where $j - i = k$. The term *diagonal*, by itself, refers to the 0th diagonal, or main diagonal, of a matrix. The kth diagonal entry of A is a_{kk}.

The number of nonzero entries (*nonzeros* for short) in a matrix or vector is $|A|$, and $|a|$ denotes the absolute value of a scalar.

A *permutation matrix* P is a row or column permutation of the identity matrix. Any given row or column of P contains a single nonzero entry, equal to 1. The *LU factorization* of a square nonsingular matrix A has the form $LU = A$, where L is lower triangular and U is upper triangular. With *partial pivoting* and row interchanges, the factorization is $LU = PA$. A matrix A is *positive definite* if and only if $x^T Ax > 0$ for all nonzero vectors x. It is *positive semidefinite* if $x^T Ax \geq 0$. The *Cholesky factorization* of a square symmetric positive definite matrix A has the form $LL^T = A$, where L is lower triangular with positive diagonal entries. Pivoting is not required for stability. A square matrix A is *diagonally dominant* by rows if $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$ for all i. It is strictly diagonally dominant by rows if $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$ for all i. It is (strictly) diagonally dominant by columns if A^T is (strictly) diagonally dominant by rows. A square strictly diagonally dominant matrix is nonsingular. *Gaussian elimination* without pivoting (a form of LU factorization) is stable for any diagonally dominant matrix (by rows or by columns).

A *QR factorization* of a rectangular matrix A is $QR = A$, where Q is orthogonal and R is upper triangular. For a square matrix A, $Ax = \lambda x$ holds for an *eigenvalue* λ and its *eigenvector* x.

Sets are denoted in calligraphic letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{L}$, $\mathcal{R}$, $\mathcal{V}$, $\mathcal{W}$, $\mathcal{X}$, and $\mathcal{Y}$. These typically arise from the nonzero pattern of the corresponding matrix or vector. For example, $\mathcal{A}_{*j} = \{i \,|\, a_{ij} \neq 0\}$, and $\mathcal{X} = \{i \,|\, x_i \neq 0\}$. The $*$ in the subscript is dropped when the context is clear.

The terms *dense* and *sparse* refer to the data structure used to store a matrix. A matrix $A \in \mathbb{R}^{m \times n}$ is dense if it is stored as a full array of m rows and n columns with mn entries. This is called a `full` matrix in MATLAB. All entries are stored,

even if some of them are zero. A sparse matrix is stored in a data structure that can exploit sparsity by not storing numerically zero entries. Numerically zero entries may be stored in a sparse matrix, typically as a result of numerical cancellation.

1.2 Graph theory, algorithms, and data structures

A *graph* $G = (V, E)$ is a set of *nodes* $V = \{1, \ldots, n\}$ and a set of *edges* $E = \{(i, j) \mid i, j \in V\}$ connecting those nodes. In an *undirected* graph, (i, j) and (j, i) are the same edge; they are different edges in a *directed* graph. In a directed graph, there can be an edge (i, j) but no edge (j, i). The *neighbors* of a node i, or equivalently the *adjacency set* of i, is $\mathrm{Adj}(i) = \{j \mid (i, j) \in E\}$. For a directed graph, this is the *out-adjacency* of node i. The *in-adjacency* of node i is $\mathrm{InAdj}(i) = \{j \mid (j, i) \in E\}$. The out-adjacency and in-adjacency of a node in an undirected graph are identical. The *adjacency matrix* A of a graph G is a binary n-by-n matrix with $a_{ij} = 1$ if $(i, j) \in E$, and $a_{ij} = 0$ otherwise. It is symmetric if G is undirected, and may be unsymmetric otherwise. A graph can also be represented as a set of n *adjacency lists*, $\mathcal{A}_{i*} = \{j \mid a_{ij} \neq 0\}$ or $\mathcal{A}_{*j} = \{i \mid a_{ij} \neq 0\}$. This relates the sparsity pattern of an n-by-n matrix A to its corresponding undirected or directed graph G_A. The diagonal a_{kk} is the self-edge (k, k); it is typically excluded from G_A. The *degree* of a node in an undirected graph is the size of its adjacency list, $|\mathcal{A}_{*i}| = |\mathcal{A}_{i*}| = |\mathrm{Adj}(i)|$. Two graphs are *isomorphic* if they can be made identical by renumbering their nodes.

A node i is *incident* on the edge (a, b) if $a = i$ or $b = i$; the edge (a, b) is also said to be incident on node i. A *node-induced subgraph* $\overline{G} = (\overline{V}, \overline{E})$ of G is defined by a subset of the nodes $\overline{V} \subseteq V$, where $\overline{E} = \{(i, j) \mid (i, j) \in E \wedge i \in \overline{V} \wedge j \in \overline{V}\}$. An *edge-induced subgraph* $\overline{G} = (\overline{V}, \overline{E})$ is defined by a set of edges $\overline{E} \subseteq E$, where the nodes $\overline{V}$ are all the nodes incident on any edge in $\overline{E}$.

A graph is completely connected if $E = V \times V$; it has an edge between every pair of nodes. A *clique* is a completely connected subgraph.

A *path* $v_0 \rightsquigarrow v_k$ of length k is a sequence of nodes $(v_0, \ldots, v_k)$ where an edge (v_{i-1}, v_i) exists between each pair of adjacent nodes in the sequence. The path is simple if no node appears more than once. Node j is *reachable* from node i if a path $i \rightsquigarrow j$ exists in the graph. The set of all nodes reachable from i in the graph G is $\mathrm{Reach}_G(i)$. A graph is *strongly connected* if there exists a path from any node to any other node. That is, $\mathrm{Reach}_G(i)$ is the entire graph for any node i. A graph is *connected* if its underlying undirected graph is strongly connected. The *underlying* undirected graph of G is the same as G but with all the edge directions ignored.

A *cycle* is a path $i \rightsquigarrow i$ where the first and last nodes are the same. It is a simple cycle if no edge or node appears more than once, except for i itself. A graph with no cycles is *acyclic*. A directed acyclic graph is often called a *DAG*, for short. An undirected acyclic graph is a *forest*. A *tree* is a connected forest. There is a unique simple path between every pair of nodes in a tree. One node of a rooted tree is designated as the *root* r. The unique node following i in the path $i \rightsquigarrow r$ is called the *parent* p of i; the root itself has no parent. Node i is a *child* of p; a node has at most one parent but can have more than one child. A node with no children is a *leaf*. The path $i \rightsquigarrow r$ is the set of *ancestors* of i. Node i is a *descendant* of all nodes

in the path $i \rightsquigarrow r$. The set of *proper* ancestors or descendants of node i excludes node i itself. The *subtree* rooted at node i consists of the subgraph induced by the nodes i and its descendants. In a *postordered tree*, the d proper descendants of any node k are nodes $k-d$ through $k-1$.

A *bipartite* graph is an undirected graph whose nodes are partitioned into two subsets, and every edge connects two nodes in different subsets. A *node separator* is a subset $S \subseteq V$ of the nodes of $G = (V, E)$ such that the graph induced by the nodes $V \setminus S$ is unconnected. An *edge separator* is a subset $S \subseteq E$ of the edges of $G = (V, E)$ such that the graph $G = (V, E \setminus S)$ is unconnected. A *node cover* is a set of nodes $S \subseteq V$ such that all edges E are incident on at least one node in S.

Informally, *asymptotic notation* describes the gist of a function $f(x)$: how fast it grows as a function of x. Scalar factors and lower order terms are ignored. For example, all quadratic polynomials $ax^2 + bx + 1$ are $O(x^2)$, unless $a = 0$. More formally, a function $f(x)$ is $O(g(x))$ if there exist positive constants c and x_0 such that $0 \leq f(x) \leq cg(x)$ for all $x \geq x_0$. This provides an asymptotic upper bound. The asymptotic lower bound is defined similarly. A function $f(x)$ is $\Omega(g(x))$ if there exist positive constants c and x_0 such that $0 \leq cg(x) \leq f(x)$ for all $x \geq x_0$. If $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$, then it has a tight asymptotic bound, $\Theta(g(x))$.

Algorithm analysis refers to methods for determining asymptotic bounds on the run time, memory usage, and other aspects of an algorithm. The run time and other statistics are typically expressed as functions of the size of the input. Worst-case analysis finds an upper bound ($O(...)$) that always holds, regardless of the input. Average-case analysis looks at the typical case. These are often the same but not always. The time complexity of an algorithm is the asymptotic bound on its run time.

An *amortized analysis* considers the total run time of a sequence of related operations. If the time for n operations is $T(n)$, the amortized time for one operation is $T(n)/n$. For example, if a function is called n times and takes one unit of time $n-1$ times but $n+1$ units of time just once, the worst-case time complexity is $O(n)$ for any one usage of this function. However, if the cost of all n operations are added together and amortized across all operations, the amortized cost per operation becomes just two, which is $O(1)$. The latter is a more useful picture of the time complexity of one operation in a sequence.

An example of where amortized time complexity is useful is the *dynamic table* algorithm. Consider a table of size k that starts out with $k = 1$ and an operation that inserts an item at the end of the table. If the size k of the table is insufficient, it is doubled in size, taking $O(k)$ time to make the larger copy. With n insertions, it may appear that total time would be $O(n^2)$, but this is not the case. Even with this extra work of copying the table, the amortized cost of a single insertion is $O(1)$, because only the ith insertions when i is a power of two plus one require a complete copy of the table. For n insertions, the time is

$$n + \sum_{i=0}^{\lfloor \log_2(n-1) \rfloor} 2^i \leq 3n,$$

where $\lfloor x \rfloor$ is the largest integer not greater than x. The total time for n insertions

is less than or equal to $3n$. The amortized time for any one insertion is at most 3.

A common class of graph algorithms consists of methods for traversing the nodes and edges of a graph. The *depth-first search* of a graph starts at a node j and finds all nodes reachable from node j. It explores recursively by always examining the outgoing edges of the latest node i just seen. When all edges of i have been explored, it backtracks to the node from which i was first discovered. Nodes are marked so that they are not searched twice. The time taken by a depth-first search is $O(s+e)$, where $s = |\mathrm{Reach}(i)|$ and e is the number of edges in the subgraph induced by s. This subgraph is connected by the way it is constructed. Traversing the entire graph in a depth-first manner requires the traversal to be repeated until all nodes are visited. A depth-first search produces a list of nodes of a DAG in *topological order*; i appears before j if $i \rightsquigarrow j$ is a path in G.

The *breadth-first search* traverses a graph in a different order. Starting at node i, it first examines all nodes adjacent to i. Next, it examines all nodes j whose shortest path $i \rightsquigarrow j$ is of length 2, then length 3, and so on. Like the depth-first search, it too traverses all nodes in $\mathrm{Reach}_G(i)$. Unlike the depth-first search, it traverses these nodes in order of the shortest path from i, not in topological order.

A graph is denoted as G or $\mathcal{G}$, and $\mathcal{T}$ denotes a tree or forest. $\mathcal{E}$ denotes the element lists in the minimum degree ordering algorithm, discussed in Chapter 7.

1.3 Further reading

Golub and Van Loan [114] provide an in-depth coverage of numerical linear algebra and matrix computations for dense and structured matrices; Strang [193] gives an introduction to linear algebra. Moler [157] provides an introduction to numerical computing with a strong MATLAB focus. Stewart presents an in-depth look at matrix decompositions [190] and eigenvalue problems [191]. Higham [135] discusses the behavior of numerical algorithms in finite precision arithmetic.

Cormen, Leiserson, and Rivest [23] discuss algorithms and data structures and their analysis, including graph algorithms. Kernighan and Ritchie [141] give a concise coverage of the C programming language. Higham and Higham [133], Davis and Sigmon [38], or the online documentation for MATLAB are good places to learn more about MATLAB.

The books by Duff, Erisman, and Reid [53] and George and Liu [89] both deal with direct methods for sparse matrices; the latter focuses on symmetric positive definite matrices. Gilbert [101] and Liu [150] provide an overview of much of the graph theory related to sparse direct methods. Stewart [192] provides a tutorial-level description of sparse Cholesky. Gould, Hu, and Scott [116] survey a wide range of software packages for the factorization of sparse symmetric matrices. Iterative methods for sparse linear systems and the incomplete factorization methods they rely on are discussed by Saad [178], Greenbaum [117], and Barrett et al. [15]. Björck [17] presents direct and iterative methods for sparse least squares problems. Parlett [164] provides an in-depth look at the symmetric eigenvalue problem for sparse matrices. Demmel [39] interleaves a discussion of numerical linear algebra with a description of related software for sparse and dense problems.

Chapter 2

Basic algorithms

A sparse matrix is one whose entries are mostly zero. There are many ways of storing a sparse matrix. Whichever method is chosen, some form of compact data structure is required that avoids storing the numerically zero entries in the matrix. It needs to be simple and flexible so that it can be used in a wide range of matrix operations. This need is met by the primary data structure in CSparse, a compressed-column matrix. Basic matrix operations that operate on this data structure are presented below, including matrix-vector multiplication, matrix-matrix multiplication, matrix addition, and transpose.

2.1 Sparse matrix data structures

The simplest sparse matrix data structure is a list of the nonzero entries in arbitrary order. The list consists of two integer arrays `i` and `j` and one real array `x` of length equal to the number of entries in the matrix. For example, the matrix

$$A = \begin{bmatrix} 4.5 & 0 & 3.2 & 0 \\ 3.1 & 2.9 & 0 & 0.9 \\ 0 & 1.7 & 3.0 & 0 \\ 3.5 & 0.4 & 0 & 1.0 \end{bmatrix} \tag{2.1}$$

is represented in *zero-based triplet* form below. A zero-based data structure for an `m`-by-`n` matrix contains row and column indices in the range `0` to `m-1` and `n-1`, respectively. A *one-based* data structure has row and column indices that start with one. The one-based convention is used in linear algebra and is presented to the MATLAB user. Internally in MATLAB and also in CSparse, all algorithms and data structures are zero-based. Thus, both conventions are used in this book, depending on the context. In particular, all C code is zero-based. All MATLAB expressions, and all linear algebraic expressions, are one-based. All pseudocode is zero-based, since it closely relates to a corresponding C code. Graph examples are one-based, since they usually relate to an example matrix (which are also one-based).

```
int i [ ]    = { 2,   1,   3,   0,   1,   3,   3,   1,   0,   2   } ;
int j [ ]    = { 2,   0,   3,   2,   1,   0,   1,   3,   0,   1   } ;
double x [ ] = { 3.0, 3.1, 1.0, 3.2, 2.9, 3.5, 0.4, 0.9, 4.5, 1.7 } ;
```

The triplet form is simple to create but difficult to use in most sparse matrix algorithms. The *compressed-column* form is more useful and is used in almost all functions in CSparse. An `m`-by-`n` sparse matrix that can contain up to `nzmax` entries is represented with an integer array `p` of length `n+1`, an integer array `i` of length `nzmax`, and a real array `x` of length `nzmax`. Row indices of entries in column `j` are stored in `i[p[j]]` through `i[p[j+1]-1]`, and the corresponding numerical values are stored in the same locations in `x`. The first entry `p[0]` is always zero, and `p[n]` $\leq$ `nzmax` is the number of actual entries in the matrix. The example matrix (2.1) is represented as

```
int p [ ]    = { 0,             3,             6,       8,       10 } ;
int i [ ]    = { 0,   1,   3,   1,   2,   3,   0,   2,   1,   3   } ;
double x [ ] = { 4.5, 3.1, 3.5, 2.9, 1.7, 0.4, 3.2, 3.0, 0.9, 1.0 } ;
```

MATLAB uses a compressed-column data structure much like `cs` for its sparse matrices. It requires the row indices in each column to appear in ascending order, and no zero entries may be present. Those two restrictions are relaxed in CSparse. The triplet form and the compressed-column data structures are both encapsulated in the `cs` structure:

```
typedef struct cs_sparse    /* matrix in compressed-column or triplet form */
{
    int nzmax ;     /* maximum number of entries */
    int m ;         /* number of rows */
    int n ;         /* number of columns */
    int *p ;        /* column pointers (size n+1) or col indices (size nzmax) */
    int *i ;        /* row indices, size nzmax */
    double *x ;     /* numerical values, size nzmax */
    int nz ;        /* # of entries in triplet matrix, -1 for compressed-col */
} cs ;
```

The array `p` contains the column pointers for the compressed-column form (of size `n+1`) or the column indices for the triplet form (of size `nzmax`). The matrix is in compressed-column form if `nz` is negative. Any given CSparse function expects its sparse matrix input in one form or the other, except for `cs_print`, `cs_spalloc`, `cs_spfree`, and `cs_sprealloc`, which can operate on either form.

Within a mexFunction written in C or Fortran (but callable from MATLAB), several functions are available that extract the parts of a MATLAB sparse matrix; `mxGetJc` returns a pointer to the equivalent of the `A->p` column pointer array of the `cs` matrix `A`. The functions `mxGetIr`, `mxGetPr`, `mxGetM`, `mxGetN`, and `mxGetNzmax` return `A->i`, `A->x`, `A->m`, `A->n`, and `A->nzmax`, respectively. These `mx` functions are not available to a MATLAB statement typed in the MATLAB command window or in a MATLAB M-file but only in a compiled C or Fortran mexFunction. The compressed-column data structures used in MATLAB and CSparse are identical, except that MATLAB can handle complex matrices as well. MATLAB 7.2 forbids explicit zero entries and requires row indices to be in order in each column.

Access to a column of `A` is simple, equivalent to `c=A(:,j)` in MATLAB, where `j` is a scalar. This assignment takes $O(|c|)$ time in MATLAB, which is optimal. Accessing the rows of a sparse matrix in `cs` form, or in MATLAB, is difficult. The MATLAB statement `r=A(i,:)` for a scalar `i` accesses a row of `A`. To implement this, MATLAB must examine every column of `A`, looking for row index `i` in each column. This is costly compared with accessing a column. Transposing a sparse matrix and accessing its columns is better than repeatedly accessing its rows.

The `cs` data structure can contain numerically zero entries, which brings up the important practical and theoretical issue of numerical cancellation. Exact numerical cancellation is rare, and most algorithms ignore it. An entry in the data structure that is computed but found to be numerically zero is still called a "nonzero" in this book. Leaving these entries in the matrix leads to much simpler algorithms and more elegant graph theoretical statements about the algorithms, in particular matrix-matrix multiplication, factorization, and the solution of $Lx = b$ when b is sparse. Zero entries can always be dropped afterward (see Section 2.7); this is what MATLAB does. Modifying the nonzero pattern of a compressed-column matrix is not trivial. Deleting or adding single entries can take $O(|A|)$ time, since no gaps can appear between columns. For example, to delete the first entry in a matrix requires that all other entries be shifted up by one position. The MATLAB statements `A(1,1)=0 ; A(1,1)=1` are very costly because MATLAB always removes zero entries whenever they occur.

A *numerically rank-deficient* matrix is rank deficient in the usual sense. The *structural rank* of a matrix is the largest rank that can be obtained by reassigning the numerical values of the entries in its data structure. An m-by-n matrix is *structurally rank deficient* if its structural rank is less than $\min(m, n)$. For example, A is numerically rank deficient but has structural full rank, while C is both numerically and structurally rank deficient:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

2.2 Matrix-vector multiplication

One of the simplest sparse matrix algorithms is matrix-vector multiplication, $z = Ax + y$, where y and x are dense vectors and A is sparse. If A is split into n column vectors, the result $z = Ax + y$ is

$$z = \begin{bmatrix} A_{*1} & \dots & A_{*n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + y.$$

Allowing the result to overwrite the input vector y, the jth iteration computes $y = y + A_{*j}x_j$. The pseudocode for computing $y = Ax + y$ is given below.

for $j = 0$ to $n - 1$ **do**
  for each i for which $a_{ij} \neq 0$ **do**
    $y_i = y_i + a_{ij}x_j$

Most algorithms are presented here directly in C, since the pseudocode directly translates into C with little modification. Below is the complete C version of the algorithm. Note how the `for (p = ...)` loop in the `cs_gaxpy` function takes the place of the **for each** i loop in the pseudocode (the name is short for generalized A times x plus y). The MATLAB equivalent of `cs_gaxpy(A,x,y)` is `y=A*x+y`. Detailed descriptions of the inputs, outputs, and return values of all CSparse functions are given in Chapter 9.

```
int cs_gaxpy (const cs *A, const double *x, double *y)
{
    int p, j, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !x || !y) return (0) ;       /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            y [Ai [p]] += Ax [p] * x [j] ;
        }
    }
    return (1) ;
}

#define CS_CSC(A) (A && (A->nz == -1))
#define CS_TRIPLET(A) (A && (A->nz >= 0))
```

The function first checks its inputs to ensure they exist, and returns false (zero) if they do not. This protects against a caller that ran out of memory. `CS_CSC(A)` is true for a compressed-column matrix; `CS_TRIPLET(A)` is true for a matrix in triplet form. The next line (`n=A->n ; ...`) extracts the contents of the matrix `A`—its dimension, column pointers, row indices, and numerical values.

2.3 Utilities

A sparse matrix algorithm such as `cs_gaxpy` requires a sparse matrix in `cs` form as input. A few utility functions are required to create this data structure. The `cs_malloc`, `cs_calloc`, `cs_realloc`, and `cs_free` functions are simple wrappers around the equivalent ANSI C or MATLAB memory management functions.

```
void *cs_malloc (int n, size_t size)
{
    return (malloc (CS_MAX (n,1) * size)) ;
}

void *cs_calloc (int n, size_t size)
{
    return (calloc (CS_MAX (n,1), size)) ;
}

void *cs_free (void *p)
{
    if (p) free (p) ;       /* free p if it is not already NULL */
    return (NULL) ;         /* return NULL to simplify the use of cs_free */
}
```

`cs_realloc` changes the size of a block of memory. If successful, it returns a pointer to a block of memory of size equal to `n*size`, and sets `ok` to true. If it fails, it returns the original pointer `p` and sets `ok` to false.

```
void *cs_realloc (void *p, int n, size_t size, int *ok)
{
    void *pnew ;
    pnew = realloc (p, CS_MAX (n,1) * size) ; /* realloc the block */
    *ok = (pnew != NULL) ;                    /* realloc fails if pnew is NULL */
    return ((*ok) ? pnew : p) ;               /* return original p if failure */
}
```

The `cs_spalloc` function creates an `m`-by-`n` sparse matrix that can hold up to `nzmax` entries. Numerical values are allocated if `values` is true. A triplet or compressed-column matrix is allocated depending on whether `triplet` is true or false. `cs_spfree` frees a sparse matrix, and `cs_sprealloc` changes the maximum number of entries that a `cs` sparse matrix can contain (either triplet or compressed-column).

```
cs *cs_spalloc (int m, int n, int nzmax, int values, int triplet)
{
    cs *A = cs_calloc (1, sizeof (cs)) ;    /* allocate the cs struct */
    if (!A) return (NULL) ;                 /* out of memory */
    A->m = m ;                              /* define dimensions and nzmax */
    A->n = n ;
    A->nzmax = nzmax = CS_MAX (nzmax, 1) ;
    A->nz = triplet ? 0 : -1 ;              /* allocate triplet or comp.col */
    A->p = cs_malloc (triplet ? nzmax : n+1, sizeof (int)) ;
    A->i = cs_malloc (nzmax, sizeof (int)) ;
    A->x = values ? cs_malloc (nzmax, sizeof (double)) : NULL ;
    return ((!A->p || !A->i || (values && !A->x)) ? cs_spfree (A) : A) ;
}

cs *cs_spfree (cs *A)
{
    if (!A) return (NULL) ;     /* do nothing if A already NULL */
    cs_free (A->p) ;
    cs_free (A->i) ;
    cs_free (A->x) ;
    return (cs_free (A)) ;      /* free the cs struct and return NULL */
}

int cs_sprealloc (cs *A, int nzmax)
{
    int ok, oki, okj = 1, okx = 1 ;
    if (!A) return (0) ;
    if (nzmax <= 0) nzmax = (CS_CSC (A)) ? (A->p [A->n]) : A->nz ;
    A->i = cs_realloc (A->i, nzmax, sizeof (int), &oki) ;
    if (CS_TRIPLET (A)) A->p = cs_realloc (A->p, nzmax, sizeof (int), &okj) ;
    if (A->x) A->x = cs_realloc (A->x, nzmax, sizeof (double), &okx) ;
    ok = (oki && okj && okx) ;
    if (ok) A->nzmax = nzmax ;
    return (ok) ;
}
```

MATLAB provides similar utilities. `cs_spalloc(m,n,nzmax,1,0)` is identical to the MATLAB `spalloc(m,n,nzmax)`, and `cs_spfree(A)` is the same as `clear A`. The

number of nonzeros in a compressed-column cs matrix A is given by A->p[A->n], the last column pointer value; this is identical to nnz(A) in MATLAB if the cs matrix A has no explicit zeros. The MATLAB statement nzmax(A) is the same as A->nzmax.

2.4 Triplet form

The utility functions can allocate space for a sparse matrix, but they do not define its contents. The simplest way to construct a cs matrix is to first allocate a matrix in triplet form. Applications would normally create a matrix in this way, rather than statically defining them as done in Section 2.1. For example,

```
cs *T ;
int *Ti, *Tj ;
double *Tx ;
T = cs_spalloc (m, n, nz, 1, 1) ;
Ti = T->i ; Tj = T->p ; Tx = T->x ;
```

Next, place each entry of the sparse matrix in the Ti, Tj, and Tx arrays. The kth entry has row index $i =$ Ti[k], column index $j =$ Tj[k], and numerical value $a_{ij} =$ Tx[k]. The entries can appear in arbitrary order. Set T->nz to be the number of entries in the matrix. Section 2.1 gives an example of a matrix in triplet form. If multiple entries with identical row and column indices exist, the corresponding numerical value is the sum of all such *duplicate* entries.

The cs_entry function is useful if the number of entries in the matrix is not known when the matrix is first allocated. If space is not sufficient for the next entry, the size of the T->i, T->j, and T->x arrays is doubled. The dimensions of T are increased as needed.

```
int cs_entry (cs *T, int i, int j, double x)
{
    if (!CS_TRIPLET (T) || i < 0 || j < 0) return (0) ;      /* check inputs */
    if (T->nz >= T->nzmax && !cs_sprealloc (T,2*(T->nzmax))) return (0) ;
    if (T->x) T->x [T->nz] = x ;
    T->i [T->nz] = i ;
    T->p [T->nz++] = j ;
    T->m = CS_MAX (T->m, i+1) ;
    T->n = CS_MAX (T->n, j+1) ;
    return (1) ;
}
```

The cs_compress function converts this triplet-form T into a compressed-column matrix C. First, C and a size-n workspace are allocated. Next, the number of entries in each column of C is computed, and the column pointer array Cp is constructed as the cumulative sum of the column counts. The counts in w are also replaced with a copy of Cp. cs_compress iterates through each entry in the triplet matrix. The column pointer w[Tj[k]] is found and postincremented. This determines the location p where the row index Ti[k] and numerical value Tx[k] are placed in C. Finally, the workspace is freed and the result C is returned.

```
cs *cs_compress (const cs *T)
{
    int m, n, nz, p, k, *Cp, *Ci, *w, *Ti, *Tj ;
    double *Cx, *Tx ;
    cs *C ;
    if (!CS_TRIPLET (T)) return (NULL) ;                /* check inputs */
    m = T->m ; n = T->n ; Ti = T->i ; Tj = T->p ; Tx = T->x ; nz = T->nz ;
    C = cs_spalloc (m, n, nz, Tx != NULL, 0) ;          /* allocate result */
    w = cs_calloc (n, sizeof (int)) ;                   /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;    /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (k = 0 ; k < nz ; k++) w [Tj [k]]++ ;           /* column counts */
    cs_cumsum (Cp, w, n) ;                              /* column pointers */
    for (k = 0 ; k < nz ; k++)
    {
        Ci [p = w [Tj [k]]++] = Ti [k] ;    /* A(i,j) is the pth entry in C */
        if (Cx) Cx [p] = Tx [k] ;
    }
    return (cs_done (C, w, NULL, 1)) ;      /* success; free w and return C */
}
```

The cs_done function returns a cs sparse matrix and frees any workspace.

```
cs *cs_done (cs *C, void *w, void *x, int ok)
{
    cs_free (w) ;                       /* free workspace */
    cs_free (x) ;
    return (ok ? C : cs_spfree (C)) ;   /* return result if OK, else free it */
}
```

Computing the cumulative sum will be useful in other CSparse functions, so it appears as its own function, cs_cumsum. It sets p[i] equal to the sum of c[0] through c[i-1]. It returns the sum of c[0...n-1]. On output, c[0...n-1] is overwritten with p[0...n-1].

```
double cs_cumsum (int *p, int *c, int n)
{
    int i, nz = 0 ;
    double nz2 = 0 ;
    if (!p || !c) return (-1) ;     /* check inputs */
    for (i = 0 ; i < n ; i++)
    {
        p [i] = nz ;
        nz += c [i] ;
        nz2 += c [i] ;              /* also in double to avoid int overflow */
        c [i] = p [i] ;             /* also copy p[0..n-1] back into c[0..n-1]*/
    }
    p [n] = nz ;
    return (nz2) ;                  /* return sum (c [0..n-1]) */
}
```

The MATLAB statement C=sparse(i,j,x,m,n) performs the same function as cs_compress, except that it returns a matrix with sorted columns, and sums up duplicate entries (see Sections 2.5 and 2.6).

2.5 Transpose

The algorithm for transposing a sparse matrix ($C = A^T$) is very similar to the cs_compress function because it can be viewed not just as a linear algebraic function but as a method for converting a compressed-column sparse matrix into a compressed-row sparse matrix as well. The algorithm computes the row counts of A, computes the cumulative sum to obtain the row pointers, and then iterates over each nonzero entry in A, placing the entry in its appropriate row vector. If the resulting sparse matrix C is interpreted as a matrix in compressed-row form, then C is equal to A, just in a different format. If C is viewed as a compressed-column matrix, then C contains A^T. It is simpler to describe `cs_transpose` with C as a row-oriented matrix.

```
cs *cs_transpose (const cs *A, int values)
{
    int p, q, j, *Cp, *Ci, n, m, *Ap, *Ai, *w ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;     /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (n, m, Ap [n], values && Ax, 0) ;       /* allocate result */
    w = cs_calloc (m, sizeof (int)) ;                      /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;       /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (p = 0 ; p < Ap [n] ; p++) w [Ai [p]]++ ;          /* row counts */
    cs_cumsum (Cp, w, m) ;                                 /* row pointers */
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            Ci [q = w [Ai [p]]++] = j ; /* place A(i,j) as entry C(j,i) */
            if (Cx) Cx [q] = Ax [p] ;
        }
    }
    return (cs_done (C, w, NULL, 1)) ;  /* success; free w and return C */
}
```

First, the output matrix C and workspace w are allocated. Next, the row counts and their cumulative sum are computed. The cumulative sum defines the row pointer array Cp. Finally, `cs_transpose` traverses each column j of A, placing column index j into each row i of C for which a_{ij} is nonzero. The position q of this entry in C is given by `q = w[i]`, which is then postincremented to prepare for the next entry to be inserted into row i. Compare `cs_transpose` and `cs_compress`. Their only significant difference is what kind of data structure their inputs are in. The statement `C=cs_transpose(A)` is identical to the MATLAB statement `C=A'`, except that the latter can also compute the complex conjugate transpose. For real matrices the MATLAB statements `C=A'` and `C=A.'` are identical. The `values` parameter is true (nonzero) to signify that the numerical values of C are to be computed or false (zero) otherwise.

Sorting the columns of a sparse matrix is particularly simple. The statement `C=cs_transpose(A)` computes the transpose of A. Each row of C is constructed one column index at a time, from column 0 to `C->n-1`. Thus, it is a sorted matrix;

`cs_transpose` is a linear-time bucket sort algorithm. `A` can be sorted by transposing it twice. A `cs_sort` function is left as an exercise. The total time required is $O(m + n + |A|)$. Rather than transposing a matrix twice, it is sometimes possible to create the transpose first and then sort it with a single call to `cs_transpose`.

MATLAB has no explicit function to sort its sparse matrices. Each function or operator that returns a sparse matrix is required to return it with sorted columns.

2.6 Summing up duplicate entries

Finite-element methods generate a matrix as a collection of *elements* or dense submatrices. The complete matrix is a summation of the elements. If two elements contribute to the same entry, their values should be summed. The `cs_compress` function leaves these duplicate entries in its output matrix. They can be summed with the `cs_dupl` function.

```
int cs_dupl (cs *A)
{
    int i, j, p, q, nz = 0, n, m, *Ap, *Ai, *w ;
    double *Ax ;
    if (!CS_CSC (A)) return (0) ;                   /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    w = cs_malloc (m, sizeof (int)) ;               /* get workspace */
    if (!w) return (0) ;                            /* out of memory */
    for (i = 0 ; i < m ; i++) w [i] = -1 ;          /* row i not yet seen */
    for (j = 0 ; j < n ; j++)
    {
        q = nz ;                                    /* column j will start at q */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;                            /* A(i,j) is nonzero */
            if (w [i] >= q)
            {
                Ax [w [i]] += Ax [p] ;              /* A(i,j) is a duplicate */
            }
            else
            {
                w [i] = nz ;                        /* record where row i occurs */
                Ai [nz] = i ;                       /* keep A(i,j) */
                Ax [nz++] = Ax [p] ;
            }
        }
        Ap [j] = q ;                                /* record start of column j */
    }
    Ap [n] = nz ;                                   /* finalize A */
    cs_free (w) ;                                   /* free workspace */
    return (cs_sprealloc (A, 0)) ;                  /* remove extra space from A */
}
```

The function uses a size-`m` integer workspace; `w[i]` records the location in `Ai` and `Ax` of the most recent entry with row index `i`. If this position is within the current column `j`, then it is a duplicate entry and must be summed. Otherwise, the entry is kept and `w[i]` is updated to reflect the position of this entry.

MATLAB does not have an explicit function to sum duplicate entries of a sparse matrix. It is combined with the MATLAB `sparse` function that converts a triplet matrix to a compressed sparse matrix.

2.7 Removing entries from a matrix

CSparse does not require its sparse matrices to be free of numerically zero entries, but its MATLAB interface does. Rather than writing a special-purpose function to drop zeros from a matrix, the `cs_fkeep` function is used. It takes as an argument a pointer to a function `fkeep(i,j,aij,other)` which is evaluated for each entry a_{ij} in the matrix. An entry is kept if `fkeep` is true for that entry. Dropping entries from `A` requires each column to be shifted; `Ap[j]` must be decremented by the number of entries dropped from columns `0` to `j-1`. When a `cs` matrix `A` is returned to MATLAB, `cs_dropzeros(A)` is normally performed first. The `cs_chol` mexFunction optionally keeps zero entries in L, so that `cs_updown` can work properly.

```
int cs_fkeep (cs *A, int (*fkeep) (int, int, double, void *), void *other)
{
    int j, p, nz = 0, n, *Ap, *Ai ;
    double *Ax ;
    if (!CS_CSC (A) || !fkeep) return (-1) ;    /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        p = Ap [j] ;                        /* get current location of col j */
        Ap [j] = nz ;                       /* record new location of col j */
        for ( ; p < Ap [j+1] ; p++)
        {
            if (fkeep (Ai [p], j, Ax ? Ax [p] : 1, other))
            {
                if (Ax) Ax [nz] = Ax [p] ;  /* keep A(i,j) */
                Ai [nz++] = Ai [p] ;
            }
        }
    }
    Ap [n] = nz ;                           /* finalize A */
    cs_sprealloc (A, 0) ;                   /* remove extra space from A */
    return (nz) ;
}

static int cs_nonzero (int i, int j, double aij, void *other)
{
    return (aij != 0) ;
}
int cs_dropzeros (cs *A)
{
    return (cs_fkeep (A, &cs_nonzero, NULL)) ;  /* keep all nonzero entries */
}
```

Additional arguments can be passed to `fkeep` via the `void *other` parameter to `cs_fkeep`. This is demonstrated by `cs_droptol`, which removes entries whose magnitude is less than or equal to `tol`.

The MATLAB equivalent for `cs_droptol(A,tol)` is `A = A.*(abs(A)>tol)`.

```
static int cs_tol (int i, int j, double aij, void *tol)
{
    return (fabs (aij) > *((double *) tol)) ;
}
int cs_droptol (cs *A, double tol)
{
    return (cs_fkeep (A, &cs_tol, &tol)) ;    /* keep all large entries */
}
```

2.8 Matrix multiplication

Since matrices are stored in compressed-column form in CSparse, the matrix multiplication $C = AB$, where C is m-by-n, A is m-by-k, and B is k-by-n, should access A and B by column and create C one column at a time. If C_{*j} and B_{*j} denote column j of C and B, then $C_{*j} = AB_{*j}$. Splitting A into its k columns and B_{*j} into its k individual entries,

$$C_{*j} = \begin{bmatrix} A_{*1} & \cdots & A_{*k} \end{bmatrix} \begin{bmatrix} b_{1j} \\ \vdots \\ b_{kj} \end{bmatrix} = \sum_{i=1}^{k} A_{*i} b_{ij}. \tag{2.2}$$

The nonzero pattern of C is given by the following theorem.

Theorem 2.1 (Gilbert [101]). *The nonzero pattern of C_{*j} is the set union of the nonzero pattern of A_{*i} for all i for which b_{ij} is nonzero. If $\mathcal{C}_j$, $\mathcal{A}_i$, and $\mathcal{B}_j$ denote the set of row indices of nonzero entries in C_{*j}, A_{*i}, and B_{*j}, then*

$$\mathcal{C}_j = \bigcup_{i \in \mathcal{B}_j} \mathcal{A}_i. \tag{2.3}$$

A matrix multiplication algorithm must compute both C_{*j} and $\mathcal{C}_j$. Note that (2.3) is correct only if numerical cancellation is ignored. It is implemented with `cs_scatter` and `cs_multiply` below. A dense vector `x` is used to construct C_{*j}. The set $\mathcal{C}_j$ is stored directly in `C`, but another work vector `w` is needed to determine if a given row index i is in the set already. The vector `w` starts out cleared. When computing column `j`, `w[i]<j+1` will denote a row index `i` that is not yet in $\mathcal{C}_j$. When `i` is inserted in $\mathcal{C}_j$, `w[i]` is set to `j+1`. The `cs_scatter` function computes one iteration of (2.2) and (2.3) for a single value of `i`, using a *scatter* operation to copy a sparse vector into a dense one. The matrix multiplication function `cs_multiply` first allocates the `w` and `x` workspace and the output matrix `C`. Next, it iterates over each column `j` of the result `C`. After a series of scatter operations, the dense vector `x` is *gathered* into a sparse vector (a column of `C`). Since the number of nonzeros in `C` is not known at the beginning, it is increased in size as needed.

Computing `nnz(A*B)` is actually much harder than computing `nnz(chol(A))`. The latter is discussed in Chapter 4. An alternate approach that computes `nnz(A*B)` in an initial pass and then `C=A*B` in a second pass is left as an exercise (Problem 2.20).

```
cs *cs_multiply (const cs *A, const cs *B)
{
    int p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values, *Bi ;
    double *x, *Bx, *Cx ;
    cs *C ;
    if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;      /* check inputs */
    m = A->m ; anz = A->p [A->n] ;
    n = B->n ; Bp = B->p ; Bi = B->i ; Bx = B->x ; bnz = Bp [n] ;
    w = cs_calloc (m, sizeof (int)) ;                    /* get workspace */
    values = (A->x != NULL) && (Bx != NULL) ;
    x = values ? cs_malloc (m, sizeof (double)) : NULL ; /* get workspace */
    C = cs_spalloc (m, n, anz + bnz, values, 0) ;        /* allocate result */
    if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
    Cp = C->p ;
    for (j = 0 ; j < n ; j++)
    {
        if (nz + m > C->nzmax && !cs_sprealloc (C, 2*(C->nzmax)+m))
        {
            return (cs_done (C, w, x, 0)) ;              /* out of memory */
        }
        Ci = C->i ; Cx = C->x ;          /* C->i and C->x may be reallocated */
        Cp [j] = nz ;                    /* column j of C starts here */
        for (p = Bp [j] ; p < Bp [j+1] ; p++)
        {
            nz = cs_scatter (A, Bi [p], Bx ? Bx [p] : 1, w, x, j+1, C, nz) ;
        }
        if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
    }
    Cp [n] = nz ;                        /* finalize the last column of C */
    cs_sprealloc (C, 0) ;                /* remove extra space from C */
    return (cs_done (C, w, x, 1)) ;      /* success; free workspace, return C */
}

int cs_scatter (const cs *A, int j, double beta, int *w, double *x, int mark,
    cs *C, int nz)
{
    int i, p, *Ap, *Ai, *Ci ;
    double *Ax ;
    if (!CS_CSC (A) || !w || !CS_CSC (C)) return (-1) ;      /* check inputs */
    Ap = A->p ; Ai = A->i ; Ax = A->x ; Ci = C->i ;
    for (p = Ap [j] ; p < Ap [j+1] ; p++)
    {
        i = Ai [p] ;                              /* A(i,j) is nonzero */
        if (w [i] < mark)
        {
            w [i] = mark ;                        /* i is new entry in column j */
            Ci [nz++] = i ;                       /* add i to pattern of C(:,j) */
            if (x) x [i] = beta * Ax [p] ;        /* x(i) = beta*A(i,j) */
        }
        else if (x) x [i] += beta * Ax [p] ;      /* i exists in C(:,j) already */
    }
    return (nz) ;
}
```

When `cs_multiply` is finished, the matrix `C` is resized to the actual number of entries it contains, and the workspace is freed. The `cs_scatter` function computes `x=x+beta*A(:,j)`, and accumulates the nonzero pattern of `x` in `C->i`, starting at

position nz. The new value of nz is returned. Row index i is in the pattern of x if w[i] is equal to mark.

The time taken by cs_multiply is $O(n + f + |B|)$, where f is the number of floating-point operations performed (f dominates the run time unless A has one or more columns with no entries, in which case either n or $|B|$ can be greater than f). If the columns of C need to be sorted, either $C = ((AB)^T)^T$ or $C = (B^T A^T)^T$ can be computed. The latter is better if C has many more entries than A or B. The MATLAB equivalent C=A*B uses a similar algorithm to the one presented here.

2.9 Matrix addition

The cs_add function performs matrix addition, $C = \alpha A + \beta B$. Matrix addition can be written as a multiplication of two matrices,

$$C = \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} \alpha I \\ \beta I \end{bmatrix}, \tag{2.4}$$

where I is an identity matrix of the appropriate size. Although it is not implemented this way, the function cs_add looks very much like cs_multiply because of (2.4). The innermost loop differs slightly; no reallocation is needed, and the for p loop is replaced with two calls to cs_scatter. Like cs_multiply, it does not return C with sorted columns. The MATLAB equivalent is C=alpha*A+beta*B.

```
cs *cs_add (const cs *A, const cs *B, double alpha, double beta)
{
    int p, j, nz = 0, anz, *Cp, *Ci, *Bp, m, n, bnz, *w, values ;
    double *x, *Bx, *Cx ;
    cs *C ;
    if (!CS_CSC (A) || !CS_CSC (B)) return (NULL) ;         /* check inputs */
    m = A->m ; anz = A->p [A->n] ;
    n = B->n ; Bp = B->p ; Bx = B->x ; bnz = Bp [n] ;
    w = cs_calloc (m, sizeof (int)) ;                       /* get workspace */
    values = (A->x != NULL) && (Bx != NULL) ;
    x = values ? cs_malloc (m, sizeof (double)) : NULL ;    /* get workspace */
    C = cs_spalloc (m, n, anz + bnz, values, 0) ;           /* allocate result*/
    if (!C || !w || (values && !x)) return (cs_done (C, w, x, 0)) ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (j = 0 ; j < n ; j++)
    {
        Cp [j] = nz ;                   /* column j of C starts here */
        nz = cs_scatter (A, j, alpha, w, x, j+1, C, nz) ;   /* alpha*A(:,j)*/
        nz = cs_scatter (B, j, beta, w, x, j+1, C, nz) ;    /* beta*B(:,j) */
        if (values) for (p = Cp [j] ; p < nz ; p++) Cx [p] = x [Ci [p]] ;
    }
    Cp [n] = nz ;                       /* finalize the last column of C */
    cs_sprealloc (C, 0) ;               /* remove extra space from C */
    return (cs_done (C, w, x, 1)) ;     /* success; free workspace, return C */
}
```

2.10 Vector permutation

An n-by-n permutation matrix P can be represented by a sparse matrix `P` with a one in each row and column, or by a length-`n` integer vector `p` called a *permutation vector*, where `p[k]=i` means that $p_{ki} = 1$. A permutation matrix P is orthogonal, so its inverse is simply $P^{-1} = P^T$. The inverse permutation vector is given by `pinv[i]=k` if $p_{ki} = 1$, since this implies $(P^T)_{ik} = 1$.

Some MATLAB functions return permutation vectors; others return permutation matrices. If `p` and `q` are MATLAB permutation vectors of length `n`, converting between these forms is done as follows:

`[p j x] = find(P')`	convert row permutation `P*A` to `A(p,:)`
`[q j x] = find(Q)`	convert column permutation `A*Q` to `A(:,q)`
`P=sparse(1:n, p, 1)`	convert row permutation `A(p,:)` to `P*A`
`Q=sparse(q, 1:n, 1)`	convert column permutation `A(:,q)` to `A*Q`

If $x = Pb$, row `k` of x is row `p[k]` of b. The function `cs_pvec` computes $x = Pb$, or `x=b(p)` in MATLAB, where `x` and `b` are vectors of length `n`. The function `cs_ipvec` computes $x = P^T b$, or `x(p)=b` in MATLAB.

```
int cs_pvec (const int *p, const double *b, double *x, int n)
{
    int k ;
    if (!x || !b) return (0) ;                              /* check inputs */
    for (k = 0 ; k < n ; k++) x [k] = b [p ? p [k] : k] ;
    return (1) ;
}


int cs_ipvec (const int *p, const double *b, double *x, int n)
{
    int k ;
    if (!x || !b) return (0) ;                              /* check inputs */
    for (k = 0 ; k < n ; k++) x [p ? p [k] : k] = b [k] ;
    return (1) ;
}
```

The inverse, or transpose, of a permutation vector `p[k]=i` is the vector `pinv`, where `pinv[i]=k`. This is computed by `cs_pinv`. In MATLAB, `pinv(p) = 1:n` computes the inverse `pinv` of a permutation vector `p` of length `n` (this assumes that `pinv` is initially not defined or a vector of length `n` or less).

```
int *cs_pinv (int const *p, int n)
{
    int k, *pinv ;
    if (!p) return (NULL) ;                     /* p = NULL denotes identity */
    pinv = cs_malloc (n, sizeof (int)) ;        /* allocate result */
    if (!pinv) return (NULL) ;                  /* out of memory */
    for (k = 0 ; k < n ; k++) pinv [p [k]] = k ;/* invert the permutation */
    return (pinv) ;                             /* return result */
}
```

2.11 Matrix permutation

The cs_permute function permutes a sparse matrix, $C = PAQ$ (C=A(p,q) in MATLAB). It takes as input a column permutation vector q of length n and an inverse row permutation pinv (not p) of length m, where A is m-by-n. Row i of A becomes row k of C if pinv[i]=k. The algorithm traverses the columns of j of A in permuted order via q. Each row index in A is mapped to its permuted row in C.

```
cs *cs_permute (const cs *A, const int *pinv, const int *q, int values)
{
    int t, j, k, nz = 0, m, n, *Ap, *Ai, *Cp, *Ci ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;     /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (m, n, Ap [n], values && Ax != NULL, 0) ;  /* alloc result */
    if (!C) return (cs_done (C, NULL, NULL, 0)) ;   /* out of memory */
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (k = 0 ; k < n ; k++)
    {
        Cp [k] = nz ;                    /* column k of C is column q[k] of A */
        j = q ? (q [k]) : k ;
        for (t = Ap [j] ; t < Ap [j+1] ; t++)
        {
            if (Cx) Cx [nz] = Ax [t] ;  /* row i of A is row pinv[i] of C */
            Ci [nz++] = pinv ? (pinv [Ai [t]]) : Ai [t] ;
        }
    }
    Cp [n] = nz ;                        /* finalize the last column of C */
    return (cs_done (C, NULL, NULL, 1)) ;
}
```

CSparse functions that operate on symmetric matrices use just the upper triangular part, just like chol in MATLAB. If A is symmetric with only the upper triangular part stored, C=A(p,p) is not upper triangular. The cs_symperm function computes C=A(p,p) for a symmetric matrix A whose upper triangular part is stored, returning C in the same format. Entries below the diagonal are ignored.

The first for j loop counts how many entries are in each column of C. Suppose $i \leq j$, and A(i,j) is permuted to become entry C(i2,j2). If $i2 \leq j2$, this entry is in the upper triangular part of C. Otherwise, C(i2,j2) is in the lower triangular part of C, and the entry must be placed in C as C(j2,i2) instead. After the column counts of C are computed (in w), the cumulative sum is computed to obtain the column pointers Cp. The second for loop constructs C, much like cs_permute.

```
cs *cs_symperm (const cs *A, const int *pinv, int values)
{
    int i, j, p, q, i2, j2, n, *Ap, *Ai, *Cp, *Ci, *w ;
    double *Cx, *Ax ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;                    /* check inputs */
    n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    C = cs_spalloc (n, n, Ap [n], values && (Ax != NULL), 0) ; /* alloc result*/
    w = cs_calloc (n, sizeof (int)) ;                   /* get workspace */
    if (!C || !w) return (cs_done (C, w, NULL, 0)) ;    /* out of memory */
```

```
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    for (j = 0 ; j < n ; j++)           /* count entries in each column of C */
    {
        j2 = pinv ? pinv [j] : j ;      /* column j of A is column j2 of C */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (i > j) continue ;       /* skip lower triangular part of A */
            i2 = pinv ? pinv [i] : i ;  /* row i of A is row i2 of C */
            w [CS_MAX (i2, j2)]++ ;     /* column count of C */
        }
    }
    cs_cumsum (Cp, w, n) ;              /* compute column pointers of C */
    for (j = 0 ; j < n ; j++)
    {
        j2 = pinv ? pinv [j] : j ;      /* column j of A is column j2 of C */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (i > j) continue ;       /* skip lower triangular part of A*/
            i2 = pinv ? pinv [i] : i ;  /* row i of A is row i2 of C */
            Ci [q = w [CS_MAX (i2, j2)]++] = CS_MIN (i2, j2) ;
            if (Cx) Cx [q] = Ax [p] ;
        }
    }
    return (cs_done (C, w, NULL, 1)) ;  /* success; free workspace, return C */
}
```

2.12 Matrix norm

Computing the 2-norm of a sparse matrix ($\|A\|_2$) is not trivial, since it is the largest singular value of A. MATLAB does not provide a function for computing the 2-norm of a sparse matrix, although it can compute a good estimate using `normest`. The ∞-norm is the maximum row-sum, the computation of which requires a workspace of size `n` if `A` is accessed by column. The simplest norm to use for a sparse matrix stored in compressed-column form is the 1-norm, $\|A\|_1 = \max_j \sum_{i=1}^{m} |a_{ij}|$, which is computed by the `cs_norm` function below. Note that it does not make use of the `A->i` row index array. The MATLAB `norm` function can compute the 1-norm, ∞-norm, or Frobenius norm of a sparse matrix.

```
double cs_norm (const cs *A)
{
    int p, j, n, *Ap ;
    double *Ax, norm = 0, s ;
    if (!CS_CSC (A) || !A->x) return (-1) ;             /* check inputs */
    n = A->n ; Ap = A->p ; Ax = A->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (s = 0, p = Ap [j] ; p < Ap [j+1] ; p++) s += fabs (Ax [p]) ;
        norm = CS_MAX (norm, s) ;
    }
    return (norm) ;
}
```

2.13 Reading a matrix from a file

The cs_load function reads in a triplet matrix from a file. The matrix T is initially allocated as a 0-by-0 triplet matrix with space for just one entry. The dimensions of T are determined by the maximum row and column index read from the file.

```
cs *cs_load (FILE *f)
{
    int i, j ;
    double x ;
    cs *T ;
    if (!f) return (NULL) ;                          /* check inputs */
    T = cs_spalloc (0, 0, 1, 1, 1) ;                 /* allocate result */
    while (fscanf (f, "%d %d %lg\n", &i, &j, &x) == 3)
    {
        if (!cs_entry (T, i, j, x)) return (cs_spfree (T)) ;
    }
    return (T) ;
}
```

2.14 Printing a matrix

cs_print prints the contents of a cs matrix in triplet form or compressed-column form. Only the first few entries are printed if brief is true.

```
int cs_print (const cs *A, int brief)
{
    int p, j, m, n, nzmax, nz, *Ap, *Ai ;
    double *Ax ;
    if (!A) { printf ("(null)\n") ; return (0) ; }
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    nzmax = A->nzmax ; nz = A->nz ;
    printf ("CSparse Version %d.%d.%d, %s.  %s\n", CS_VER, CS_SUBVER,
        CS_SUBSUB, CS_DATE, CS_COPYRIGHT) ;
    if (nz < 0)
    {
        printf ("%d-by-%d, nzmax: %d nnz: %d, 1-norm: %g\n", m, n, nzmax,
                Ap [n], cs_norm (A)) ;
        for (j = 0 ; j < n ; j++)
        {
            printf ("    col %d : locations %d to %d\n", j, Ap [j], Ap [j+1]-1);
            for (p = Ap [j] ; p < Ap [j+1] ; p++)
            {
                printf ("      %d : %g\n", Ai [p], Ax ? Ax [p] : 1) ;
                if (brief && p > 20) { printf ("  ...\n") ; return (1) ; }
            }
        }
    }
    else
    {
        printf ("triplet: %d-by-%d, nzmax: %d nnz: %d\n", m, n, nzmax, nz) ;
        for (p = 0 ; p < nz ; p++)
        {
            printf ("    %d %d : %g\n", Ai [p], Ap [p], Ax ? Ax [p] : 1) ;
            if (brief && p > 20) { printf ("  ...\n") ; return (1) ; }
        }
    }
    return (1) ;
}
```

2.15 Sparse matrix collections

Arbitrary random matrices are easy to generate; random sparse matrices with specific properties are not simple to generate (type the command `type sprand` in MATLAB and compare the 3-input versus 4-input usage of the function). Both can give misleading performance results. Sparse matrices from real applications are better, such as those from the Rutherford-Boeing collection[4] [55], the NIST Matrix Market,[5] and the UF Sparse Matrix Collection.[6] The UFget package distributed with CSparse provides a simple MATLAB interface to the UF Sparse Matrix Collection. For example, `UFget('HB/arc130')` downloads the `arc130` matrix and loads it into MATLAB. `UFweb('HB/arc130')` brings up a web browser with the web page for the same matrix. Matrix properties are listed in an index, which makes it simple to write a MATLAB program that uses a selected subset of matrices (for example, all symmetric positive definite matrices in order of increasing number of nonzeros). As of April 2006, the UF Sparse Matrix Collection contains 1,377 matrices, with order 5 to 5 million, and as few as 15 and as many as 99 million nonzeros. The submission of new matrices not represented by the collection is always welcome.

2.16 Further reading

The CHOLMOD [30] package provides some of the sparse matrix operators in MATLAB. Other sparse matrix packages have similar functions; see the HSL[7] and the BCSLIB-EXT[8] packages in particular. Gilbert, Moler, and Schreiber present the early development of sparse matrices in MATLAB [105]. Gustavson discusses sparse matrix permutation, transpose, and multiplication [121]. The Sparse BLAS [43, 44, 56, 70] includes many of these operations.

Exercises

2.1. Write a `cs_gatxpy` function that computes $y = A^T x + y$ without forming A^T.

2.2. Write a function `cs_find` that converts a `cs` matrix into a triplet-form matrix, like the `find` function in MATLAB.

2.3. Write a variant of `cs_gaxpy` that computes $y = Ax + y$, where A is a symmetric matrix with only the upper triangular part present. Ignore entries in the lower triangular part.

2.4. Write a function with prototype `void cs_scale(cs *A, double *r, double *c)` that overwrites A with RAC, where R and C are diagonal matrices; `r[k]` and `c[k]` are the `k`th diagonal entries of R and C, respectively.

2.5. Write a function similar to `cs_entry` that adds a dense submatrix to a triplet

[4] www.cse.clrc.ac.uk/nag/hb

[5] math.nist.gov/MatrixMarket

[6] www.cise.ufl.edu/research/sparse/matrices; see also www.siam.org/books/fa02

[7] www.cse.clrc.ac.uk/nag/hsl

[8] www.boeing.com/phantom/bcslib-ext

matrix. `i` and `j` should be integer arrays of length `k`, and `x` should be a `k`-by-`k` dense matrix.

2.6. Show how to transpose a `cs` matrix in triplet form in $O(1)$ time.

2.7. Write a function `cs_sort` that sorts a `cs` matrix. Its prototype should be `cs *cs_sort (cs *A)`. Use two calls to `cs_transpose`. Why is `C=cs_transpose (cs_transpose (A))` incorrect?

2.8. Write a function that sorts a matrix one column at a time, using the ANSI C quicksort function, `qsort`. Compare its performance (time and memory usage) with the solution to Problem 2.7.

2.9. Write a function that creates a compressed-column matrix from a triplet matrix with sorted columns, no duplicates, and no numerically zero entries.

2.10. Show how to multiply a matrix in triplet form times a dense vector.

2.11. Sorting a matrix with a double transpose does extra work that is not required. The second transpose counts the entries in each row, but these are equal to the original column counts. Write a `cs_sort` function that avoids extra work.

2.12. Write a function `cs_ok` that checks a matrix to see if it is valid and optionally prints the matrix with prototype `int cs_ok (cs *A, int sorted, int values, int print)`. If `values` is negative, `A->x` is ignored and may be `NULL`; otherwise, it must be non-`NULL`. If `sorted` is true, then the columns must be sorted. If `values` is positive, then there can be no numerically zero entries in `A`. The time and workspace are $O(m + n + |A|)$ and $O(m)$.

2.13. Write a function that determines if a sparse matrix is symmetric.

2.14. Write a function `cs *cs_copy (cs *A)` that returns a copy of `A`.

2.15. Write a function `cs_band(A,k1,k2)` that removes all entries from `A` except for those in diagonals `k1` to `k2` of `A`. Entries outside the band should be dropped. Hint: use `cs_fkeep`.

2.16. Write a function that creates a sparse matrix copy of a dense matrix stored in column-major form.

2.17. How much time does it take to transpose a column vector? How much space does a sparse row vector take if stored in compressed-column form?

2.18. How much time and space does it take to compute $x^T y$ for two sparse column vectors x and y, using `cs_transpose` and `cs_multiply`? Write a more efficient routine with prototype `double cs_dot (cs *x, cs *y)`, which assumes `x` and `y` are column vectors. Consider two cases: (1) The row indices of `x` and `y` are not sorted. A `double` workspace `w` of size `x->m` will need to be allocated. (2) The row indices of `x` and `y` are sorted. No workspace is required. Both cases take $O(|x| + |y|)$ time.

2.19. The first call to `cs_scatter` in each iteration of the `j` loop in both `cs_multiply` and `cs_add` does more work than is necessary, since `w[i]<mark` is always true in this case. Write a more efficient version.

2.20. Consider an alternative algorithm for `cs_multiply` that uses two passes. The first pass computes the number of entries in each column of `C` (or just the total number of entries), and the second pass performs the matrix multiplication.

No `cs_sprealloc` is needed. Compare with the original `cs_multiply`.

2.21. How efficient is `cs_add` when `A` and `B` are sparse column vectors? Hint: how much time does `calloc` take? Write faster function `cs_saxpy` that takes an initialized workspace (`w` and `x`) as input, computes the result, and returns the workspace ready to use in a subsequent call to `cs_saxpy`.

2.22. Write two functions `cs_hcat` and `cs_vcat` that perform the horizontal and vertical concatenation of `A` and `B`, respectively, just like the MATLAB statements `C = [A B]` and `C = [A ; B]`.

2.23. Write a function that implements the MATLAB statement `C=A(i1:i2,j1:j2)`. This is much simpler than the next two problems.

2.24. The MATLAB statement `C=A(i,j)`, where `i` and `j` are integer vectors, creates a submatrix `C` of `A` of dimension `length(i)`-by-`length(j)`. Write a function that performs this operation. Either assume that `i` and `j` do not contain duplicate indices or that they may contain duplicates (MATLAB allows for duplicates).

2.25. The MATLAB statement `A(i,j)=C`, where `i` and `j` are integer vectors, replaces the entries in the `A(i,j)` submatrix with the `length(i)`-by-`length(j)` matrix `C`. Write a function that performs this operation. Either assume that `i` and `j` do not contain duplicate indices or that they may contain duplicates (MATLAB allows for duplicates).

2.26. Write a function combining `cs_permute` and `cs_transpose` that computes the permuted transpose, just as in the MATLAB statement `C=A(p,q)'`, where `p` and `q` are permutation vectors. It should use one pass over the matrix to count the number of entries in `C` and another to copy entries from `A` to `C`.

2.27. Create three versions of `cs_gaxpy` that operate on dense matrices `X` and `Y` (`A` is still sparse). The first should assume `X` and `Y` are in column-major form. The second should use row-major form. The third should use column-major form but operate on blocks of (say) 32 columns of `X` at a time. Compare their performance.

2.28. Repeat Problem 2.27 but for `cs_gatxpy` instead (described in Problem 2.1).

2.29. Write four functions that modify a sparse matrix `A`, adding `k` *empty* rows or columns (an empty row or column has no entries in it). Adding empty rows takes $O(|A|)$ if added to the top or $O(1)$ if added to the bottom. Adding empty columns takes $O(n+k)$ time.

2.30. Experiment with the time taken by the MATLAB statement `r=A(i,:)` for an `m`-by-`n` matrix and a scalar `i`. Does MATLAB use a binary search (taking $O(\sum \log |\texttt{A}(:,\texttt{j})|)$ time)? Or does it use a linear search of each column? Does it exploit special cases, such as `r=A(1,:)` and `r=A(m,:)`?

2.31. Which CSparse functions work properly if duplicate entries are present?

Chapter 3
Solving triangular systems

Solving a triangular system, $Lx = b$, where L is square and lower triangular, is a key mathematical kernel. It will be used in Chapter 4 as part of a sparse Cholesky factorization algorithm and in Chapter 6 as part of a sparse LU factorization algorithm. The nonzero pattern of x will be used to construct the nonzero pattern of a column of R for the QR factorization presented in Chapter 5. Solving $Lx = b$ is also essential for solving $Ax = b$ after either a Cholesky or LU factorization of A.

3.1 A dense right-hand side

There are many ways of solving $Lx = b$ (a *forward solve*), but if L is stored as a compressed-column sparse matrix, accessing L by columns is the most natural. Consider the 2-by-2 block decomposition,

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \tag{3.1}$$

where L_{22} is the lower right $(n-1)$-by-$(n-1)$ submatrix of L; l_{21}, x_2, and b_2 are column vectors of length $n-1$; and l_{11}, x_1, and b_1 are scalars. This leads to two equations, $l_{11}x_1 = b_1$ and $l_{21}x_1 + L_{22}x_2 = b_2$. To solve $Lx = b$, the first can be solved ($x_1 = b_1/l_{11}$) to obtain the first entry in x. The second equation is a lower triangular system of the form $L_{22}x_2 = b_2 - l_{21}x_1$ that can be solved recursively for x_2. Unwinding the tail recursion leads naturally to an algorithm that iterates over the columns of L. Note that b_1 and b_2 are used just once; this allows x to overwrite b in the implementation:

$x = b$
for $j = 0$ to $n-1$ **do**
  $x_j = x_j / l_{jj}$
  for each $i > j$ for which $l_{ij} \neq 0$ **do**
    $x_i = x_i - l_{ij} x_j$

If `x` is a dense vector but `L` is sparse, the algorithm and code are very similar to the matrix-vector multiplication, `cs_gaxpy`. On input, `x` contains the right-hand side

b; on output it contains the solution to $Lx = b$. The `cs_lsolve` function assumes that the diagonal entry of L is always present and is the first entry in each column. Otherwise, the row indices in each column of L can appear in any order.

```
int cs_lsolve (const cs *L, double *x)
{
    int p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                    /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = 0 ; j < n ; j++)
    {
        x [j] /= Lx [Lp [j]] ;
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [Li [p]] -= Lx [p] * x [j] ;
        }
    }
    return (1) ;
}
```

Solving $L^T x = b$ (a *backsolve*) is best done by accessing L^T by rows, since L is stored by column. The 2-by-2 block decomposition becomes

$$\begin{bmatrix} l_{11} & l_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$

where each entry in L has the same size as in (3.1).

```
int cs_ltsolve (const cs *L, double *x)
{
    int p, j, n, *Lp, *Li ;
    double *Lx ;
    if (!CS_CSC (L) || !x) return (0) ;                    /* check inputs */
    n = L->n ; Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (j = n-1 ; j >= 0 ; j--)
    {
        for (p = Lp [j]+1 ; p < Lp [j+1] ; p++)
        {
            x [j] -= Lx [p] * x [Li [p]] ;
        }
        x [j] /= Lx [Lp [j]] ;
    }
    return (1) ;
}
```

To solve $Ux = b$, where U is stored by column, yet another 2-by-2 decomposition is used:

$$\begin{bmatrix} U_{11} & u_{12} \\ 0 & u_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$

where U_{11} is $(n-1)$-by-$(n-1)$. This results in the two equations $U_{11}x_1 + u_{12}x_2 = b_1$ and $u_{22}x_2 = b_2$. The second equation can be solved for $x_2 = b_2/u_{22}$, and the first becomes $U_{11}x_1 = b_1 - u_{12}x_2$. These equations are encapsulated in the function `cs_usolve`. It assumes the diagonal entry is always present and appears as the last

entry in each column. Row indices in the columns of U can otherwise be in any order.

```
int cs_usolve (const cs *U, double *x)
{
    int p, j, n, *Up, *Ui ;
    double *Ux ;
    if (!CS_CSC (U) || !x) return (0) ;                    /* check inputs */
    n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
    for (j = n-1 ; j >= 0 ; j--)
    {
        x [j] /= Ux [Up [j+1]-1] ;
        for (p = Up [j] ; p < Up [j+1]-1 ; p++)
        {
            x [Ui [p]] -= Ux [p] * x [j] ;
        }
    }
    return (1) ;
}
```

The cs_utsolve function solves $U^T x = b$, where U is upper triangular. Its derivation is left as an exercise.

```
int cs_utsolve (const cs *U, double *x)
{
    int p, j, n, *Up, *Ui ;
    double *Ux ;
    if (!CS_CSC (U) || !x) return (0) ;                    /* check inputs */
    n = U->n ; Up = U->p ; Ui = U->i ; Ux = U->x ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Up [j] ; p < Up [j+1]-1 ; p++)
        {
            x [j] -= Ux [p] * x [Ui [p]] ;
        }
        x [j] /= Ux [Up [j+1]-1] ;
    }
    return (1) ;
}
```

cs_lsolve(L,x), cs_ltsolve(L,x), cs_usolve(U,x), and cs_utsolve(U,x) correspond to x=L\x, x=L'\x, x=U\x, and x=U'\x in MATLAB, except that the transposed solvers cs_ltsolve and cs_ltsolve do not transpose their inputs.

3.2 A sparse right-hand side

The Cholesky and LU factorization algorithms presented in Chapters 4 and 6 rely on the solution to $Lx = b$ to compute one row or column of the factors, where L, x, and b are all sparse. The QR factorization algorithm presented in Chapter 5 uses the nonzero pattern of x to construct one column of R. This small change, from a dense right-hand side in cs_lsolve to a sparse right-hand side, has a large impact on the algorithm and its underlying theory.

To simplify the discussion, assume that L has a unit diagonal (this will be the case for its use in LU factorization). The algorithm for solving $Lx = b$ becomes

$x = b$
for $j = 0$ to $n - 1$ **do**
  if $x_j \neq 0$
    for each $i > j$ for which $l_{ij} \neq 0$ **do**
      $x_i = x_i - l_{ij} x_j$

The sparse vector x can be temporarily stored in a dense vector of size n, assumed to be initially zero. Thus, the two statements $x = b$ and $x_i = x_i - l_{ij} x_j$ can be done efficiently. If this algorithm is implemented as in the above pseudocode, the time taken would be $O(n + |b| + f)$, where f is the number of floating-point operations performed and $|b|$ is the number of nonzeros in b. Normally, $|b| < f$, so the time is $O(n + f)$. This looks efficient, but it is not. The floating-point operation count can easily be dominated by n. If b is all zero except for b_n, f is $O(1)$, but the total work is $O(n)$. Basing an LU factorization algorithm on this method for solving $Lx = b$ would lead to an $\Omega(n^2)$-time factorization, which is clearly unacceptable. Factorizing a tridiagonal matrix should take $O(n)$ time, not $O(n^2)$ time.

The problem is the **for** j loop. A better method would assume that the algorithm starts with a list of indices j for which x_j will be nonzero, $\mathcal{X} = \{j \mid x_j \neq 0\}$, sorted in ascending order. The algorithm would then be

$x = b$
for each $j \in \mathcal{X}$ **do**
  for each $i > j$ for which $l_{ij} \neq 0$ **do**
    $x_i = x_i - l_{ij} x_j$

Assuming $\mathcal{X}$ is already given, the run time drops to $O(|b| + f)$, which is essentially $O(f)$, an ideal target.

The problem now becomes how to determine $\mathcal{X}$ and how to sort it. Entries in x become nonzero in two places, the first and last lines of the above pseudocode. If numerical cancellation is neglected, these two statements can be written as two logical implications:

1. $b_i \neq 0 \Rightarrow x_i \neq 0$.
2. $x_j \neq 0 \wedge \exists i (l_{ij} \neq 0) \Rightarrow x_i \neq 0$.

These two rules can be expressed as a graph traversal problem. Consider a directed graph $G_L = (V, E)$, where $V = \{1 \ldots n\}$ and $E = \{(j, i) \mid l_{ij} \neq 0\}$ (note that this is actually the graph of L^T). The graph is acyclic. If marked nodes in G_L correspond to nonzero entries in x, rule one translates into marking all those nodes $i \in \mathcal{B}$, where $\mathcal{B} = \{i \mid b_i \neq 0\}$. Rule two states that if node j is marked, and there is an edge from node j to node i, then node i must be marked. The set $\mathcal{X}$ becomes the set of all nodes in G_L that can be reached via a path from one or more nodes in $\mathcal{B}$. These rules are illustrated in Figure 3.1. In graph terminology, $\mathcal{X} = \text{Reach}_{G_L}(\mathcal{B})$, or more simply $\mathcal{X} = \text{Reach}_L(\mathcal{B})$, to avoid double subscripts. This gives a formal proof of the following theorem.

Theorem 3.1 (Gilbert and Peierls [109]). *Define the directed graph $G_L = (V, E)$ with nodes $V = \{1 \ldots n\}$ and edges $E = \{(j, i) \mid l_{ij} \neq 0\}$. Let $\text{Reach}_L(i)$ denote*

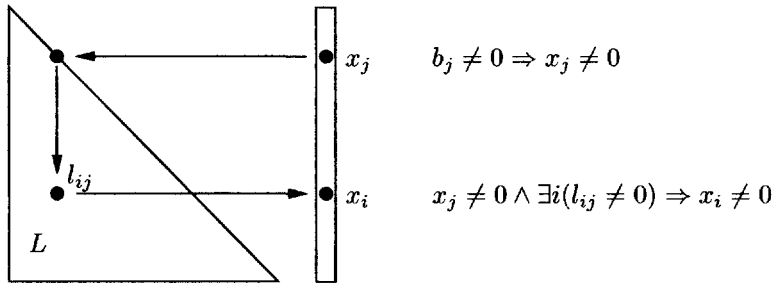


Figure 3.1. *Sparse triangular solve*

the set of nodes reachable from node i via paths in G_L, and let Reach($\mathcal{B}$), *for a set $\mathcal{B}$, be the set of all nodes reachable from any node in $\mathcal{B}$. The nonzero pattern $\mathcal{X} = \{j \mid x_j \neq 0\}$ of the solution x to the sparse linear system $Lx = b$ is given by $\mathcal{X} = \text{Reach}_L(\mathcal{B})$, where $\mathcal{B} = \{i \mid b_i \neq 0\}$, assuming no numerical cancellation.*

The set $\mathcal{X}$ can be computed by a depth-first search of the directed graph G_L, starting at nodes in $\mathcal{B}$. The time taken by a depth-first search is proportional to the number of edges traversed, plus the number of initial nodes in $\mathcal{B}$. Each edge reflects exactly two floating-point operations in the numerical solution to $Lx = b$, so the total time is thus $O(|b| + f)$. A depth-first search does not sort the set $\mathcal{X}$, however. Fortunately, the update $x_i = x_i - l_{ij}x_j$ can be computed as soon as x_j is known. This update translates into two nodes j and i in $\mathcal{X}$ with an edge from j to i in the directed graph G_L. An ordering of $\mathcal{X}$ that preserves this precedence is called a *topological order*, and a depth-first search can compute $\mathcal{X}$ in topological order (a breadth-first search cannot).

A depth-first search is most easily written as a recursive algorithm, stated in pseudocode below. The *reach* function computes $\mathcal{X} = \text{Reach}_L(\mathcal{B})$ by starting a depth-first search at each node $i \in \mathcal{B}$.

```
function X = reach (L, B)
    assume all nodes are unmarked
    for each i for which b_i ≠ 0 do
        if node i is unmarked
            dfs (i)

function dfs (j)
    mark node j
    for each i for which l_ij ≠ 0 do
        if node i is unmarked
            dfs (i)
    push j onto stack for X
```

These two pseudocodes can be implemented with `reachr` and `dfsr` below.

```
int reachr (const cs *L, const cs *B, int *xi, int *w)
{
    int p, n = L->n ;
    int top = n ;                                   /* stack is empty */
    for (p = B->p [0] ; p < B->p [1] ; p++)         /* for each i in pattern of b */
    {
        if (w [B->i [p]] != 1)                      /* if i is unmarked */
        {
            dfsr (B->i [p], L, &top, xi, w) ;       /* start a dfs at i */
        }
    }
    return (top) ;                                  /* return top of stack */
}

void dfsr (int j, const cs *L, int *top, int *xi, int *w)
{
    int p ;
    w [j] = 1 ;                                     /* mark node j */
    for (p = L->p [j] ; p < L->p [j+1] ; p++)       /* for each i in L(:,j) */
    {
        if (w [L->i [p]] != 1)                      /* if i is unmarked */
        {
            dfsr (L->i [p], L, top, xi, w) ;        /* start a dfs at i */
        }
    }
    xi [--(*top)] = j ;                             /* push j onto the stack */
}
```

The `reachr` function computes $\mathcal{X} = \mathrm{Reach}_L(\mathcal{B})$, where $\mathcal{B}$ is the nonzero pattern of the n-by-1 sparse column vector b. The function returns $\mathcal{X}$ in the `xi` array, in locations `top` to `n-1`, in topological order. The array `w` is of size `n` and must be all zero on input. In `reachr`, a depth-first search is started at each node in $\mathcal{B}$, unless that node is already marked. The `dfsr` function starts a depth-first search at node `j`. It simply marks node `j` and then starts a depth-first search at any unmarked neighbors. When it finishes, it pushes `j` onto a stack containing $\mathcal{X}$ on output, topologically sorted.

Recursive algorithms are easy to understand, but they can cause stack overflow if the recursion goes too deep. The stack depth is up to n, which limits the algorithm to solving matrices of modest size. Thus, CSparse does not rely on `dfrs` and `reachr` (they are included but only for reference). The `cs_dfs` function below uses its own recursion stack `xi[0]` to `xi[head]` that does not overlap with the output stack in `xi[top]` to `xi[n-1]`, since no node can be in both stacks at the same time. The input matrix is called `G`, because it need not be lower triangular. Two parameters are added in anticipation of Chapter 6 (`pinv` and `k`). The parameter `k` specifies which column of `B` contains the right-hand side b. For now, assume `pinv` is `NULL`.

Initializing the workspace `w` of size `n` takes $O(n)$ time. This can be avoided by marking nodes using the matrix `G` itself. To denote a marked node `j`, `Gp[j]` is set to `CS_FLIP(Gp[j])`, which exploits the fact that `Gp[j]` ≥ 0 in an unmodified matrix `G`. A marked node `j` will have `Gp[j]` < 0. To unmark a node or to obtain the original value of `Gp[j]`, `CS_FLIP` can be applied again, since the function is its own inverse. The name "flip" is used because the function "flips" its input about the integer `-1`. `CS_UNFLIP(i)` "flips" `i` if it is negative or returns `i` otherwise.

```
#define CS_FLIP(i) (-(i)-2)
#define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
#define CS_MARKED(w,j) (w [j] < 0)
#define CS_MARK(w,j) { w [j] = CS_FLIP (w [j]) ; }

int cs_reach (cs *G, const cs *B, int k, int *xi, const int *pinv)
{
    int p, n, top, *Bp, *Bi, *Gp ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi) return (-1) ;    /* check inputs */
    n = G->n ; Bp = B->p ; Bi = B->i ; Gp = G->p ;
    top = n ;
    for (p = Bp [k] ; p < Bp [k+1] ; p++)
    {
        if (!CS_MARKED (Gp, Bi [p]))    /* start a dfs at unmarked node i */
        {
            top = cs_dfs (Bi [p], G, top, xi, xi+n, pinv) ;
        }
    }
    for (p = top ; p < n ; p++) CS_MARK (Gp, xi [p]) ;  /* restore G */
    return (top) ;
}

int cs_dfs (int j, cs *G, int top, int *xi, int *pstack, const int *pinv)
{
    int i, p, p2, done, jnew, head = 0, *Gp, *Gi ;
    if (!CS_CSC (G) || !xi || !pstack) return (-1) ;    /* check inputs */
    Gp = G->p ; Gi = G->i ;
    xi [0] = j ;                /* initialize the recursion stack */
    while (head >= 0)
    {
        j = xi [head] ;         /* get j from the top of the recursion stack */
        jnew = pinv ? (pinv [j]) : j ;
        if (!CS_MARKED (Gp, j))
        {
            CS_MARK (Gp, j) ;       /* mark node j as visited */
            pstack [head] = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew]) ;
        }
        done = 1 ;                  /* node j done if no unvisited neighbors */
        p2 = (jnew < 0) ? 0 : CS_UNFLIP (Gp [jnew+1]) ;
        for (p = pstack [head] ; p < p2 ; p++)  /* examine all neighbors of j */
        {
            i = Gi [p] ;            /* consider neighbor node i */
            if (CS_MARKED (Gp, i)) continue ;   /* skip visited node i */
            pstack [head] = p ;     /* pause depth-first search of node j */
            xi [++head] = i ;       /* start dfs at node i */
            done = 0 ;              /* node j is not done */
            break ;                 /* break, to start dfs (i) */
        }
        if (done)               /* depth-first search at node j is done */
        {
            head-- ;            /* remove j from the recursion stack */
            xi [--top] = j ;    /* and place in the output stack */
        }
    }
    return (top) ;
}
```

The `cs_dfs` function starts by placing `j` in the recursion stack at `xi[0]`. Each

iteration of the `while` loop starts, or continues, the `j`th instance of `cs_dfs`. If `j` is on the recursion stack and it is not marked, then this is the first time it has been visited. In this case, the node is marked, and `pstack[head]` is set to point to the first outgoing edge of node `j`. If an unmarked node `i` is found, it is placed on the recursion stack, and the iteration for node `j` is paused. The next `while` loop iteration will then start the depth-first search for node `i`. When the depth-first search for node `j` eventually finishes, it is removed from the recursion stack and placed in the output stack.

The `cs_reach` function is nearly identical to `reachr`. It computes $\mathcal{X} = \mathrm{Reach}_G(\mathcal{B}_k)$, where $\mathcal{B}_k$ is the nonzero pattern of column `k` of B.

With `cs_reach` defined, solving $Lx = b$, where L, x, and b are all sparse, becomes a straightforward translation of the pseudocode. The `cs_spsolve` function computes the solution to $Lx = b_k$ (if `lo` is nonzero), where b_k is the kth column of B. When `lo` is nonzero, the function assumes `G` $= L$ is lower triangular with the diagonal entry as the first entry in each column. It takes an optimal $O(|b|+f)$ time. Solving an upper triangular system $Ux = b$ is almost identical to solving $Lx = b$. Its derivation is left as an exercise. With `lo` equal to zero, the `cs_spsolve` function assumes `G` $= U$ is upper triangular with the diagonal entry as the last entry in each column.

```
int cs_spsolve (cs *G, const cs *B, int k, int *xi, double *x, const int *pinv,
    int lo)
{
    int j, J, p, q, px, top, n, *Gp, *Gi, *Bp, *Bi ;
    double *Gx, *Bx ;
    if (!CS_CSC (G) || !CS_CSC (B) || !xi || !x) return (-1) ;
    Gp = G->p ; Gi = G->i ; Gx = G->x ; n = G->n ;
    Bp = B->p ; Bi = B->i ; Bx = B->x ;
    top = cs_reach (G, B, k, xi, pinv) ;        /* xi[top..n-1]=Reach(B(:,k)) */
    for (p = top ; p < n ; p++) x [xi [p]] = 0 ;    /* clear x */
    for (p = Bp [k] ; p < Bp [k+1] ; p++) x [Bi [p]] = Bx [p] ; /* scatter B */
    for (px = top ; px < n ; px++)
    {
        j = xi [px] ;                               /* x(j) is nonzero */
        J = pinv ? (pinv [j]) : j ;                 /* j maps to col J of G */
        if (J < 0) continue ;                       /* column J is empty */
        x [j] /= Gx [lo ? (Gp [J]) : (Gp [J+1]-1)] ;/* x(j) /= G(j,j) */
        p = lo ? (Gp [J]+1) : (Gp [J]) ;            /* lo: L(j,j) 1st entry */
        q = lo ? (Gp [J+1]) : (Gp [J+1]-1) ;        /* up: U(j,j) last entry */
        for ( ; p < q ; p++)
        {
            x [Gi [p]] -= Gx [p] * x [j] ;          /* x(i) -= G(i,j) * x(j) */
        }
    }
    return (top) ;                                  /* return top of stack */
}
```

The function returns the nonzero pattern $\mathcal{X}$ in `xi[top]` through `xi[n-1]`, an array of size `2*n`. The first `n` entries of `xi` holds the output stack and the recursion stack for `j`. The second `n` entries holds the stack for `p` in `cs_dfs`. The numerical values are in the dense vector `x`, which need not be initialized on input. To solve $Lx = b$, a `NULL` pointer must be passed for `pinv`, and `lo` must be nonzero.

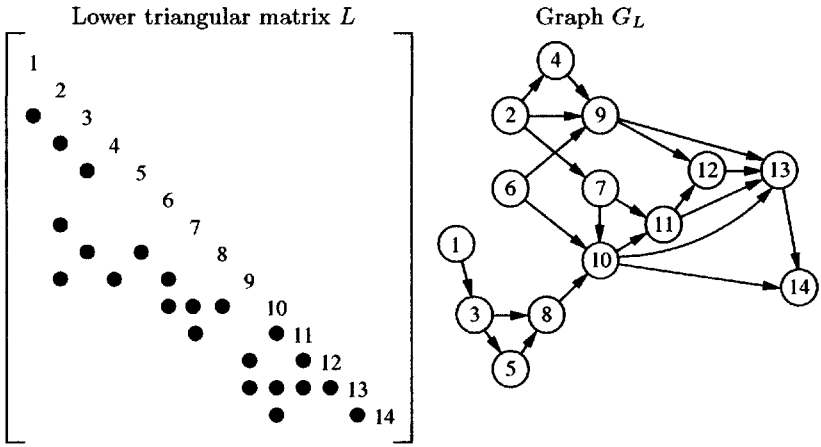


Figure 3.2. *Solving $Lx = b$ where L, x, and b are sparse*

An example is shown in Figure 3.2. Suppose $\mathcal{B} = \{4, 6\}$. The depth-first search can start at either node 4 or node 6, and the neighbors of any node can be searched in any order. If node 4 is searched first and the row indices in each column of L are sorted, Reach(4) $= \{4, 9, 12, 13, 14\}$ in order. This list of 5 nodes is placed on the stack `xi`, and node 6 is searched next; Reach(6) $= \{6, 9, 10, 11, 12, 13, 14\}$, but some of these nodes are already marked. The stack `xi` will contain the list $\{6, 10, 11, 4, 9, 12, 13, 14\}$ in topological order. The forward solve will access the columns of L in this order. The work done at columns 6, 10, and 11 is not affected by the work done at columns 4 and 9.

MATLAB does not have an exact equivalent to `cs_reach` or `cs_spsolve`, except as used internally in `[L,U,P]=lu(A)`. The MATLAB expression `x=L\b` takes $O(n + |L|)$ time to compute a sparse `x` if `L` is sparse and `b` is a sparse vector.

3.3 Further reading

The sparse triangular solve forms the basis of the GPLU algorithm in MATLAB [105, 109]. All sparse matrix packages with direct methods include forward solvers and backsolvers. Not all provide transposed solvers such as `cs_ltsolve` or sparse solvers such as `cs_spsolve`. Cormen, Leiserson, and Rivest [23] present algorithms for the depth-first search and topological sort of a graph.

Exercises

3.1. Derive the algorithm used by `cs_utsolve`.

3.2. *Try* to describe an algorithm for solving $Lx = b$, where L is stored in triplet form, and x and b are dense vectors. What goes wrong?

3.3. The MATLAB statement `[L,U]=lu(A)` finds a permuted lower triangular matrix `L` and an upper triangular matrix `U` so that `L*U=A`. The rows of `L` have been permuted via partial pivoting, but the permutation itself is not available. Write a permuted triangular solver that computes `x=L\b` without modifying `L` and with only $O(n)$ extra workspace. Two passes of the matrix `L` are required, each taking $O(|L|)$ time. The first pass finds the diagonal entry in each column. Start at the last column and work backwards, flagging rows as they are seen. There will be exactly one nonzero entry in an unflagged row in each column. This is the diagonal entry of the unpermuted lower triangular matrix. In any given column, if there are no entries in unflagged rows, or more than one, then the matrix is not a permuted lower triangular matrix. The second pass then performs the permuted forward solve.

3.4. Repeat Problem 3.3 for a matrix `U` that is an upper triangular matrix whose rows have been permuted. The algorithm is almost identical to Problem 3.3.

3.5. Repeat Problem 3.3 for a matrix `L` that is a lower triangular matrix whose columns have been permuted. Assume the first entry in each column has the smallest row index. This problem is simpler than Problem 3.3. Two passes of `L` are still required, but the first takes only $O(n)$ time.

3.6. Repeat Problem 3.5 for a matrix `U` that is an upper triangular matrix whose columns have been permuted. The solution will be similar to Problem 3.5.

3.7. This problem is a generalization of Problems 3.3 through 3.6 and is the method used in MATLAB (due to Gilbert). Consider a matrix A that may be a permuted upper or lower triangular matrix with both rows and columns permuted by unknown permutations P and Q. Write an algorithm that determines if the matrix is in this form and, if so, solves $Ax = b$. A single integer `s` can represent a set of integers of arbitrary size that supports the following operations: `s=0` clears the set; `s=s^j` (the exclusive-or) removes `j` from the set if `j` is not in the set or adds it to the set otherwise; `j=s` gets the member of the set if the set has size one. Let `r[i]` be the count of nonzeros in row `i` of `A`. Let `z[i]` represent the `i`th set; it starts as the exclusive-or of all column indices of nonzeros in row `i`. Create a linked list of all row singletons (rows with only one entry). For `n` iterations where `A` has dimension `n`, select a row `i` from the list. If the list is empty, the matrix is not a permuted triangular matrix. Otherwise, the corresponding column is `j=z[i]`. Append `i` and `j` to the permutations `p` and `q`. Remove `j` from the sets `z[t]` for all `t` $\in \mathcal{A}_{*j}$ and decrement their row counts by one. Add any new row singletons to the linked list. If successful, follow this by a permuted triangular solve using the newly discovered permutations `p` and `q`. The permuted matrix `A(p,q)` need not be formed explicitly.

3.8. The `cs_lsolve` and `cs_usolve` functions assume that `b` contains no zero entries. The time can be reduced if it does. Note that the inner `for` loop in the two functions can be skipped if `x[j]` is zero. Add this test and compare the run times of the modified and original functions. The modified functions take $O(n + f)$ time; the original ones take $O(|L|)$ and $O(|U|)$ time.

3.9. Prove that the `cs_spsolve` function solves $Ux = b$ when `lo` is zero.

Chapter 4

Cholesky factorization

The *Cholesky factorization* of a sparse symmetric positive definite matrix A is the product $A = LL^T$, where L is a lower triangular matrix with positive entries on its diagonal. Entries in L that do not appear in A are called *fill-in*. Let G_{L+L^T} be the undirected graph of $L + L^T$; it is called the *filled graph* of A. The structure of G_{L+L^T} is given by the following theorem.

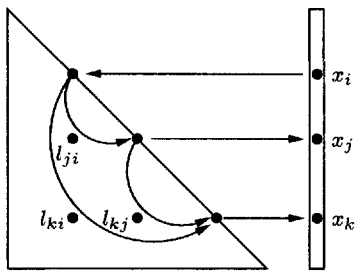
Theorem 4.1 (Rose, Tarjan, and Lueker [175]). *The edge (i, j) is in the undirected graph G_{L+L^T} of $L+L^T$ if and only if there exists a path $i \rightsquigarrow j$ in the undirected graph of A where all nodes in the path except i and j are numbered less than* $\min(i, j)$.

Numeric Cholesky factorization is typically preceded by a *symbolic analysis* step that determines either G_{L+L^T} or some of its key properties. The goal is to keep the numeric factorization as simple as possible in terms of time complexity, memory usage, and clarity of code. The analysis step presented here finds the *elimination tree*, computes its *postordering*, and then computes the *column counts*, which are the number of nonzeros in each column of L. Some numeric Cholesky factorization algorithms also need the nonzero pattern of L.

There are many ways to compute the Cholesky factorization $A = LL^T$ and the graph G_{L+L^T}. The sparse triangular solve, $Lx = b$, forms a common thread throughout this book; it is used as the basis of an *up-looking* sparse Cholesky factorization described here. Consider a 2-by-2 block decomposition $LL^T = A$,

$$\begin{bmatrix} L_{11} & \\ l_{12}^T & l_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} \\ & l_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix}, \tag{4.1}$$

where L_{11} and A_{11} are $(n-1)$-by-$(n-1)$. The three equations that lead to the up-looking Cholesky factorization algorithm are $L_{11}L_{11}^T = A_{11}$, $L_{11}l_{12} = a_{12}$, and $l_{12}^T l_{12} + l_{22}^2 = a_{22}$. The first equation can be solved recursively to obtain L_{11}, followed by a sparse triangular solve using the second equation to compute l_{12}. Finally, a sparse dot product and scalar square root, $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$, result in l_{22}. If the matrix is positive definite, then $a_{22} > l_{12}^T l_{12}$ holds, and the Cholesky



Thm. 4.2: $a_{ik} \neq 0$ implies $l_{ki} \neq 0$
Thm. 4.3: $l_{ji} \neq 0$ and $l_{ki} \neq 0$ implies $l_{kj} \neq 0$
Thus l_{ki} redundant for $\mathcal{X} = \text{Reach}_L(i)$

Figure 4.1. *Pruning the directed graph G_L yields the elimination tree $\mathcal{T}$*

factorization exists. The nonrecursive version of this algorithm is demonstrated by chol_up below. It is called up-looking because it looks up (accessing L_{11} or rows 1 to $k-1$ of L) to construct the kth row of L.

```
function L = chol_up (A)
n = size (A) ;
L = zeros (n) ;
for k = 1:n
    L (k,1:k-1) = (L (1:k-1,1:k-1) \ A (1:k-1,k))' ;
    L (k,k) = sqrt (A (k,k) - L (k,1:k-1) * L (k,1:k-1)') ;
end
```

4.1 Elimination tree

The *elimination tree* is even more important to sparse matrix algorithms than the sparse triangular solve. It appears in many algorithms and many theorems, both in this book and elsewhere. One motivation for deriving the tree is to reduce the time required to compute the graph reachability (Theorem 3.1) for the sparse triangular solve presented in Section 3.2. This triangular solve is required by the up-looking sparse Cholesky factorization algorithm just described in the previous section.

Consider (4.1). The vector l_{12} is computed with a sparse triangular solve, $L_{11}l_{12} = a_{12}$, and its transpose becomes the kth row of L. Its nonzero pattern is thus $\mathcal{L}_k = \text{Reach}_{G_{k-1}}(\mathcal{A}_k)$, where G_{k-1} is the directed graph of L_{11}, $\mathcal{L}_k$ denotes the nonzero pattern of the kth row of L, and $\mathcal{A}_k$ denotes the nonzero pattern of the upper triangular part of the kth column of A.

The depth-first search of G_{k-1} is sufficient for computing $\mathcal{L}_k$, but a simpler method exists, taking only $O(|\mathcal{L}_k|)$ time. Consider any $i < j < k$, where $l_{ji} \neq 0$ and $a_{ik} \neq 0$, as shown in Figure 4.1. A graph traversal of G_{k-1} will start at node i. Thus, $i \in \mathcal{L}_k$, the nonzero pattern of the solution to the triangular system. This becomes the kth row of L, and thus $l_{ki} \neq 0$. The traversal will visit node j because of the edge (i, j) (corresponding to the nonzero l_{ji}). Thus, $j \in \mathcal{L}_k$, and $l_{kj} \neq 0$; two nonzeros in column i (l_{ji} and l_{ki}) imply that l_{kj} is nonzero also. In terms of the directed graph G_L (not just the graph of L_{11}), edges (i, j) and (i, k) imply edge (j, k). These facts are summarized in the following theorems that completely define the nonzero pattern of the Cholesky factor L.

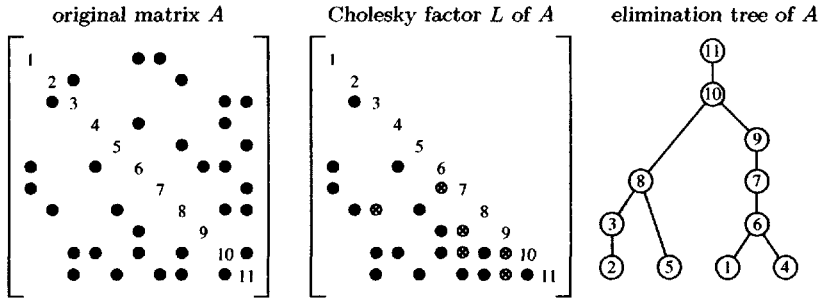


Figure 4.2. *Example matrix A, factor L, and elimination tree*

Theorem 4.2. *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $a_{ij} \neq 0 \Rightarrow l_{ij} \neq 0$. That is, if a_{ij} is nonzero, then l_{ij} will be nonzero as well.*

Theorem 4.3 (Parter [165]). *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $i < j < k \wedge l_{ji} \neq 0 \wedge l_{ki} \neq 0 \Rightarrow l_{kj} \neq 0$. That is, if both l_{ji} and l_{ki} are nonzero where $i < j < k$, then l_{kj} will be nonzero as well.*

Since there is a path from i to k via j that does not traverse the edge (i, k), the edge (i, k) is not needed to compute Reach(i). The set Reach(t) for any other node $t < i$ with a path $t \rightsquigarrow i$ is also not affected if (i, k) is removed from the directed graph G_L. This removal of edges leaves at most one outgoing edge from node i in the pruned graph, all the while not affecting Reach(i). If $j > i$ is the least numbered node for which $l_{ji} \neq 0$, all other nonzeros l_{ki} where $k > j$ are redundant.

The result is the elimination tree. The parent of node i in the tree is j, where the first off-diagonal nonzero in column i has row index j (the smallest $j > i$ for which $l_{ji} \neq 0$). Node i is a root of the tree if column i has no off-diagonal nonzero entries; it has no parent. The tree may actually be a forest, with multiple roots, if the graph of A consists of multiple connected components (there will be one tree per component of A). By convention, it is still called a tree. Assume the edges of the tree are directed, from a child to its parent. Let T denote the elimination tree of L, and let T_k denote the elimination tree of submatrix $L_{1\ldots k,1\ldots k}$, the first k rows and columns of L. An example matrix A, its Cholesky factor L, and its elimination tree T are shown in Figure 4.2. In the factor L, *fill-in* (entries that appear in L but not in A) are shown as circled x's. Diagonal entries are numbered for easy reference.

The existence of the elimination tree has been shown; it is now necessary to compute it. A few more theorems are required.

Theorem 4.4 (Schreiber [181]). *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $l_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree T; equivalently, $i \rightsquigarrow k$ is a path in T.*

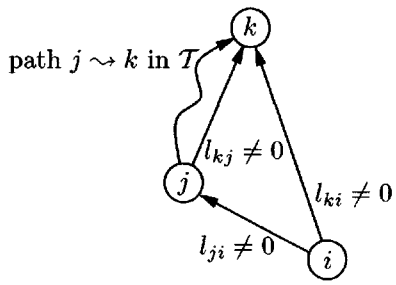


Figure 4.3. *Illustration of Theorem* 4.4

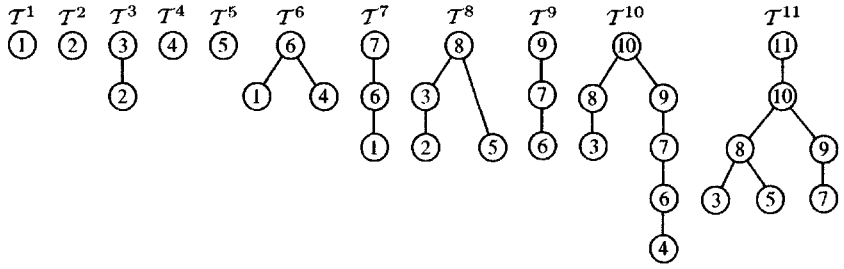


Figure 4.4. *Row subtrees of the example in Figure* 4.2

Proof. Refer to Figure 4.3. The proof is by induction on i for a fixed k. Let $j = \min\{j \mid l_{ji} \neq 0 \wedge j > i\}$ be the parent of i. The parent $j > i$ must exist because $l_{ki} \neq 0$. For the base case, if $k = j$, then k is the parent of i and thus $i \rightsquigarrow k$ is a path in $\mathcal{T}$. For the inductive step, $k > j > i$ must hold, and there are thus two nonzero entries l_{ki} and l_{ji}. From Theorem 4.3, $l_{kj} \neq 0$. By induction, l_{kj} implies the path $j \rightsquigarrow k$ exists in $\mathcal{T}$. Combined with the edge (i, j), this means there is a path $i \rightsquigarrow k$ in $\mathcal{T}$. □

Removing edges from the directed graph G_L to obtain the elimination tree $\mathcal{T}$ does not affect the Reach(i) of any node. The result is the following theorem.

Theorem 4.5 (Liu [148]). *The nonzero pattern $\mathcal{L}_k$ of the kth row of L is given by*

$$\mathcal{L}_k = Reach_{G_{k-1}}(\mathcal{A}_k) = Reach_{\mathcal{T}_{k-1}}(\mathcal{A}_k). \tag{4.2}$$

Theorem 4.5 defines $\mathcal{L}_k$. The kth row subtree, denoted $\mathcal{T}^k$, is the subtree of $\mathcal{T}$ induced by the nodes in $\mathcal{L}_k$. The 11 row subtrees $\mathcal{T}^1, \ldots, \mathcal{T}^{11}$ of the matrix shown in Figure 4.2 are shown in Figure 4.4. Each row subtree is characterized by its leaves, which correspond to entries in A. This fact is summarized by the following theorem.

Theorem 4.6 (Liu [148]). *Node j is a leaf of $\mathcal{T}^k$ if and only if both $a_{jk} \neq 0$ and $a_{ik} = 0$ for every descendant i of j in the elimination tree $\mathcal{T}$.*

A corollary to Theorem 4.4 fully characterizes the elimination tree $\mathcal{T}$.

Corollary 4.7 (Liu [148]). *For a Cholesky factorization $LL^T = A$, and neglecting numerical cancellation, $a_{ki} \neq 0$ and $k > i$ imply that i is a descendant of k in the elimination tree $\mathcal{T}$; equivalently, $i \rightsquigarrow k$ is a path in $\mathcal{T}$.*

Theorem 4.4 and Corollary 4.7 lead to an algorithm that computes the elimination tree $\mathcal{T}$ in almost $O(|A|)$ time. Suppose $\mathcal{T}_{k-1}$ is known. This tree is a subset of $\mathcal{T}_k$. To compute $\mathcal{T}_k$ from $\mathcal{T}_{k-1}$, the children of k (which are root nodes in $\mathcal{T}_{k-1}$) must be found. Since $a_{ki} \neq 0$ implies the path $i \rightsquigarrow k$ exists in $\mathcal{T}$, this path can be traversed in $\mathcal{T}_{k-1}$ until reaching a root node. This node must be a child of k, since the path $i \rightsquigarrow k$ must exist.

To speed up the traversal of the partially constructed elimination tree $\mathcal{T}_{k-1}$, a set of *ancestors* is kept. The ancestor of i, ideally, would simply be the root r of the partially constructed tree $\mathcal{T}_{k-1}$ that contains i. Traversing the path $i \rightsquigarrow r$ would take $O(1)$ time, simply by finding the ancestor r of i. This goal can nearly be met by a *disjoint-set-union* data structure. An optimal one would result in a total time complexity of $O(|A|\alpha(|A|, n))$ for the $|A|$ path traversals that need to be made, where $\alpha(|A|, n)$ is the inverse Ackermann function, a very slowly growing function. However, a simpler method is used that leads to an $O(|A| \log n)$ time algorithm. The $\log n$ upper bound is never reached in practice, however, and the resulting algorithm takes practically $O(|A|)$ time and is faster (in practice, not asymptotically) than the $O(|A|\alpha(|A|, n))$-time disjoint-set-union algorithm. The time complexity of `cs_etree` is called *nearly* $O(|A|)$ time.

The `cs_etree` function computes the elimination tree of the Cholesky factorization of A (assuming `ata` is false), using just `A` and returning the `int` array `parent`, of size `n`. It iterates over each column `k` and considers every entry a_{ik} in the upper triangular part of A. It updates the tree, following the path from `i` to the root of the tree. Rather than following the path via the `parent` array, an array `ancestor` is kept, where `ancestor[i]` is the highest known ancestor of `i`, not necessarily the root of the tree in $\mathcal{T}_{k-1}$ containing `i`. If `r` is a root, it has no ancestor (`ancestor[r]` is `-1`). Since the path is guaranteed to lead to node `k` in $\mathcal{T}_k$, the ancestors of all nodes along this path are set to `k` (*path compression*). If a root node is reached in $\mathcal{T}_{k-1}$ that is not `k`, it must be a child of `k` in $\mathcal{T}_k$; `parent` is updated to reflect this.

If the input parameter `ata` is true, `cs_etree` computes the elimination tree of A^TA without forming A^TA. This is the *column elimination tree*. It will be used in the QR and LU factorization algorithms in Chapters 5 and 6. Row i of A creates a dense submatrix, or clique, in the graph of A^TA. Rather than using the graph of A^TA (with one node corresponding to each column of A), a new graph is constructed dynamically (also with one node per column of A). If the nonzero pattern of row i contains column indices $j_1, j_2, j_3, j_4, \ldots$, the new graph is given edges (j_1, j_2), (j_2, j_3), (j_3, j_4), and so on. Each row i creates a path in this new graph. In the tree, these edges ensure $j_1 \rightsquigarrow j_2 \rightsquigarrow j_3 \rightsquigarrow j_4 \ldots$ is a path in $\mathcal{T}$.

The clique in A^TA has edges between all nodes $j_1, j_2, j_3, j_4, \ldots$ and will have the same ancestor/descendant relationship. Thus, the elimination tree of A^TA and this new graph will be the same. The path is constructed dynamically as the algorithm progresses, using the `prev` array. `prev[i]` starts out equal to `-1` for all `i`. Let $\mathcal{A}_k$ be the nonzero pattern of `A(:,k)`. When column `k` is considered, the edge (`prev[i]`,`k`) is created for each $i \in \mathcal{A}_k$. This edge is used to update the elimination tree, traversing from `prev[i]` up to `k` in the tree. After this traversal, `prev[i]` is set to `k`, to prepare for the next edge in this row `i`, for a subsequent column in the outer `for k` loop.

```
int *cs_etree (const cs *A, int ata)
{
    int i, k, p, m, n, inext, *Ap, *Ai, *w, *parent, *ancestor, *prev ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ;
    parent = cs_malloc (n, sizeof (int)) ;              /* allocate result */
    w = cs_malloc (n + (ata ? m : 0), sizeof (int)) ;   /* get workspace */
    if (!w || !parent) return (cs_idone (parent, NULL, w, 0)) ;
    ancestor = w ; prev = w + n ;
    if (ata) for (i = 0 ; i < m ; i++) prev [i] = -1 ;
    for (k = 0 ; k < n ; k++)
    {
        parent [k] = -1 ;                   /* node k has no parent yet */
        ancestor [k] = -1 ;                 /* nor does k have an ancestor */
        for (p = Ap [k] ; p < Ap [k+1] ; p++)
        {
            i = ata ? (prev [Ai [p]]) : (Ai [p]) ;
            for ( ; i != -1 && i < k ; i = inext)   /* traverse from i to k */
            {
                inext = ancestor [i] ;              /* inext = ancestor of i */
                ancestor [i] = k ;                  /* path compression */
                if (inext == -1) parent [i] = k ;   /* no anc., parent is k */
            }
            if (ata) prev [Ai [p]] = k ;
        }
    }
    return (cs_idone (parent, NULL, w, 1)) ;
}
```

The `cs_idone` function used by `cs_etree` returns an `int` array and frees any workspace.

```
int *cs_idone (int *p, cs *C, void *w, int ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    return (ok ? p : cs_free (p)) ;     /* return result if OK, else free it */
}
```

The MATLAB statement `parent=etree(A)` computes the elimination tree of a symmetric matrix `A`, represented as a size-`n` array; `parent(i)=k` if `k` is the parent of `i`. To compute the column elimination tree, a second parameter is added: `parent=etree(A,'col')`. Both algorithms in MATLAB use the same method as in `cs_etree` (using `cholmod_etree`).

4.2 Sparse triangular solve

Solving $Lx = b$ when L, x, and b are sparse has already been discussed in the general case in Section 3.2. Here, L is a more specific lower triangular matrix, arising from a Cholesky factorization. The set $\text{Reach}_L(i)$ for a Cholesky factor L can be computed in time proportional to the size of the set via a simple tree traversal that starts at node i and traverses the path to the root of the elimination tree. This is much less than the time to compute $\text{Reach}(i)$ in the general case.

The `cs_ereach` function computes the nonzero pattern of the kth row of L, $\mathcal{L}_k = \text{Reach}_{k-1}(\mathcal{A}_k)$, where Reach_{k-1} denotes the graph of `L(1:k-1,1:k-1)`. It uses the elimination tree $\mathcal{T}$ of L to traverse the kth row subtree, $\mathcal{T}^k$. The set $\mathcal{A}_k$ denotes the nonzero pattern of the kth column of the upper triangular part of A. This is the same as the kth row of the lower triangular part of A, which is how the kth row subtree $\mathcal{T}^k$ is traversed. The first part of the code iterates for each i in this set $\mathcal{A}_k$. Next, the path from i towards the root of the tree is traversed. The traversal marks the nodes and stops if it encounters a marked node. Nodes are marked using the `w` array of size `n`; all entries in `w` must be greater than or equal to zero on input, and the contents of `w` are restored on output. This subpath is then pushed onto the output stack. On output, the set $\mathcal{L}_k$ is contained in `s[top]` through `s[n-1]` (except for the diagonal entry). It is created in topological order.

```
int cs_ereach (const cs *A, int k, const int *parent, int *s, int *w)
{
    int i, p, n, len, top, *Ap, *Ai ;
    if (!CS_CSC (A) || !parent || !s || !w) return (-1) ;   /* check inputs */
    top = n = A->n ; Ap = A->p ; Ai = A->i ;
    CS_MARK (w, k) ;                /* mark node k as visited */
    for (p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        i = Ai [p] ;                /* A(i,k) is nonzero */
        if (i > k) continue ;       /* only use upper triangular part of A */
        for (len = 0 ; !CS_MARKED (w,i) ; i = parent [i]) /* traverse up etree*/
        {
            s [len++] = i ;         /* L(k,i) is nonzero */
            CS_MARK (w, i) ;        /* mark i as visited */
        }
        while (len > 0) s [--top] = s [--len] ; /* push path onto stack */
    }
    for (p = top ; p < n ; p++) CS_MARK (w, s [p]) ;    /* unmark all nodes */
    CS_MARK (w, k) ;                /* unmark node k */
    return (top) ;                  /* s [top..n-1] contains pattern of L(k,:)*/
}
```

The total time taken by the algorithm is $O(|\mathcal{L}_k|)$, the number of nonzeros in row k of L. This is much faster in general than the `cs_reach` function that computes $\text{Reach}_L(\mathcal{B})$ for an arbitrary lower triangular L. Solving $Lx = b$ is an integrated part of the up-looking Cholesky factorization; a stand-alone $Lx = b$ solver when L is a Cholesky factor is left as an exercise.

The `cs_ereach` function can be used to construct the elimination tree itself by extending the tree one node at a time. The time taken would be $O(|L|)$. As a by-product, the entries in L are created one at a time. They can be kept to obtain

the nonzero pattern of L, or they can be counted and discarded to obtain a count of nonzeros in each row and column of L. The latter function is done more efficiently with `cs_post` and `cs_counts`, discussed in the next three sections.

MATLAB uses a code similar to `cs_ereach` in its sparse Cholesky factorization methods (an up-looking sparse Cholesky `cholmod_rowfac` and a supernodal symbolic factorization, `cholmod_super_symbolic`). However, it cannot use the tree when computing `x=L\b`, for several reasons. The elimination tree is discarded after `L` is computed. MATLAB does not keep track of how `L` was computed, and `L` may be modified prior to using it in `x=L\b`. It may be an arbitrary sparse lower triangular system, whose nonzero pattern is not governed by the tree. Numerically zero entries are dropped from `L`, so even if `L` is not modified by the application, the tree cannot be determined from the first off-diagonal entry in each column of `L`. For these reasons, the MATLAB statement `x=L\b` determines only that `L` is sparse and lower triangular (see Section 8.5) and uses an algorithm much like `cs_lsolve`.

4.3 Postordering a tree

Once the elimination tree is found, its *postordering* can be found. A postorder of the tree is required for computing the number of nonzeros in each column of L in the algorithm presented in the next section. It is also useful in its own right. If A is permuted according to the postordering P (`C=A(p,p)` in MATLAB notation), then the number of nonzeros in $LL^T = C$ is unchanged, but the nonzero pattern of L will be more structured and the numerical factorization will often be faster as a result, even with no change to the factorization code.

Theorem 4.8 (Liu [150]). *The filled graphs of A and PAP^T are isomorphic if P is a postordering of the elimination tree of A. Likewise, the elimination trees of A and PAP^T are isomorphic.*

In a postordered tree, the d proper descendants of any node k are numbered $k-d$ through $k-1$. If `post` represents the postordering permutation, `post[k]=i` means node `i` of the original tree is node `k` of the postordered tree. The most natural postordering preserves the relative ordering of the children of each node; if nodes $c_1 < c_2 < \cdots < c_t$ are the t children of node k, then $\texttt{post}[c_1] < \texttt{post}[c_2] < \cdots < \texttt{post}[c_t]$. A recursive depth-first search of the tree can compute the postordering:

function *postorder* (T)
    $k = 0$
    for each root node j of T **do**
        dfstree (j)

function *dfstree* (j)
    for each child i of j **do**
        dfstree (i)
    $\texttt{post}[k] = j$
    $k = k + 1$

However, the depth of the elimination tree can easily be $O(n)$, causing stack overflow for large matrices. A nonrecursive implementation is better, as shown in the `cs_post` function below.

```
int *cs_post (const int *parent, int n)
{
    int j, k = 0, *post, *w, *head, *next, *stack ;
    if (!parent) return (NULL) ;                        /* check inputs */
    post = cs_malloc (n, sizeof (int)) ;                /* allocate result */
    w = cs_malloc (3*n, sizeof (int)) ;                 /* get workspace */
    if (!w || !post) return (cs_idone (post, NULL, w, 0)) ;
    head = w ; next = w + n ; stack = w + 2*n ;
    for (j = 0 ; j < n ; j++) head [j] = -1 ;           /* empty linked lists */
    for (j = n-1 ; j >= 0 ; j--)                /* traverse nodes in reverse order*/
    {
        if (parent [j] == -1) continue ;        /* j is a root */
        next [j] = head [parent [j]] ;          /* add j to list of its parent */
        head [parent [j]] = j ;
    }
    for (j = 0 ; j < n ; j++)
    {
        if (parent [j] != -1) continue ;        /* skip j if it is not a root */
        k = cs_tdfs (j, k, head, next, post, stack) ;
    }
    return (cs_idone (post, NULL, w, 1)) ;  /* success; free w, return post */
}

int cs_tdfs (int j, int k, int *head, const int *next, int *post, int *stack)
{
    int i, p, top = 0 ;
    if (!head || !next || !post || !stack) return (-1) ;    /* check inputs */
    stack [0] = j ;                 /* place j on the stack */
    while (top >= 0)                /* while (stack is not empty) */
    {
        p = stack [top] ;           /* p = top of stack */
        i = head [p] ;              /* i = youngest child of p */
        if (i == -1)
        {
            top-- ;                 /* p has no unordered children left */
            post [k++] = p ;        /* node p is the kth postordered node */
        }
        else
        {
            head [p] = next [i] ;   /* remove i from children of p */
            stack [++top] = i ;     /* start dfs on child node i */
        }
    }
    return (k) ;
}
```

First, workspace is allocated, and a set of `n` linked lists is initialized. The `j`th linked list contains a list of all the children of node `j` in ascending order. Next, nodes `j` from `0` to `n-1` are traversed, corresponding to the **for** j loop in the *postorder* pseudocode. If `j` is a root, a depth-first search of the tree is performed, using `cs_tdfs`. The `cs_tdfs` function places the root `j` on a stack. Each iteration of the `while` loop considers the node `p` at the top of the stack. If it has no unordered children

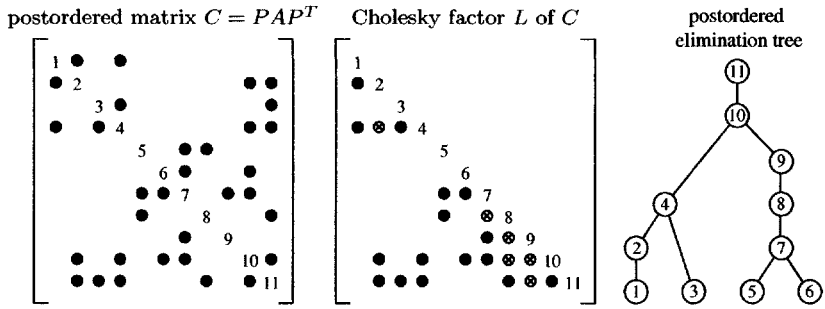


Figure 4.5. *After elimination tree postordering*

left, it is removed from the stack and ordered as the `k`th node in the postordering. Otherwise, its youngest unordered child `i` is removed from the head of the `p`th linked list and placed on the stack. The next iteration of the `while` loop will commence the depth-first search at this node `i`.

The `cs_post` function takes as input the elimination tree T represented as the `parent` array of size `n`. The `parent` array is not modified. The function returns a pointer to an integer vector `post`, of size `n`, that contains the postordering. The total time taken by the postordering is $O(n)$.

Figure 4.5 illustrates the matrix PAP^T, its Cholesky factor, and its elimination tree, where P is the postordering of the elimination tree in Figure 4.2.

The MATLAB statement `[parent,post] = etree(A)` computes the elimination tree and its postordering, using the same algorithms (`cholmod_etree` and `cholmod_postorder`). Node `i` is the `k`th node in the postordered tree if `post(k)=i`. A matrix can be permuted with this postordering via `C=A(post,post)`; the number of nonzeros in `chol(A)` and `chol(C)` will be the same. Looking ahead, Chapter 7 discusses how to find a fill-reducing ordering, `p`, where the permuted matrix is `A(p,p)`. This permutation vector `p` can be combined with the postordering of `A(p,p)`, using the following MATLAB code:

```
[parent, post] = etree (A (p,p)) ;
p = p (post) ;
```

4.4 Row counts

The *row counts* of a Cholesky factorization are the numbers of nonzeros in each row of L. The numeric factorization presented in Section 4.7 requires the *column counts* (the number of nonzeros in each column of L), not the row counts. However, the row count algorithm is simpler, and many of the ideas and theorems needed to understand the simpler row count algorithm are used in the column count algorithm.

A simple (yet nonoptimal) method of computing both the row and column counts is to traverse each row subtree. For each row i, consider each nonzero a_{ij}. Traverse up the tree from node j marking each node and stop if node i or a marked node is found (the marks must be cleared for each row i). The row count for

row i is the number of nodes in the row subtree T^i, and the column counts can be accumulated while traversing each node of the ith row subtree. The number of nonzeros in column j of L is the number of row subtrees that contain node j. However, this method requires $O(|L|)$ time. The goal of this section and the following one is to show how to compute the row and column counts in nearly $O(|A|)$ time.

To reduce the time complexity to nearly $O(|A|)$, five concepts must be introduced: (1) the *least common ancestor* of two nodes, (2) *path decomposition*, (3) the *first descendant* of each node, (4) the *level* of a node in the elimination tree, and (5) the *skeleton matrix*. The basic idea is to decompose each row subtree into a set of disjoint paths, each starting with a leaf node and terminating at the least common ancestor of the current leaf and the prior leaf node. The paths are not traversed one node at a time. Instead, the length of these paths are found via the difference in the levels of their starting and ending nodes, where the level of a node is its distance from the root. The row count algorithm exploits the fact that all subtrees are related to each other; they are all subtrees of the elimination tree.

The first step in the row count algorithm is to find the level and first descendant of each node of the elimination tree. The *first descendant* of a node j is the smallest postordering of any descendant of j. The first descendant and level of each node of the tree can be easily computed in $O(n)$ time by the `firstdesc` function below.

```
void firstdesc (int n, int *parent, int *post, int *first, int *level)
{
    int len, i, k, r, s ;
    for (i = 0 ; i < n ; i++) first [i] = -1 ;
    for (k = 0 ; k < n ; k++)
    {
        i = post [k] ;      /* node i of etree is kth postordered node */
        len = 0 ;           /* traverse from i towards the root */
        for (r = i ; r != -1 && first [r] == -1 ; r = parent [r], len++)
            first [r] = k ;
        len += (r == -1) ? (-1) : level [r] ;   /* root node or end of path */
        for (s = i ; s != r ; s = parent [s]) level [s] = len-- ;
    }
}
```

A node `i` whose first descendant has not yet been computed has `first[i]` equal to `-1`. The function starts at the first node (`k=0`) in the postordered elimination tree and traverses up towards the root. All nodes `r` along this path from node zero to the root have a first descendant of zero, and `first[r]=k` is set accordingly. For `k>0`, the traversal can also terminate at a node `r` whose `first[r]` and `level[r]` have already been determined. Once the path has been found, it is retraversed to set the levels of each node along this path.

Once the first descendant and level of each node are found, the row subtree is decomposed into disjoint paths. To do this, the leaves of the row subtrees must be found. The entries corresponding to these leaves form the *skeleton matrix* $\widehat{A}$; an entry a_{ij} is defined to be in the skeleton matrix $\widehat{A}$ of A if node j is a leaf of the ith row subtree. The nonzero patterns of the Cholesky factorization of the skeleton matrix of A and the original matrix A are identical. If node j is a leaf of the ith row

Case 1: t not a descendant of j Case 2: t a descendant of j

Figure 4.6. *Descendants in a postordered tree*

subtree, a_{ij} must be nonzero, but the converse is not true. For example, consider row 11 of the matrix A in Figure 4.2 and its corresponding row subtree $\mathcal{T}^{11}$ in Figure 4.4. The nonzero entries in row 11 of A are in columns 3, 5, 7, 8, 10, and 11, but only the first three are leaves of the 11th row subtree.

Suppose the matrix and the elimination tree are postordered. The first descendant of each node determines the leaves of the row subtrees, using the following *skeleton* function.

```
function skeleton
    maxfirst[0...n-1] = -1
    for j = 0 to n-1 do
        for each i > j for which a_ij ≠ 0
            if first[j] > maxfirst[i]
                node j is a leaf in the ith subtree
                maxfirst[i] = first[j]
```

The algorithm considers node j in all row subtrees i that contain node j, where j iterates from 0 to $n-1$. Let `first[j]` be the first descendant of node j in the elimination tree. Let `maxfirst[i]` be the largest `first[j]` seen so far for any nonzero a_{ij} in the ith subtree. If `first[j]` is less than or equal to `maxfirst[i]`, then node j must have a descendant $d < j$ in the ith row subtree, for which `first[d]=maxfirst[i]` will equal or exceed `first[j]`. Node j is thus not a leaf of the ith row subtree. If `first[j]` exceeds `maxfirst[i]`, then node j has no descendant in the ith row subtree, and node j is a leaf. The correctness of *skeleton* depends on Corollary 4.11 below.

Lemma 4.9. *Let $f_j \leq j$ denote the first descendant of j in a postordered tree. The descendants of j are all nodes $f_j, f_j + 1, \ldots, j-1, j$.*

Theorem 4.10. *Consider two nodes $t < j$ in a postordered tree. Then either* (1) *$f_t \leq t < f_j \leq j$ and t is not a descendant of j, or* (2) *$f_j \leq f_t \leq t < j$ and t is a descendant of j.*

Proof. The two cases of Theorem 4.10 are illustrated in Figure 4.6. A triangle represents the subtree rooted at a node j, and a small circle represents f_j. Case

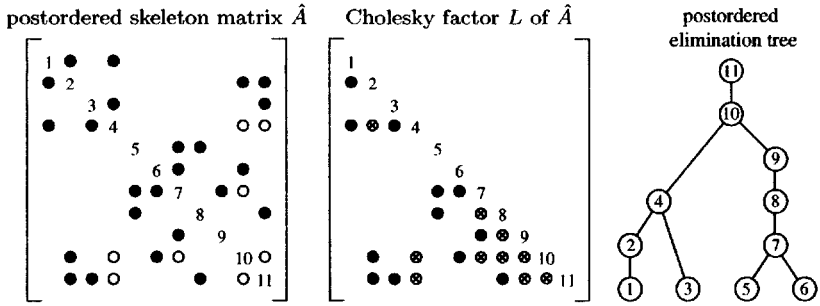


Figure 4.7. *Postordered skeleton matrix, its factor, and its elimination tree*

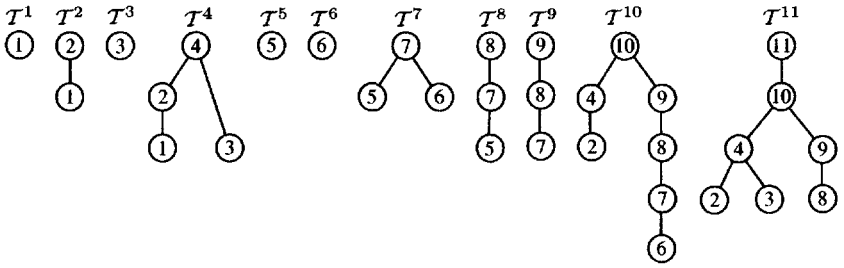


Figure 4.8. *Postordered row subtrees*

(1): Node t is not a descendant of j if and only if $t < f_j$, because of Lemma 4.9. Case (2): If t is a descendant of j, then f_t is also a descendant of j, and thus $f_j \leq f_t$. If $f_j \leq f_t$, then all nodes f_t through t must be descendants of j (Lemma 4.9). □

Corollary 4.11. *Consider a node j in a postordered tree and any set of nodes S where all nodes $s \in S$ are numbered less than j. Let t be the node in S with the largest first descendant f_t. Node j has a descendant in S if and only if $f_t \geq f_j$.*

Figure 4.7 shows the postordered skeleton matrix of A, denoted $\hat{A}$, its factor L, and its elimination tree (compare with Figure 4.5). Figure 4.8 shows the postordered row subtrees (compare with Figure 4.4). Entry a_{ij} (where $i > j$) is present in the skeleton matrix if and only if j is a leaf of the (postordered) ith subtree; they are shown as dark circles (the entry a_{ji} is also shown in the upper triangular part of $\hat{A}$). A white circle denotes an entry in A that is not in the skeleton matrix $\hat{A}$.

The leaves of the row subtree can be used to decompose the row subtree into a set of disjoint paths in a process called path decomposition. Consider the first (least numbered) leaf of a row subtree. The first disjoint path starts at this node and leads all the way to the root. Consider two consecutive leaves of a row subtree, $j_{\text{prev}} < j$. The next disjoint path starts at j and ends at the child of the least

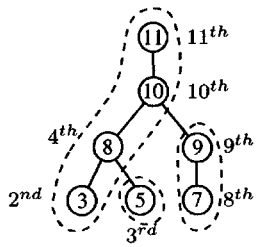


Figure 4.9. *Path decomposition of T^{11} from Figure 4.4*

common ancestor of j_{prev} and j. The *least common ancestor* of two nodes a and b is the least numbered node that is an ancestor of both a and b and is denoted as $q = lca(a, b)$. In T^{11}, shown in Figure 4.9, the first path is from node 3 (the 2nd node in the postordered tree) to the root node 11. The next path starts at node 5 (the 3rd node in the postordered tree) and terminates at node 5 (node 8 is the least common ancestor of the two leaves 3 and 5). The third and last path starts at node 7 and terminates at the child node 9 of the least common ancestor (node 10) of nodes 5 and 7. Figure 4.9 shows the path decomposition of T^{11} into these three disjoint paths. Each node is labeled with its corresponding column index in A and its postordering (node 8 is the 4th node in the postordered tree, for example).

Once the kth row subtree is decomposed into its disjoint paths, the kth row count is computed as the sum of the lengths of these paths. An efficient method for finding the least common ancestors of consecutive leaves j_{prev} and j of the row subtree is needed. Given the least common ancestor q of these two leaves, the length of the path from j to q can be added to the row count (excluding q itself). The lengths of the paths can be found by taking the difference of the starting and ending nodes of each path.

Theorem 4.12. *Assume that the elimination tree T is postordered. The least common ancestor of two nodes a and b, where $a < b$, can be found by traversing the path from a towards the root. The first node $q \geq b$ found along this path is the least common ancestor of a and b.*

The `rowcnt` function takes as input the matrix A, its elimination tree, and a postordering of the elimination tree. It uses a disjoint-set-union data structure to efficiently compute the least common ancestors of successive pairs of leaves of the row subtrees. Since it is not actually part of CSparse, it does not check any out-of-memory error conditions. Unlike `cs_etree`, the function uses the lower triangular part of A only and omits an option for computing the row counts of the Cholesky factor of A^TA. The `cs_leaf` function determines if j is a leaf of the ith row subtree, T^i. If it is, it computes the *lca* of j_{prev} (the previous leaf found in T^i) and node j.

To compute $q = lca(j_{\text{prev}}, j)$ efficiently in `cs_leaf`, an ancestor of each node is maintained, using a disjoint-set-union data structure. Initially, each node is in

its own set, and ancestor$[i] = i$ for all nodes i. If a node i is the root of a set, it is its own ancestor. For all other nodes i, traversing the ancestor tree and hitting a root q determines the representative (q) of the set containing node i.

```
int *rowcnt (cs *A, int *parent, int *post) /* return rowcount [0..n-1] */
{
    int i, j, k, len, s, p, jprev, q, n, sparent, jleaf, *Ap, *Ai, *maxfirst,
        *ancestor, *prevleaf, *w, *first, *level, *rowcount ;
    n = A->n ; Ap = A->p ; Ai = A->i ;                  /* get A */
    w = cs_malloc (5*n, sizeof (int)) ;                 /* get workspace */
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    level = w+4*n ;
    rowcount = cs_malloc (n, sizeof (int)) ;    /* allocate result */
    firstdesc (n, parent, post, first, level) ; /* find first and level */
    for (i = 0 ; i < n ; i++)
    {
        rowcount [i] = 1 ;      /* count the diagonal of L */
        prevleaf [i] = -1 ;     /* no previous leaf of the ith row subtree */
        maxfirst [i] = -1 ;     /* max first[j] for node j in ith subtree */
        ancestor [i] = i ;      /* every node is in its own set, by itself */
    }
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in the postordered etree */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf) ;
            if (jleaf) rowcount [i] += (level [j] - level [q]) ;
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    cs_free (w) ;
    return (rowcount) ;
}

int cs_leaf (int i, int j, const int *first, int *maxfirst, int *prevleaf,
    int *ancestor, int *jleaf)
{
    int q, s, sparent, jprev ;
    if (!first || !maxfirst || !prevleaf || !ancestor || !jleaf) return (-1) ;
    *jleaf = 0 ;
    if (i <= j || first [j] <= maxfirst [i]) return (-1) ;  /* j not a leaf */
    maxfirst [i] = first [j] ;      /* update max first[j] seen so far */
    jprev = prevleaf [i] ;          /* jprev = previous leaf of ith subtree */
    prevleaf [i] = j ;
    *jleaf = (jprev == -1) ? 1: 2 ; /* j is first or subsequent leaf */
    if (*jleaf == 1) return (i) ;   /* if 1st leaf, q = root of ith subtree */
    for (q = jprev ; q != ancestor [q] ; q = ancestor [q]) ;
    for (s = jprev ; s != q ; s = sparent)
    {
        sparent = ancestor [s] ;    /* path compression */
        ancestor [s] = q ;
    }
    return (q) ;                    /* q = least common ancester (jprev,j) */
}
```

Assuming the matrix is already postordered, rowcnt considers each node j

one at a time, where j iterates from 0 to $n-1$. For each node j, all row subtrees i that contain it are considered (all row indices i corresponding to nonzero entries a_{ij}, where $i > j$). Since j_{prev} and j are leaves of the same row subtree, they will have a least common ancestor q that is greater than j and which will be the representative of the set containing j_{prev}. Traversing from node j_{prev} towards the root determines node q. After this path is traversed, it is compressed to speed up any remaining path traversals. After all row subtrees containing node j are considered, it is merged into the set corresponding to its parent. Assuming the elimination tree is connected, no nodes 0 to j are now root nodes of any set. This ensures that traversing a path in the ancestor tree will find the least common ancestor for subsequent nodes j (see Theorem 4.12).

4.5 Column counts

Computing the number of nonzeros in each column of L can be done in nearly $O(|A|)$ time, similar to the row counts. A characterization of the nonzero pattern of each column of L is required. Let $\mathcal{A}_j$ denote the nonzero pattern of the jth column of the strictly lower triangular part of A, and let $\widehat{\mathcal{A}}_j$ denote entries in the same part of the skeleton matrix $\widehat{A}$. Let $\mathcal{L}_j$ denote the nonzero pattern of column j of L, and let $c_j = |\mathcal{L}_j|$ denote the number of entries in that column (including the diagonal).

Theorem 4.13 and its corollary show how to compute the nonzero pattern of L in a left-looking manner. It states that the nonzero pattern of `L(:,j)` is the union of the nonzero patterns of the children of `j` in the elimination tree and the nonzero pattern of `A(:,j)`.

Theorem 4.13 (George and Liu [89]). *If $\mathcal{L}_j$ denotes the nonzero pattern of the jth column of L, and $\mathcal{A}_j$ denotes the nonzero pattern of the strictly lower triangular part of the jth column of A, then*

$$\mathcal{L}_j = \mathcal{A}_j \cup \{j\} \cup \left(\bigcup_{j=parent(s)} \mathcal{L}_s \setminus \{s\} \right). \tag{4.3}$$

Proof. Refer to Figure 4.10. Consider any descendant d of j and any row $i \in \mathcal{L}_d$. That is, $l_{id} \neq 0$ and the path $d \rightsquigarrow j$ exists in $\mathcal{T}$. Theorem 4.5 states that the nonzero pattern of row i is given by the ith row subtree, $\mathcal{T}^i$. Thus, the path $d \rightsquigarrow s \rightarrow j$ exists in $\mathcal{T}^i$ for some s, and row index i is present in $\mathcal{L}_j$ and in $\mathcal{L}_s$ of the child s of j (also true if $d = s$). To construct the nonzero pattern of column j ($\mathcal{L}_j$), only $\mathcal{L}_s$ of the children s of j need to be considered. Likewise, there can be no $i \in \mathcal{L}_j$ not accounted for by (4.3). If $i \in \mathcal{L}_j$, then j must be in $\mathcal{T}^i$. Either j is a leaf (and thus $i \in \widehat{\mathcal{A}}_j \subseteq \mathcal{A}_j$), or it is not a leaf (and thus j has a child s in $\mathcal{T}^i$, and $i \in \mathcal{L}_s$). □

Corollary 4.14 (Schreiber [181]). *The nonzero pattern of the jth column of L is a subset of the path $j \rightsquigarrow r$ from j to the root of the elimination tree $\mathcal{T}$.*

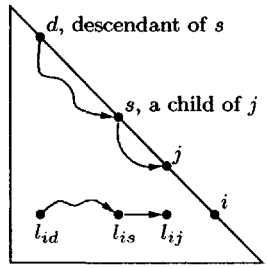


Figure 4.10. *The nonzero pattern of* `L(:,j)` *is the union of its children*

Computing the column counts $c_j = |\mathcal{L}_j|$ can be done in $O(|L|)$ time by using Theorem 4.13 or by traversing each row subtree explicitly and keeping track of how many times j appears in any row subtree. Using the least common ancestors of successive pairs of leaves of the row subtree reduces the time to nearly $O(|A|)$.

Consider the case where j is a leaf of the elimination tree $\mathcal{T}$. The column count c_j is simply $c_j = |\mathcal{A}_j| + 1 = |\widehat{\mathcal{A}}_j| + 1$, since j has no children and each entry in column j of A is also in the skeleton matrix $\widehat{A}$. Consider the case where j is not a leaf of the elimination tree. Theorem 4.13 states that $\mathcal{L}_j$ is the union of $\mathcal{A}_j \cup \{j\}$ and the nonzero patterns of its children, $\mathcal{L}_s \setminus \{s\}$. Since $\widehat{\mathcal{A}}_j$ is disjoint from each child $\mathcal{L}_s$, and since $s \in \mathcal{L}_s$,

$$c_j = |\widehat{\mathcal{A}}_j| + \left| \bigcup_{j=\text{parent}(s)} \mathcal{L}_s \setminus \{s\} \right| = |\widehat{\mathcal{A}}_j| - e_j + \left| \bigcup_{j=\text{parent}(s)} \mathcal{L}_s \right|,$$

where e_j is the number of children of j in the elimination tree $\mathcal{T}$. Suppose there was an efficient method for computing the *overlap* between the nonzero patterns of the children of j. That is, let o_j be the term required so that

$$c_j = |\widehat{\mathcal{A}}_j| - e_j - o_j + \sum_{j=\text{parent}(s)} c_s. \tag{4.4}$$

As an example, consider column $j = 4$ of Figure 4.7. Its two children are $\mathcal{L}_2 = \{2, 4, 10, 11\}$ and $\mathcal{L}_3 = \{3, 4, 11\}$. The skeleton matrix is empty in this column, so

$$\mathcal{L}_4 = \widehat{\mathcal{A}}_4 \cup \mathcal{L}_2 \setminus \{2\} \cup \mathcal{L}_3 \setminus \{3\} = \emptyset \cup \{4, 10, 11\} \cup \{4, 11\} = \{4, 10, 11\}.$$

The overlap $o_4 = 2$, because rows 4 and 11 each appear twice in the children. The number of children is $e_4 = 2$. Thus, $c_4 = 0 - 2 - 2 + 4 + 3 = 3$. If the overlap and the skeleton matrix are known for each column j, (4.4) can be used to compute the column counts. Note that the diagonal entry $j = 4$ does not appear in $\widehat{\mathcal{A}}_4$. Instead, $4 \in \mathcal{L}_4$ appears in each child, and the overlap accounts for all but one of these entries.

The overlap can be computed by considering the row subtrees. There are three cases to consider. Recall that the ith row subtree $\mathcal{T}^i$ determines the nonzero pattern of the ith row of L. Node j is present in the ith subtree if and only if $i \in \mathcal{L}_j$.

1. If $j \notin \mathcal{T}^i$, then $i \notin \mathcal{L}_j$ and row i does not contribute to the overlap o_j.
2. If j is a leaf of $\mathcal{T}^i$, then by definition a_{ij} is in the skeleton matrix. Row i does not contribute to the overlap o_j, because it appears in none of the children of j. Row i contributes exactly one to c_j, since $i \in \widehat{\mathcal{A}}_j$.
3. If j is not a leaf of $\mathcal{T}^i$, let d_{ij} denote the number of children of j that are in $\mathcal{T}^i$. These children are a subset of the children of j in the elimination tree $\mathcal{T}$. Row i is present in the nonzero patterns of each of these d_{ij} children. Thus, row i contributes $d_{ij} - 1$ to the overlap o_j. If j has just one child, row i appears only in that one child and there is no overlap.

Case 3, above, is the only place where the overlap needs to be considered: nodes in the row subtrees with more than one child. Refer to Figure 4.8, and consider column 4. Node 4 has two children in subtrees $\mathcal{T}^4$ and $\mathcal{T}^{11}$.

The key observation is to see that if j has d children in a row subtree, then it will be the least common ancestor of exactly $d - 1$ successive pairs of leaves in that row subtree. For example, node 4 is the least common ancestor of leaves 1 and 3 in $\mathcal{T}^4$ and the least common ancestor of leaves 2 and 3 in $\mathcal{T}^{11}$. Thus, each time column j becomes a least common ancestor of any successive pair of leaves in the row count algorithm, the overlap o_j can be incremented.

These modifications can be folded into a single correction term, Δ_j. If j is a leaf of the elimination tree, $c_j = \Delta_j = |\widehat{\mathcal{A}}_j| + 1$. Otherwise, $\Delta_j = |\widehat{\mathcal{A}}_j| - e_j - o_j$. Equation (4.4) becomes

$$c_j = \Delta_j + \sum_{j=\text{parent}(s)} c_s \tag{4.5}$$

for both cases. The term Δ_j is initialized as 1 if j is a leaf or 0 otherwise. It is incremented once for each entry a_{ij} in the skeleton matrix, decremented once for each child of j, and decremented once each time j becomes the least common ancestor of a successive pair of leaves in a row subtree. Once Δ_j is computed, (4.5) can be used to compute the column counts.

The column count algorithm `cs_counts` is given below. It takes as input the matrix `A`, the elimination tree (`parent`), and its postordering (`post`). It also takes a parameter `ata` that determines whether the Cholesky factor of A or A^TA is to be computed. If `ata` is nonzero, the column counts of the Cholesky factor of A^TA are computed. To compute the column counts for the Cholesky factorization of A, `ata` is zero, the `init_ata` function is not used, and the `for J` loop iterates just once with `J = j`.

Unlike `rowcnt`, the `cs_counts` function uses the upper triangular part of `A` to compute the column counts; it transposes `A` internally. None of the inputs are modified. The function returns an array of size `n` of the column counts or `NULL` on error. Compare `cs_counts` with `rowcnt`. It is quite simple to combine these two functions into a single function that computes both the row and column counts. They are split into two functions here, to introduce the concepts more gradually and because CSparse does not require the row counts.

```
#define HEAD(k,j) (ata ? head [k] : j)
#define NEXT(J)   (ata ? next [J] : -1)
```

```
static void init_ata (cs *AT, const int *post, int *w, int **head, int **next)
{
    int i, k, p, m = AT->n, n = AT->m, *ATp = AT->p, *ATi = AT->i ;
    *head = w+4*n, *next = w+5*n+1 ;
    for (k = 0 ; k < n ; k++) w [post [k]] = k ;    /* invert post */
    for (i = 0 ; i < m ; i++)
    {
        for (k = n, p = ATp[i] ; p < ATp[i+1] ; p++) k = CS_MIN (k, w [ATi[p]]);
        (*next) [i] = (*head) [k] ;     /* place row i in linked list k */
        (*head) [k] = i ;
    }
}
int *cs_counts (const cs *A, const int *parent, const int *post, int ata)
{
    int i, j, k, n, m, J, s, p, q, jleaf, *ATp, *ATi, *maxfirst, *prevleaf,
        *ancestor, *head = NULL, *next = NULL, *colcount, *w, *first, *delta ;
    cs *AT ;
    if (!CS_CSC (A) || !parent || !post) return (NULL) ;    /* check inputs */
    m = A->m ; n = A->n ;
    s = 4*n + (ata ? (n+m+1) : 0) ;
    delta = colcount = cs_malloc (n, sizeof (int)) ;    /* allocate result */
    w = cs_malloc (s, sizeof (int)) ;                   /* get workspace */
    AT = cs_transpose (A, 0) ;                          /* AT = A' */
    if (!AT || !colcount || !w) return (cs_idone (colcount, AT, w, 0)) ;
    ancestor = w ; maxfirst = w+n ; prevleaf = w+2*n ; first = w+3*n ;
    for (k = 0 ; k < s ; k++) w [k] = -1 ;      /* clear workspace w [0..s-1] */
    for (k = 0 ; k < n ; k++)                   /* find first [j] */
    {
        j = post [k] ;
        delta [j] = (first [j] == -1) ? 1 : 0 ;  /* delta[j]=1 if j is a leaf */
        for ( ; j != -1 && first [j] == -1 ; j = parent [j]) first [j] = k ;
    }
    ATp = AT->p ; ATi = AT->i ;
    if (ata) init_ata (AT, post, w, &head, &next) ;
    for (i = 0 ; i < n ; i++) ancestor [i] = i ; /* each node in its own set */
    for (k = 0 ; k < n ; k++)
    {
        j = post [k] ;          /* j is the kth node in postordered etree */
        if (parent [j] != -1) delta [parent [j]]-- ;    /* j is not a root */
        for (J = HEAD (k,j) ; J != -1 ; J = NEXT (J))   /* J=j for LL'=A case */
        {
            for (p = ATp [J] ; p < ATp [J+1] ; p++)
            {
                i = ATi [p] ;
                q = cs_leaf (i, j, first, maxfirst, prevleaf, ancestor, &jleaf);
                if (jleaf >= 1) delta [j]++ ;   /* A(i,j) is in skeleton */
                if (jleaf == 2) delta [q]-- ;   /* account for overlap in q */
            }
        }
        if (parent [j] != -1) ancestor [j] = parent [j] ;
    }
    for (j = 0 ; j < n ; j++)           /* sum up delta's of each child */
    {
        if (parent [j] != -1) colcount [parent [j]] += colcount [j] ;
    }
    return (cs_idone (colcount, AT, w, 1)) ;    /* success: free workspace */
}
```

The column count algorithm presented here can also be used for the QR and LU factorization of a square or rectangular matrix A. For QR factorization, the nonzero pattern of R is identical to L^T in the Cholesky factorization $LL^T = A^TA$ (assuming no numerical cancellation and mild assumptions discussed in Chapter 5). This same matrix R provides an upper bound on the nonzero pattern of U for an LU factorization of A. Details are presented in Chapters 5 and 6.

One method for finding the row counts of R is to compute A^TA explicitly and then find the column counts of its Cholesky factorization. This can be expensive both in time and memory. A better method taking nearly $O(|A|+n+m)$ time is to find a symmetric matrix with fewer nonzeros than A^TA but whose Cholesky factor has the same nonzero pattern as A^TA. One matrix that satisfies this property is the *star* matrix. It has $O(|A|)$ entries and can be found in $O(|A|+n+m)$ time. Each row of A defines a clique in the graph of A^TA. Let $\mathcal{A}_i$ denote the nonzero pattern of the ith row of A. Consider the lowest numbered column index k of nonzeros in row i of A; that is, $k = \min \mathcal{A}_i$. The clique in A^TA corresponding to row i of A is the set of entries $\mathcal{A}_i \times \mathcal{A}_i$. Consider an entry $(A^TA)_{ab}$, where $a \in \mathcal{A}_i$ and $b \in \mathcal{A}_i$. If both $a > k$ and $b > k$, then this entry is not needed. It can be removed without changing the nonzero pattern of the Cholesky factor of A^TA. Without loss of generality, assume $a > b$. The entries $(A^TA)_{bk}$ and $(A^TA)_{ak}$ will both be nonzero. Theorems 4.2 and 4.3 imply that l_{ab} is nonzero, regardless of whether or not $(A^TA)_{ab}$ is nonzero.

The nonzero pattern of the kth row and column of the star matrix is thus the union of all $\mathcal{A}_i$, where $k = \min \mathcal{A}_i$. Fortunately, this union need not be formed explicitly, since the row and column count algorithms (and specifically the skeleton function) implicitly ignore duplicate entries. To traverse all entries in the kth column of the star matrix, all rows $\mathcal{A}_i$, where $k = \min \mathcal{A}_i$, are considered. In `cs_counts`, this is implemented by placing each row in a linked list corresponding to its least numbered nonzero column index (using a `head` array of size `n+1` and a `next` array of size `n`).

In MATLAB, `c = symbfact(A)` uses the same algorithms given here, returning the column counts of the Cholesky factorization of A. The column counts of the Cholesky factorization of A^TA are given by `c = symbfact(A,'col')`. Both forms use the CHOLMOD function `cholmod_rowcolcounts`.

4.6 Symbolic analysis

The *symbolic analysis* of a matrix is a precursor to its numerical factorization. It includes computations that typically depend only on the nonzero pattern, not the numerical values. This allows the numerical factorization to be repeated for a sequence of matrices with identical nonzero pattern (a situation that often arises when solving nonlinear equations). *Symbolic factorization* includes the computation of an explicit representation of the nonzero pattern of the factorization; some sparse Cholesky algorithms require this. Permuting a matrix has a large impact on the amount of fill-in. Typically a fill-reducing permutation P is found so that the factorization of PAP^T is sparser than that of A. The `cs_schol` function performs

the symbolic analysis for the up-looking sparse Cholesky factorization presented in the next section.

```
typedef struct cs_symbolic  /* symbolic Cholesky, LU, or QR analysis */
{
    int *pinv ;     /* inverse row perm. for QR, fill red. perm for Chol */
    int *q ;        /* fill-reducing column permutation for LU and QR */
    int *parent ;   /* elimination tree for Cholesky and QR */
    int *cp ;       /* column pointers for Cholesky, row counts for QR */
    int *leftmost ; /* leftmost[i] = min(find(A(i,:))), for QR */
    int m2 ;        /* # of rows for QR, after adding fictitious rows */
    double lnz ;    /* # entries in L for LU or Cholesky; in V for QR */
    double unz ;    /* # entries in U for LU; in R for QR */
} css ;

css *cs_schol (int order, const cs *A)
{
    int n, *c, *post, *P ;
    cs *C ;
    css *S ;
    if (!CS_CSC (A)) return (NULL) ;        /* check inputs */
    n = A->n ;
    S = cs_calloc (1, sizeof (css)) ;       /* allocate result S */
    if (!S) return (NULL) ;                 /* out of memory */
    P = cs_amd (order, A) ;                 /* P = amd(A+A'), or natural */
    S->pinv = cs_pinv (P, n) ;              /* find inverse permutation */
    cs_free (P) ;
    if (order && !S->pinv) return (cs_sfree (S)) ;
    C = cs_symperm (A, S->pinv, 0) ;        /* C = spones(triu(A(P,P))) */
    S->parent = cs_etree (C, 0) ;           /* find etree of C */
    post = cs_post (S->parent, n) ;         /* postorder the etree */
    c = cs_counts (C, S->parent, post, 0) ; /* find column counts of chol(C) */
    cs_free (post) ;
    cs_spfree (C) ;
    S->cp = cs_malloc (n+1, sizeof (int)) ; /* allocate result S->cp */
    S->unz = S->lnz = cs_cumsum (S->cp, c, n) ; /* find column pointers for L */
    cs_free (c) ;
    return ((S->lnz >= 0) ? S : cs_sfree (S)) ;
}

css *cs_sfree (css *S)
{
    if (!S) return (NULL) ;     /* do nothing if S already NULL */
    cs_free (S->pinv) ;
    cs_free (S->q) ;
    cs_free (S->parent) ;
    cs_free (S->cp) ;
    cs_free (S->leftmost) ;
    return (cs_free (S)) ;      /* free the css struct and return NULL */
}
```

`cs_schol` does not compute the nonzero pattern of L. First, a `css` structure `S` is allocated. For a sparse Cholesky factorization, `S->pinv` is the fill-reducing permutation (stored as an inverse permutation vector), `S->parent` is the elimination tree, `S->cp` is the column pointer of `L`, and `S->lnz` $= |L|$. This symbolic structure will also be used for sparse LU and QR factorizations. Next, `p` is found via a minimum

degree ordering of $A + A^T$ (see Chapter 7) if `order` is 1. No permutation is used if `order` is 0 (`p` is `NULL` to denote the identity permutation). This vector is inverted to obtain `pinv`. The upper triangular part of `A` is permuted to obtain `C` (`C=A(p,p)` in MATLAB notation). The elimination tree of `C` is found and postordered, and the column counts of L are found. A cumulative sum of the column counts gives both `S->cp`, the column pointers of `L`, and `S->lnz`= $|L|$. `cs_free` frees a symbolic analysis.

4.7 Up-looking Cholesky

A great deal of theory and algorithms have been covered to reach this point: a simple, concise sparse Cholesky factorization algorithm. The `cs_chol` function presented below implements the up-looking algorithm described in the preface to this chapter. To clarify the discussion, the bold paragraph headings are tied to comments in the code starting with `---`. This style is also used elsewhere in this book.

`cs_chol` computes the Cholesky factorization of `C=A(p,p)`. The input matrix `A` is assumed to be symmetric; only the upper triangular part is accessed. The function requires the symbolic analysis from `cs_schol`. It returns a numeric factorization, consisting of just the `N->L` matrix. First, workspace is allocated, the contents of the symbolic analysis are retrieved, `L` is allocated, and `A` is permuted (`E` is `A(p,p)` if `A` is permuted, or `NULL` otherwise, so that `A(p,p)` can be freed by `cs_ndone`).

Nonzero pattern of `L(k,:)`: The kth iteration of the `for` loop computes the kth row of `L`. It starts by computing the nonzero pattern of this kth row, placing the result in `s[top]` through `s[n-1]` in topological order. The entry `x[k]` is cleared to ensure that `x[0]` through `x[k]` are all zero. The kth column of `C` is scattered into the dense vector `x`. The diagonal entry of `C` is retrieved from `x`, and `x[k]` is cleared to prepare for iteration `k+1` of the outer loop.

Triangular solve: The numerical solution to $L_{0\ldots k-1,0\ldots k-1}x = C_{0\ldots k-1,k}$ is found, which defines the numerical values in the row `k` of `L`. As each entry l_{ki} is computed, the workspace `x[i]` is cleared, the numerical value l_{ki} and row index `k` are placed in column `i` of `L`, and the dot product $L_{k,0\ldots k-1}L^T_{k,0\ldots k-1}$ is added to `d`.

Compute `L(k,k)`: The kth diagonal entry of `L` is computed. The function frees `N` and returns if the matrix `A` is not positive definite. When the factorization finishes successfully, the workspace is freed and `N` is returned.

The time taken by `cs_chol` to compute the Cholesky factorization of a symmetric positive definite matrix is $O(f)$, the number of floating-point operations performed; $f = \sum |\mathtt{L}(:,\mathtt{k})|^2$. To perform a complete sparse Cholesky factorization, including a fill-reducing preordering and symbolic analysis, use `S=cs_schol(order,A)` followed by `N=cs_chol(A,S)`, where `order` is 0 to use the natural ordering or 1 to use a fill-reducing minimum degree ordering of $A + A^T$.

In MATLAB, the sparse Cholesky factorization `R=chol(A)` returns `R=L'`. If `A` is very sparse, it uses an up-looking algorithm (`cholmod_rowfac`, much like [29]). Otherwise, it uses a left-looking supernodal factorization (`cholmod_super_numeric`), discussed in the next section.

```
csn *cs_chol (const cs *A, const css *S)
```

```
{
    double d, lki, *Lx, *x, *Cx ;
    int top, i, p, k, n, *Li, *Lp, *cp, *pinv, *s, *c, *parent, *Cp, *Ci ;
    cs *L, *C, *E ;
    csn *N ;
    if (!CS_CSC (A) || !S || !S->cp || !S->parent) return (NULL) ;
    n = A->n ;
    N = cs_calloc (1, sizeof (csn)) ;       /* allocate result */
    c = cs_malloc (2*n, sizeof (int)) ;     /* get int workspace */
    x = cs_malloc (n, sizeof (double)) ;    /* get double workspace */
    cp = S->cp ; pinv = S->pinv ; parent = S->parent ;
    C = pinv ? cs_symperm (A, pinv, 1) : ((cs *) A) ;
    E = pinv ? C : NULL ;           /* E is alias for A, or a copy E=A(p,p) */
    if (!N || !c || !x || !C) return (cs_ndone (N, E, c, x, 0)) ;
    s = c + n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    N->L = L = cs_spalloc (n, n, cp [n], 1, 0) ;    /* allocate result */
    if (!L) return (cs_ndone (N, E, c, x, 0)) ;
    Lp = L->p ; Li = L->i ; Lx = L->x ;
    for (k = 0 ; k < n ; k++) Lp [k] = c [k] = cp [k] ;
    for (k = 0 ; k < n ; k++)       /* compute L(:,k) for L*L' = C */
    {
        /* --- Nonzero pattern of L(k,:) ------------------------------------ */
        top = cs_ereach (C, k, parent, s, c) ;      /* find pattern of L(k,:) */
        x [k] = 0 ;                                 /* x (0:k) is now zero */
        for (p = Cp [k] ; p < Cp [k+1] ; p++)       /* x = full(triu(C(:,k))) */
        {
            if (Ci [p] <= k) x [Ci [p]] = Cx [p] ;
        }
        d = x [k] ;                     /* d = C(k,k) */
        x [k] = 0 ;                     /* clear x for k+1st iteration */
        /* --- Triangular solve --------------------------------------------- */
        for ( ; top < n ; top++)    /* solve L(0:k-1,0:k-1) * x = C(:,k) */
        {
            i = s [top] ;               /* s [top..n-1] is pattern of L(k,:) */
            lki = x [i] / Lx [Lp [i]] ; /* L(k,i) = x (i) / L(i,i) */
            x [i] = 0 ;                 /* clear x for k+1st iteration */
            for (p = Lp [i] + 1 ; p < c [i] ; p++)
            {
                x [Li [p]] -= Lx [p] * lki ;
            }
            d -= lki * lki ;            /* d = d - L(k,i)*L(k,i) */
            p = c [i]++ ;
            Li [p] = k ;                /* store L(k,i) in column i */
            Lx [p] = lki ;
        }
        /* --- Compute L(k,k) ----------------------------------------------- */
        if (d <= 0) return (cs_ndone (N, E, c, x, 0)) ; /* not pos def */
        p = c [k]++ ;
        Li [p] = k ;                /* store L(k,k) = sqrt (d) in column k */
        Lx [p] = sqrt (d) ;
    }
    Lp [n] = cp [n] ;               /* finalize L */
    return (cs_ndone (N, E, c, x, 1)) ; /* success: free E,s,x; return N */
}
```

The csn structure contains a numeric Cholesky, LU, or QR factorization.

For a sparse Cholesky factorization, only N->L is used. cs_nfree frees a numeric factorization. cs_ndone frees any workspace and returns a numeric factorization.

```
typedef struct cs_numeric   /* numeric Cholesky, LU, or QR factorization */
{
    cs *L ;         /* L for LU and Cholesky, V for QR */
    cs *U ;         /* U for LU, R for QR, not used for Cholesky */
    int *pinv ;     /* partial pivoting for LU */
    double *B ;     /* beta [0..n-1] for QR */
} csn ;

csn *cs_nfree (csn *N)
{
    if (!N) return (NULL) ;     /* do nothing if N already NULL */
    cs_spfree (N->L) ;
    cs_spfree (N->U) ;
    cs_free (N->pinv) ;
    cs_free (N->B) ;
    return (cs_free (N)) ;      /* free the csn struct and return NULL */
}

csn *cs_ndone (csn *N, cs *C, void *w, void *x, int ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    cs_free (x) ;
    return (ok ? N : cs_nfree (N)) ;    /* return result if OK, else free it */
}
```

4.8 Left-looking and supernodal Cholesky

The *left-looking* Cholesky factorization algorithm is more commonly used than the up-looking algorithm. The MATLAB function chol_left implements this algorithm.

```
function L = chol_left (A)
n = size (A,1) ;
L = zeros (n) ;
for k = 1:n
    L (k,k) = sqrt (A (k,k) - L (k,1:k-1) * L (k,1:k-1)') ;
    L (k+1:n,k) = (A (k+1:n,k) - L (k+1:n,1:k-1) * L (k,1:k-1)') / L (k,k) ;
end
```

It computes L one column at a time and can be derived from the expression

$$\begin{bmatrix} L_{11} & & \\ l_{12}^T & l_{22} & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} & L_{31}^T \\ & l_{22} & l_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{31}^T \\ a_{12}^T & a_{22} & a_{32}^T \\ A_{31} & a_{32} & A_{33} \end{bmatrix}, \tag{4.6}$$

where the middle row and column of each matrix are the kth row and column of L, L^T, and A, respectively. If the first $k-1$ columns of L are known, $l_{22} = \sqrt{a_{22} - l_{12}^T l_{12}}$ can be computed first, followed by $l_{32} = (a_{32} - L_{31} l_{12})/l_{22}$. For the sparse case, an amplified version is given below.

```
function L = chol_left (A)
n = size (A,1) ;
L = sparse (n,n) ;
a = sparse (n,1) ;
for k = 1:n
    a (k:n) = A (k:n,k) ;
    for j = find (L (k,:))
        a (k:n) = a (k:n) - L (k:n,j) * L (k,j) ;
    end
    L (k,k) = sqrt (a (k)) ;
    L (k+1:n,k) = a (k+1:n) / L (k,k) ;
end
```

This method requires the nonzero pattern of the columns of L to be computed (in sorted order) prior to numerical factorization. In contrast, the up-looking method requires only the cumulative sum of the column counts (Lp) prior to numerical factorization and computes both the pattern (in sorted order) and numerical values in a single pass. Computing the pattern of L can be done in $O(|L|)$ time, using `cs_ereach` (see Problem 4.10).

The left-looking numerical factorization needs access to row k of L. The nonzero pattern of `L(k,:)` is given by the kth row subtree (4.2), T^k. This is enough information for the up-looking method, but the left-looking method also needs access to the numerical values of `L(k,:)` and the submatrix `L(k:n,1:k-1)`. To access the numerical values of the kth row, an array c of size n is maintained in the same way as the c array in the up-looking algorithm `cs_chol`. The row indices and numerical values in `L(k:n,j)` are given by `Li[c[j] ... Lp[j+1]-1]` and `Lx[c[j] ... Lp[j+1]-1]`, respectively. A left-looking sparse Cholesky factorization is left as an exercise (see Problem 4.11).

The left-looking algorithm forms the basis for the *supernodal* method. The `chol_super` function is a working prototype for a supernodal left-looking Cholesky factorization.

```
function L = chol_super (A,s)
n = size (A) ;
L = zeros (n) ;
ss = cumsum ([1 s]) ;
for j = 1:length (s)
    k1 = ss (j) ;
    k2 = ss (j+1) ;
    k = k1:(k2-1) ;
    L (k,k) = chol (A (k,k) - L (k,1:k1-1) * L (k,1:k1-1)')' ;
    L (k2:n,k) = (A (k2:n,k) - L (k2:n,1:k1-1) * L (k,1:k1-1)') / L (k,k)' ;
end
```

Consider (4.6), and let the middle row and column of the three matrices represent a block of $s_j \geq 1$ rows and columns. This block of columns is selected so that the nonzero patterns of these s_j columns are all identical, except for the diagonal block L_{22}, which is dense. In MATLAB notation, s is an integer vector where `all(s>0)` is true, and `sum(s)=n`. The `j`th supernode consists of `s(j)` columns of L which can be stored as a dense matrix of dimension $|\mathcal{L}_f|$ by s_j, where f is the column of L represented as the leftmost column in the jth supernode. `chol_super` relies on four key operations, all of which can exploit dense matrix kernels:

1. A symmetric update, `A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)'`. In the sparse case, `A(k,k)` is a dense matrix. `L(k,1:k1-1)` represents the rows in a subset of the descendants of the `j`th supernode. The update from each descendant can be done with a single dense matrix multiplication.

2. A dense Cholesky factorization, `chol`.

3. A sparse matrix product, `A(k2:n,k)-L(k2:n,1:k1-1)*L(k,1:k1-1)'`, where the two `L` terms come from the descendants of the `j`th supernode.

4. A dense triangular solve `(...)/L(k,k)'` using the `k`th diagonal block of `L`.

A supernodal Cholesky factorization based on dense matrix kernels (the BLAS) can achieve a substantial fraction of a computer's theoretical peak performance. MATLAB uses this method (`cholmod_super_numeric`) for a sparse symmetric positive definite A, except when A is very sparse, in which case it uses the up-looking algorithm described in Section 4.7.

4.9 Right-looking and multifrontal Cholesky

The *right-looking* Cholesky factorization is based on the following matrix expression, where l_{11} is a scalar:

$$\begin{bmatrix} l_{11} & \\ l_{21} & L_{22} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21}^T \\ & L_{22}^T \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21}^T \\ a_{21} & A_{22} \end{bmatrix}. \tag{4.7}$$

The first equation, $l_{11}^2 = a_{11}$, is solved for l_{11}, followed by $l_{21} = a_{21}/l_{11}$. Next, the Cholesky factorization $L_{22}L_{22}^T = A_{22} - l_{21}l_{21}^T$ is computed. The `chol_right` function is the MATLAB expression of this algorithm.

```
function L = chol_right (A)
n = size (A) ;
L = zeros (n) ;
for k = 1:n
    L (k,k) = sqrt (A (k,k)) ;
    L (k+1:n,k) = A (k+1:n,k) / L (k,k) ;
    A (k+1:n,k+1:n) = A (k+1:n,k+1:n) - L (k+1:n,k) * L (k+1:n,k)' ;
end
```

It forms the basis for the *multifrontal* method, which is similar to `chol_right`, except that the summation of the outer product $l_{21}l_{21}^T$ is postponed. Only a brief overview of the multifrontal method is given here. See Section 6.3 for more details. Just as in the supernodal method, the columns of L are grouped together; each group is represented by a dense *frontal matrix*. Let $\mathcal{L}_f$ be the nonzero pattern of the first column in a frontal matrix. The frontal matrix has dimension $|\mathcal{L}_f|$-by-$|\mathcal{L}_f|$. Within this frontal matrix, $k \geq 1$ steps of factorization are computed, and a rank-k outer product is computed. These steps can use the dense matrix BLAS, and thus they too can obtain very high performance.

Unlike `chol_right`, the outer product computed in the frontal matrix is not immediately added into the sparse matrix A. Let column e be the last pivot column of L represented by a frontal matrix (e = `k2-1` in `chol_super`). Its contribution is held in the frontal matrix until its parent is factorized. Its parent is that frontal matrix whose first column is the parent of e in the elimination tree of L. When a frontal matrix is factorized, the contribution blocks of its children are first added together.

MATLAB does not use a multifrontal sparse Cholesky method. It does use the multifrontal method for its sparse LU factorization (see Section 6.3).

4.10 Modifying a Cholesky factorization

Given the sparse Cholesky factorization $LL^T = A$, some applications require the factorization of A after a low-rank change, $A \pm WW^T$, where W is n-by-k with $k \ll n$. Computing the factorization $\overline{L}\overline{L}^T = A + WW^T$ is a rank-k *update*, and computing $\overline{L}\overline{L}^T = A - WW^T$ is a rank-k *downdate*. If A is positive definite, $A + WW^T$ is always positive definite, but $A - WW^T$ may not be.

Only the rank-1 case is considered here. There are many methods for computing $\overline{L}$ from L and w. The `chol_update` function below performs a rank-1 update and is used as the basis for the sparse rank-1 update. For details of its derivation, see the references discussed in Section 4.11. The function returns a modified $\overline{L}$ that is the Cholesky factor of the matrix `L*L'+w*w'`. It also computes one additional result, `w=L\w`, which reveals a key feature exploited in the sparse case.

```
function [L, w] = chol_update (L, w)
beta = 1 ;
n = size (L,1) ;
for j = 1:n
    alpha = w (j) / L (j,j) ;
    beta2 = sqrt (beta^2 + alpha^2) ;
    gamma = alpha / (beta2 * beta) ;
    delta = beta / beta2 ;
    L (j,j) = delta * L (j,j) + gamma * w (j) ;
    w (j) = alpha ;
    beta = beta2 ;
    if (j == n) return, end
    w1 = w (j+1:n) ;
    w (j+1:n) = w (j+1:n) - alpha * L (j+1:n,j) ;
    L (j+1:n,j) = delta * L (j+1:n,j) + gamma * w1 ;
end
```

Consider the sparse case. A key observation is to note that the columns of L that are modified correspond to the nonzero pattern of the solution to the triangular system $Lx = w$. At the `j`th step, the variable `alpha` is equal to x_j. This can be seen by removing everything from the algorithm except the modifications to `w`; all that is left is just a lower triangular solve. If `alpha` is zero, `beta2` and `beta` are identical, `gamma` is zero, and `delta` is one. No change is made to the `j`th column of `L` in this case. Thus, the `j`th step can be skipped if $x_j = 0$.

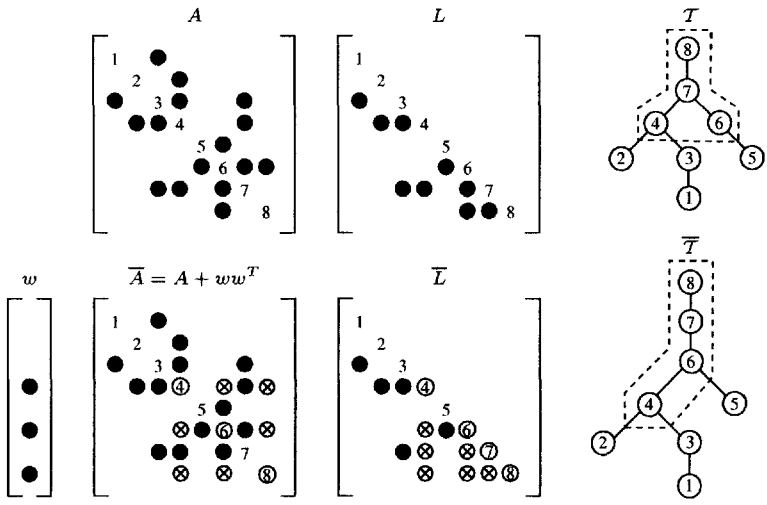


Figure 4.11. *Rank-1 update*

Let $\mathcal{X}$ be the nonzero pattern of the solution to $Lx = w$. Let $f = \min \mathcal{W}$, where $\mathcal{W} = \{i \,|\, w_i \neq 0\}$ is the set of row indices of nonzero entries in w. Applying Theorem 4.5 to the system $Lx = w$ reveals that $\mathcal{X} = \text{Reach}_T(\mathcal{W})$. That is, $\mathcal{X}$ is a set of paths in the elimination tree, starting at nodes $i \in \mathcal{W}$ and walking up the tree to the root.

Assume that numerical cancellation is ignored. If the nonzero pattern of L changes as a result of an update or downdate, the set $\mathcal{X}$ becomes a single path in the elimination tree $\overline{T}$ of $\overline{L}$. This is because the first nontrivial step $f = \min \mathcal{W}$ adds the nonzero pattern of $\mathcal{W}$ to the nonzero pattern of column f of L, and thus $\mathcal{W} \subseteq \overline{\mathcal{L}}_f$. Corollary 4.14 states that $\overline{\mathcal{L}}_f$ is a subset of the path from f to the root r in the elimination tree of $\overline{L}$. Thus, $\mathcal{W}$ is a subset of this path $f \rightsquigarrow r$, and computing a rank-1 update requires the traversal of a single path $f \rightsquigarrow r$ in the elimination tree $\overline{T}$ of $\overline{\mathcal{L}}$. An example rank-1 update is shown in Figure 4.11, where $\mathcal{W} = \{4, 6, 8\}$. Entries in $\overline{A} = A + ww^T$ that differ from A are circled x's or circled numbers on the diagonal, as are entries in $\overline{L}$. The columns that are modified in $\overline{L}$ correspond to the path $\{4, 6, 7, 8\}$; these nodes are highlighted in both elimination trees.

Modifying the nonzero pattern of L is not trivial, since adding entries to individual columns of the `cs` data structure requires all subsequent columns to be shifted. Modifying both the pattern and the values of L can be done in time proportional to the number of entries in L that change, but the full algorithm is beyond the concise scope of this book. The simpler case assumes the pattern of L does not change. This case occurs if and only if $\mathcal{W} \subseteq \mathcal{L}_f$, in which case the elimination tree of L and $\overline{L}$ are the same, and a rank-1 update traverses the path $f \rightsquigarrow r$ in T.

The MATLAB `chol_downdate` function below performs the rank-1 downdate, returning a new `Lb` that is the Cholesky factor of `L*L'-w*w'`.

```
function [L, w] = chol_downdate (L, w)
beta = 1 ;
n = size (L,1) ;
for j = 1:n
    alpha = w (j) / L (j,j) ;
    beta2 = sqrt (beta^2 - alpha^2) ;
    if (~isreal (beta2)) error ('not positive definite') , end
    gamma = alpha / (beta2 * beta) ;
    delta = beta2 / beta ;
    L (j,j) = delta * L (j,j) ;
    w (j) = alpha ;
    beta = beta2 ;
    if (j == n) return, end
    w (j+1:n) = w (j+1:n) - alpha * L (j+1:n,j) ;
    L (j+1:n,j) = delta * L (j+1:n,j) - gamma * w (j+1:n) ;
end
```

The cs_updown function computes a rank-1 update if `sigma=1` or a rank-1 downdate if `sigma=-1`. It assumes that $\mathcal{W}$ is a subset of the first column to be modified, $\mathcal{L}_f$; it does not modify the nonzero pattern of `L`. The sparse column w is passed in as the first column of the `C` matrix. The elimination tree, `parent`, is also required on input.

```
int cs_updown (cs *L, int sigma, const cs *C, const int *parent)
{
    int p, f, j, *Lp, *Li, *Cp, *Ci ;
    double *Lx, *Cx, alpha, beta = 1, delta, gamma, w1, w2, *w, n,  beta2 = 1 ;
    if (!CS_CSC (L) || !CS_CSC (C) || !parent) return (0) ;  /* check inputs */
    Lp = L->p ; Li = L->i ; Lx = L->x ; n = L->n ;
    Cp = C->p ; Ci = C->i ; Cx = C->x ;
    if ((p = Cp [0]) >= Cp [1]) return (1) ;        /* return if C empty */
    w = cs_malloc (n, sizeof (double)) ;            /* get workspace */
    if (!w) return (0) ;                            /* out of memory */
    f = Ci [p] ;
    for ( ; p < Cp [1] ; p++) f = CS_MIN (f, Ci [p]) ;  /* f = min (find (C)) */
    for (j = f ; j != -1 ; j = parent [j]) w [j] = 0 ;  /* clear workspace w */
    for (p = Cp [0] ; p < Cp [1] ; p++) w [Ci [p]] = Cx [p] ; /* w = C */
    for (j = f ; j != -1 ; j = parent [j])          /* walk path f up to root */
    {
        p = Lp [j] ;
        alpha = w [j] / Lx [p] ;                    /* alpha = w(j) / L(j,j) */
        beta2 = beta*beta + sigma*alpha*alpha ;
        if (beta2 <= 0) break ;                     /* not positive definite */
        beta2 = sqrt (beta2) ;
        delta = (sigma > 0) ? (beta / beta2) : (beta2 / beta) ;
        gamma = sigma * alpha / (beta2 * beta) ;
        Lx [p] = delta * Lx [p] + ((sigma > 0) ? (gamma * w [j]) : 0) ;
        beta = beta2 ;
        for (p++ ; p < Lp [j+1] ; p++)
        {
            w1 = w [Li [p]] ;
            w [Li [p]] = w2 = w1 - alpha * Lx [p] ;
            Lx [p] = delta * Lx [p] + gamma * ((sigma > 0) ? w1 : w2) ;
        }
    }
    cs_free (w) ;
    return (beta2 > 0) ;
}
```

`cs_updown` returns 0 if it runs out of memory or if the downdated matrix is not positive definite, and 1 otherwise. To keep the user interface as simple as possible, the function allocates its own workspace `w` of size `n`. Not all of this will be used, and care is taken to initialize only the part of the workspace `w` that will be needed. The time taken by the function is proportional to the number of entries in L that change; this can be much less than `n` (consider the case where `f=n`, for example). Since all columns along the path $f \rightsquigarrow r$ are a subset of this path, only those entries `w[i]` where $i \in \{f \rightsquigarrow r\}$ will be used. An alternative approach (used by CHOLMOD) is to allocate and initialize `w` only once and to set it to zero as the path is traversed. In this case, the traversal of the path to clear `w` can be skipped.

The update/downdate algorithms in CHOLMOD can modify the nonzero pattern of `L`, but this requires a more complex data structure than the `cs` sparse matrix. CHOLMOD can also perform a multiple-rank update/downdate of a supernodal Cholesky factorization and can add or delete rows and columns from L.

The MATLAB interface for `cs_updown` adds a substantial amount of overhead, since a MATLAB mexFunction should not modify its inputs. The interface makes a copy of `L` (taking $O(n + |L|)$ time) and then modifies it with a call to `cs_updown` (taking time proportional to the number of entries in L that change). The latter can be as small as $\Omega(1)$ and as high as $O(|L|)$, so the time to make the copy can be substantially larger than the time to update L. The `cs_updown` function also requires a matrix L for which no entries have been dropped due to numerical cancellation. The MATLAB statement `L=chol(A)'` drops zeros due to numerical cancellation, but `L=cs_chol(A,0)` leaves them in the matrix. The MATLAB function `cholupdate` computes a rank-1 update or downdate for full matrices only.

4.11 Further reading

George and Liu [88, 89] cover sparse Cholesky factorization in depth, prior to the development of elimination trees, supernodal factorization, or many of the algorithms in this chapter. They include a full description of SPARSPAK, which includes the left-looking method described in Section 4.8. SPARSPAK uses linked lists, first used in YSMP (Eisenstat et al.[73] and Eisenstat, Schultz, and Sherman [76]), rather than the $\mathcal{T}^k$ traversal. Gilbert [101] catalogs methods for computing nonzero patterns in sparse matrix computations. Schreiber [181] provides the first formal definition of the elimination tree and also introduces the row subtree $\mathcal{T}^k$. Liu describes the many uses of the elimination tree and how to compute it [148, 150], discusses the row-oriented sparse Cholesky factorization [151], and gives an overview of the multifrontal method [152]. Gilbert et al. [102] and Gilbert, Ng, and Peyton [107] discuss how to compute the row and column counts for Cholesky, QR, and LU factorization and how to compute the elimination tree of A^TA. Davis [29] presents an up-looking LDL^T factorization algorithm with an $O(|L|)$-time column count algorithm that traverses each row subtree explicitly.

The row and column count algorithms `rowcnt` and `cs_counts` are due to Gilbert, Ng, and Peyton [107], except for a few minor modifications. The algorithms described here use a slightly different method for determining the skeleton

matrix. Tarjan [195] discusses how the disjoint-set-union data structure can be used efficiently to compute a sequence of least common ancestors.

Many software packages are available for factorizing sparse symmetric positive definite or symmetric indefinite matrices. Details of these packages are summarized in Section 8.6, including references to papers that discuss the supernodal and multifrontal methods. Gould, Hu, and Scott [116] compare many of these packages.

The BLAS (Dongarra et al. [46]) and LAPACK (Anderson et al. [8]) are two of the many software packages that provide dense matrix operations and factorization methods. Optimized BLAS can obtain near-peak performance on many computers (Goto and van de Geijn [115]).[9]

Applications that require the update or downdate of a sparse Cholesky factorization include optimization algorithms, least squares problems in statistics, the analysis of electrical circuits and power systems, structural mechanics, boundary condition changes in partial differential equations, domain decomposition methods, and boundary element methods, as discussed by Hager [124]. Gill et al. [110] and Stewart [190] provide an extensive summary of update/downdate methods. Stewart [189] introduced the term *downdating* and analyzed its error properties. LINPACK includes a rank-1 dense update/downdate [45]; it is used in the MATLAB `cholupdate` function. The `chol_update` function above is Carlson's algorithm [20], and `chol_downdate` is from Pan [163]. The `cs_updown` function is based on Bischof, Pan, and Tang's combination of Carlson's update and Pan's downdate [16]. Davis and Hager developed an optimal sparse multiple-rank supernodal update/downdate method, including a method to add and delete rows from A (CHOLMOD [35, 36, 37]).

Exercises

4.1. Use `cs_ereach` to implement an $O(|L|)$-time algorithm for computing the elimination tree and the number of entries in each row and column of L. It should operate using only the matrix `A` and $O(n)$ additional workspace. The matrix `A` should not be modified.

4.2. Compare and contrast `cs_chol` with the LDL package [29] and with `cholmod_rowfac`. Both can be downloaded from www.siam.org/books/fa02.

4.3. Write a function that solves $Lx = b$ when L, x, and b are sparse and L comes from a Cholesky factorization, using the elimination tree. Assume the elimination tree is already available; it can be passed in a `parent` array, or it can be found by examining `L` directly, since `L` has sorted columns.

4.4. Repeat Problem 4.3, but solve $L^T x = b$ instead.

4.5. The `cs_ereach` function can be simplified if `A` is known to be permuted according to a postordering of its elimination tree and if the row indices in each column of `A` are known to be sorted. Consider two successive row indices i_1 and i_2 in a column of A. When traversing up the elimination tree from node

[9] www.tacc.utexas.edu/resources/software

i_1, the least common ancestor of i_1 and i_2 is the first node $a > i_2$. Let p be the next-to-the-last node along the path $i_1 \rightsquigarrow a$ (where $p < i_2 < a$). Include the path $i_1 \rightsquigarrow p$ in an output queue (not a stack). Continue traversing the tree, starting at node i_2. The resulting queue will be in perfectly sorted order. The `while(len>0)` loop in `cs_ereach` can then be removed.

4.6. Compute the *height* of the elimination tree, which is the length of the longest path from a root to any leaf. The time taken should be $O(n)$. The result should be the same as the second output of the MATLAB `symbfact` function.

4.7. Why is `head` of size `n+1` in `cs_counts`?

4.8. How does the *skeleton* function implicitly ignore duplicate entries?

4.9. The `cs_schol` function computes a postordering, but does not combine it with the fill-reducing ordering, because the ordering from `cs_amd` includes an approximate postordering of the elimination tree. However, `cs_amd` might not be called. Add an option to `cs_schol` to combine the fill-reducing order (or the natural ordering) with the postordering.

4.10. Write a function that computes the symbolic Cholesky factorization of A (the nonzero pattern of L). Hint: start with `cs_chol` and remove any numerical computations. The algorithm should compute the pattern of L in $O(|L|)$ time and return a matrix L with sorted columns. The `s` array can be removed, since the row indices can be stored immediately into `L` in any order. It should allocate both `N->L->i` and `N->L->x` for use in Problem 4.11. Allocating `N->L->x` can be postponed, but allocating it here makes it simpler to write a MATLAB mexFunction interface for this problem.

4.11. Write a sparse left-looking Cholesky factorization algorithm with prototype `int cs_leftchol(cs *A, css *S, csn *N)`. It should assume the nonzero pattern of `L` has already been computed (see Problem 4.10). Compare its performance with `cs_chol` and `cs_rechol` in Problem 4.12. The algorithm is very similar to `cs_chol`. The initializations are identical, except that `x` should be created with `cs_calloc`, not `cs_malloc`. The `N` structure should be passed in with all of `N->L` preallocated. The `s` array is not needed if `cs_ereach` is merged with `cs_leftchol` (the topological order is not required).

4.12. Write a function with prototype `int cs_rechol(cs *A, css *S, csn *N)` that computes the Cholesky factorization of A using the up-looking method. It should assume that the nonzero pattern of `L` has already been computed in a prior call to `cs_chol` (or by Problem 4.10). The nonzero pattern of `A` should be the same as in the prior call to `cs_chol`.

4.13. An *incomplete Cholesky factorization* computes an approximation to L with fewer nonzeros. It is useful as a preconditioner for iterative methods, as discussed in detail by Saad [178]. One method for computing it is to drop small entries from L as they are computed. Another is to use a fixed nonzero pattern (typically the nonzero entries in A) and keep only entries in L within that pattern. Write an incomplete Cholesky factorization based on `cs_chol` or `cs_leftchol` (Problem 4.11). See Problem 6.13 for more details. See also the MATLAB `cholinc` function.

Chapter 5

Orthogonal methods

The most reliable methods for solving least squares problems use orthogonal transformations. This chapter considers QR factorization based on Householder reflections and Givens rotations.

5.1 Householder reflections

A *Householder reflection* is an orthogonal matrix of the form $H = I - \beta v v^T$, where β is a scalar and v is a column vector. The vector v and scalar β can be chosen based on a vector x so that Hx is zero except for the first entry $(Hx)_1 = \pm\|x\|_2$. The MATLAB function `[v,beta,s]=gallery('house',x,2)` computes `v`, `beta`, and `s` $= (Hx)_1$. Applying H to a vector x of length n takes only about $4n$ floating-point operations ($Hx = x - v(\beta(v^T x))$). The matrix H is symmetric and orthogonal ($HH^T = H^T H = I$), as a consequence of how β and v are chosen. The `cs_house` function ensures `s` $= \|x\|_2$. It computes β and `s` and overwrites `x` with `v`.

```
double cs_house (double *x, double *beta, int n)
{
    double s, sigma = 0 ;
    int i ;
    if (!x || !beta) return (-1) ;          /* check inputs */
    for (i = 1 ; i < n ; i++) sigma += x [i] * x [i] ;
    if (sigma == 0)
    {
        s = fabs (x [0]) ;                  /* s = |x(0)| */
        (*beta) = (x [0] <= 0) ? 2 : 0 ;
        x [0] = 1 ;
    }
    else
    {
        s = sqrt (x [0] * x [0] + sigma) ;  /* s = norm (x) */
        x [0] = (x [0] <= 0) ? (x [0] - s) : (-sigma / (x [0] + s)) ;
        (*beta) = -1. / (s * x [0]) ;
    }
    return (s) ;
}
```

Let $\mathcal{V}$ and $\mathcal{X}$ be the nonzero pattern of v and x, respectively. If $x_1 \neq 0$, then $\mathcal{V} = \mathcal{X}$. The theorems and algorithms for the sparse QR factorization are simplified if this condition always holds, and thus this discussion assumes that x_1 is a structural entry in x. That is, x_1 will always be an entry in the sparse vector x, but it might be numerically zero. Then, ignoring numerical cancellation, $\mathcal{V} = \mathcal{X}$ is always true. Normally, a Householder reflection H is applied not only to the vector x it was created from but also to other vectors or matrices as well. If applied to a vector y ($\overline{y} = Hy$), and assuming $x_1 \neq 0$, the nonzero pattern of $\overline{y}$ becomes $\overline{\mathcal{Y}} = \mathcal{Y} \cup \mathcal{V} = \mathcal{Y} \cup \mathcal{X}$ if $\mathcal{Y} \cap \mathcal{X}$ is **not** empty, and $\overline{\mathcal{Y}} = \mathcal{Y}$ otherwise.

The function `cs_happly` applies a Householder reflection to a dense vector x, where v is sparse. It overwrites x with `x-v*(beta*(v'*x))`. The vector v is obtained from the ith column of the matrix V.

```
int cs_happly (const cs *V, int i, double beta, double *x)
{
    int p, *Vp, *Vi ;
    double *Vx, tau = 0 ;
    if (!CS_CSC (V) || !x) return (0) ;      /* check inputs */
    Vp = V->p ; Vi = V->i ; Vx = V->x ;
    for (p = Vp [i] ; p < Vp [i+1] ; p++)    /* tau = v'*x */
    {
        tau += Vx [p] * x [Vi [p]] ;
    }
    tau *= beta ;                            /* tau = beta*(v'*x) */
    for (p = Vp [i] ; p < Vp [i+1] ; p++)    /* x = x - v*tau */
    {
        x [Vi [p]] -= Vx [p] * tau ;
    }
    return (1) ;
}
```

5.2 Left- and right-looking QR factorization

In this section, two Householder-based QR factorization algorithms are described that factorize the m-by-n matrix A into the product QR, where Q is orthogonal and R is upper triangular (or upper trapezoidal if $m < n$): a *left-looking* algorithm and a *right-looking* algorithm.

Suppose A is m-by-n with $m \geq n$. A sequence of n Householder reflections $H_1, \ldots, H_n$ is chosen to reduce A to upper triangular form. Let $A^{[k]}$ denote the product $H_k H_{k-1} \ldots H_1 A$. The first Householder reflection, H_1, is constructed from the first column of A; thus, the first column of $H_1 A$ is all zero except for the diagonal entry, $a_{11} = \|A_{*1}\|_2$. The kth Householder reflection is chosen based on $x = A^{[k-1]}_{k \ldots m,k}$, a vector of length $m-k+1$, resulting in β_k (a scalar) and $\overline{v}_k$ (a vector of length $m-k+1$). Let v_k be a column vector of length m that is zero in rows 1 to $k-1$ and $(v_k)_{k \ldots m} = \overline{v}_k$. Then $H_k = I - \beta_k v_k v_k^T$.

Theorem 5.1 (Golub [113]). *The QR factorization $QR = A$, where $A \in \mathbb{R}^{m \times n}$ and $m \geq n$, is $Q = H_1 H_2 \cdots H_n = \prod_{k=1}^{n} H_k$ and $R = Q^T A = H_n \cdots H_2 H_1 A = (\prod_{k=n}^{1} H_k) A = A^{[n]}$.*

The sequence of Householder reflections can be applied in a left-looking or right-looking manner. The right-looking algorithm `qr_right` simply applies each Householder reflection to all of `A` as soon as it is constructed. The `qr_right` function is difficult to implement in a concise sparse matrix algorithm, although it forms the basis of the multifrontal sparse QR method.

```
function [V,Beta,R] = qr_right (A)
[m n] = size (A) ;
V = zeros (m,n) ;
Beta = zeros (1,n) ;
for k = 1:n
    [v,beta,s] = gallery ('house', A (k:m,k), 2) ;
    V (k:m,k) = v ;
    Beta (k) = beta ;
    A (k:m,k:n) = A (k:m,k:n) - v * (beta * (v' * A (k:m,k:n))) ;
end
R = A ;
```

The left-looking algorithm `qr_left` applies the Householder reflections only to the current column `k`, one column at a time, and is simpler to implement for the sparse case.

```
function [V,Beta,R] = qr_left (A)
[m n] = size (A) ;
V = zeros (m,n) ;
Beta = zeros (1,n) ;
R = zeros (m,n) ;
for k = 1:n
    x = A (:,k) ;
    for i = 1:k-1
        v = V (i:m,i) ;
        beta = Beta (i) ;
        x (i:m) = x (i:m) - v * (beta * (v' * x (i:m))) ;
    end
    [v,beta,s] = gallery ('house', x (k:m), 2) ;
    V (k:m,k) = v ;
    Beta (k) = beta ;
    R (1:(k-1),k) = x (1:(k-1)) ;
    R (k,k) = s ;
end
```

Note that `qr_right` and `qr_left` do not compute an explicit representation of Q; this is also true of the sparse left-looking QR factorization described next.

5.3 Householder-based sparse QR factorization

The left-looking QR factorization algorithm (`qr_left`) forms the basis of the sparse QR factorization algorithm presented below.

Let $\mathcal{R}_{i*}$ and $\mathcal{R}_{*j}$ be the nonzero pattern of row i and column j of R, respectively. Let $\mathcal{A}_{i*}^{[k]}$ and $\mathcal{A}_{*j}^{[k]}$ denote the nonzero pattern of row i and column j of $A^{[k]}$, respectively. Let $\mathcal{V}_k$ be the nonzero pattern of v_k. Let $\mathcal{T}$ be the column elimination tree of A (the elimination tree of $A^T A$).

Some of the theorems stated here require $A_{kk}^{[k-1]}$ to be structurally nonzero (that is, it is an entry in the data structure even if numerically zero). If this is not the case, then the rows of A can be permuted, or the sparse matrix `A` can be modified by adding explicit zero entries, to ensure that this condition holds. All of the theorems ignore numerical cancellation. Some theorems require the matrix to have the *strong Hall property*; they provide loose upper bounds on the nonzero pattern otherwise. Because of space constraints, the proofs are brief or omitted. The definition of *strong Hall* is given in Section 7.3.

Theorem 5.2 (George, Liu, and Ng [95]). *Consider $HA = A - v(\beta(v^T A))$. Then $(HA)_{i*}$, where $i \notin \mathcal{V}$ is equal to row i of A. For any row $i \in \mathcal{V}$, the nonzero pattern of $(HA)_{i*}$ is*

$$\bigcup_{i \in \mathcal{V}} \mathcal{A}_{i*}. \tag{5.1}$$

That is, in HA, the nonzero pattern of any modified row $i \in \mathcal{V}$ is replaced with the set union of all rows that are modified by the Householder reflection H.

Proof. From Theorem 2.1, the nonzero pattern of $v^T A = (A^T v)^T$ is given by (5.1). The entry $(v(v^T A))_{ij}$ (β can be ignored) is nonzero if and only if $i \in \mathcal{V}$ and $j \in \cup_{i \in \mathcal{V}} \mathcal{A}_{i*}$. The matrix $v(v^T A)$ is then subtracted from A to obtain HA, modifying all rows $i \in \mathcal{V}$. Each of the corresponding rows of A was used to construct the set union (5.1), so all of them now have the same nonzero pattern, given by (5.1). □

Theorem 5.3 (Golub and Van Loan [114]). *If $A^T A$ is positive definite, and its Cholesky factorization is $LL^T = A^T A$, then $L = R_{1\ldots n,1\ldots n}^T$.*

Proof. $A^T A = (QR)^T QR = R^T Q^T QR = R^T R$. □

Theorem 5.4 (George and Ng [97]). *If $A_{kk}^{[k-1]}$ is structurally nonzero for all $1 \leq k \leq n$, then $\mathcal{R}_{i*} = \mathcal{A}_{i*}^{[k]}$.*

Theorem 5.5 (Coleman, Edenbrandt, and Gilbert [22], George and Heath [83]). *Assuming the matrix A has the strong Hall property, $\mathcal{R}_{*k} = \mathcal{L}_{k*}$, where $\mathcal{L}_{k*}$ denotes the nonzero pattern of the kth row of the symbolic Cholesky factor of $A^T A$. If A does not have the strong Hall property, $\mathcal{R}_{*k} \subseteq \mathcal{L}_{k*}$.*

Corollary 5.6. *$\mathcal{R}_{*k} = Reach_{T_k}(\mathcal{C}_k)$, where $\mathcal{C}_k$ is the nonzero pattern of the upper triangular part of column k of $A^T A$ (assuming A has the strong Hall property).*

A more concise method for computing $\mathcal{R}_{*k}$ is based on the following theorem.

Theorem 5.7. *$\mathcal{R}_{*k} = Reach_{T_k}(\{\min \mathcal{A}_{i*} \mid i \in \mathcal{A}_{*k}\})$ (assuming A has the strong*

Hall property).

Proof. Consider column k of A and the nonzero pattern of $C = A^T A$. The matrix C can be written as a summation of outer products, one for each row, $C = \sum_{i=1}^{m} A_{i*}^T A_{i*}$. Consider the entry $a_{ik} \neq 0$ in column k. The nonzero pattern of $A_{i*}^T A_{i*}$ is a dense submatrix of C; every c_{k_1,k_2} is nonzero for any pair of column indices k_1 and k_2 in $\mathcal{A}_{i*}$. From Theorem 4.13, $k_1 < k_2$ implies $k_1 \rightsquigarrow k_2$ in $\mathcal{T}_k$. Thus, k_2 is redundant for computing $\text{Reach}_{\mathcal{T}_k}(\mathcal{C}_k)$. Only $j = \min \mathcal{A}_{i*}$, the smallest column index of nonzero entries in row i, needs to be considered for any entry a_{ik} in column k of A. This j corresponds to the leftmost entry a_{ij} in row i of A. Thus $\mathcal{R}_{*k} = \text{Reach}_{\mathcal{T}_k}(\mathcal{C}_k) = \text{Reach}_{\mathcal{T}_k}(\{\min \mathcal{A}_{i*} \mid i \in \mathcal{A}_{*k}\})$. □

By definition, $\mathcal{V}_k = \mathcal{A}_{*k}^{[k-1]} \setminus \{1 \ldots k-1\}$. It can be computed using either of the next two theorems.

Theorem 5.8.

$$\mathcal{V}_k = \left(\bigcup_{k=parent(i)} \mathcal{V}_i \setminus \{i\} \right) \cup \{i \mid k = \min \mathcal{A}_{i*}\},$$

where each set in the above expression is disjoint from all other sets, and A has the strong Hall property. That is,

$$|\mathcal{V}_k| = \left(\sum_{k=parent(i)} |\mathcal{V}_i| - 1 \right) + |\{i \mid k = \min \mathcal{A}_{i*}\}|. \tag{5.2}$$

If A does not have the strong Hall property, this is an upper bound on $\mathcal{V}$.

The left-looking sparse QR algorithm `cs_qr` uses a less concise theorem to compute $\mathcal{V}_k$ to avoid the need for finding the set $|\{i \mid k = \min \mathcal{A}_{i*}\}|$.

Theorem 5.9.

$$\mathcal{V}_k = \left(\bigcup_{k=parent(i)} \mathcal{V}_i \setminus \{i\} \right) \cup (\mathcal{A}_{*k} \setminus \{1, \ldots, k-1\}), \tag{5.3}$$

where A has the strong Hall property. Otherwise, (5.3) is an upper bound.

These theorems are illustrated with an example matrix, its QR factorization in Figure 5.1, and its column elimination tree. The upper triangular matrix R and lower triangular matrix V containing the Householder vectors are shown as a single matrix. Fill-in entries that appear in R or V but not in A are shown as circled x's.

With these theorems, a left-looking sparse QR factorization algorithm can be stated in the following pseudocode, `sparse_qr_left`. It assumes $A_{kk}^{[k-1]}$ is struc-

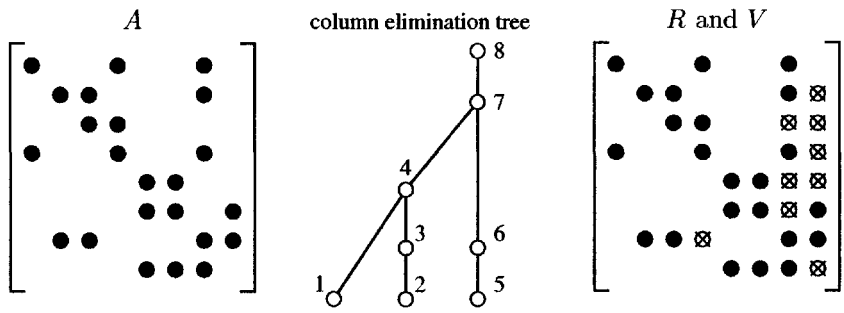


Figure 5.1. *QR factorization*

turally nonzero or, equivalently, $k \in \mathcal{V}_k$. Compare the following pseudocode function with the MATLAB function `qr_left`.

function $[V, \beta_{1\ldots n}, R] =$ `sparse_qr_left`(A)
  $\mathcal{T}$ = elimination tree of $A^T A$
  compute $|R|$ using `cs_counts` of $A^T A$
  compute $|\mathcal{V}_{1\ldots n}|$ using (5.2)
  for $k = 0$ to $n-1$ **do**
    $\mathcal{R}_{*k} = \text{Reach}_{\mathcal{T}_k}(\{\min \mathcal{A}_{i*} \mid i \in \mathcal{A}_{*k}\})$
    $x = A_{*k}$
    $\mathcal{V}_k = \mathcal{A}_{*k}$
    for each $i \in \mathcal{R}_{*k}$ **do**
      $x = x - v_i(\beta_i(v_i^T x))$
      if `parent`$(i) = k$ **then**
        $\mathcal{V}_k = \mathcal{V}_k \cup \mathcal{V}_i \setminus \{i\}$
    $R_{1\ldots k-1,k} = x_{1\ldots k-1}$
    $[v_k, \beta_k, r_{kk}] =$ `house`$(x_{k\ldots m})$

The `cs_vcount` function evaluates (5.2). It also computes a row permutation P (represented by `pinv`) that ensures the diagonal entries $(PA)_{kk}^{[k-1]}$ are all structurally nonzero and finds `leftmost[i]` $= \min \mathcal{A}_{i*}$ for all rows i. It finds $|\mathcal{V}_{1\ldots n}|$ by creating and updating a set of n linked lists. At step k of the algorithm, list k will contain the set $\mathcal{V}_k$. The lists are initialized by placing each row i in the list $\min \mathcal{A}_{i*}$. The primary `for k` loop computes the row permutation and nonzero counts of V. In this loop, the first entry in the kth linked list is selected as the "pivot row"; this row becomes the kth row of PA to ensure that $(PA)_{kk}^{[k-1]}$ is structurally nonzero. If no such row exists, then the matrix A is structurally rank deficient and a fictitious row is created. Next, all rows in the kth linked list except this first row are removed from list k and placed in the linked list of the parent of k. After $|\mathcal{V}_{1\ldots n}|$ is computed, any unordered rows are assigned. It computes `S->lnz`, which is the number of entries in V, and `S->m2`, which is the number of rows in A after fictitious rows are added.

```
static int cs_vcount (const cs *A, css *S)
{
    int i, k, p, pa, n = A->n, m = A->m, *Ap = A->p, *Ai = A->i, *next, *head,
        *tail, *nque, *pinv, *leftmost, *w, *parent = S->parent ;
    S->pinv = pinv = cs_malloc (m+n, sizeof (int)) ;        /* allocate pinv, */
    S->leftmost = leftmost = cs_malloc (m, sizeof (int)) ;  /* and leftmost */
    w = cs_malloc (m+3*n, sizeof (int)) ;   /* get workspace */
    if (!pinv || !w || !leftmost)
    {
        cs_free (w) ;                       /* pinv and leftmost freed later */
        return (0) ;                        /* out of memory */
    }
    next = w ; head = w + m ; tail = w + m + n ; nque = w + m + 2*n ;
    for (k = 0 ; k < n ; k++) head [k] = -1 ;   /* queue k is empty */
    for (k = 0 ; k < n ; k++) tail [k] = -1 ;
    for (k = 0 ; k < n ; k++) nque [k] = 0 ;
    for (i = 0 ; i < m ; i++) leftmost [i] = -1 ;
    for (k = n-1 ; k >= 0 ; k--)
    {
        for (p = Ap [k] ; p < Ap [k+1] ; p++)
        {
            leftmost [Ai [p]] = k ;         /* leftmost[i] = min(find(A(i,:)))*/
        }
    }
    for (i = m-1 ; i >= 0 ; i--)            /* scan rows in reverse order */
    {
        pinv [i] = -1 ;                     /* row i is not yet ordered */
        k = leftmost [i] ;
        if (k == -1) continue ;             /* row i is empty */
        if (nque [k]++ == 0) tail [k] = i ; /* first row in queue k */
        next [i] = head [k] ;               /* put i at head of queue k */
        head [k] = i ;
    }
    S->lnz = 0 ;
    S->m2 = m ;
    for (k = 0 ; k < n ; k++)               /* find row permutation and nnz(V)*/
    {
        i = head [k] ;                      /* remove row i from queue k */
        S->lnz++ ;                          /* count V(k,k) as nonzero */
        if (i < 0) i = S->m2++ ;            /* add a fictitious row */
        pinv [i] = k ;                      /* associate row i with V(:,k) */
        if (--nque [k] <= 0) continue ;     /* skip if V(k+1:m,k) is empty */
        S->lnz += nque [k] ;                /* nque [k] is nnz (V(k+1:m,k)) */
        if ((pa = parent [k]) != -1)        /* move all rows to parent of k */
        {
            if (nque [pa] == 0) tail [pa] = tail [k] ;
            next [tail [k]] = head [pa] ;
            head [pa] = next [i] ;
            nque [pa] += nque [k] ;
        }
    }
    for (i = 0 ; i < m ; i++) if (pinv [i] < 0) pinv [i] = k++ ;
    cs_free (w) ;
    return (1) ;
}
```

The cs_sqr function does the ordering and analysis for a sparse QR factor-

ization. Two parameters determine behavior of `cs_sqr`: `order` and `qr`. The `order` parameter specifies the ordering to use; the natural ordering (`order=0`) or a minimum degree ordering of A^TA (`order=3`) are good choices for a QR factorization. The `qr` parameter must be true (nonzero) for a sparse QR factorization. `cs_sqr` first finds a fill-reducing column permutation `S->q`. The function then finds the permuted matrix $C = AQ$ (where Q is the column permutation, not the orthogonal factor Q), determines the elimination tree of C^TC and postorders it, and finds the column counts of L (equivalently, the row counts of R). It then calls `cs_vcount` to find the column counts $|\mathcal{V}_{1\ldots n}|$ of the `V` matrix that holds the Householder vectors. The `cs_qr` function performs the numerical QR factorization.

```
css *cs_sqr (int order, const cs *A, int qr)
{
    int n, k, ok = 1, *post ;
    css *S ;
    if (!CS_CSC (A)) return (NULL) ;          /* check inputs */
    n = A->n ;
    S = cs_calloc (1, sizeof (css)) ;         /* allocate result S */
    if (!S) return (NULL) ;                   /* out of memory */
    S->q = cs_amd (order, A) ;                /* fill-reducing ordering */
    if (order && !S->q) return (cs_sfree (S)) ;
    if (qr)                                   /* QR symbolic analysis */
    {
        cs *C = order ? cs_permute (A, NULL, S->q, 0) : ((cs *) A) ;
        S->parent = cs_etree (C, 1) ;         /* etree of C'*C, where C=A(:,q) */
        post = cs_post (S->parent, n) ;
        S->cp = cs_counts (C, S->parent, post, 1) ;  /* col counts chol(C'*C) */
        cs_free (post) ;
        ok = C && S->parent && S->cp && cs_vcount (C, S) ;
        if (ok) for (S->unz = 0, k = 0 ; k < n ; k++) S->unz += S->cp [k] ;
        ok = ok && S->lnz >= 0 && S->unz >= 0 ;     /* int overflow guard */
        if (order) cs_spfree (C) ;
    }
    else
    {
        S->unz = 4*(A->p [n]) + n ;           /* for LU factorization only, */
        S->lnz = S->unz ;                     /* guess nnz(L) and nnz(U) */
    }
    return (ok ? S : cs_sfree (S)) ;          /* return result S */
}

csn *cs_qr (const cs *A, const css *S)
{
    double *Rx, *Vx, *Ax, *Beta, *x ;
    int i, k, p, m, n, vnz, p1, top, m2, len, col, rnz, *s, *leftmost, *Ap, *Ai,
        *parent, *Rp, *Ri, *Vp, *Vi, *w, *pinv, *q ;
    cs *R, *V ;
    csn *N ;
    if (!CS_CSC (A) || !S) return (NULL) ;
    m = A->m ; n = A->n ; Ap = A->p ; Ai = A->i ; Ax = A->x ;
    q = S->q ; parent = S->parent ; pinv = S->pinv ; m2 = S->m2 ;
    vnz = S->lnz ; rnz = S->unz ; leftmost = S->leftmost ;
    w = cs_malloc (m2+n, sizeof (int)) ;          /* get int workspace */
    x = cs_malloc (m2, sizeof (double)) ;         /* get double workspace */
    N = cs_calloc (1, sizeof (csn)) ;             /* allocate result */
```

```
    if (!w || !x || !N) return (cs_ndone (N, NULL, w, x, 0)) ;
    s = w + m2 ;                                    /* s is size n */
    for (k = 0 ; k < m2 ; k++) x [k] = 0 ;          /* clear workspace x */
    N->L = V = cs_spalloc (m2, n, vnz, 1, 0) ;      /* allocate result V */
    N->U = R = cs_spalloc (m2, n, rnz, 1, 0) ;      /* allocate result R */
    N->B = Beta = cs_malloc (n, sizeof (double)) ;  /* allocate result Beta */
    if (!R || !V || !Beta) return (cs_ndone (N, NULL, w, x, 0)) ;
    Rp = R->p ; Ri = R->i ; Rx = R->x ;
    Vp = V->p ; Vi = V->i ; Vx = V->x ;
    for (i = 0 ; i < m2 ; i++) w [i] = -1 ; /* clear w, to mark nodes */
    rnz = 0 ; vnz = 0 ;
    for (k = 0 ; k < n ; k++)               /* compute V and R */
    {
        Rp [k] = rnz ;                      /* R(:,k) starts here */
        Vp [k] = p1 = vnz ;                 /* V(:,k) starts here */
        w [k] = k ;                         /* add V(k,k) to pattern of V */
        Vi [vnz++] = k ;
        top = n ;
        col = q ? q [k] : k ;
        for (p = Ap [col] ; p < Ap [col+1] ; p++)   /* find R(:,k) pattern */
        {
            i = leftmost [Ai [p]] ;         /* i = min(find(A(i,q))) */
            for (len = 0 ; w [i] != k ; i = parent [i]) /* traverse up to k */
            {
                s [len++] = i ;
                w [i] = k ;
            }
            while (len > 0) s [--top] = s [--len] ; /* push path on stack */
            i = pinv [Ai [p]] ;             /* i = permuted row of A(:,col) */
            x [i] = Ax [p] ;                /* x (i) = A(:,col) */
            if (i > k && w [i] < k)         /* pattern of V(:,k) = x (k+1:m) */
            {
                Vi [vnz++] = i ;            /* add i to pattern of V(:,k) */
                w [i] = k ;
            }
        }
        for (p = top ; p < n ; p++) /* for each i in pattern of R(:,k) */
        {
            i = s [p] ;                     /* R(i,k) is nonzero */
            cs_happly (V, i, Beta [i], x) ; /* apply (V(i),Beta(i)) to x */
            Ri [rnz] = i ;                  /* R(i,k) = x(i) */
            Rx [rnz++] = x [i] ;
            x [i] = 0 ;
            if (parent [i] == k) vnz = cs_scatter (V, i, 0, w, NULL, k, V, vnz);
        }
        for (p = p1 ; p < vnz ; p++)        /* gather V(:,k) = x */
        {
            Vx [p] = x [Vi [p]] ;
            x [Vi [p]] = 0 ;
        }
        Ri [rnz] = k ;                     /* R(k,k) = norm (x) */
        Rx [rnz++] = cs_house (Vx+p1, Beta+k, vnz-p1) ; /* [v,beta]=house(x) */
    }
    Rp [n] = rnz ;                          /* finalize R */
    Vp [n] = vnz ;                          /* finalize V */
    return (cs_ndone (N, NULL, w, x, 1)) ;  /* success */
}
```

The cs_qr function uses the symbolic analysis computed by cs_sqr: the column elimination tree S->parent, column preordering S->q, row permutation S->pinv, the S->leftmost array, the number of nonzeros in R and V (S->unz and S->lnz, respectively), and the number of rows S->m2 after adding fictitious rows if A is structurally rank deficient.

The function first extracts the contents of S, allocates the result N, and allocates and initializes some workspace. Next, each column k of V and R is computed. The body of the for k loop first determines where V(:,k) and R(:,k) start, and finds the column A(:,col) corresponding to C(:,k). The nonzero pattern $\mathcal{R}_{*k}$ of the kth column of R is found using a symbolic sparse triangular solve (Theorem 5.7). Prior Householder reflections are applied to column k, one for each nonzero entry in $|\mathcal{R}_k|$, and $\mathcal{V}_k$ is computed (5.3). The modified column $x = H_{k-1} \dots H_1 C_{*k}$ is gathered from its dense vector representation x as the kth column of V, and overwritten with the kth Householder vector. A complete symbolic and numeric QR factorization, including a fill-reducing column preordering, can be computed with S=cs_sqr(3,A,1) followed by N=cs_qr(A,S).

In MATLAB, [Q,R]=qr(A) computes the QR factorization of A. The fill-reducing column permutation must be applied to A prior to calling qr. The MATLAB qr function is based on Givens rotations, not Householder reflections. It returns the orthogonal matrix Q, not the more compact representation of V, Beta, and pinv that cs_qr uses.

The cs_qright and cs_qleft M-files apply the Householder reflections (V, Beta, and p as computed by cs_qr) to the left or right of a matrix. cs_qleft is similar to cs_happly, except that it applies all of the Householder vectors.

```
function X = cs_qright (V, Beta, p, Y)
%CS_QRIGHT apply Householder vectors on the right.
%   X = cs_qright(V,Beta,p,Y) computes X = Y*P'*H1*H2*...*Hn = Y*Q where Q is
%   represented by the Householder vectors V, coefficients Beta, and
%   permutation p.  p can be [], which denotes the identity permutation.
%   To obtain Q itself, use Q = cs_qright(V,Beta,p,speye(size(V,1))).
%
%   See also CS_QR, CS_QLEFT.

[m n] = size (V) ;
X = Y ;
if (~isempty (p)) X = X (:,p) ; end
for k = 1:n
    X = X - (X * (Beta (k) * V (:,k))) * V (:,k)' ;
end

function X = cs_qleft (V, Beta, p, Y)
%CS_QLEFT apply Householder vectors on the left.
%   X = cs_qleft(V,Beta,p,Y) computes X = Hn*...*H2*H1*P*Y = Q'*Y where Q is
%   represented by the Householder vectors V, coefficients Beta, and
%   permutation p.  p can be [], which denotes the identity permutation.
%
%   See also CS_QR, CS_QRIGHT.

[m2 n] = size (V) ;
[m ny] = size (Y) ;
X = Y ;
```

```
if (m2 > m)
    if (issparse (Y))
        X = [X ; sparse(m2-m,ny)] ;
    else
        X = [X ; zeros(m2-m,ny)] ;
    end
end
if (~isempty (p)) X = X (p,:) ; end
for k = 1:n
    X = X - V (:,k) * (Beta (k) * (V (:,k)' * X)) ;
end
```

The Householder vectors stored in V are typically much sparser than the explicit representation of Q. Try this short experiment in MATLAB, which compares Q (with 38,070 nonzeros) and V (with only 3,906 nonzeros):

```
load west0479
q = colamd (west0479) ;
[Q,R] = qr (west0479 (:,q)) ;
[V,beta,p,R2] = cs_qr (west0479 (:,q)) ;
Q2 = cs_qright (V, beta, p, speye(size(V,1))) ;
```

5.4 Givens rotations

A *Givens rotation* is a 2-by-2 orthogonal matrix that can be applied to a 2-by-1 vector to zero out a selected entry. If a and b are scalars, c and s are selected so that

$$Gx = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix},$$

where $r = \pm\|x\|_2$. The coefficients c and s are computed using the following `givens2` MATLAB function.

```
function g = givens2(a,b)
if (b == 0)
    c = 1 ; s = 0 ;
elseif (abs (b) > abs (a))
    tau = -a/b ; s = 1 / sqrt (1+tau^2) ; c = s*tau ;
else
    tau = -b/a ; c = 1 / sqrt (1+tau^2) ; s = c*tau ;
end
g = [c -s ; s c] ;
```

If applied to a 2-by-n sparse matrix X (equivalently, to two rows selected from a larger matrix), the nonzero patterns of the rows of GX are both equal to the union of the nonzero pattern of the rows of X, except that any one entry in GX can be selected to become zero.

5.5 Row-merge sparse QR factorization

A Givens-rotation-based QR factorization requires 50% more floating-point operations than a Householder-based QR factorization for a full matrix, but it has some

advantages in the sparse case. Rows can be ordered to reduce the work below that of the Householder-based sparse QR. The disadvantage of using Givens rotations is that the resulting QR factorization is less suitable for multifrontal techniques.

MATLAB uses Givens rotations for its sparse QR factorization. It operates on the rows of R and A. The matrix R starts out equal to zero but with enough space allocated to hold the final R. Each step of the factorization brings in a new row of A and eliminates its entries with the existing R until it is either all zero or it can be placed as a new row of R. The `qr_givens_full` algorithm for full matrices is shown below. It assumes the diagonal of A is nonzero. The innermost loop annihilates the a_{ik} entry via a Givens rotation of the incoming ith row of A and the kth row of R.

```
function R = qr_givens_full (A)
[m n] = size (A) ;
for i = 2:m
    for k = 1:min(i-1,n)
        A ([k i],k:n) = givens2 (A(k,k), A(i,k)) * A ([k i],k:n) ;
        A (i,k) = 0 ;
    end
end
R = A ;
```

For the sparse case, the rotation to zero out a_{ik} must be skipped if it is already zero. The entries k that must be annihilated correspond to the nonzero pattern $\mathcal{V}_{i*}$ of the ith row of the Householder matrix V, discussed in the previous section.

Theorem 5.10 (George, Liu, and Ng [95]). *Assume A has a zero-free diagonal. The nonzero pattern $\mathcal{V}_{i*}$ of the ith row of the Householder matrix V is given by the path $f \rightsquigarrow \min(i, r)$ in the elimination tree $\mathcal{T}$ of A^TA, where $f = \min \mathcal{A}_{i*}$ is the leftmost nonzero entry in the ith row, and r is the root of the tree.*

This method is illustrated in the `qr_givens` M-file below.

```
function R = qr_givens (A)
[m n] = size (A) ;
parent = cs_etree (sparse (A), 'col') ;
A = full (A) ;
for i = 2:m
    k = min (find (A (i,:))) ;
    if (isempty (k))
        continue ;
    end
    while (k > 0 && k <= min (i-1,n))
        A ([k i],k:n) = givens2 (A(k,k), A(i,k)) * A ([k i],k:n) ;
        A (i,k) = 0 ;
        k = parent (k) ;
    end
end
R = sparse (A) ;
```

The M-file is not meant as an efficient implementation. Since operating on the rows of a sparse matrix is very slow, the matrix is converted into a full matrix first. In an efficient implementation, only the ith row of A is held in a working array of

size n. The matrix R is allocated with enough space for each final row, but it starts out empty. The ith row of A is annihilated with rows k in the path $f \rightsquigarrow \min(i, r)$ until encountering an empty row k of R, at which point the elimination stops and the partially eliminated row of A becomes the kth row of R. If A has a zero-free diagonal, this happens when $i = k$. This method is called the *row-merge* QR factorization algorithm; it could also be called an up-looking sparse QR, since at the ith step it accesses only row i of A and rows 1 to $\min(i, n)$ of R. The `qr_givens` function assumes `A` has a zero-free diagonal.

5.6 Further reading

George and Heath [83] present the Givens-rotation row-merge algorithm; the `qr` function in MATLAB is Gilbert's implementation of this method. The row-merge scheme of Liu [149] is a generalization of the George–Heath method. George and Liu [90] compare Householder reflections and Givens rotations for sparse QR factorization. Heath [128] surveys a range of methods for solving sparse linear least squares problems. Givens [112] and Householder [138] discuss the transformations later named after them. Householder reflections, Givens rotations, QR factorization, and the Gram–Schmidt method are all discussed at length by Golub and Van Loan [114], Higham [135], and Stewart [190].

Many papers have considered the sparsity pattern of Q, R, and the Householder matrix V and data structures to represent them. Coleman, Edenbrandt, and Gilbert [22] showed that a sequence of symbolic Givens rotations gives a tight upper bound on the structure of R when A is strong Hall. George, Liu, and Ng [95] describe an efficient data structure for storing the Householder matrix V, using paths in the elimination tree. The Gram–Schmidt method is not typically used in the sparse case, since it computes Q explicitly, and an explicit sparse Q has many more nonzeros than the Householder matrix V. Ng and Peyton [158] describe an efficient data structure for an explicit sparse Q.

Gilbert, Ng, and Peyton [108] discuss the use of separators for predicting structure in sparse QR factorization. Hare et al. [127] show how to compute the nonzero pattern of Q and R. Pothen [168] extended these results by showing that there exist matrices that reach these bounds. Ng and Peyton [160] determine a bound on the nonzero pattern of Q based on the Householder matrix V. Gilbert et al. [102] and Gilbert, Ng, and Peyton [107] discuss how to compute the row and column counts for Cholesky, QR, and LU factorization and how to compute the elimination tree of $A^T A$. Oliveira [162] discusses how to predict the pattern of QR factorization without computing the Dulmage–Mendelsohn decomposition (see also Section 7.4). Many of these methods for determining the sparsity patterns for both QR and LU factorization are surveyed by Gilbert and Ng [106].

Row orderings have a large impact on the intermediate fill-in and total work required for the row-merge sparse QR, as shown by George and Ng [96], who present a nested dissection approach for determining a good row permutation. See also George, Liu, and Ng's three-part series [92, 93, 94] and Gillespie and Olesky [111].

Sparse multifrontal QR factorization is typically based on Householder reflec-

tions; see Lu and Barlow [154], Matstoms [156], Amestoy, Duff, and Puglisi [6], and Pierce and Lewis [167] (who also present an approximate rank-revealing multifrontal QR algorithm).

Underdetermined systems can be solved with QR factorization applied to A^T, as discussed by George, Heath, and Ng [84].

Exercises

5.1. Write a function that computes the nonzero pattern of `V` and `R`.

5.2. Modify `cs_qr` so that it can handle `m`-by-`n` matrices where `m` $<$ `n`. One simple solution is to append empty rows onto `A`, but this will not be efficient if `m` is much smaller than `n`.

5.3. Write a function `cs_reqr (cs *A, css *S, csn *N)` that computes a QR factorization. It should assume that the nonzero patterns of `V` and `R` are already computed.

5.4. Add column pivoting to `cs_qr`. If a column has a norm less than or equal to a given tolerance, permute it to the end of the matrix. The matrices `V` and `R` will need to be dynamically reallocated (see `cs_lu` in Chapter 6), since permuting the columns breaks the symbolic preanalysis.

5.5. Combine the postordering with the fill-reducing ordering in `cs_sqr` (see Problem 4.9 for details).

Chapter 6
LU factorization

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

6.1 Upper bound on fill-in

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

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

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

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

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

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

- `order=0`: No column permutation is used; $LU = PA$. This is useful if A is already known to have a good column ordering.

- `order=1`: The column permutation Q is found from a fill-reducing ordering of $A + A^T$. During factorization, an attempt is made to ensure $P = Q^T$ (preference for selecting the pivot is given to the diagonal entry of $Q^T AQ$). This strategy is well suited to the many unsymmetric matrices arising in practice that have a roughly symmetric nonzero structure and reasonably large entries on the diagonal.

- `order=2`: Q is obtained from a fill-reducing ordering of $S^T S$, where $S = A$ except that all entries in "dense" rows of S are removed. A row in A with more entries than a heuristic threshold is considered dense. With partial pivoting, the (optimistic) hope is that these rows will not be selected as pivot rows until very late in the factorization.

- `order=3`: Q is obtained from the ordering of $A^T A$ with no dense rows dropped. In this case, the ordering tries to reduce the QR upper bounds on L and U given in Theorems 6.1 and 6.2.

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

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

6.2 Left-looking LU

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3 Right-looking and multifrontal LU

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

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

Cholesky factorization,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This inductive derivation is demonstrated by the `lu_rightpr` function below. It is not a tail-recursive procedure, since P_2 must be applied to l_{21} after the recursive call completes, and P must be constructed as well. These permutations can be postponed and applied when the factorization is complete; this is what the `cs_lu`

function does. For a dense matrix factorization, access to the rows of L is much simpler, and the permutations can be applied immediately, as done by `lu_left`. Either method leads to the same $LU = PA$ factorization. After replacing the recursion in `lu_rightpr` with its nonrecursive implementation and allowing `A` to be overwritten with its LU factorization, the conventional outer-product form of Gaussian elimination is obtained, as demonstrated by the `lu_rightp` function, shown below.

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

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

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

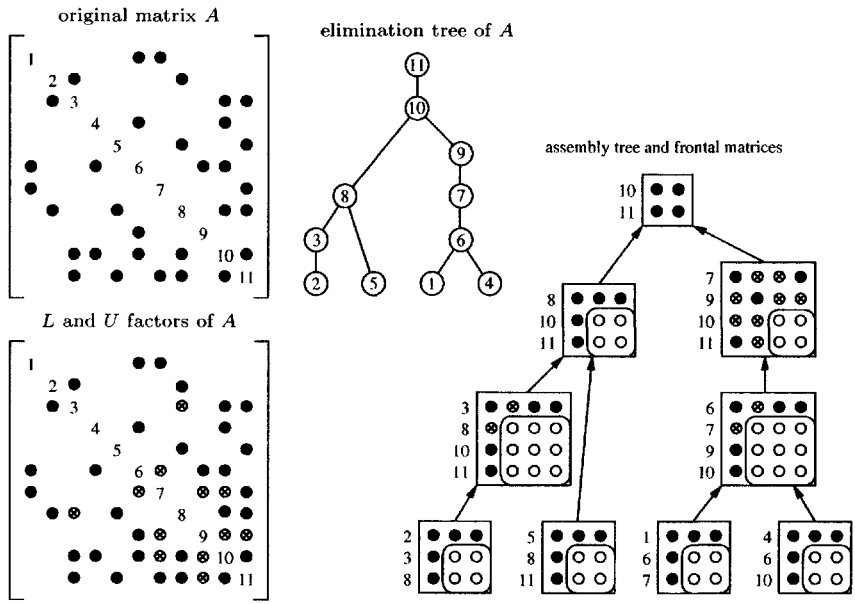


Figure 6.1. *Multifrontal example*

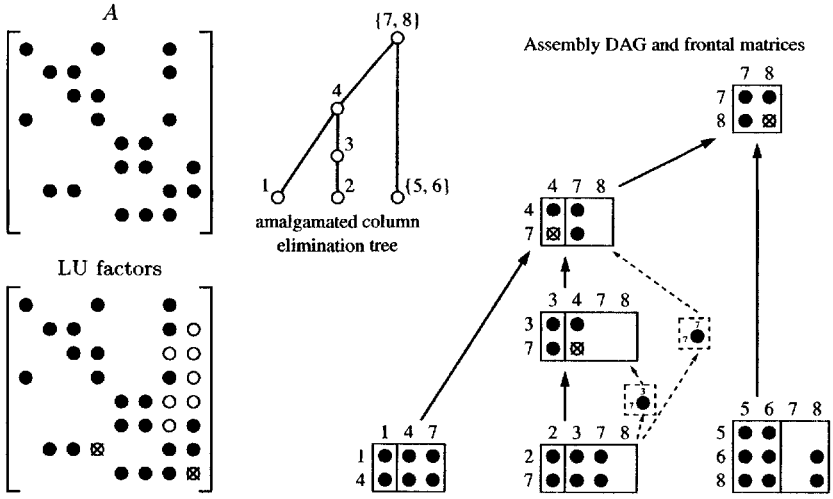


Figure 6.2. *Unsymmetric-pattern multifrontal example*

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

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

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

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

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

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

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

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

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

6.4 Further reading

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

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

Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This page intentionally left blank

Chapter 7
Fill-reducing orderings

The fill-minimization problem can be stated as follows. *Given a matrix A, find a row and column permutation P and Q (with the added constraint that $Q = P^T$ for a sparse Cholesky factorization) such that the number of nonzeros in the factorization of PAQ, or the amount of work required to compute it, are minimized.* The problem is impossible to solve in practice, so heuristics are used that attempt to reduce fill-in. Three basic strategies exist; they are sometimes combined to obtain hybrid strategies: (1) minimum degree and its variants (minimum fill, for example), (2) nested dissection (recursive graph partitioning), and (3) band reduction. The first strategy is most common and is presented in detail below. Features of the other two are highlighted. The Dulmage–Mendelsohn decomposition is a permutation that reduces the work required for LU and QR factorization. It consists of two primary steps: a permutation to obtain a zero-free diagonal and another to permute the matrix into block triangular form. The method is also useful for converting an edge separator into a node separator, which is used in many nested dissection methods.

7.1 Minimum degree ordering

The *minimum degree* algorithm is a widely used heuristic for finding a permutation P so that PAP^T has fewer nonzeros in its factorization than A. It is a greedy method that selects the sparsest pivot row and column during the course of a right-looking sparse Cholesky factorization (see Section 4.9). Consider the following MATLAB fragment, which does no pivoting:

```
for k = 1:n
    L (k,k) = sqrt (A (k,k)) ;
    L (k+1:n,k) = A (k+1:n,k) / L (k,k) ;
    A (k+1:n,k+1:n) = A (k+1:n,k+1:n) - L (k+1:n,k) * L (k+1:n,k)' ;
end
```

The kth step updates `A` with the outer product `L(:,k)*L(:,k)'`. Let $A^{[k]}$ denote the matrix `A(k:n,k:n)` at the start of the kth iteration, above.[11] Consider the

[11] This use of the $A^{[k]}$ notation differs from its use in Chapter 5, in which $A^{[k]} = H_k \ldots H_1 A$.

undirected graph of $A^{[k]}$, $G^{[k]} = (V, E)$ with nodes $V = \{k \ldots n\}$ and $(i, j) \in E$ if $a_{ij}^{[k]} \neq 0$. The graph $G^{[k]}$ is called the *elimination graph*. If $\mathcal{L}_k$ is the nonzero pattern of `L(:,k)`, the update to $G^{[k]}$ and $A^{[k]}$ corresponds to adding a dense submatrix to `A`, and adding a clique in $G^{[k]}$ and removing node k. Rather than representing $A^{[k]}$ as a simple graph, a *quotient graph* $\mathcal{G}^{[k]}$ represents this clique implicitly. Node k is replaced with *element* k with neighbors $\mathcal{L}_k$. The term "element" is borrowed from finite-element methods, since they too are a collection of dense submatrices or cliques in a graph. The adjacency list of an uneliminated node $i > k$ has two kinds of nodes adjacent to it: $\mathcal{A}_i$ is a list corresponding to original nonzeros a_{ij}, and $\mathcal{E}_i$ is a list of the elements adjacent to i. The nonzero pattern of row or column i is thus

$$\left(\mathcal{A}_i \cup \bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) \setminus \{i\}, \tag{7.1}$$

excluding the diagonal (no self-edges occur in G or $\mathcal{G}$). The degree d_i of node i is the size of the set (7.1). When node k is eliminated, any elements adjacent to it are no longer required to represent the nonzero pattern of $A^{[k]}$ (a consequence of Theorem 4.13); these elements can be removed (called *element absorption*). An example sequence of graphs G and quotient graphs $\mathcal{G}$ is given in Figure 7.1. In the graphs, a plain circle represents a node in $\mathcal{G}$, while a dark circle represents an element. In the matrices, a filled-in circle represents an edge between two nodes in $\mathcal{G}$, a circle is an edge no longer in G, and a circled x is an edge in G represented by an element in $\mathcal{G}$.

Additional terms in $\mathcal{G}$ can be pruned. If two nodes i and j are both in the pivotal element $\mathcal{L}_k$, then j and i can be removed from $\mathcal{A}_i$ and $\mathcal{A}_j$, respectively. They may have been adjacent due to an original entry a_{ij} and are still adjacent in $\mathcal{G}$ because they are both adjacent to element k. That is, $\mathcal{A}_i$ can be replaced with the smaller set $\mathcal{A}_i \setminus \mathcal{L}_k$ for all $i \in \mathcal{L}_k$ (referred to here as *pruning*). With element absorption and pruning of the $\mathcal{A}_i$ sets, the graph $\mathcal{G}$ can be represented in place (its size never exceeds $|A|$).

With this graph representation $\mathcal{G}$, the minimum degree algorithm consists simply of a greedy reordering of the nodes. Rather than selecting node k at the kth step, the algorithm selects the node with the least degree. When an element k is created, the degree of all nodes $i \in \mathcal{L}_k$ must be recomputed, using (7.1). This is the most costly part of the algorithm.

The cost can be reduced by exploiting supernodes, also called *indistinguishable nodes*. If two nodes i and j become identical ($\mathcal{E}_i = \mathcal{E}_j$ and $\mathcal{A}_i = \mathcal{A}_j$), they will remain identical until one of them is eliminated (either both are adjacent to the pivotal element or both are not adjacent). When one of them becomes the node of least degree, the other node is also a node of least degree, and eliminating both causes no more edges in $\mathcal{G}$ than when eliminating just one of them. Thus, if two nodes i and j are found to be indistinguishable, they can be merged into a single supernode that represents both of them. This is done by removing one of the nodes (j, say) and letting i be a representative of the supernode containing both i and j (j has been *absorbed* into i). The minimum degree algorithm selects a supernode k of least degree and eliminates it. All nodes start out simply representing themselves.

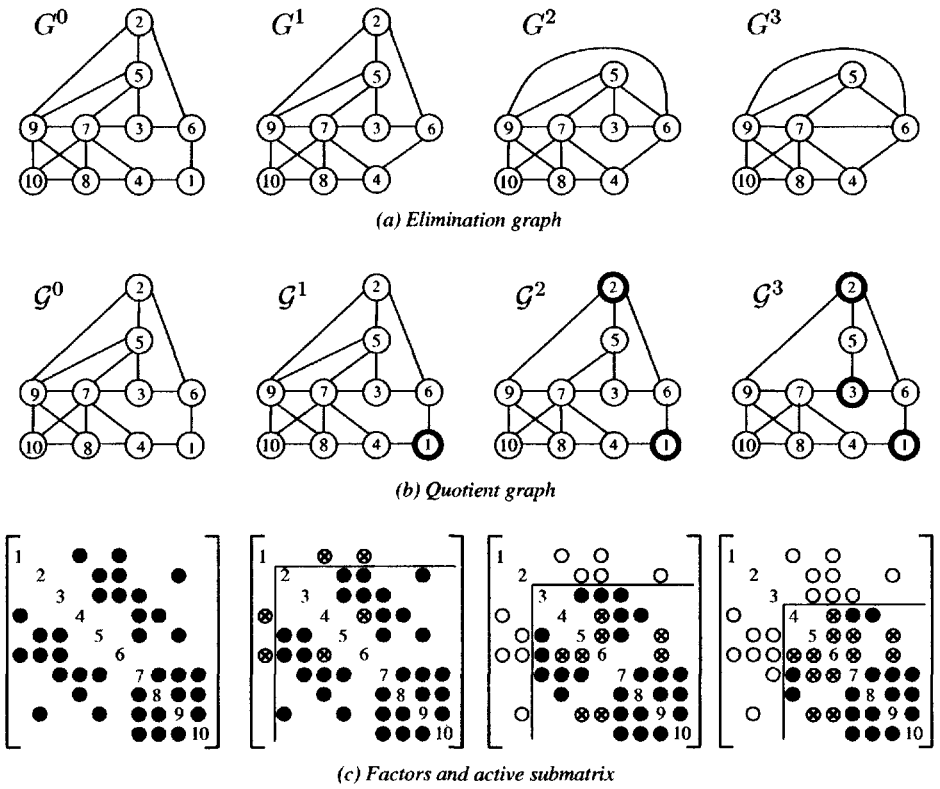


Figure 7.1. *Graph elimination*

After k is eliminated, if a node i is left with just an edge to k ($\mathcal{E}_i = \{k\}$ and $\mathcal{A}_i$ is empty), it can be eliminated immediately (called *mass elimination*). Let $|i|$ denote the number of nodes represented by supernode i. To keep the notation simple, when dealing with set expressions the use of i as a member of a set should be interpreted as the set of nodes represented by i.

Supernodes and mass elimination reduce the number of times (7.1) must be evaluated. Another technique discards the use of (7.1), and uses an approximation $\overline{d}_i$ to the true degree d_i instead, which is cheaper to compute, where

$$\overline{d}_i = |\mathcal{A}_i| + |\mathcal{L}_k \setminus \{i\}| + \sum_{e \in \mathcal{E}_i \setminus \{k\}} |\mathcal{L}_e \setminus \mathcal{L}_k| \tag{7.2}$$

and k is the current pivot element, and where $\mathcal{A}_i$ is assumed to have already been pruned. Note that $\overline{d}_i = d_i$ if $|\mathcal{E}_i| \leq 2$ (where $\mathcal{E}_i$ includes k), because $\mathcal{A}_i$ and $\mathcal{L}_k$ are disjoint after pruning. Otherwise $\overline{d}_i \geq d_i$. Using $\overline{d}$ in place of d results in the *approximate minimum degree* algorithm, or AMD. At first glance, (7.2) looks no simpler than computing the set (7.1) and then finding its size. The *scan1* algorithm

below shows how to compute the set differences $|\mathcal{L}_e \setminus \mathcal{L}_k|$ efficiently. In (7.2) and in the rest of the discussion, $|\mathcal{A}_i|$ and $|\mathcal{L}_e|$ refer to the sum of $|j|$ for the nodes j that they contain.

function *scan1*
  assume $w(e) < 0$ for all $e = 1, \ldots, n$
  for each node $i \in \mathcal{L}_k$ **do**
    for each element $e \in \mathcal{E}_i$ **do**
      if $(w(e) < 0)$ **then** $w(e) = |\mathcal{L}_e|$
      $w(e) = w(e) - |i|$

Then $w(e) = |\mathcal{L}_e \setminus \mathcal{L}_k|$ if $w(e) \geq 0$. If $w(e) < 0$, then the sets $\mathcal{L}_e$ and $\mathcal{L}_k$ are disjoint, and $|\mathcal{L}_e \setminus \mathcal{L}_k| = |\mathcal{L}_e|$. Once the set differences are known, a second pass over all $i \in \mathcal{L}_k$ evaluates (7.2) to compute $\overline{d}_i$. The amortized time for computing the set differences, computing $\overline{d}_i$, and pruning the set $\mathcal{A}_i$ is $O(|\mathcal{A}_i| + |\mathcal{E}_i|)$. This is much less than the $O(d_i)$ required to compute (7.1).

The minimum degree algorithm (AMD) is the most complex of the codes presented in this book. A concise version of AMD is presented below as the `cs_amd` function. It uses slightly more workspace than AMD, leading to a simpler code. It uses the tree postordering `cs_tdfs`, rather than AMD's own postordering. It has no control parameters, as AMD does, and does not compute any of the statistics that AMD does (such as $|L|$ and the floating-point operation count for a subsequent Cholesky factorization). It also has a simpler dense-node removal strategy. However, even with these simplifications, `cs_amd` generates orderings of the same quality as AMD and is just as fast.

Construct matrix `C`: The function accepts the matrix `A` as input and returns a permutation vector `p`. The `cs_amd` function operates on a symmetric matrix, so one of three symmetric matrices is formed. If `order` is `0`, a natural ordering `p=NULL` is returned. If `order` is `1` and the matrix is square, `C=A+A'` is formed, which is appropriate for a Cholesky factorization or an LU factorization of a matrix with substantial entries on the diagonal and a roughly symmetric nonzero pattern (using `tol<1` for `cs_lu`). If `order` is 2, `C=A'*A` is formed after removing "dense" rows from `A`. This is suitable for LU factorization of unsymmetric matrices and is similar to what COLAMD computes. If `order` is 3, `C=A'*A` is computed, which is best used for QR factorization or for LU factorization if `A` has no dense rows. A "dense" row is a row with more than $10\sqrt{n}$ entries.

Diagonal entries are removed from `C` (since `cs_amd` requires a graph with no self-edges), and extra "elbow room" is added to `C->i` via `cs_sprealloc`. The contents of `C` will be destroyed during the elimination (it holds $\mathcal{G}^{[k]}$). After `C` is formed, the output `p` and workspace of size $8(n+1)$ are allocated. The input `A` is not modified. To simplify the remainder of the discussion, the superscript $[k]$ is dropped.

Initialize quotient graph: The quotient graph $\mathcal{G}$ is represented by the arrays `Cp`, `Ci`, `w`, `nv`, `elen`, and `len`, and `degree`, each of size `n+1`, except for `Ci` of size `nzmax`. There are four kinds of nodes and elements that must be represented:

- A *live* node is a node `i` (or a supernode) that has not been selected as a

pivot and has not been merged into another supernode. In this case, $\mathcal{E}_i$ is represented as `Ci [Cp[i] ... Cp[i]+elen[i]-1]`, where `elen[i]` ≥ 0. The set $\mathcal{A}_i$ is represented as `Ci [Cp[i]+elen[i] ... Cp[i]+len[i]-1]`. Note that `Cp[i]` is greater than or equal to zero. The number of original nodes represented by `i` is given by $|i|$ = `nv[i]`, which is thus greater than zero. The degree $\overline{d}_i$ is `degree[i]`.

- A *dead* node `i` is one that has been removed from the graph, having been absorbed into node `r = CS_FLIP(Cp[i])`, where `CS_FLIP(x)` is defined as `-(x)-2`. Note that `Cp[i]` is less than zero. Note that node `r` might itself be absorbed into yet another node. In this case `Cp` forms an *assembly tree*, very similar to the elimination tree. The adjacency list of i is not stored. `elen[i]` is set to `-1` to denote the fact that node `i` is dead. The size of node `i`, $|i|$ = `nv[i]`, is zero.

- A live element `e` is one that is in the graph $\mathcal{G}$, having been formed when node `e` was selected as the pivot. `elen[e]` is set to `-2`, and `w[e]` will always be greater than zero. The sets $\mathcal{A}_e$ and $\mathcal{E}_e$ do not exist. Instead, the set $\mathcal{L}_e$ is stored in `Ci [Cp[e] ... Cp[e]+len[e]-1]`. `degree[e]` is $|\mathcal{L}_e|$, which is not the same as `len[e]`; the latter is smaller because $\mathcal{L}_e$ is a list of supernodes. The size of node `e`, $|e|$ = `nv[e]`, is greater than zero. It represents the number of nodes represented by supernode `e` when it was selected as the pivot.

- A dead element `e` is one that as been absorbed into a subsequent element `s = CS_FLIP(Cp[e])`. `elen[e]` is `-2` and `w[e]` is set to zero to denote that `e` is a dead element. $|e|$ = `nv[e]` > 0 is the same as for live elements.

`cs_amd` initializes the quotient graph $\mathcal{G}$ and two sets of `n` linked lists: the *degree lists* and the *hash buckets*. Degree list `d` is a doubly linked list containing a list of all nodes with approximate degree `d`. The head of list `d` is `head[d]`. The nodes preceding and following node `i` in the list are `last[i]` and `next[i]`, respectively. The hash buckets share some of this workspace; `hhead[h]` is the head of the `h`th hash bucket, a singly linked list. Since a node is never in both lists, `next[i]` is the node following `i` in the list, and `last[i]` is the hash key for node `i`. The degree lists are used to determine the node of minimum degree, and the hash buckets are used for supernode detection.

Initialize degree lists: Each node is placed in its degree lists. Nodes of zero degree are eliminated immediately. Nodes with degree $\geq$ `dense` are also eliminated and merged into a placeholder node `n`, a dead element. These dense nodes will appear last in the output permutation `p`.

Select node of minimum approximate degree: `cs_amd` is now ready to start eliminating the graph. It first finds a node `k` of minimum degree and removes it from its degree list. The variable `nel` keeps track of how many nodes have been eliminated; the elimination of `k` increments this count by $|k|$ = `nv[k]`. Because nodes are not eliminated in order `0` through `n-1`, this pivot node `k` is not equivalent to the `k` discussed above, but it serves the same purpose.

Garbage collection: The new element $\mathcal{L}_k$ requires space in `Ci`. It will be placed at the end of this array, in `Ci[cnz ... cnz +` $|\mathcal{L}_k| - 1$`]`, if $|\mathcal{E}_k| > 0$ (more

precisely, less space than this will be used; the exact degree $d_k = |\mathcal{L}_k|$ is an upper bound, and $\overline{d}_k \geq d_k$ is yet a higher upper bound). If not enough space is available, garbage collection is performed to pack $\mathcal{G}$ in the `Ci` array.

Live nodes and elements need not appear in any particular order in `Ci`. To pack `Ci` efficiently, the method relies on the fact that all entries in `Ci[0...cnz-1]` are nonnegative, and `Cp` will be redefined for live nodes and elements. The `for j` loop copies the first entry from each live node and element `j` into `Cp[j]` and places `CS_FLIP(j)` in the first position of each live object. The second loop scans all of `Ci`, looking for negative entries. When a negative entry is found, the live node or element `j` is compacted. Garbage collection occurs rarely.

Construct new element: The new element $\mathcal{L}_k$ is constructed, using (7.1). It is constructed in place if $|\mathcal{E}_k| = 0$. `nv[i]` is negated for all nodes `i` $\in \mathcal{L}_k$ to flag them as members of this set. Each node `i` is removed from the degree lists. All elements `e` $\in \mathcal{E}_k$ are absorbed into element k.

Find set differences: The *scan1* function now computes the set differences $|\mathcal{L}_e \setminus \mathcal{L}_k|$ for all elements e. At the start of the scan, no entry in the `w` array is greater than or equal to `mark`. A test is made to ensure `mark` + $\max |\mathcal{L}_e|$ does not cause integer overflow. If it does, then `w` is safely reset and the algorithm continues. The value of `mark` is used as an offset; $w(e)$ in the *scan1* pseudocode is replaced with `w[e]-mark` in the code.

Degree update: The second pass (*scan2*) computes the approximate degree $\overline{d}_i$ using (7.2), prunes the sets $\mathcal{E}_i$ and $\mathcal{A}_i$, and computes a hash function $h(i)$ for all nodes in $\mathcal{L}_k$. The hash function will be used in the next step for supernode detection. If a live element e is found where $|\mathcal{L}_e \setminus \mathcal{L}_k| = 0$, then *aggressive* element absorption is performed. Element e is a subset of k, so it is not needed to represent $\mathcal{G}$. At this point, `degree[i]` = `d` = $\overline{d}_i - |\mathcal{L}_k \setminus \{i\}|$ is computed for node `i` (7.2). The $|\mathcal{L}_k \setminus \{i\}|$ term is added later, after mass elimination and supernode detection. If `d` is zero, node `i` is mass eliminated along with element `k`; otherwise, node `i` remains alive. Element `k` is added to $\mathcal{E}_i$, and node `i` is placed in the `hth` hash bucket. Finally, `mark` is incremented by $\max |\mathcal{L}_e|$ to ensure that all entries in `w` are less than `mark`.

Supernode detection: Supernode detection relies on the hash function $h(i)$ computed for each node `i`. If two nodes have identical adjacency lists, their hash functions will be identical. Each hash bucket containing any node `i` $\in \mathcal{L}_k$ is considered. The first node `i` in the hash bucket is compared with all other nodes `j`; this is repeated until the hash bucket is empty. To compare `i` with a set of other nodes `j`, `w[s]=mark` is set for each node or element `s` in $\mathcal{A}_i$ or $\mathcal{E}_i$. These lists have been pruned in *scan2* of all dead nodes and elements. If the adjacent lists of `i` and `j` have the same lengths and all `s` in $\mathcal{A}_j$ or $\mathcal{E}_j$ are flagged, then `i` and `j` are identical. In this case, `j` is absorbed into `i` and removed from the hash bucket. The `mark` is incremented to clear the array `w` for the next iteration.

Finalize new element: The elimination of node `k` is nearly complete. All nodes `i` in $\mathcal{L}_k$ are scanned one last time. Node `i` is removed from $\mathcal{L}_k$ if it is dead (it may have been absorbed during supernode detection). The flagged status of `nv[i]` is cleared. The degree $\overline{d}_i$ is finalized, and node i is placed in its corresponding degree list. The new minimum degree is found when nodes are placed back into the degree lists. Note that the degree of the current element, `dk` = $|\mathcal{L}_k \setminus \{i\}|$, is finally added to

the degree of each node during this final pass to complete the approximate degree computation for (7.2). This term was not added in *scan2*, because it was modified during that scan due to mass elimination. Finally, `nv[k]` is updated to reflect the number of nodes represented by `k` (this may have increased, since `k` was selected as the pivot, due to mass elimination). If the set $\mathcal{L}_k$ is empty, element `k` is a root of the assembly tree, and element `k` is removed from the graph.

Postordering: The elimination is complete, but no permutation has been computed. All that is left of the graph is the assembly tree (`Cp`) and a set of dead nodes and elements (`i` is a dead node if `nv[i]` is zero and a dead element if `nv[i]` > 0). It is from this information only that the final permutation is computed. The tree is restored by unflipping all of `Cp`. It now forms a tree; `Cp[x]` is the parent of `x` or `-1` if `x` is a root. This is not the elimination tree, but it is quite similar.

If an element `e` has been absorbed into its parent `Cp[e]`, then `e` must precede `Cp[e]` in the output permutation `p`. Likewise, a node `i` must precede its parent `Cp[i]`. A distinction must be made between nodes and elements. The parent of an element is always an element. The parent of a node can be either another node or an element, but a node can never be a root of the tree. The children of an element `e` must appear before it in `p`, but all child elements must appear before all child nodes, because child nodes (and their descendants in the assembly tree) reflect a set of nodes that were absorbed into supernode `e` when it was selected as a pivot.

A postordering of the assembly tree gives the permutation `p`. The list of children of any node `x` are partitioned; the child elements appear first, followed by the child nodes. The dead element `n` is a placeholder for any dense rows and columns of `C`, so it too is included in the postordering; it and its descendants will be ordered at the very last, followed by `n=p[n]` itself. Thus, `p[0...n-1]` is the resulting fill-reducing permutation. This postordering is much simpler than the postordering in AMD, yet just as effective.

```
int *cs_amd (int order, const cs *A)  /* order 0:natural, 1:Chol, 2:LU, 3:QR */
{
    cs *C, *A2, *AT ;
    int *Cp, *Ci, *last, *W, *len, *nv, *next, *P, *head, *elen, *degree, *w,
        *hhead, *ATp, *ATi, d, dk, dext, lemax = 0, e, elenk, eln, i, j, k, k1,
        k2, k3, jlast, ln, dense, nzmax, mindeg = 0, nvi, nvj, nvk, mark, wnvi,
        ok, cnz, nel = 0, p, p1, p2, p3, p4, pj, pk, pk1, pk2, pn, q, n, m, t ;
    unsigned int h ;
    /* --- Construct matrix C ----------------------------------------------- */
    if (!CS_CSC (A) || order <= 0 || order > 3) return (NULL) ; /* check */
    AT = cs_transpose (A, 0) ;              /* compute A' */
    if (!AT) return (NULL) ;
    m = A->m ; n = A->n ;
    dense = CS_MAX (16, 10 * sqrt ((double) n)) ;   /* find dense threshold */
    dense = CS_MIN (n-2, dense) ;
    if (order == 1 && n == m)
    {
        C = cs_add (A, AT, 0, 0) ;          /* C = A+A' */
    }
    else if (order == 2)
    {
        ATp = AT->p ;                       /* drop dense columns from AT */
        ATi = AT->i ;
```

```
    for (p2 = 0, j = 0 ; j < m ; j++)
    {
        p = ATp [j] ;                   /* column j of AT starts here */
        ATp [j] = p2 ;                  /* new column j starts here */
        if (ATp [j+1] - p > dense) continue ;   /* skip dense col j */
        for ( ; p < ATp [j+1] ; p++) ATi [p2++] = ATi [p] ;
    }
    ATp [m] = p2 ;                      /* finalize AT */
    A2 = cs_transpose (AT, 0) ;         /* A2 = AT' */
    C = A2 ? cs_multiply (AT, A2) : NULL ;  /* C=A'*A with no dense rows */
    cs_spfree (A2) ;
}
else
{
    C = cs_multiply (AT, A) ;           /* C=A'*A */
}
cs_spfree (AT) ;
if (!C) return (NULL) ;
cs_fkeep (C, &cs_diag, NULL) ;          /* drop diagonal entries */
Cp = C->p ;
cnz = Cp [n] ;
P = cs_malloc (n+1, sizeof (int)) ;     /* allocate result */
W = cs_malloc (8*(n+1), sizeof (int)) ; /* get workspace */
t = cnz + cnz/5 + 2*n ;                 /* add elbow room to C */
if (!P || !W || !cs_sprealloc (C, t)) return (cs_idone (P, C, W, 0)) ;
len  = W           ; nv     = W +   (n+1) ; next   = W + 2*(n+1) ;
head = W + 3*(n+1) ; elen   = W + 4*(n+1) ; degree = W + 5*(n+1) ;
w    = W + 6*(n+1) ; hhead  = W + 7*(n+1) ;
last = P ;                              /* use P as workspace for last */
/* --- Initialize quotient graph ---------------------------------------- */
for (k = 0 ; k < n ; k++) len [k] = Cp [k+1] - Cp [k] ;
len [n] = 0 ;
nzmax = C->nzmax ;
Ci = C->i ;
for (i = 0 ; i <= n ; i++)
{
    head [i] = -1 ;                     /* degree list i is empty */
    last [i] = -1 ;
    next [i] = -1 ;
    hhead [i] = -1 ;                    /* hash list i is empty */
    nv [i] = 1 ;                        /* node i is just one node */
    w [i] = 1 ;                         /* node i is alive */
    elen [i] = 0 ;                      /* Ek of node i is empty */
    degree [i] = len [i] ;              /* degree of node i */
}
mark = cs_wclear (0, 0, w, n) ;         /* clear w */
elen [n] = -2 ;                         /* n is a dead element */
Cp [n] = -1 ;                           /* n is a root of assembly tree */
w [n] = 0 ;                             /* n is a dead element */
/* --- Initialize degree lists ------------------------------------------ */
for (i = 0 ; i < n ; i++)
{
    d = degree [i] ;
    if (d == 0)                         /* node i is empty */
    {
        elen [i] = -2 ;                 /* element i is dead */
        nel++ ;
```

```
            Cp [i] = -1 ;                   /* i is a root of assembly tree */
            w [i] = 0 ;
        }
        else if (d > dense)                 /* node i is dense */
        {
            nv [i] = 0 ;                    /* absorb i into element n */
            elen [i] = -1 ;                 /* node i is dead */
            nel++ ;
            Cp [i] = CS_FLIP (n) ;
            nv [n]++ ;
        }
        else
        {
            if (head [d] != -1) last [head [d]] = i ;
            next [i] = head [d] ;           /* put node i in degree list d */
            head [d] = i ;
        }
    }
    while (nel < n)                         /* while (selecting pivots) do */
    {
        /* --- Select node of minimum approximate degree -------------------- */
        for (k = -1 ; mindeg < n && (k = head [mindeg]) == -1 ; mindeg++) ;
        if (next [k] != -1) last [next [k]] = -1 ;
        head [mindeg] = next [k] ;          /* remove k from degree list */
        elenk = elen [k] ;                  /* elenk = |Ek| */
        nvk = nv [k] ;                      /* # of nodes k represents */
        nel += nvk ;                        /* nv[k] nodes of A eliminated */
        /* --- Garbage collection ------------------------------------------- */
        if (elenk > 0 && cnz + mindeg >= nzmax)
        {
            for (j = 0 ; j < n ; j++)
            {
                if ((p = Cp [j]) >= 0)      /* j is a live node or element */
                {
                    Cp [j] = Ci [p] ;       /* save first entry of object */
                    Ci [p] = CS_FLIP (j) ;  /* first entry is now CS_FLIP(j) */
                }
            }
            for (q = 0, p = 0 ; p < cnz ; ) /* scan all of memory */
            {
                if ((j = CS_FLIP (Ci [p++])) >= 0)  /* found object j */
                {
                    Ci [q] = Cp [j] ;       /* restore first entry of object */
                    Cp [j] = q++ ;          /* new pointer to object j */
                    for (k3 = 0 ; k3 < len [j]-1 ; k3++) Ci [q++] = Ci [p++] ;
                }
            }
            cnz = q ;                       /* Ci [cnz...nzmax-1] now free */
        }
        /* --- Construct new element ---------------------------------------- */
        dk = 0 ;
        nv [k] = -nvk ;                     /* flag k as in Lk */
        p = Cp [k] ;
        pk1 = (elenk == 0) ? p : cnz ;      /* do in place if elen[k] == 0 */
        pk2 = pk1 ;
        for (k1 = 1 ; k1 <= elenk + 1 ; k1++)
        {
```

```
    if (k1 > elenk)
    {
        e = k ;                     /* search the nodes in k */
        pj = p ;                    /* list of nodes starts at Ci[pj]*/
        ln = len [k] - elenk ;      /* length of list of nodes in k */
    }
    else
    {
        e = Ci [p++] ;              /* search the nodes in e */
        pj = Cp [e] ;
        ln = len [e] ;              /* length of list of nodes in e */
    }
    for (k2 = 1 ; k2 <= ln ; k2++)
    {
        i = Ci [pj++] ;
        if ((nvi = nv [i]) <= 0) continue ; /* node i dead, or seen */
        dk += nvi ;                 /* degree[Lk] += size of node i */
        nv [i] = -nvi ;             /* negate nv[i] to denote i in Lk*/
        Ci [pk2++] = i ;            /* place i in Lk */
        if (next [i] != -1) last [next [i]] = last [i] ;
        if (last [i] != -1)         /* remove i from degree list */
        {
            next [last [i]] = next [i] ;
        }
        else
        {
            head [degree [i]] = next [i] ;
        }
    }
    if (e != k)
    {
        Cp [e] = CS_FLIP (k) ;      /* absorb e into k */
        w [e] = 0 ;                 /* e is now a dead element */
    }
}
if (elenk != 0) cnz = pk2 ;         /* Ci [cnz...nzmax] is free */
degree [k] = dk ;                   /* external degree of k - |Lk\i| */
Cp [k] = pk1 ;                      /* element k is in Ci[pk1..pk2-1] */
len [k] = pk2 - pk1 ;
elen [k] = -2 ;                     /* k is now an element */
/* --- Find set differences ----------------------------------- */
mark = cs_wclear (mark, lemax, w, n) ;  /* clear w if necessary */
for (pk = pk1 ; pk < pk2 ; pk++)    /* scan 1: find |Le\Lk| */
{
    i = Ci [pk] ;
    if ((eln = elen [i]) <= 0) continue ;/* skip if elen[i] empty */
    nvi = -nv [i] ;                     /* nv [i] was negated */
    wnvi = mark - nvi ;
    for (p = Cp [i] ; p <= Cp [i] + eln - 1 ; p++)  /* scan Ei */
    {
        e = Ci [p] ;
        if (w [e] >= mark)
        {
            w [e] -= nvi ;          /* decrement |Le\Lk| */
        }
        else if (w [e] != 0)        /* ensure e is a live element */
        {
```

```
            w [e] = degree [e] + wnvi ; /* 1st time e seen in scan 1 */
        }
    }
}
/* --- Degree update ------------------------------------------------- */
for (pk = pk1 ; pk < pk2 ; pk++)    /* scan2: degree update */
{
    i = Ci [pk] ;                   /* consider node i in Lk */
    p1 = Cp [i] ;
    p2 = p1 + elen [i] - 1 ;
    pn = p1 ;
    for (h = 0, d = 0, p = p1 ; p <= p2 ; p++)    /* scan Ei */
    {
        e = Ci [p] ;
        if (w [e] != 0)             /* e is an unabsorbed element */
        {
            dext = w [e] - mark ;   /* dext = |Le\Lk| */
            if (dext > 0)
            {
                d += dext ;         /* sum up the set differences */
                Ci [pn++] = e ;     /* keep e in Ei */
                h += e ;            /* compute the hash of node i */
            }
            else
            {
                Cp [e] = CS_FLIP (k) ;  /* aggressive absorb. e->k */
                w [e] = 0 ;             /* e is a dead element */
            }
        }
    }
    elen [i] = pn - p1 + 1 ;        /* elen[i] = |Ei| */
    p3 = pn ;
    p4 = p1 + len [i] ;
    for (p = p2 + 1 ; p < p4 ; p++) /* prune edges in Ai */
    {
        j = Ci [p] ;
        if ((nvj = nv [j]) <= 0) continue ; /* node j dead or in Lk */
        d += nvj ;                  /* degree(i) += |j| */
        Ci [pn++] = j ;             /* place j in node list of i */
        h += j ;                    /* compute hash for node i */
    }
    if (d == 0)                     /* check for mass elimination */
    {
        Cp [i] = CS_FLIP (k) ;      /* absorb i into k */
        nvi = -nv [i] ;
        dk -= nvi ;                 /* |Lk| -= |i| */
        nvk += nvi ;                /* |k| += nv[i] */
        nel += nvi ;
        nv [i] = 0 ;
        elen [i] = -1 ;             /* node i is dead */
    }
    else
    {
        degree [i] = CS_MIN (degree [i], d) ;   /* update degree(i) */
        Ci [pn] = Ci [p3] ;         /* move first node to end */
        Ci [p3] = Ci [p1] ;         /* move 1st el. to end of Ei */
        Ci [p1] = k ;               /* add k as 1st element in of Ei */
```

```
        len [i] = pn - p1 + 1 ;     /* new len of adj. list of node i */
        h %= n ;                    /* finalize hash of i */
        next [i] = hhead [h] ;      /* place i in hash bucket */
        hhead [h] = i ;
        last [i] = h ;              /* save hash of i in last[i] */
    }
}                                   /* scan2 is done */
degree [k] = dk ;                   /* finalize |Lk| */
lemax = CS_MAX (lemax, dk) ;
mark = cs_wclear (mark+lemax, lemax, w, n) ;    /* clear w */
/* --- Supernode detection ------------------------------------------ */
for (pk = pk1 ; pk < pk2 ; pk++)
{
    i = Ci [pk] ;
    if (nv [i] >= 0) continue ;         /* skip if i is dead */
    h = last [i] ;                      /* scan hash bucket of node i */
    i = hhead [h] ;
    hhead [h] = -1 ;                    /* hash bucket will be empty */
    for ( ; i != -1 && next [i] != -1 ; i = next [i], mark++)
    {
        ln = len [i] ;
        eln = elen [i] ;
        for (p = Cp [i]+1 ; p <= Cp [i] + ln-1 ; p++) w [Ci [p]] = mark;
        jlast = i ;
        for (j = next [i] ; j != -1 ; ) /* compare i with all j */
        {
            ok = (len [j] == ln) && (elen [j] == eln) ;
            for (p = Cp [j] + 1 ; ok && p <= Cp [j] + ln - 1 ; p++)
            {
                if (w [Ci [p]] != mark) ok = 0 ;    /* compare i and j*/
            }
            if (ok)                     /* i and j are identical */
            {
                Cp [j] = CS_FLIP (i) ;  /* absorb j into i */
                nv [i] += nv [j] ;
                nv [j] = 0 ;
                elen [j] = -1 ;         /* node j is dead */
                j = next [j] ;          /* delete j from hash bucket */
                next [jlast] = j ;
            }
            else
            {
                jlast = j ;             /* j and i are different */
                j = next [j] ;
            }
        }
    }
}
/* --- Finalize new element------------------------------------------ */
for (p = pk1, pk = pk1 ; pk < pk2 ; pk++)   /* finalize Lk */
{
    i = Ci [pk] ;
    if ((nvi = -nv [i]) <= 0) continue ;/* skip if i is dead */
    nv [i] = nvi ;                      /* restore nv[i] */
    d = degree [i] + dk - nvi ;         /* compute external degree(i) */
    d = CS_MIN (d, n - nel - nvi) ;
    if (head [d] != -1) last [head [d]] = i ;
```

```
            next [i] = head [d] ;               /* put i back in degree list */
            last [i] = -1 ;
            head [d] = i ;
            mindeg = CS_MIN (mindeg, d) ;       /* find new minimum degree */
            degree [i] = d ;
            Ci [p++] = i ;                      /* place i in Lk */
        }
        nv [k] = nvk ;                      /* # nodes absorbed into k */
        if ((len [k] = p-pk1) == 0)         /* length of adj list of element k*/
        {
            Cp [k] = -1 ;                   /* k is a root of the tree */
            w [k] = 0 ;                     /* k is now a dead element */
        }
        if (elenk != 0) cnz = p ;           /* free unused space in Lk */
    }
    /* --- Postordering ----------------------------------------------------- */
    for (i = 0 ; i < n ; i++) Cp [i] = CS_FLIP (Cp [i]) ;/* fix assembly tree */
    for (j = 0 ; j <= n ; j++) head [j] = -1 ;
    for (j = n ; j >= 0 ; j--)              /* place unordered nodes in lists */
    {
        if (nv [j] > 0) continue ;          /* skip if j is an element */
        next [j] = head [Cp [j]] ;          /* place j in list of its parent */
        head [Cp [j]] = j ;
    }
    for (e = n ; e >= 0 ; e--)              /* place elements in lists */
    {
        if (nv [e] <= 0) continue ;         /* skip unless e is an element */
        if (Cp [e] != -1)
        {
            next [e] = head [Cp [e]] ;      /* place e in list of its parent */
            head [Cp [e]] = e ;
        }
    }
    for (k = 0, i = 0 ; i <= n ; i++)       /* postorder the assembly tree */
    {
        if (Cp [i] == -1) k = cs_tdfs (i, k, head, next, P, w) ;
    }
    return (cs_idone (P, C, W, 1)) ;
}
```

The `cs_wclear` function is used in `cs_amd` to clear the `w` array. The condition is true just once in the first call to `cs_wclear` and then when integer overflow is near (in which case `w` is safely reset and the algorithm continues). `cs_diag` is used to drop diagonal entries.

```
static int cs_wclear (int mark, int lemax, int *w, int n)
{
    int k ;
    if (mark < 2 || (mark + lemax < 0))
    {
        for (k = 0 ; k < n ; k++) if (w [k] != 0) w [k] = 1 ;
        mark = 2 ;
    }
    return (mark) ;     /* at this point, w [0..n-1] < mark holds */
}

static int cs_diag (int i, int j, double aij, void *other) { return (i != j) ; }
```

In MATLAB, `p=amd(A)`[12] or `p=symamd(A)` compute essentially the same ordering as `p=cs_amd(A)`. Slight differences exist due to *tie-breaking*. The input matrix `C` is computed differently in AMD and `cs_amd`; the adjacency lists are the same, but neither produce sorted adjacency lists $\mathcal{A}$. This affects the order that nodes are replaced in the degree lists, and thus the pivot order can be affected. There is often more than one node of identical minimum degree in the degree lists, and both AMD and `cs_amd` simply select the first node in the first nonempty degree list.

7.2 Maximum matching

A *maximum matching* (also called a *maximum transversal*) is a permutation of any matrix A so that its kth diagonal is zero-free and $|k|$ is uniquely minimized (except when A is completely zero). This permutation determines the structural rank of a matrix and is a precursor to LU or QR factorization or to the block triangular form and Dulmage–Mendelsohn decomposition described in the next sections. With this maximum matching, a matrix has structural full rank if and only if $k = 0$ and is structurally rank deficient otherwise. The number of entries on this diagonal gives the *structural rank* of a matrix A which is an upper bound on the numerical rank of any matrix with the same nonzero pattern as A. It is `sprank(A)` in MATLAB.

Consider the bipartite graph $G = (V, E)$ of an m-by-n matrix A with m row nodes, n column nodes, and undirected edges $E = \{(i, j) \mid a_{ij} \neq 0\}$; no edge connects pairs of row nodes or pairs of column nodes. Let $\mathcal{A}_j$ denote the nonzeros in column j or, equivalently, the rows adjacent to j in G. Note that although the edges are undirected, (i, j) and (j, i) are different edges. A *matching* is a set of rows $\mathcal{R}$ and columns $\mathcal{C}$ where each row in $i \in \mathcal{R}$ is paired with a unique $j \in \mathcal{C}$, where $(i, j) \in E$. A row $i \in \mathcal{R}$ is called a matched row, a column $j \in \mathcal{C}$ is called a matched column, and an edge (i, j) where both $i \in \mathcal{R}$ and $j \in \mathcal{C}$ is called a matched edge. All other rows, columns, and edges are unmatched. The matching defines a zero-free diagonal of a permuted submatrix (`diag(A(r,c))` in MATLAB notation). A maximum matching of G has a size greater than or equal to any other matching in G. A matching is *row-perfect* if all rows are matched and *column-perfect* if all columns are matched. A graph can have many maximum matchings (there are $n!$ maximum matchings in the graph of a square dense matrix). The size of this maximum matching is the structural rank. Fortunately, the algorithms and theorems presented here can use any maximum matching, and the matching algorithm presented below is guaranteed to find one of them.

The algorithm starts with an empty matching. Let `jmatch[i]` equal `j` if row $i \in \mathcal{R}$ is matched to column $j \in \mathcal{C}$ or `-1` if row $i \notin \mathcal{R}$. The algorithm works by extending the matching one column $j \notin \mathcal{C}$ at a time by finding an *alternating augmenting path*. The path starts at an unmatched column k and then traverses any edge to a row i_1. Since k is unmatched, this edge (k, i_1) is not matched. When reaching an unmatched row, the path stops. If i_1 is matched, the path traverses the

[12]The `amd` function is an internal part of MATLAB, but MATLAB 7.2 does not include an interface for it. It will likely be added to a future version of MATLAB. In the meantime, use `symamd` or use the version of `amd` at www.siam.org/books/fa02.

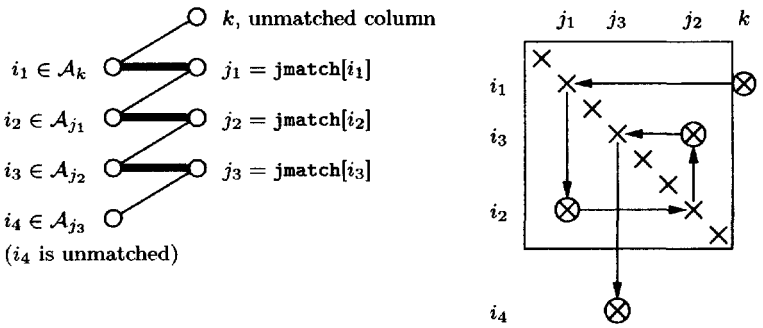


Figure 7.2. *An alternating augmenting path*

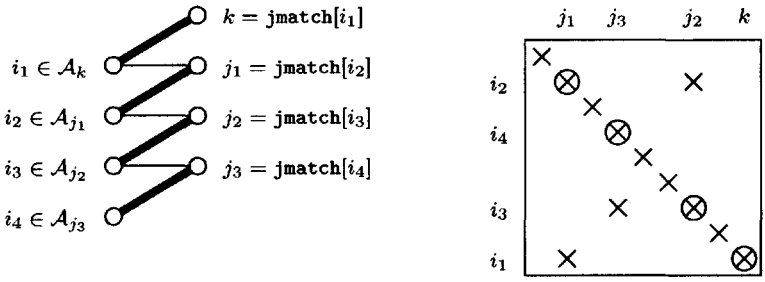


Figure 7.3. *Matching extended via an alternating augmenting path*

matched edge (i_1, j_1), where $j_1 =$ jmatch$[i_1]$, and then an unmatched edge (j_1, i_2). The path continues until it stops at an unmatched row. The path is *alternating* because every other edge in the path is matched. In general the path can be of any odd length, and no node or edge appears in the path twice. An alternating augmenting path of length 7 with matched edges shown in bold,

$$(k, i_1), (\mathbf{i_1}, \mathbf{j_1}), (j_1, i_2), (\mathbf{i_2}, \mathbf{j_2}), (j_2, i_3), (\mathbf{i_3}, \mathbf{j_3}), (j_3, i_4), \tag{7.3}$$

is shown in Figure 7.2. The figure also gives a matrix view of the same path. Matched edges in the graph are shown in bold; the same edges correspond to the diagonal elements of the permuted submatrix (drawn as a box). In the matrix view, entries corresponding to unmatched edges are circled. To extend the matching (as shown in Figure 7.3), k and i_4 are added to $\mathcal{C}$ and $\mathcal{R}$, respectively, and the matching along the path is changed so that the path becomes

$$(\mathbf{k}, \mathbf{i_1}), (i_1, j_1), (\mathbf{j_1}, \mathbf{i_2}), (i_2, j_2), (\mathbf{j_2}, \mathbf{i_3}), (i_3, j_3), (\mathbf{j_3}, \mathbf{i_4}). \tag{7.4}$$

That is, k = `jmatch`$[i_1]$, j_1 = `jmatch`$[i_2]$, and so on. The matching has been extended by one additional edge. Note that any matched row or column remains matched; k and i_4 are added to $\mathcal{C}$ and $\mathcal{R}$, respectively, and no nodes are removed from $\mathcal{C}$ or $\mathcal{R}$. If no unmatched row i_4 is found, no such path exists and the matching is not extended (this can occur only if A is structurally rank deficient). The modified graph and matrix are shown in Figure 7.3. The four entries circled in Figure 7.2 are still circled in Figure 7.3; the rows have been permuted to place them on the diagonal, and the box denoting the current match is one row and column larger. The three formerly matched entries are no longer on the diagonal. There can of course be other unmatched edges incident on these 8 nodes, and correspondingly more off-diagonal entries in the matrix. They are left out to make the figures clearer.

The path (7.3) can be found via a depth-first search of G, starting at an unmatched column k and traversing only alternating paths. If the whole graph is searched at every step, the time complexity is $O(|A|n)$ to find the entire maximum matching for a square matrix, but typically only a small part of G needs to be traversed before finding an alternating path (at which point the search stops). To reduce the typical cost of finding a path, a one-step breadth-first search is performed at each column j before continuing in a depth-first manner (called a *cheap match*). Once a row i is matched, it remains matched (although the column `jmatch`$[i]$ it is matched to may change). The breadth-first search exploits this fact by splitting $\mathcal{A}_j$ into two parts, the first of which contains only matched rows. When considering a column j, rows in the second part of $\mathcal{A}_j$ are considered, and the splitting is extended until an unmatched row (if any) is found. Thus, any edge is considered only once in this breadth-first search, adding only $O(|A|)$ to the time for finding the entire maximum matching but greatly reducing the average-case complexity of the algorithm.

The `maxtrans` and `augment` functions are not part of CSparse, since they rely on a recursive depth-first search, which can cause stack overflow for very large matrices. However, they are simpler to understand than the nonrecursive versions, so they are discussed first. `maxtrans` allocates workspace (`w` and `cheap`) and the result `jmatch`. Initially, all rows are unmatched. `w[j]` is used to mark column node `j` during a depth-first search; `w[j]=k` if column `j` has been visited during the `k`th step of the algorithm or `w[j]<k` otherwise. The splitting of $\mathcal{A}_j$ for the one-step breadth-first search is given by `cheap[j]`; `Ai[Ap[j] ... cheap[j]-1]` is known to contain only matched rows, whereas `Ai[cheap[j] ... Ap[j+1]-1]` may contain both matched and unmatched rows.

After these initializations, the one-line `for k` loop computes the matching. It searches for an alternating augmenting path starting at column `k` and augments the matching if this path is found. At the start of the `k`th iteration, $\mathcal{C}$ is a subset of $\{0 \ldots k-1\}$, and it may be extended by adding column k. When the algorithm completes, `j=jmatch[i]` $\geq$ `0` if row `i` is matched to column `j`. If row `i` is not matched, `jmatch[i]=-1`.

The recursive function `augment` is called by `maxtrans`, starting at node `j=k` (`j` will be modified when `augment` calls itself recursively, but `k` is kept unchanged). When at node `j`, it first attempts to find a cheap match (the first `for` loop). `cheap[j]` is modified to point to the remaining part of $\mathcal{A}_j$. If no cheap match

is found, all edges $i \in \mathcal{A}_j$ are considered in a depth-first manner. All of these rows i must be already matched; otherwise, a cheap match would have already been found and this loop would be skipped. If column jmatch[i] has not been considered during this kth step, a recursive call is made to find an augmenting path starting at column jmatch[i] (corresponding, for example, to column j_1 in Figure 7.2). The loop is terminated if a path is found, and the matching is revised by matching this new row i to column j.

```
int *maxtrans (cs *A)   /* returns jmatch [0..m-1] */
{
    int i, j, k, n, m, *Ap, *jmatch, *w, *cheap ;
    if (!A) return (NULL) ;                          /* check inputs */
    n = A->n ; m = A->m ; Ap = A->p ;
    jmatch = cs_malloc (m, sizeof (int)) ;           /* allocate result */
    w = cs_malloc (2*n, sizeof (int)) ;              /* allocate workspace */
    if (!w || !jmatch) return (cs_idone (jmatch, NULL, w, 0)) ;
    cheap = w + n ;
    for (j = 0 ; j < n ; j++) cheap [j] = Ap [j] ;   /* for cheap assignment */
    for (j = 0 ; j < n ; j++) w [j] = -1 ;           /* all columns unflagged */
    for (i = 0 ; i < m ; i++) jmatch [i] = -1 ;      /* no rows matched yet */
    for (k = 0 ; k < n ; k++) augment (k, A, jmatch, cheap, w, k) ;
    return (cs_idone (jmatch, NULL, w, 1)) ;
}

int augment (int k, cs *A, int *jmatch, int *cheap, int *w, int j)
{
    int found = 0, p, i = -1, *Ap = A->p, *Ai = A->i ;
    /* --- Start depth-first-search at node j ------------------------------ */
    w [j] = k ;                                      /* mark j as visited for kth path */
    for (p = cheap [j] ; p < Ap [j+1] && !found ; p++)
    {
        i = Ai [p] ;                                 /* try a cheap assignment (i,j) */
        found = (jmatch [i] == -1) ;
    }
    cheap [j] = p ;                                  /* start here next time for j */
    /* --- Depth-first-search of neighbors of j ---------------------------- */
    for (p = Ap [j] ; p < Ap [j+1] && !found ; p++)
    {
        i = Ai [p] ;                                 /* consider row i */
        if (w [jmatch [i]] == k) continue ; /* skip col jmatch [i] if marked */
        found = augment (k, A, jmatch, cheap, w, jmatch [i]) ;
    }
    if (found) jmatch [i] = j ;                      /* augment jmatch if path found */
    return (found) ;
}
```

Just as with the recursive depth-first search algorithm dfsr described in Section 3.2, using recursion limits augment to solving problems of modest size. The nonrecursive cs_maxtrans below is essentially the same as maxtrans; it simply allocates more workspace and calls the nonrecursive cs_augment instead. The extra workspace is a stack that holds the intermediate values of the variables j, i, and p in the recursive augment. The algorithm is more efficient than maxtrans for a matrix A with more rows than columns (it transposes the matrix in this case). It also returns quickly if the diagonal is already zero-free.

The cs_augment function does the same thing as augment(k, ..., k). First, the node j is placed on the stack js. The while loop continues until either the stack is empty or the last node in an augmenting path is found.

```
int *cs_maxtrans (const cs *A, int seed)  /*[jmatch [0..m-1]; imatch [0..n-1]]*/
{
    int i, j, k, n, m, p, n2 = 0, m2 = 0, *Ap, *jimatch, *w, *cheap, *js, *is,
        *ps, *Ai, *Cp, *jmatch, *imatch, *q ;
    cs *C ;
    if (!CS_CSC (A)) return (NULL) ;                /* check inputs */
    n = A->n ; m = A->m ; Ap = A->p ; Ai = A->i ;
    w = jimatch = cs_calloc (m+n, sizeof (int)) ;   /* allocate result */
    if (!jimatch) return (NULL) ;
    for (k = 0, j = 0 ; j < n ; j++)    /* count nonempty rows and columns */
    {
        n2 += (Ap [j] < Ap [j+1]) ;
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            w [Ai [p]] = 1 ;
            k += (j == Ai [p]) ;        /* count entries already on diagonal */
        }
    }
    if (k == CS_MIN (m,n))              /* quick return if diagonal zero-free */
    {
        jmatch = jimatch ; imatch = jimatch + m ;
        for (i = 0 ; i < k ; i++) jmatch [i] = i ;
        for (      ; i < m ; i++) jmatch [i] = -1 ;
        for (j = 0 ; j < k ; j++) imatch [j] = j ;
        for (      ; j < n ; j++) imatch [j] = -1 ;
        return (cs_idone (jimatch, NULL, NULL, 1)) ;
    }
    for (i = 0 ; i < m ; i++) m2 += w [i] ;
    C = (m2 < n2) ? cs_transpose (A,0) : ((cs *) A) ; /* transpose if needed */
    if (!C) return (cs_idone (jimatch, (m2 < n2) ? C : NULL, NULL, 0)) ;
    n = C->n ; m = C->m ; Cp = C->p ;
    jmatch = (m2 < n2) ? jimatch + n : jimatch ;
    imatch = (m2 < n2) ? jimatch : jimatch + m ;
    w = cs_malloc (5*n, sizeof (int)) ;             /* get workspace */
    if (!w) return (cs_idone (jimatch, (m2 < n2) ? C : NULL, w, 0)) ;
    cheap = w + n ; js = w + 2*n ; is = w + 3*n ; ps = w + 4*n ;
    for (j = 0 ; j < n ; j++) cheap [j] = Cp [j] ;  /* for cheap assignment */
    for (j = 0 ; j < n ; j++) w [j] = -1 ;          /* all columns unflagged */
    for (i = 0 ; i < m ; i++) jmatch [i] = -1 ;     /* nothing matched yet */
    q = cs_randperm (n, seed) ;                     /* q = random permutation */
    for (k = 0 ; k < n ; k++)   /* augment, starting at column q[k] */
    {
        cs_augment (q ? q [k]: k, C, jmatch, cheap, w, js, is, ps) ;
    }
    cs_free (q) ;
    for (j = 0 ; j < n ; j++) imatch [j] = -1 ;     /* find row match */
    for (i = 0 ; i < m ; i++) if (jmatch [i] >= 0) imatch [jmatch [i]] = i ;
    return (cs_idone (jimatch, (m2 < n2) ? C : NULL, w, 1)) ;
}
```

```
static void cs_augment (int k, const cs *A, int *jmatch, int *cheap, int *w,
        int *js, int *is, int *ps)
{
    int found = 0, p, i = -1, *Ap = A->p, *Ai = A->i, head = 0, j ;
    js [0] = k ;                        /* start with just node k in jstack */
    while (head >= 0)
    {
        /* --- Start (or continue) depth-first-search at node j ------------ */
        j = js [head] ;                 /* get j from top of jstack */
        if (w [j] != k)                 /* 1st time j visited for kth path */
        {
            w [j] = k ;                 /* mark j as visited for kth path */
            for (p = cheap [j] ; p < Ap [j+1] && !found ; p++)
            {
                i = Ai [p] ;            /* try a cheap assignment (i,j) */
                found = (jmatch [i] == -1) ;
            }
            cheap [j] = p ;             /* start here next time j is traversed*/
            if (found)
            {
                is [head] = i ;         /* column j matched with row i */
                break ;                 /* end of augmenting path */
            }
            ps [head] = Ap [j] ;        /* no cheap match: start dfs for j */
        }
        /* --- Depth-first-search of neighbors of j ------------------------- */
        for (p = ps [head] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;                /* consider row i */
            if (w [jmatch [i]] == k) continue ; /* skip jmatch [i] if marked */
            ps [head] = p + 1 ;         /* pause dfs of node j */
            is [head] = i ;             /* i will be matched with j if found */
            js [++head] = jmatch [i] ;  /* start dfs at column jmatch [i] */
            break ;
        }
        if (p == Ap [j+1]) head-- ;     /* node j is done; pop from stack */
    }                                   /* augment the match if path found: */
    if (found) for (p = head ; p >= 0 ; p--) jmatch [is [p]] = js [p] ;
}
```

Start (or continue) depth-first search of node j: At each iteration of the `while` loop in `cs_augment`, the node `j` at the top of the stack is considered. If this is the first time `j` is considered, `w[j]` will not be equal to `k`; this corresponds to the start of a call to `augment(k,...,j)`. Thus, the next seven lines of code (from `w[j]=k` to `cheap[j]=p`) are identical to the same seven lines in `augment`. If a cheap match is found, this is recorded in the stack `is`, and the "recursion" will start to unwind (the `while` loop terminates). Otherwise, the starting position `p=Ap[j]` in $\mathcal{A}_j$ for the depth-first part is saved in `ps`, corresponding to the initialization of the `for (p=Ap[j] ...)` loop in `augment`.

Depth-first search of neighbors of j: The `for (p=ps[head] ...)` loop starts (or continues) the corresponding `for (p=Ap[j] ...)` loop in `augment`. If an unmarked column `jmatch[i]` is found, the iteration for node `j` is paused, and the new column `jmatch[i]` is now at the top of the stack. The `break` statement terminates the `for` loop so that the outermost `while` loop can consider this node

`jmatch[i]` at the top of the stack. If the `for` loop terminates after searching all rows $i \in A_j$, then no match is found (yet), and `j` is popped from the stack by decrementing `head`. Finally, if an augmenting path is found the "recursion" unwinds by revising the matching for all unmatched edges `(i,j)` in this path, corresponding to the `jmatch[i]=j` statement in `augment`.

If a column-perfect matching is found, `imatch[0...n-1]` is a permutation of a subset of the columns of `A` (or all of the columns if `A` is square), and `A(imatch,:)` has a zero-free diagonal. The MATLAB statement `p=dmperm(A)` is identical (that is, `p` is `imatch`).

The worst-case time complexity of `cs_maxtrans` is $O(|A|n)$, but this rarely occurs in practice. `cs_maxtrans` can match the columns of A in reverse order (from `n-1` to `0`), or in a randomized order, which can help avoid this worst-case behavior. The `cs_randperm` computes a random permutation used by `cs_maxtrans`. If `seed` is zero, the identity permutation is returned (`p=NULL`). If `seed` is `-1`, the reverse permutation is returned (`p=n-1:-1:0` in MATLAB notation). Otherwise, a random permtutation is returned.

```
int *cs_randperm (int n, int seed)
{
    int *p, k, j, t ;
    if (seed == 0) return (NULL) ;      /* return p = NULL (identity) */
    p = cs_malloc (n, sizeof (int)) ;   /* allocate result */
    if (!p) return (NULL) ;             /* out of memory */
    for (k = 0 ; k < n ; k++) p [k] = n-k-1 ;
    if (seed == -1) return (p) ;        /* return reverse permutation */
    srand (seed) ;                      /* get new random number seed */
    for (k = 0 ; k < n ; k++)
    {
        j = k + (rand ( ) % (n-k)) ;    /* j = rand int in range k to n-1 */
        t = p [j] ;                     /* swap p[k] and p[j] */
        p [j] = p [k] ;
        p [k] = t ;
    }
    return (p) ;
}
```

7.3 Block triangular form

The *block triangular form* and its generalization the Dulmage–Mendelsohn decomposition (described in the next section) is a useful tool for many sparse matrix algorithms and theorems. It is a permutation of a matrix A that reduces the work required for LU and QR factorization and provides a precise characterization of structurally rank-deficient matrices.

An m-by-n matrix A has the *strong Hall* [13] property if every set of k columns has nonzero entries in at least $k+1$ rows for all k in the range 1 to $n-1$. If A is square and has structural full rank but is not strong Hall, it can be permuted to

[13] An m-by-n matrix A has the *Hall* property if every set of k columns has nonzero entries in at least k rows for all k in the range 1 to n. Equivalently, A has the Hall property if and only if it has full structural column rank.

block triangular form,

$$PAQ = \begin{bmatrix} A_{11} & \cdots & A_{1k} \\ & \ddots & \vdots \\ & & A_{kk} \end{bmatrix}, \tag{7.5}$$

where each diagonal block is square with a zero-free diagonal and has the strong Hall property. The strong Hall property implies full structural rank. The block triangular form (7.5) is unique, except that the blocks can sometimes be interchanged. There is often a choice of ordering within the blocks (the diagonal must remain zero-free). To solve $Ax = b$ with LU factorization, only the diagonal blocks need to be factorized, followed by a block backsolve for the off-diagonal blocks. No fill-in occurs in the off-diagonal blocks. Because each diagonal block is strong Hall, the theorems in Chapters 5 and 6 provide tighter bounds on the nonzero pattern of the factors.

The inverse of a strong Hall matrix has no zero entries (ignoring numerical cancellation), and thus should very rarely be computed in practice.

Permuting a square matrix with a zero-free diagonal into block triangular form is identical to finding the *strongly connected components* of a directed graph, $G(A)$. The directed graph is defined as $G(A) = (V, E)$, where $V = \{1, \ldots, n\}$ and $E = \{(i,j) \,|\, a_{ij} \neq 0\}$. That is, the nonzero pattern of A is the adjacency matrix of the directed graph $G(A)$. A strongly connected component is a maximal set of nodes such that for any pair of nodes i and j in the component, the paths $i \rightsquigarrow j$ and $j \rightsquigarrow i$ both exist in the graph.

The strongly connected components of a graph can be found in many ways. The simplest method uses two depth-first traversals, one of $G(A)$ and the second of the graph $G(A^T)$. This is simple in CSparse, because `cs_dfs` can already perform this depth-first traversal. It was presented in the context of a directed acyclic graph (the graph of L) to find the nonzero pattern $\mathcal{X} = \text{Reach}_L(\mathcal{B})$ for the sparse triangular solve in Section 3.2, but nothing in the design of `cs_dfs` limits it to acyclic graphs. In general, the graph of A can have cycles, unlike the graph of L.

The first depth-first search returns a set $\mathcal{X}$ that contains all the nodes of the graph. As a set, this is not very interesting. However, the *order* in which nodes appear in $\mathcal{X}$ is very important. A node j is placed in the stack $\mathcal{X}$ in the order in which its corresponding $dfs(j)$ finishes. A second depth-first traversal of $G(A^T)$, where nodes are considered in the reverse order of their finish times (from the top of the stack $\mathcal{X}$ to the bottom), determines the strongly connected components. Every new node i found in a new depth-first search in the second pass, and all nodes reachable from it in $G(A^T)$, define a unique strongly connected component of $G(A)$, denoted as C_b. The algorithm and its implementation (`cs_scc`) are given below. The components are actually computed in reverse order; this detail is not included in the *scc* function below. See Section 7.7 for more details on why it works.

Since A is stored in compressed-column form, $\mathcal{A}_j$ is the adjacency list of node j in the graph $G(A^T)$. However, the *scc* algorithm will find a permutation that puts the adjacency matrix of the graph in block lower triangular form. A block upper triangular form is more conventional for sparse matrix computations, so `cs_scc` can be applied to the transpose. These two transposes cancel each other, so the

cs_scc algorithm is identical to the pseudocode *scc*. It finds a permutation vector p such that C=A(p,p) is in block upper triangular form (7.5) and an array r that determines where the blocks are in C; the kth block is C(k1:k2,k1:k2) in MATLAB notation, where k1=r[k] and k2=r[k+1]-1.

function *scc*(A)
  $\mathcal{X} = \text{Reach}_A(\{1 \ldots n\})$
  $b = 0$
  for each node $i \in \mathcal{X}$
    if i is unmarked
      $b = b + 1$
      $\mathcal{C}_b$ =*dfs*(i) of the graph $G(A^T)$

```
csd *cs_scc (cs *A)     /* matrix A temporarily modified, then restored */
{
    int n, i, k, b, nb = 0, top, *xi, *pstack, *p, *r, *Ap, *ATp, *rcopy, *Blk ;
    cs *AT ;
    csd *D ;
    if (!CS_CSC (A)) return (NULL) ;                /* check inputs */
    n = A->n ; Ap = A->p ;
    D = cs_dalloc (n, 0) ;                          /* allocate result */
    AT = cs_transpose (A, 0) ;                      /* AT = A' */
    xi = cs_malloc (2*n+1, sizeof (int)) ;          /* get workspace */
    if (!D || !AT || !xi) return (cs_ddone (D, AT, xi, 0)) ;
    Blk = xi ; rcopy = pstack = xi + n ;
    p = D->p ; r = D->r ; ATp = AT->p ;
    top = n ;
    for (i = 0 ; i < n ; i++)   /* first dfs(A) to find finish times (xi) */
    {
        if (!CS_MARKED (Ap, i)) top = cs_dfs (i, A, top, xi, pstack, NULL) ;
    }
    for (i = 0 ; i < n ; i++) CS_MARK (Ap, i) ; /* restore A; unmark all nodes*/
    top = n ;
    nb = n ;
    for (k = 0 ; k < n ; k++)   /* dfs(A') to find strongly connnected comp */
    {
        i = xi [k] ;            /* get i in reverse order of finish times */
        if (CS_MARKED (ATp, i)) continue ;  /* skip node i if already ordered */
        r [nb--] = top ;        /* node i is the start of a component in p */
        top = cs_dfs (i, AT, top, p, pstack, NULL) ;
    }
    r [nb] = 0 ;                /* first block starts at zero; shift r up */
    for (k = nb ; k <= n ; k++) r [k-nb] = r [k] ;
    D->nb = nb = n-nb ;         /* nb = # of strongly connected components */
    for (b = 0 ; b < nb ; b++)  /* sort each block in natural order */
    {
        for (k = r [b] ; k < r [b+1] ; k++) Blk [p [k]] = b ;
    }
    for (b = 0 ; b <= nb ; b++) rcopy [b] = r [b] ;
    for (i = 0 ; i < n ; i++) p [rcopy [Blk [i]]++] = i ;
    return (cs_ddone (D, AT, xi, 1)) ;
}
```

The last part of the cs_scc function sorts the permutation vector p in linear time ($O(n)$), so that the rows and columns in each block appear in their natural order. This is not essential but useful because a subsequent fill-reducing ordering algorithm can tend to give slightly better results if it is provided with the matrix in natural order, $P = I$ if the matrix consists of a single strongly connected component, and cs_dmspy looks prettier in MATLAB.

The cs_dalloc, cs_dfree, and cs_ddone functions allocate, free, and return a csd object that represents the strongly connected components found by cs_scc.

```
typedef struct cs_dmperm_results    /* cs_dmperm or cs_scc output */
{
    int *p ;        /* size m, row permutation */
    int *q ;        /* size n, column permutation */
    int *r ;        /* size nb+1, block k is rows r[k] to r[k+1]-1 in A(p,q) */
    int *s ;        /* size nb+1, block k is cols s[k] to s[k+1]-1 in A(p,q) */
    int nb ;        /* # of blocks in fine dmperm decomposition */
    int rr [5] ;    /* coarse row decomposition */
    int cc [5] ;    /* coarse column decomposition */
} csd ;


csd *cs_dalloc (int m, int n)
{
    csd *D ;
    D = cs_calloc (1, sizeof (csd)) ;
    if (!D) return (NULL) ;
    D->p = cs_malloc (m, sizeof (int)) ;
    D->r = cs_malloc (m+6, sizeof (int)) ;
    D->q = cs_malloc (n, sizeof (int)) ;
    D->s = cs_malloc (n+6, sizeof (int)) ;
    return ((!D->p || !D->r || !D->q || !D->s) ? cs_dfree (D) : D) ;
}


csd *cs_dfree (csd *D)
{
    if (!D) return (NULL) ;     /* do nothing if D already NULL */
    cs_free (D->p) ;
    cs_free (D->q) ;
    cs_free (D->r) ;
    cs_free (D->s) ;
    return (cs_free (D)) ;
}


csd *cs_ddone (csd *D, cs *C, void *w, int ok)
{
    cs_spfree (C) ;                     /* free temporary matrix */
    cs_free (w) ;                       /* free workspace */
    return (ok ? D : cs_dfree (D)) ;    /* return result if OK, else free it */
}
```

In MATLAB, [p,q,r,s]=dmperm(A) applied to a square matrix A with a zero-free diagonal will return the same block triangular form A(p,p) (where p=q and r=s), subject to the nonuniqueness of this form, as described above.

7.4 Dulmage–Mendelsohn decomposition

The *Dulmage–Mendelsohn decomposition* is a generalization of (7.5). Suppose a maximum matching of A has been found (A need not be square). Let $\mathcal{R}$ be the set of matched rows and $\mathcal{C}$ be the set of matched columns. This matching partitions the rows and columns of A into four disjoint subsets:

$\overline{\mathcal{R}}$	the set of unmatched rows
$\overline{\mathcal{C}}$	the set of unmatched columns
$\mathcal{R}_1$	the set of all rows reachable from any column in $\overline{\mathcal{C}}$ via an alternating path
$\mathcal{C}_1$	the columns matched to $\mathcal{R}_1$
$\mathcal{C}_3$	the set of all columns reachable from any row in $\overline{\mathcal{R}}$ via an alternating path
$\mathcal{R}_3$	the rows matched to $\mathcal{C}_3$
$\mathcal{R}_2$	any row not in any of the above sets
$\mathcal{C}_2$	any column not in any of the above sets

Note that all rows in $\mathcal{R}_1$ are matched; if they were not, an alternating augmenting path could be found, extending the maximum matching (which is a contradiction). Likewise, all columns in $\mathcal{C}_3$ are matched. Also note that if $\mathcal{C}_1$ and $\mathcal{C}_3$ had a column in common, there would be an alternating augmenting path from $\overline{\mathcal{R}}$ to $\overline{\mathcal{C}}$ through that column. Similarly, $\mathcal{R}_1$ and $\mathcal{R}_3$ are disjoint. All rows in $\mathcal{R}_2$ are matched to some row in $\mathcal{C}_2$. Thus, $\mathcal{R}$ is divided into three disjoint subsets $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$, and $\mathcal{C}$ is divided into three disjoint subsets $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$. Given this four-way partition of the rows and columns, *any* matrix A can be permuted into the 4-by-4 block matrix

$$PAQ = \begin{array}{c} \begin{array}{cccc} \overline{\mathcal{C}} & \mathcal{C}_1 & \mathcal{C}_2 & \mathcal{C}_3 \end{array} \\ \left[\begin{array}{cccc} A_{11} & A_{12} & A_{13} & A_{14} \\ & & A_{23} & A_{24} \\ & & & A_{34} \\ & & & A_{44} \end{array}\right] \end{array} \begin{array}{l} \mathcal{R}_1 \\ \mathcal{R}_2 \\ \mathcal{R}_3 \\ \overline{\mathcal{R}} \end{array}, \tag{7.6}$$

where A_{12}, A_{23}, and A_{34} are square with a zero-free diagonal. The transpose of the matrix

$$[A_{11}A_{12}] \tag{7.7}$$

and the matrix

$$\begin{bmatrix} A_{34} \\ A_{44} \end{bmatrix} \tag{7.8}$$

are both rectangular and have the strong Hall property. The matrix (7.7) has a perfect row-matching, and the matrix (7.8) has a perfect column-matching. If $\overline{\mathcal{C}}$ is empty, the matrix A has a column-perfect matching, and both $\mathcal{R}_1$ and $\mathcal{C}_1$ will be empty. Likewise, if $\overline{\mathcal{R}}$ is empty, the matrix A has a row-perfect matching, and both $\mathcal{R}_3$ and $\mathcal{C}_3$ will be empty. Thus, it is possible for the two matrices (7.7) and (7.8) to be empty (with no rows and columns). If they do exist, (7.7) will have more columns than rows, corresponding to the structurally underdetermined part of the system $Ax = b$, and (7.8) will have more rows than columns, corresponding to the structurally overdetermined part of the system $Ax = b$. The matrix A_{23} need not have the strong Hall property. If it does not, it can be permuted into block upper

triangular form, as described in Section 7.3. It has structural full rank because it is square with a zero-free diagonal. Thus, for any matrix A, LU or QR factorization can be applied to submatrices, all of which have the strong Hall property.

The permutation and partitioning of A given in (7.6) is unique, except that a different maximum matching can swap columns between $\overline{\mathcal{C}}$ and $\mathcal{C}_1$ and can swap rows between $\overline{\mathcal{R}}$ and $\mathcal{R}_3$ (but not arbitrarily; $\mathcal{C}_1$ must still be matchable to the set $\mathcal{R}_1$). Otherwise the eight sets, and their sizes, are unique. The row or column ordering within each of the eight sets is not unique in general.

The `cs_dmperm` function computes the Dulmage–Mendelsohn decomposition. It returns a `csd` object containing the row and column permutation vectors `p` and `q`. The four subsets of these permutation vectors are given by `cc` and `rr`; this determines the *coarse* decomposition, given in (7.6). The eight sets are given by

$\overline{\mathcal{C}}$	`q[0 ... cc[1]-1]`		$\mathcal{R}_1$	`p[0 ... rr[1]-1]`	
$\mathcal{C}_1$	`q[cc[1] ...`	`cc[2]-1]`	$\mathcal{R}_2$	`p[rr[1] ...`	`rr[2]-1]`
$\mathcal{C}_2$	`q[cc[2] ...`	`cc[3]-1]`	$\mathcal{R}_3$	`p[rr[2] ...`	`rr[3]-1]`
$\mathcal{C}_3$	`q[cc[3] ...`	`cc[4]-1]`	$\overline{\mathcal{R}}$	`p[rr[3] ...`	`rr[4]-1]`

The *fine* decomposition includes the permutation of the A_{23} submatrix into its strongly connected components, (7.5). It is given by `r` and `s`. If `C=A(p,q)`, the `k`th block consists of rows `r[k]` through `r[k+1]-1` and columns `s[k]` through `s[k+1]-1` of `C`. The first block is the rectangular matrix (7.7) and the last block is (7.8) if they are not 0-by-0. The middle blocks are the strongly connected components of A_{23}. Note that (7.7) can have columns but no rows (A_{11} is 0-by-`cc[1]` and A_{12} is 0-by-0). Similarly, (7.8) can have rows but no columns.

Maximum matching: `cs_dmperm` finds the maximum matching `jmatch[i]=j` and its inverse `imatch[j]=i`.

Coarse decomposition: A breadth-first search starting from unmatched columns $\overline{\mathcal{C}}$ finds $\mathcal{C}_1$ and $\mathcal{R}_1$. Using the matrix A^T, another breadth-first search starting from rows in $\overline{\mathcal{R}}$ determines $\mathcal{C}_3$ and $\mathcal{R}_3$. In the code, $\overline{\mathcal{C}}$ and $\overline{\mathcal{R}}$ are `C0` and `R0`, respectively. At this point, the coarse decomposition is determined solely by the flag arrays `wi` and `wj` and the matching `jmatch` and `imatch`. These arrays are scanned to find the permutations `p` and `q` and the coarse set sizes `cc` and `rr`.

Fine decomposition: The strongly connected components of the `A(R2,C2)` submatrix are found. The `A(R2,C2)` matrix is formed by first computing `C=A(p,q)` and then removing all rows and columns not in the set `R2` or `C2`. The `cs_fkeep` function is used to drop entries not in `R2`, and then the size of `R1` is subtracted from all row indices. The strongly connected components could be found without forming `C=A(R2,C2)` explicitly, but this method results in a simpler `cs_scc` function.

Combine decompositions: The fine and coarse decompositions are combined into the `r` and `s` vectors that determine the boundaries of the blocks. The permutation vectors `p` and `q` from the coarse decomposition are combined with the permutation `scc->p` of `A(R2,C2)` that reveals its strongly connected components.

```
csd *cs_dmperm (const cs *A, int seed)
{
    int m, n, i, j, k, cnz, nc, *jmatch, *imatch, *wi, *wj, *pinv, *Cp, *Ci,
        *ps, *rs, nb1, nb2, *p, *q, *cc, *rr, *r, *s, ok ;
    cs *C ;
    csd *D, *scc ;
    /* --- Maximum matching ------------------------------------------------ */
    if (!CS_CSC (A)) return (NULL) ;            /* check inputs */
    m = A->m ; n = A->n ;
    D = cs_dalloc (m, n) ;                      /* allocate result */
    if (!D) return (NULL) ;
    p = D->p ; q = D->q ; r = D->r ; s = D->s ; cc = D->cc ; rr = D->rr ;
    jmatch = cs_maxtrans (A, seed) ;            /* max transversal */
    imatch = jmatch + m ;                       /* imatch = inverse of jmatch */
    if (!jmatch) return (cs_ddone (D, NULL, jmatch, 0)) ;
    /* --- Coarse decomposition -------------------------------------------- */
    wi = r ; wj = s ;                           /* use r and s as workspace */
    for (j = 0 ; j < n ; j++) wj [j] = -1 ;     /* unmark all cols for bfs */
    for (i = 0 ; i < m ; i++) wi [i] = -1 ;     /* unmark all rows for bfs */
    cs_bfs (A, n, wi, wj, q, imatch, jmatch, 1) ;       /* find C1, R1 from C0*/
    ok = cs_bfs (A, m, wj, wi, p, jmatch, imatch, 3) ;  /* find R3, C3 from R0*/
    if (!ok) return (cs_ddone (D, NULL, jmatch, 0)) ;
    cs_unmatched (n, wj, q, cc, 0) ;                    /* unmatched set C0 */
    cs_matched (n, wj, imatch, p, q, cc, rr, 1, 1) ;    /* set R1 and C1 */
    cs_matched (n, wj, imatch, p, q, cc, rr, 2, -1) ;   /* set R2 and C2 */
    cs_matched (n, wj, imatch, p, q, cc, rr, 3, 3) ;    /* set R3 and C3 */
    cs_unmatched (m, wi, p, rr, 3) ;                    /* unmatched set R0 */
    cs_free (jmatch) ;
    /* --- Fine decomposition ---------------------------------------------- */
    pinv = cs_pinv (p, m) ;             /* pinv=p' */
    if (!pinv) return (cs_ddone (D, NULL, NULL, 0)) ;
    C = cs_permute (A, pinv, q, 0) ;/* C=A(p,q) (it will hold A(R2,C2)) */
    cs_free (pinv) ;
    if (!C) return (cs_ddone (D, NULL, NULL, 0)) ;
    Cp = C->p ;
    nc = cc [3] - cc [2] ;              /* delete cols C0, C1, and C3 from C */
    if (cc [2] > 0) for (j = cc [2] ; j <= cc [3] ; j++) Cp [j-cc[2]] = Cp [j] ;
    C->n = nc ;
    if (rr [2] - rr [1] < m)            /* delete rows R0, R1, and R3 from C */
    {
        cs_fkeep (C, cs_rprune, rr) ;
        cnz = Cp [nc] ;
        Ci = C->i ;
        if (rr [1] > 0) for (k = 0 ; k < cnz ; k++) Ci [k] -= rr [1] ;
    }
    C->m = nc ;
    scc = cs_scc (C) ;                  /* find strongly connected components of C*/
    if (!scc) return (cs_ddone (D, C, NULL, 0)) ;
    /* --- Combine coarse and fine decompositions -------------------------- */
    ps = scc->p ;                       /* C(ps,ps) is the permuted matrix */
    rs = scc->r ;                       /* kth block is rs[k]..rs[k+1]-1 */
    nb1 = scc->nb  ;                    /* # of blocks of A(R2,C2) */
    for (k = 0 ; k < nc ; k++) wj [k] = q [ps [k] + cc [2]] ;
    for (k = 0 ; k < nc ; k++) q [k + cc [2]] = wj [k] ;
    for (k = 0 ; k < nc ; k++) wi [k] = p [ps [k] + rr [1]] ;
    for (k = 0 ; k < nc ; k++) p [k + rr [1]] = wi [k] ;
    nb2 = 0 ;                           /* create the fine block partitions */
```

```
    r [0] = s [0] = 0 ;
    if (cc [2] > 0) nb2++ ;          /* leading coarse block A (R1, [C0 C1]) */
    for (k = 0 ; k < nb1 ; k++)      /* coarse block A (R2,C2) */
    {
        r [nb2] = rs [k] + rr [1] ; /* A (R2,C2) splits into nb1 fine blocks */
        s [nb2] = rs [k] + cc [2] ;
        nb2++ ;
    }
    if (rr [2] < m)
    {
        r [nb2] = rr [2] ;           /* trailing coarse block A ([R3 R0], C3) */
        s [nb2] = cc [3] ;
        nb2++ ;
    }
    r [nb2] = m ;
    s [nb2] = n ;
    D->nb = nb2 ;
    cs_dfree (scc) ;
    return (cs_ddone (D, C, NULL, 1)) ;
}
```

The breadth-first search is performed by the `cs_bfs` function below.

```
static int cs_bfs (const cs *A, int n, int *wi, int *wj, int *queue,
    const int *imatch, const int *jmatch, int mark)
{
    int *Ap, *Ai, head = 0, tail = 0, j, i, p, j2 ;
    cs *C ;
    for (j = 0 ; j < n ; j++)            /* place all unmatched nodes in queue */
    {
        if (imatch [j] >= 0) continue ; /* skip j if matched */
        wj [j] = 0 ;                     /* j in set C0 (R0 if transpose) */
        queue [tail++] = j ;             /* place unmatched col j in queue */
    }
    if (tail == 0) return (1) ;          /* quick return if no unmatched nodes */
    C = (mark == 1) ? ((cs *) A) : cs_transpose (A, 0) ;
    if (!C) return (0) ;                 /* bfs of C=A' to find R3,C3 from R0 */
    Ap = C->p ; Ai = C->i ;
    while (head < tail)                  /* while queue is not empty */
    {
        j = queue [head++] ;             /* get the head of the queue */
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            i = Ai [p] ;
            if (wi [i] >= 0) continue ; /* skip if i is marked */
            wi [i] = mark ;              /* i in set R1 (C3 if transpose) */
            j2 = jmatch [i] ;            /* traverse alternating path to j2 */
            if (wj [j2] >= 0) continue ;/* skip j2 if it is marked */
            wj [j2] = mark ;             /* j2 in set C1 (R3 if transpose) */
            queue [tail++] = j2 ;        /* add j2 to queue */
        }
    }
    if (mark != 1) cs_spfree (C) ;       /* free A' if it was created */
    return (1) ;
}
```

To find $\mathcal{R}_1$ and $\mathcal{C}_1$, `cs_bfs` starts at unmatched column nodes in $\overline{\mathcal{C}}$ and traverses alternating paths, according to the maximum matching found by `cs_maxtrans`.

The order of the nodes in the sets $\mathcal{R}_1$ and $\mathcal{C}_1$ is not important, so a simpler breadth-first search can be used instead of a more complicated depth-first search. To find $\mathcal{R}_3$ and $\mathcal{C}_3$, it starts at unmatched row nodes in $\overline{\mathcal{R}}$ and searches the transpose of the graph of A. The queue array is workspace for the breadth-first queue. cs_dmperm passes p and q to cs_bfs to use as workspace for the breadth-first search queue.

cs_matched constructs the portions of the output permutations corresponding to the matched submatrices (A ([R1 R2 R3], [C1 C2 C3])).

```
static void cs_matched (int n, const int *wj, const int *imatch, int *p, int *q,
    int *cc, int *rr, int set, int mark)
{
    int kc = cc [set], j ;
    int kr = rr [set-1] ;
    for (j = 0 ; j < n ; j++)
    {
        if (wj [j] != mark) continue ;      /* skip if j is not in C set */
        p [kr++] = imatch [j] ;
        q [kc++] = j ;
    }
    cc [set+1] = kc ;
    rr [set] = kr ;
}
```

cs_unmatched constructs the R0 and C0 sets. The cs_rprune function is used with cs_fkeep to remove all rows not in the set R2.

```
static void cs_unmatched (int m, const int *wi, int *p, int *rr, int set)
{
    int i, kr = rr [set] ;
    for (i = 0 ; i < m ; i++) if (wi [i] == 0) p [kr++] = i ;
    rr [set+1] = kr ;
}

static int cs_rprune (int i, int j, double aij, void *other)
{
    int *rr = (int *) other ;
    return (i >= rr [1] && i < rr [2]) ;
}
```

In MATLAB, the corresponding decomposition is [p,q,r,s]=dmperm(A). The four outputs of dmperm are identical to the same outputs of cs_dmperm. The dmperm function in MATLAB 7.2 does not return the coarse decomposition. The matrix A([R1 R2 R3], [C1 C2 C3]) permuted by cs_dmperm has a zero-free diagonal of maximum size. This is almost the same as dmperm, except that dmperm returns the sets C0 and C1 jumbled together in no particular order, as are the sets R3 and R0. The function cs_dmperm makes the distinction between these sets clearer by returning the maximum matching as the diagonal of the square matrix A([R1 R2 R3], [C1 C2 C3]). Also, dmperm in MATLAB does not explicitly return the coarse decomposition (cc and rr). cs_dmperm returns the columns of each block in natural order, while dmperm does not. The two functions may not find the same matching, since any maximum matching is valid. Both functions return a valid Dulmage–Mendelsohn decomposition.

7.5 Bandwidth and profile reduction

The *profile* (also called the *envelope*) of a symmetric matrix is a measure of how close its entries are to the diagonal. Assuming the diagonal of A is nonzero, the profile of A is $\sum_{j=1}^{n}(j - \min \mathcal{A}_{*j})$. The bandwidth of the matrix is similar, $\max_j(j - \min \mathcal{A}_{*j})$. A permutation that reduces the profile or bandwidth of a matrix can provide a good ordering for the LU or Cholesky factorization of PAP^T. Minimizing the profile or bandwidth is NP-hard, so heuristics are used.

The reverse Cuthill–McKee algorithm is one of the simplest heuristics. It starts at a given node and orders it first. Next, all unnumbered neighbors of any ordered nodes are ordered by increasing degree. This gives the Cuthill–Mckee ordering; the profile is reduced further by reversing this ordering (the profile cannot be increased by this reversal). Finding a good starting node is critical for obtaining a good ordering. The method starts with a *pseudoperipheral node*, which is found by repeated breadth-first searches. An arbitrary starting node is selected, and a breadth-first search determines the most distant node from this starting node. The breadth-first search is repeated, starting at this distant node, and the process repeats until the distance does not increase. The pseudoperipheral node is the last node found, and becomes the starting node for the reverse Cuthill–McKee algorithm. The MATLAB function for reverse Cuthill–McKee is `p=symrcm(A)`. CSparse does not include a C version of this algorithm.

The *Fiedler vector* of a symmetric matrix A is the eigenvector corresponding to the second largest eigenvalue of the Laplacian of the graph of A (assuming the graph is connected). The Laplacian S has the same nonzero pattern as A. Off-diagonal entries are replaced with -1, and the kth diagonal is replaced by the number of off-diagonal entries in the kth column. With the permutation P obtained from sorting the Fiedler vector, the matrix PAP^T will tend to have a small profile. It can be computed using the MATLAB `eigs` function. The `eigs` function uses a shift-and-invert technique; it thus computes the factorization of the positive semidefinite matrix $S - \alpha I$. For the Fiedler vector, the Lanczos method could be used instead, which does not require the factorization of S.

```
function [p,v,d] = cs_fiedler (A)
%CS_FIEDLER the Fiedler vector of a connected graph.
%   [p,v,d] = cs_fiedler(A) computes the Fiedler vector v (the eigenvector
%   corresponding to the 2nd smallest eigenvalue d of the Laplacian of the graph
%   of A+A').  p is the permutation obtained when v is sorted.  A should be a
%   connected graph.
%
%   See also CS_SCC, EIGS, SYMRCM, UNMESH.

n = size (A,1) ;
if (n < 2) p = 1 ; v = 1 ; d = 0 ; return ; end
opt.disp = 0 ;                      % turn off printing in eigs
opt.tol = sqrt (eps) ;
S = A | A' | speye (n) ;            % compute the Laplacian of A
S = diag (sum (S)) - S ;
[v,d] = eigs (S, 2, 'SA', opt) ;    % find the Fiedler vector v
v = v (:,2) ;
d = d (2,2) ;
[ignore p] = sort (v) ;             % sort it to get p
```

7.6 Nested dissection

Nested dissection is a fill-reducing ordering well suited to matrices arising from the discretization of a problem with two-dimensional (2D) or three-dimensional (3D) geometry. The goal of this ordering is the same as the minimum degree ordering; it is a heuristic for reducing fill-in, not the profile or bandwidth. Consider the undirected graph of a matrix A with symmetric nonzero pattern. Nested dissection finds a *node separator* that splits the graph into two or more roughly equal-sized subgraphs (left and right) when the nodes in the separator (and their incident edges) are removed from the graph. The subgraphs are then ordered recursively via nested dissection for a large subgraph or minimum degree for a small one.

With a single node separator, a matrix is split into the following form, where A_{33} is the matrix corresponding to the nodes in the node separator, A_{11} corresponds to the left subgraph, and A_{22} corresponds to the right subgraph. Since the left subgraph (A_{11}) and right subgraph (A_{22}) are not joined by any edges, A_{12} is zero.

$$\begin{bmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{bmatrix}.$$

There are many methods for finding a good node separator. One class of methods starts with an *edge separator* and then converts it into a node separator. Likewise, an edge separator can be found in many ways; the method discussed here is based on the profile-reducing methods discussed in the previous section. There are many ways of finding a nested dissection ordering; this method was chosen for its simplicity of implementation. State-of-the-art methods are highlighted at the end of this section. See also Section 7.7.

Suppose a profile-reducing ordering P has been found. Divide the matrix PAP^T into its first $\lfloor n/2 \rfloor$ rows and columns and its last $n - \lfloor n/2 \rfloor$ rows and columns.

$$PAP^T = \begin{bmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{bmatrix}.$$

Since the profile of PAP^T has been reduced, the number of entries in A_{12} will be small. If these entries (edges in the graph G) are removed, the graph splits into two components of equal size (possibly more than two connected components if the graphs of A_{11} or A_{22} are unconnected). The edges in A_{12} are an edge separator of G. In practice, the matrix is not divided equally, since the size of the edge separator can often be reduced if the two subgraphs are allowed to differ in size (they are kept roughly equal in size; otherwise, a good ordering is not obtained). The `cs_esep` M-file shown below finds an edge separator using `symrcm`.

The Dulmage–Mendelsohn decomposition can convert this edge separator into a node separator by finding a minimal *node cover* of the edges in A_{12}. Consider the

bipartite graph of $S = A_{12}$ and its Dulmage–Mendelsohn decomposition,

$$S = \begin{array}{c} \begin{array}{cccc} \overline{\mathcal{C}} & \mathcal{C}_1 & \mathcal{C}_2 & \mathcal{C}_3 \end{array} \\ \left[\begin{array}{cccc} S_{11} & S_{12} & S_{13} & S_{14} \\ & & S_{23} & S_{24} \\ & & & S_{34} \\ & & & S_{44} \end{array}\right] \end{array} \begin{array}{c} \\ \mathcal{R}_1 \\ \mathcal{R}_2 \\ \mathcal{R}_3 \\ \overline{\mathcal{R}} \end{array} .$$

The method can select as the node cover either the set $\mathcal{R}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ or $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{C}_3$, both with size equal to the size of the maximal matching of A_{12} (its structural rank). Any edge in S will be incident on at least one node in one of these two sets. The `cs_sep` M-file selects $\mathcal{R}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$. In practice, the set is chosen that best balances the sizes of the left and right subgraphs. The `cs_nsep` M-file constructs an edge separator and converts it into a node separator. The recursive `cs_nd` M-file finds a node separator using `cs_nsep` and then recursively bisects the two subgraphs. Small graphs (of order 500 or less) are ordered with `cs_amd`.

```
function [a,b] = cs_esep (A)
%CS_ESEP find an edge separator of a symmetric matrix A
%   [a,b] = cs_esep(A) finds a edge separator s that splits the graph of A
%   into two parts a and b of roughly equal size.  The edge separator is the
%   set of entries in A(a,b).
%
%   See also CS_NSEP, CS_SEP, CS_ND, SYMRCM.

p = symrcm (A) ;
n2 = fix (size(A,1)/2) ;
a = p (1:n2) ;
b = p (n2+1:end) ;
```

```
function [s,as,bs] = cs_sep (A,a,b)
%CS_SEP convert an edge separator into a node separator.
%   [s,as,bs] = cs_sep (A,a,b) converts an edge separator into a node separator.
%   [a b] is a partition of 1:n, thus the edges in A(a,b) are an edge separator
%   of A.  s is the node separator, consisting of a node cover of the edges of
%   A(a,b).  as and bs are the sets a and b with s removed.
%
%   See also CS_DMPERM, CS_NSEP, CS_ESEP, CS_ND.

[p q r s cc rr] = cs_dmperm (A (a,b)) ;
s = [(a (p (1:rr(2)-1))) (b (q (cc(3):(cc(5)-1))))] ;
w = ones (1, size (A,1)) ;
w (s) = 0 ;
as = a (find (w (a))) ;
bs = b (find (w (b))) ;
```

```
function [s,a,b] = cs_nsep (A)
%CS_NSEP find a node separator of a symmetric matrix A.
%   [s,a,b] = cs_nsep(A) finds a node separator s that splits the graph of A
%   into two parts a and b of roughly equal size.  If A is unsymmetric, use
%   cs_nsep(A|A').  The permutation p = [a b s] is a one-level dissection of A.
%
%   See also CS_SEP, CS_ESEP, CS_ND.

[a b] = cs_esep (A) ;
[s a b] = cs_sep (A, a, b) ;
```

```
function p = cs_nd (A)
%CS_ND generalized nested dissection ordering.
%   p = cs_nd(A) computes the nested dissection ordering of a matrix.  Small
%   submatrices (order 500 or less) are ordered via cs_amd.  A must be sparse
%   and symmetric (use p = cs_nd(A|A') if it is not symmetric).
%
%   See also CS_AMD, CS_SEP, CS_ESEP, CS_NSEP, AMD.

n = size (A,1) ;
if (n == 1)
    p = 1 ;
elseif (n < 500)
    p = cs_amd (A) ;                    % use cs_amd on small graphs
else
    [s a b] = cs_nsep (A) ;             % find a node separator
    a = a (cs_nd (A (a,a))) ;           % order A(a,a) recursively
    b = b (cs_nd (A (b,b))) ;           % order A(b,b) recursively
    p = [a b s] ;                       % concatenate to obtain the final ordering
end
```

The Fiedler vector, or other eigenvector techniques, can lead to a smaller node separator and orderings with lower fill-in, but they are prohibitively expensive to compute for large graphs. To overcome this problem, the graph G of A can be successively *coarsened*. A node in the coarse graph G_c of A represents a unique set of nodes in G with node weight equal to the number of nodes it represents. Edge weights are used to reflect the number of edges (the sum of their weights) between sets of nodes in G. A sequence of coarser and coarser graphs G (the original graph), $G_1, G_2, \ldots, G_k$ is found until G_k is small enough to use powerful edge or node separator methods efficiently. Next, the edge or node separator is mapped to the graph of G_{k-1}, and refinement techniques (such as the Kernighan–Lin algorithm) are used to improve this partition of G_{k-1}. The refinement process continues until a separator of G is obtained.

Figure 7.4 is the graph of the matrix in Figure 4.2 on page 39 with both node and edge separators highlighted.

If applied to a 2D s-by-s mesh with node separators selected along a mesh line that most evenly divides the graph, nested dissection leads to an asymptotically optimal ordering, with $31(n \log_2 n)/8 + O(n)$ nonzeros in L, and requiring $829(n^{3/2})/84 + O(n \log n)$ floating-point operations to compute, where $n = s^2$ is the dimension of the matrix. For a 3D s-by-s-by-s mesh the dimension of the matrix is $n = s^3$. There are $O(n^{4/3})$ nonzeros in L, and $O(n^2)$ floating-point operations are required to compute L, which is also asymptotically optimal.

7.7 Further reading

Finding a good fill-reducing ordering is essential for reducing time and memory requirements for sparse factorization methods. Since it is an NP-hard problem (Yannakakis [199]), many heuristic methods have been developed. These heuristics can be divided into three broad categories: minimum degree, nested dissection, and profile reduction. The Dulmage–Mendelsohn decomposition [72] is not a heuristic, but it can be considered as a fourth category of ordering methods, since it restricts

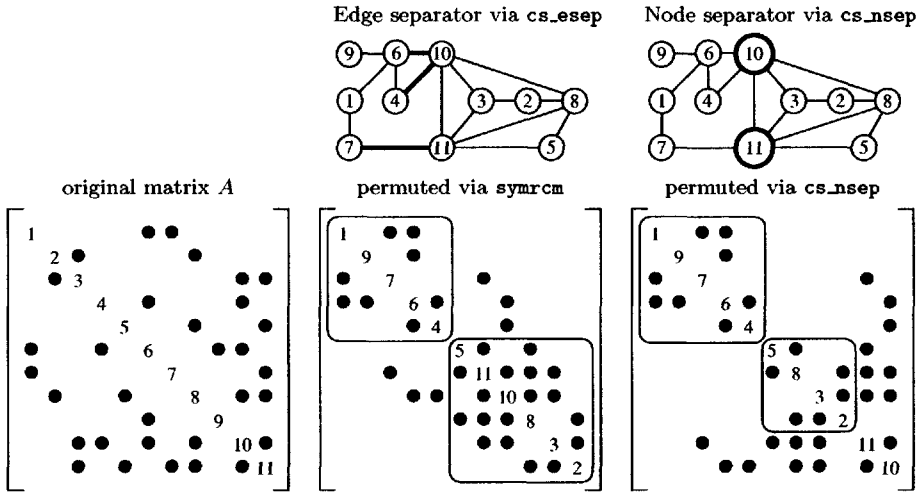


Figure 7.4. *A node and edge separator*

fill-in to the diagonal blocks.

Minimum degree and its variants are the most commonly used methods, providing low fill-in on the widest range of matrices. Good nested dissection orderings tend to be superior to minimum degree for large problems that come from 2D and 3D spatial discretizations (finite-element methods, for example), particularly when coupled with minimum degree for ordering small subgraphs. An early unsymmetric form is Markowitz's method [155] that selects a_{ij} to minimize the product of the row degree i and column degree j. MA28 by Duff and Reid [61] in the HSL package is based on an efficient implementation of the method. It is interesting to note that Markowitz received the 1990 Nobel prize [79] in economics for his pioneering work in portfolio theory, part of which is his LU ordering work.[14]

Minimum degree is a local greedy strategy, or a "bottom-up" approach, since it finds the leaves of the elimination tree first. Tinney and Walker [196] developed the first minimum degree method for symmetric matrices. Note that the word "optimal" in the title of their paper is a misnomer, since minimum degree is a non-optimal method. Its earliest efficient implementation is by George and Liu's algorithm (MMD and its precursors in SPARSPAK [85, 87, 91]). Other implementations include YSMP (Eisenstat et al. [73] and Eisenstat, Schultz, and Sherman [76]) and MA27 (Duff, Erisman, and Reid [53] and Duff and Reid [62]).

Since computing the exact degree is costly, Amestoy, Davis, and Duff developed AMD, an approximate minimum degree ordering [1, 2]; Davis et al. later extended this to compute a column ordering of A^TA without forming A^TA explicitly (COLAMD [33, 34]). The approximate degree spurred the search for approximations to the *deficiency*. The deficiency of a node is how much fill-in would occur if

[14] http://nobelprize.org/economics/laureates/1990/markowitz-autobio.html

the node were selected as the pivot. Methods based on variations of approximate deficiency have been developed by Rothberg and Eisenstat [176], Ng and Raghavan [161], and Pellegrini, Roman, and Amestoy [166].

Nested dissection is a "top-down" approach, since the first level separator includes the root of the elimination tree and its immediate descendants. Kernighan and Lin [140] present an early graph partitioning technique based on exchanging pairs of nodes in the graph. Fiduccia and Mattheyses present a more efficient node-swapping method [77]. Hager, Park, and Davis extend this idea to exchange blocks of nodes [126]. The first nested dissection algorithm for ordering sparse matrices is due to George [81, 82]. It relies on finding a good pseudoperipheral node, as discussed by George and Liu [86]. See also Duff, Erisman, and Reid's discussion of George's nested dissection method [52].

More recent approaches to nested dissection are based on multilevel methods and eigenvector techniques (in particular the Fiedler vector [78]). These include methods by Pothen, Simon, and Liou [170], Karypis and Kumar (METIS [139]), Hendrickson and Leland (CHACO [131]), Pellegrini, Roman, and Amestoy (SCOTCH [166]), and Walshaw, Cross, and Everett (JOSTLE [197]). Since many matrices arise in problems with 2D and 3D geometry, Heath and Raghavan [129] and Gilbert, Miller, and Teng (MESHPART [104]) present partitioning methods based on the geometric position of nodes in the graph. CHOLMOD includes both AMD and a partitioning method that combines METIS with a constrained approximate column minimum degree ordering algorithm, CCOLAMD [30].

Many of the early sparse matrix factorization methods used a profile or envelope data structure, so reducing the profile of a matrix had a direct impact on the memory usage of the method. Profile reduction is still a useful method for more recent factorization techniques. It can form a first step in finding a good edge or node separator for graph partitioning and nested dissection. Cuthill and McKee [25] developed one of the first techniques, which is still in use. Liu and Sherman [153] showed that reversing the Cuthill–McKee ordering never increases the profile and often reduces it. Chan and George [21] present an efficient implementation. Other profile reduction techniques include those by Crane et al. [24], Gibbs [99], Gibbs, Poole, and Stockmeyer [100], Hager [125], Reid and Scott [172, 173], Lewis [146], and Sloan [187]. Eigenvector techniques are also an effective method for reducing the profile, as discussed by Barnard, Pothen, and Simon [14], Pothen, Simon, and Liou [170], and Kumfert and Pothen [142].

Cormen, Leiserson, and Rivest [23] describe the *scc* algorithm discussed in Section 7.3. The earliest algorithm for finding the strongly connected components of a graph is due to Tarjan [194]; Duff and Reid [59, 60] implement the algorithm. Gustavson [120] discusses both the maximum matching and an implementation of Tarjan's algorithm. Duff [48, 49] presents the $O(|A|n)$-time maximum matching algorithm used by `cs_maxtrans`. Duff and Wiberg [71] implement an $O(|A|\sqrt{n})$-time maximum matching algorithm of Hopcroft and Karp [137] that is not always faster than the $O(|A|n)$-time algorithm in practice. Pothen and Fan [169] compare various methods for computing the block triangular form. Duff and Koster [57, 58] present a maximum weighted matching.

Ordering methods in MATLAB are discussed in Chapter 10.

Exercises

7.1. Download a copy of AMD from www.siam.org/books/fa02 (or from www.acm.org as ACM Algorithm 837). Compare it with `cs_amd` and make a list of the differences between the two codes. Compare the run time, memory usage, and ordering quality on a large range of symmetric matrices (use `p=cs_amd(A)` and compare with `p=amd(A)` in MATLAB). The MATLAB expression `lnz=sum(symbfact(A(p,p)))` gives the number of nonzeros in the Cholesky factor L of the matrix `A(p,p)` (ignoring numerical cancellation).

7.2. Compare the MATLAB statement `q=cs_amd(A,2)` with the permutation computed by the MATLAB statement `q=colamd(A)`. Compare the ordering time and memory usage. Use the column ordering in `[L,U,P]=lu(A(:,q))` to compare the ordering quality (`nnz(L)+nnz(U)`). Add code to `cs_amd` and COLAMD to compute their memory usage. COLAMD orders A^TA without forming it explicitly, so it will tend to use much less memory than `cs_amd(A)`. Both drop dense rows from `A`.

7.3. Compare the MATLAB statement `p=cs_amd(A,3)` with the permutation computed by the MATLAB statement `p=amd(A'*A)` (see Problem 7.1). Compare the ordering time and memory usage. Compare ordering quality; use `rnz =sum(symbfact(A(:,q),'col'))` (where `rnz` is the same as `nnz(qr(A,0))`, ignoring numerical cancellation and assuming `A` is not structurally rank deficient).

7.4. Write a function that solves $Ax = b$ by combining LU factorization with the block triangular form (the fine Dulmage–Mendelsohn decomposition). Find the blocks with `cs_dmperm` and then analyze and factorize each block with `cs_sqr` and `cs_lu`, respectively. Next, solve $Ax = b$ via a block backsolve. Compare with `cs_lu` on matrices with many diagonal blocks (these include matrices arising in circuit simulation and chemical process simulation). See also Section 8.4.

7.5. Why is the block triangular form not helpful for sparse Cholesky factorization? Hint: consider the elimination tree postordering. What happens if the elimination tree is a forest?

7.6. Heuristics for placing large entries on the diagonal of a matrix are useful methods for reducing the need for partial pivoting during factorization (see [57, 58], for example). Try the following method. First, scale a copy of the matrix A so that the largest entry in each column is equal to one. Next, remove small entries from the matrix and use `cs_maxtrans` to find a zero-free diagonal. If too many entries were dropped, decrease the drop tolerance and try again, or simply complete the matching arbitrarily. Use the matching as a column preordering Q and then order $AQ + (AQ)^T$ with minimum degree. Use a small pivot tolerance in `cs_lu` and determine how many off-diagonal pivots are found.

7.7. Compare the run time of `cs_dmperm` with different values of `seed` (`0`, `-1`, and `1`) on a wide range of matrices from real applications. Symmetric indefinite

matrices arising in optimization problems are of particular interest (many of the matrices from Gould, Hu, and Scott [116] in the `GHS_indef` group in the UF sparse matrix collection, for example). Find examples where the randomized order, reverse order, and natural order methods each outperform the other methods (the `boyd2` matrix is an extreme example).

Chapter 8

Solving sparse linear systems

Solving a sparse linear system $Ax = b$ is now a matter of putting together the methods described in Chapters 2 through 7. The typical steps are to (1) find a permutation to reduce fill-in, (2) analyze and factorize the permuted matrix A (via Cholesky, LU, or QR), (3) permute the right-hand side b, (4) solve for x using forward/backsolve or by applying Householder reflections, (5) permute the solution x, and (6) optionally perform one or two steps of iterative refinement, which can sometimes improve the solution (see Problem 8.5), even when performed in standard floating-point precision.

A naive method for solving $Ax = b$ when A is square and nonsingular is to compute the inverse A^{-1}, or `x=inv(A)*b` in MATLAB. This is numerically unstable when A is ill-conditioned in the dense case and also very costly in the sparse case. The inverse normally has no zero entries at all, as shown by the following two theorems.

Theorem 8.1 (Gilbert [101]). *Ignoring numerical cancellation, the nonzero pattern of the solution to $Ax = b$, where A has a zero-free diagonal, is $\mathcal{X} = Reach_A(\mathcal{B})$. Ignoring numerical cancellation, the solution x has no zero entries if A is strong Hall.*

Theorem 8.2 (Gilbert [101]). *The transitive closure of the directed graph of A is the graph C, where $\mathcal{C}_i = Reach_A(i)$. Ignoring numerical cancellation, $\mathcal{C}$ gives the nonzero pattern of A^{-1}. Every edge is present in C, and A^{-1} has no zero entries, if A is strong Hall.*

8.1 Using a Cholesky factorization

When A is symmetric positive definite, the system $Ax = b$ can be solved via Cholesky factorization. If P is the fill-reducing permutation, $LL^T = PAP^T$. The system $Ax = b$ becomes $PAP^TPx = Pb$. Solving $Ly = Pb$ for y, solving $L^Tz = y$ for z, and finally $x = P^Tz$ results in the solution x. In MATLAB notation,

```
p = amd (A) ;
L = chol (A) ;
x = L' \ (L \ b (p)) ;
x (p) = x ;
```

The `cs_cholsol` function overwrites its input `b` with the solution to $Ax = b$. `order` determines the input ordering used (`0` for $P = I$, or `1` for a minimum degree ordering of A). It returns true (`1`) if successful, false (`0`) if the matrix is not positive definite or if the method ran out of memory. Note that the forward/backsolve steps cannot fail because they do not allocate memory.

```
int cs_cholsol (int order, const cs *A, double *b)
{
    double *x ;
    css *S ;
    csn *N ;
    int n, ok ;
    if (!CS_CSC (A) || !b) return (0) ;      /* check inputs */
    n = A->n ;
    S = cs_schol (order, A) ;                /* ordering and symbolic analysis */
    N = cs_chol (A, S) ;                     /* numeric Cholesky factorization */
    x = cs_malloc (n, sizeof (double)) ;     /* get workspace */
    ok = (S && N && x) ;
    if (ok)
    {
        cs_ipvec (S->pinv, b, x, n) ;    /* x = P*b */
        cs_lsolve (N->L, x) ;            /* x = L\x */
        cs_ltsolve (N->L, x) ;           /* x = L'\x */
        cs_pvec (S->pinv, x, b, n) ;     /* b = P'*x */
    }
    cs_free (x) ;
    cs_sfree (S) ;
    cs_nfree (N) ;
    return (ok) ;
}
```

8.2 Using a QR factorization

The *least squares* problem is to find x that minimizes the 2-norm of the residual, $\|r\|_2$, where $r = b - Ax$ and A is m-by-n with $m \geq n$. Multiplying a vector by an orthogonal matrix Q does not change its 2-norm. If A is factorized into the product $A = QR$, then

$$\|r\|_2 = \|b - Ax\|_2 = \|Q^T b - Rx\|_2 = \left\| \begin{bmatrix} Q_1^T b - R_1 x \\ Q_2^T b \end{bmatrix} \right\|_2 = \left\| \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} \right\|_2,$$

where Q is m-by-m, R_1 is n-by-n, and Q_1 is m-by-n. Assuming A has full rank, R_1 is nonsingular and so the upper triangular system $Q_1^T b = R_1 x$ can be solved, which makes $r_1 = 0$ and minimizes $\|r\|_2$.

QR factorization also provides a reliable method for solving underdetermined systems $Ax = b$, where A is m-by-n with $m < n$. Ignoring the fill-reducing ordering, consider the QR factorization $QR = A^T$. The system becomes $R^T Q^T x = b$. If the upper triangular system $R^T y = b$ is solved for y, the solution to $Ax = b$ is $x = Qy$.

In cs_qrsol, $Q = H_1 H_2 \ldots H_n$ is represented implicitly as a product of Householder reflections, and the permuted matrix $PA\overline{Q}$ is factorized instead of A, where $\overline{Q}$ is the fill-reducing column permutation. The right-hand side `b[0...m-1]` is overwritten with the solution `x[0..n-1]` (and thus `b` must be of size `max(m,n)`). Use `order=0` for the natural ordering or 3 for a minimum degree ordering of $A^T A$.

```
int cs_qrsol (int order, const cs *A, double *b)
{
    double *x ;
    css *S ;
    csn *N ;
    cs *AT = NULL ;
    int k, m, n, ok ;
    if (!CS_CSC (A) || !b) return (0) ; /* check inputs */
    n = A->n ;
    m = A->m ;
    if (m >= n)
    {
        S = cs_sqr (order, A, 1) ;          /* ordering and symbolic analysis */
        N = cs_qr (A, S) ;                  /* numeric QR factorization */
        x = cs_calloc (S ? S->m2 : 1, sizeof (double)) ;    /* get workspace */
        ok = (S && N && x) ;
        if (ok)
        {
            cs_ipvec (S->pinv, b, x, m) ;   /* x(0:m-1) = b(p(0:m-1) */
            for (k = 0 ; k < n ; k++)       /* apply Householder refl. to x */
            {
                cs_happly (N->L, k, N->B [k], x) ;
            }
            cs_usolve (N->U, x) ;           /* x = R\x */
            cs_ipvec (S->q, x, b, n) ;      /* b(q(0:n-1)) = x(0:n-1) */
        }
    }
    else
    {
        AT = cs_transpose (A, 1) ;          /* Ax=b is underdetermined */
        S = cs_sqr (order, AT, 1) ;         /* ordering and symbolic analysis */
        N = cs_qr (AT, S) ;                 /* numeric QR factorization of A' */
        x = cs_calloc (S ? S->m2 : 1, sizeof (double)) ;    /* get workspace */
        ok = (AT && S && N && x) ;
        if (ok)
        {
            cs_pvec (S->q, b, x, m) ;       /* x(q(0:m-1)) = b(0:m-1) */
            cs_utsolve (N->U, x) ;          /* x = R'\x */
            for (k = m-1 ; k >= 0 ; k--)    /* apply Householder refl. to x */
            {
                cs_happly (N->L, k, N->B [k], x) ;
            }
            cs_pvec (S->pinv, x, b, n) ;    /* b(0:n-1) = x(p(0:n-1)) */
        }
    }
    cs_free (x) ;
    cs_sfree (S) ;
    cs_nfree (N) ;
    cs_spfree (AT) ;
    return (ok) ;
}
```

8.3 Using an LU factorization

When A is a general square matrix, solving $Ax = b$ is normally done via LU factorization. The factorization is $PAQ = LU$, where P and Q serve two purposes: reducing fill-in and maintaining numerical accuracy. In left-looking sparse LU factorization, Q is chosen to reduce fill-in and P is chosen via partial pivoting. Solving $Ly = Pb$ and $Uz = y$ gives the permuted solution, and then $x = Qz$ gives the solution to $Ax = b$.

The `cs_lusol` function solves $Ax = b$ using LU factorization. The parameter `order` determines the ordering method used. The natural order ($Q = I$) is `order=0`. If the matrix has a mostly symmetric nonzero pattern and large enough entries on its diagonal, then a minimum degree ordering of $A + A^T$ (`order=1`) with a small `tol` is best. Otherwise, an ordering of A^TA with `tol=1` is more suitable (`order=2` removes dense rows of A prior to ordering A^TA; `order=3` does not).

```
int cs_lusol (int order, const cs *A, double *b, double tol)
{
    double *x ;
    css *S ;
    csn *N ;
    int n, ok ;
    if (!CS_CSC (A) || !b) return (0) ;      /* check inputs */
    n = A->n ;
    S = cs_sqr (order, A, 0) ;               /* ordering and symbolic analysis */
    N = cs_lu (A, S, tol) ;                  /* numeric LU factorization */
    x = cs_malloc (n, sizeof (double)) ;     /* get workspace */
    ok = (S && N && x) ;
    if (ok)
    {
        cs_ipvec (N->pinv, b, x, n) ;        /* x = b(p) */
        cs_lsolve (N->L, x) ;                /* x = L\x */
        cs_usolve (N->U, x) ;                /* x = U\x */
        cs_ipvec (S->q, x, b, n) ;           /* b(q) = x */
    }
    cs_free (x) ;
    cs_sfree (S) ;
    cs_nfree (N) ;
    return (ok) ;
}
```

8.4 Using a Dulmage–Mendelsohn decomposition

The Dulmage–Mendelsohn decomposition provides a precise characterization of the structurally overdetermined, well-determined, and underdetermined parts of a linear system $Ax = b$. It allows the LU and QR factorizations to be applied to submatrices that have structural full rank, have a zero-free diagonal, and always have the strong Hall property. This simplifies the theorems and algorithms and ensures that the symbolic analysis is as tight as possible. Consider the matrix $C = PAQ$, where P and Q are the Dulmage–Mendelsohn permutations from (7.6). Let $C_{11} = [A_{11}A_{12}]$

and $C_{33} = [A_{34}; A_{44}]$. The system $Cx = b$ becomes

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ & C_{22} & C_{23} \\ & & C_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix},$$

where $C_{22} = A_{23}$. The overdetermined system $C_{33}x_3 = b_3$ can first be solved for x_3 using a QR factorization to obtain a least squares solution. Next, $C_{22}x_2 = b_2 - C_{23}x_3$ can be solved for x_2 using an LU factorization. When solving this system the block upper triangular form of C_{22} should be exploited (see Problem 7.4). Finally, the underdetermined system $C_{11}x_1 = b_1 - C_{12}x_2 - C_{13}x_3$ can be solved for x_1 using the QR factorization of C_{11}^T.

This method is illustrated by the `cs_dmsol` M-file. It is able to find consistent solutions to rank-deficient problems, assuming the structural rank and numeric rank are equal. It finds the least squares solution if A is overdetermined, even though it relies on an LU factorization for the well-determined part of the system.

```
function x = cs_dmsol (A,b)
%CS_DMSOL x=A\b using the coarse Dulmage-Mendelsohn decomposition.
%   x = cs_dmsol(A,b) computes x=A\b where A may be rectangular and/or
%   structurally rank deficient, and b is a full vector.
%
%   See also CS_QRSOL, CS_LUSOL, CS_DMPERM, SPRANK, RANK.

[m n] = size (A) ;
[p q r s cc rr] = cs_dmperm (A) ;
C = A (p,q) ;
b = b (p) ;
x = zeros (n,1) ;
if (rr(3) <= m && cc(4) <= n)
    x (cc(4):n) = cs_qrsol (C (rr(3):m, cc(4):n), b (rr(3):m)) ;
    b (1:rr(3)-1) = b (1:rr(3)-1) - C (1:rr(3)-1, cc(4):n) * x (cc(4):n) ;
end
if (rr(2) < rr (3) && cc(3) < cc(4))
    x (cc(3):cc(4)-1) = ...
        cs_lusol (C (rr(2):rr(3)-1, cc(3):cc(4)-1), b (rr(2):rr(3)-1)) ;
    b (1:rr(2)-1) = ...
        b (1:rr(2)-1) - C (1:rr(2)-1, cc(3):cc(4)-1) * x (cc(3):cc(4)-1) ;
end
if (rr(2) > 1 && cc(3) > 1)
    x (1:cc(3)-1) = cs_qrsol (C (1:rr(2)-1, 1:cc(3)-1), b (1:rr(2)-1)) ;
end
x (q) = x ;
```

The Dulmage–Mendelsohn decomposition can also be used to exploit sparsity in x and b. Consider a graph G where $(j,i) \in E$ if the (ij)th block is nonzero; G is sometimes called the *acyclic reduction* of the graph of C. Solving $Ax = b$ is then analogous to solving the sparse upper triangular system $Ux = b$, where U is sparse and k-by-k, and where k is the number of fine blocks in the Dulmage–Mendelsohn decomposition. A depth-first traversal of this graph, starting from nonzero blocks of b, will determine the blocks of x that are nonzero.

8.5 MATLAB sparse backslash

Almost the entire book (and more) is encapsulated in the single MATLAB statement `x=A\b`. The single character "\" invokes a host of powerful matrix solvers (LAPACK, the BLAS, UMFPACK, CHOLMOD, GPLU, Givens-based sparse QR factorization, Maple, and many specialized solvers). The algorithm follows the following steps in order. It uses the first method that fits the given rule and succeeds. If the linear system is not too ill-conditioned (and even sometimes when it is), backslash (`mldivide` as it is formally known) nearly always finds a good solution.

1. If `A` is a symbolic matrix, the Symbolic Toolbox (Maple) solves it symbolically.

2. If `A` is sparse and diagonal, `b` is scaled. See `cs_scale` in Problem 2.4.

3. If `A` is sparse, square, and banded (with a sufficiently dense band), either a tridiagonal solver or a band solver (in LAPACK) is used.

4. If `A` is a lower or upper triangular sparse matrix, a forward solve or backsolve is used (like `cs_lsolve` and `cs_usolve`). If it is a full matrix, the BLAS triangular solvers are used.

5. If `A` is a permuted triangular matrix, a permuted backsolve is used (one solver for the sparse case and another for the full case). See Problem 3.7.

6. If `A` is symmetric (or Hermitian) and has real positive diagonal elements, a Cholesky factorization is tried. It uses CHOLMOD in the sparse case (either an up-looking algorithm much like `cs_chol` if `A` is very sparse or a supernodal algorithm otherwise). The matrix is first permuted via an approximate minimum degree ordering algorithm (see `cs_amd`). This algorithm uses most of the theory and algorithms presented in Chapters 2 through 4 and Chapter 7. In the full case, it uses LAPACK Cholesky factorization routines. This step is terminated early if the matrix is found not to be positive definite.

7. If `A` is a full Hessenberg matrix, it is reduced to an upper triangular matrix and the BLAS backsolve is used.

8. If `A` is square and sparse, UMFPACK is used to perform an LU factorization, `L*U=P*(R\A)*Q`, where `R` is a diagonal matrix that scales the rows of `A` (GPLU can be optionally used instead; it can be faster for very sparse matrices). UMFPACK selects one of three ordering strategies: COLAMD, AMD, or a cheap diagonal matching followed by AMD, depending on the matrix properties (how symmetric its nonzero pattern is and how many entries are on the diagonal). It uses its default pivot tolerances (0.001 for diagonal entries and 0.1 for off-diagonal entries). These tolerances lead to a much sparser factorization than partial pivoting but may result in a numerically unstable factorization. If $\min_i |u_{ii}| / \max_j |u_{jj}|$ is less than machine epsilon (about 2×10^{-16}), the matrix is factorized again using partial pivoting (a tolerance of 1.0). UMFPACK is a multifrontal method, based on the column elimination tree and a symbolic QR analysis. Iterative refinement with sparse backward

error is used [9], [135, Chap. 12]. UMFPACK relies on the theory and some of the algorithms presented in nearly the whole book.

9. If `A` is square and full, LAPACK is used.

10. If `A` is sparse and not square, a sparse QR factorization based on Givens rotations is used (Section 5.5).

11. If `A` is full and not square, a QR factorization based on Householder reflections is used (in LAPACK).

The `x=b/A` statement in MATLAB is called the forward slash, or matrix right-division (`mrdivide`). It is translated immediately into `x=(A'\b')'`, and the above algorithm for backslash is used. Type `doc mldivide` in MATLAB for more details.

Even with all its host of supporting solvers, the backslash operator in MATLAB 7.2 has its limitations. It does not attempt to use iterative methods. It makes no use of ordering methods based on graph partitioning methods, and so its fill-in can be higher than it might be otherwise. It does not use the Dulmage–Mendelsohn decomposition. It uses LU factorization for symmetric indefinite matrices, rather than methods that exploit symmetry.

Gilbert, Moler, and Schreiber [105] developed the original sparse backslash for MATLAB 4.0.

8.6 Software for solving sparse linear systems

Table 8.1 summarizes most of the available software for solving sparse linear systems via direct methods as of April 2006. The first column lists the name of the package. The next four columns describe what kinds of factorizations are available: LU, Cholesky, LDL^T for symmetric indefinite matrices, and QR. If the LDL^T factorization uses 2-by-2 block pivoting a "2" is listed; a "1" is listed otherwise [18, 19]. The next column states if complex matrices (unsymmetric, symmetric, and/or Hermitian) are supported. The ordering methods available are listed in the next four columns: minimum degree and its variants (minimum fill, column minimum degree, Markowitz, and related methods), nested or one-way dissection (including all graph-based partitionings), permutation to block triangular form, and profile/bandwidth reduction (or related) methods. The next three columns indicate what level of BLAS is used (1: vector, 2: matrix-vector, 3: matrix-matrix), if the package is parallel ("s" for shared memory or "d" for distributed memory), and whether or not the package includes an out-of-core option (where most of the factors remain on disk). Most distributed-memory packages can also be used in a shared-memory environment, since most message-passing libraries (MPI in particular) are ported to shared-memory environments. A code is listed as "sd" if it includes two versions, one for shared memory and the other for distributed memory. The next column indicates if a MATLAB interface is available. The primary method(s) used in the package are listed in the final column. Table 8.2 lists the authors of the packages, relevant papers, and where to get the code. An up-to-date table will be maintained at www.siam.org/books/fa02.

Table 8.1. *Package features*

Package	LU	Cholesky	LDL^T	QR	Complex	Minimum degree	Nested dissection	Block triangular	Profile	BLAS	Parallel	Out-of-core	MATLAB	Method
BCSLIB-EXT	●	●	2	●	●	●	●	-	-	3	s	●	-	multifrontal
CHOLMOD	-	●	1	-	●	●	●	-	-	3	-	-	●	left-looking supernodal
CSparse	●	●	-	●	●	●	-	●	-	-	-	-	●	various
DSCPACK	-	●	1	-	-	●	●	-	-	3	d	-	-	multifrontal
GPLU	●	-	-	-	●	-	-	-	-	-	-	-	●	left-looking
KLU	●	-	-	-	●	●	-	●	-	-	-	-	●	left-looking
LDL	-	●	1	-	-	-	-	-	-	-	-	-	●	up-looking
MA27	-	●	2	-	●	●	-	-	-	-	-	-	-	multifrontal
MA28	●	-	-	-	●	●	-	●	-	-	-	-	-	right-looking Markowitz
MA32	●	-	-	-	●	-	-	-	●	1	-	●	-	frontal
MA37	●	-	-	-	●	●	-	-	-	-	-	-	-	multifrontal
MA38	●	-	-	-	●	●	-	●	-	3	-	-	-	unsymmetric multifrontal
MA41	●	-	-	-	●	●	-	-	-	3	s	-	-	multifrontal
MA42	●	-	-	-	●	-	-	-	●	3	-	●	-	frontal
HSL_MP42	●	-	-	-	-	-	●	-	●	3	d	●	-	frontal
MA46	●	-	-	-	-	-	-	-	-	3	-	-	-	finite-element multifrontal
MA47	-	●	2	-	●	●	-	-	-	3	-	-	-	multifrontal
MA48	●	-	-	-	●	●	-	●	-	3	-	-	●	left-looking
HSL_MP48	●	-	-	-	-	●	-	●	-	3	d	●	-	left-looking
MA49	-	-	-	●	-	●	-	●	-	3	s	-	-	multifrontal
MA57	-	●	2	-	●	●	●	-	-	3	-	-	●	multifrontal
MA62	-	●	-	-	●	-	-	-	●	3	-	●	-	frontal
HSL_MP62	-	●	-	-	-	-	●	-	●	3	d	●	-	frontal
MA67	-	●	2	-	-	●	-	-	-	3	-	-	-	right-looking Markowitz
Mathematica	●	●	-	-	●	●	●	●	-	3	-	-	-	various
MATLAB	●	●	-	●	●	●	-	●	●	3	-	-	●	various
Meschach	●	●	2	-	-	●	-	-	-	-	-	-	-	right-looking
MUMPS	●	●	2	-	●	●	●	-	-	3	d	-	●	multifrontal
NSPIV	●	-	-	-	-	-	-	-	-	-	-	-	-	up-looking
Oblio	-	●	2	-	●	●	●	-	-	3	-	●	-	left, right, multifrontal
PARDISO	●	●	2	-	●	●	●	-	-	3	s	-	●	left/right supernodal
PaStiX	●	●	1	-	●	●	●	-	-	3	d	-	-	left-looking supernodal
PSPASES	-	●	-	-	-	-	●	-	-	3	d	-	-	multifrontal
RF	●	-	-	-	-	-	-	-	●	-	-	-	-	product form of inverse
S+	●	-	-	-	-	-	-	-	-	3	d	-	-	right-looking supernodal
Sparse 1.4	●	-	-	-	●	●	-	-	-	-	-	-	-	right-looking Markowitz
SPARSPAK	●	●	-	●	-	●	●	-	●	-	-	-	-	left-looking
SPRSBLKLLT	-	●	-	-	-	●	-	-	-	3	-	-	-	left-looking supernodal
SPOOLES	●	●	2	●	●	●	●	-	-	-	sd	-	-	left-looking, multifrontal
SuperLU	●	-	-	-	●	●	-	-	-	2	-	-	●	left-looking supernodal
SuperLU_MT	●	-	-	-	-	●	-	-	-	2	s	-	-	left-looking supernodal
SuperLU_DIST	●	-	-	-	●	●	-	-	-	3	d	-	-	right-looking supernodal
TAUCS	●	●	1	-	●	●	●	-	-	3	s	●	-	left-looking, multifrontal
UMFPACK	●	-	-	-	●	●	-	-	-	3	-	-	●	multifrontal
WSMP	●	●	1	-	●	●	●	●	-	3	sd	-	-	multifrontal
Y12M	●	-	-	-	-	●	-	-	-	-	-	-	-	right-looking Markowitz

Table 8.2. *Package authors, references, and availability*

Package	Authors, references	URL and/or contact
BCSLIB-EXT	Ashcraft, Grimes, Lewis, and Pierce [10, 12, 13, 167]	www.boeing.com/phantom/bcslib-ext
CHOLMOD	Davis, Hager, Chen, and Rajamanickam [30]	www.cise.ufl.edu/research/sparse
CSparse	Davis	www.cise.ufl.edu/research/sparse
DSCPACK	Heath and Raghavan [129, 130, 171]	www.cse.psu.edu/~raghavan
GPLU	Gilbert and Peierls [109]	www.mathworks.com
KLU	Davis and Palamadai	www.cise.ufl.edu/research/sparse
LDL	Davis [29]	www.cise.ufl.edu/research/sparse
MA27	Duff and Reid [62]	www.cse.clrc.ac.uk/nag/hsl
MA28	Duff and Reid [61]	www.cse.clrc.ac.uk/nag/hsl
MA32	Duff [50]	www.cse.clrc.ac.uk/nag/hsl
MA37	Duff and Reid [63]	www.cse.clrc.ac.uk/nag/hsl
MA38	Davis and Duff [31]	www.cse.clrc.ac.uk/nag/hsl
MA41	Amestoy and Duff [3]	www.cse.clrc.ac.uk/nag/hsl
MA42	Duff and Scott [67]	www.cse.clrc.ac.uk/nag/hsl
HSL_MP42	Scott [182, 183, 184]	www.cse.clrc.ac.uk/nag/hsl
MA46	Damhaug and Reid [26]	www.cse.clrc.ac.uk/nag/hsl
MA47	Duff and Reid [64]	www.cse.clrc.ac.uk/nag/hsl
MA48	Duff and Reid [65]	www.cse.clrc.ac.uk/nag/hsl
HSL_MP48	Duff and Scott [69]	www.cse.clrc.ac.uk/nag/hsl
MA49	Amestoy, Duff, and Puglisi [6]	www.cse.clrc.ac.uk/nag/hsl
MA57	Duff [51, 66]	www.cse.clrc.ac.uk/nag/hsl
MA62	Duff and Scott [68]	www.cse.clrc.ac.uk/nag/hsl
HSL_MP62	Scott [184]	www.cse.clrc.ac.uk/nag/hsl
MA67	Reid [54]	www.cse.clrc.ac.uk/nag/hsl
Mathematica	Wolfram Research, Inc. [198]	www.wolfram.com
MATLAB	The MathWorks, Inc. [105]	www.mathworks.com
Meschach	Steward and Leyk	www.netlib.org/c/meschach
MUMPS	Amestoy, Duff, Guermouche, Koster, L'Excellent, Pralet [4, 5, 7]	www.enseeiht.fr/apo/MUMPS graal.ens-lyon.fr/MUMPS
NSPIV	Sherman [186]	www.netlib.org/toms/533
Oblio	Dobrian, Kumfert, and Pothen [42]	email: pothen@cs.odu.edu
PARDISO	Schenk, Gärtner, and Fichtner [179, 180]	www.computational.unibas.ch/cs/scicomp/software/pardiso
PaStiX	Hénon, Ramet, and Roman [132]	www.labri.fr/~ramet/pastix
PSPASES	Joshi, Karypis, Kumar, Gupta, and Gustavson [119]	www.cs.umn.edu/~mjoshi/pspases
RF	Neculai	www.ici.ro/camo/neculai/RF
S+	Fu, Jiao, and Yang [80, 185]	www.cs.ucsb.edu/projects/s+
Sparse 1.4	Kundert [143]	sparse.sourceforge.net
SPARSPAK	George and Liu [85, 89]	www.cs.uwaterloo.ca/~jageorge
SPOOLES	Ashcraft and Grimes [11]	www.netlib.org/linalg/spooles
SPRSBLKLLT	Ng and Peyton [159]	email: EGNg@lbl.gov
SuperLU	Demmel, Eisenstat, Gilbert, and Li [40]	crd.lbl.gov/~xiaoye/SuperLU
SuperLU_MT	Demmel, Gilbert, and Li [41]	crd.lbl.gov/~xiaoye/SuperLU
SuperLU_DIST	Demmel and Li [147]	crd.lbl.gov/~xiaoye/SuperLU
TAUCS	Chen, Rotkin, and Toledo [177]	www.tau.ac.il/~stoledo/taucs
UMFPACK	Davis and Duff [28, 31, 32]	www.cise.ufl.edu/research/sparse
WSMP	Gupta [118, 119]	www.cs.umn.edu/~agupta/wsmp
Y12M	Zlatev, Wasniewski, and Schaumburg [200]	www.netlib.org/y12m

Exercises

8.1. Write a sparse backslash algorithm, just like `x=A\b` in MATLAB, that solves $Ax = b$. Assume `A` is sparse, and `b` can be a full or sparse vector. Examine `A` and determine its properties. If `A` is upper or lower triangular, use `cs_usolve` or `cs_lsolve` (or `cs_spsolve` if `b` is sparse). If it is square, symmetric, and all its diagonal entries are greater than zero, try `cs_chol`. Otherwise (or if `cs_chol` fails), use `cs_lu` if it is square or `cs_qr` if it is rectangular. Order the matrix as appropriate, factorize it, and then perform the appropriate forward/backsolves. For LU factorization, optionally select `order` and `tol` based on how symmetric the nonzero pattern is and how large the diagonal entries are relative to the off-diagonal entries. For yet more possibilities, see Sections 8.4 and 8.5. Optionally allow `b` to be a matrix.

8.2. CSparse does very little error checking of its inputs. CSparse checks only a few key error conditions: if it runs out of memory, if the matrix is singular for an LU factorization, if the matrix is not positive definite for a Cholesky factorization, if the matrix has the wrong type (compressed-column versus triplet), or if the row or column index is negative in `cs_entry`. Add more error checking to CSparse. See also Problem 2.12.

8.3. Add a floating-point operation (*flop*) counter to CSparse. Avoid adding statements such as `flop++`. Use a global `double flop` variable.

8.4. Modify `cs_qrsol` so that it returns x, r, and $\|r\|_2$.

8.5. *Iterative refinement* is a process that can improve the accuracy of the solution to $Ax = b$. In the sparse case, it is most useful in LU factorization when a small pivot tolerance is used. In MATLAB notation, `x=A\b ; x=x+A\(b-A*x)`, where of course `A` needs to be factorized only once. Add iterative refinement to `cs_lusol`. Note that `b-A*x` can be computed with `cs_gaxpy`. MATLAB uses iterative refinement with sparse backward error [9], [135, Chap. 12] in `x=A\b` when `A` is sparse and unsymmetric, so in this case iterative refinement will not lead to any improvement, since it has already been done. Compare with `cs_lu` with a very small pivot tolerance and no iterative refinement.

8.6. Modify `cs_lusol`, adding a `qrbound` parameter. Use this as the `qr` input to `cs_sqr`. For a wide range of matrices, determine how close $|L|$ and $|U|$ are to their upper bounds, computed when `qrbound=1`. Determine how good the guess is when `qrbound=0`. Add a parameter α to `cs_lusol` and `cs_sqr` that modifies the initial guess (replace $|U| = 4|A| + n$ in `cs_sqr` with α times the upper bound) and experiment with this parameter.

8.7. Modify `cs_lusol` so that `b` can have multiple columns.

8.8. Write a version `cs_lusol` where `b` is a sparse `n`-by-`k` matrix.

8.9. Repeat Problems 8.7 and 8.8 for `cs_cholsol` and `cs_qrsol`.

8.10. Repeat Problem 8.1, where `b` can have multiple columns.

8.11. Write a MATLAB interface for CXSparse. Note that MATLAB and CXSparse use different methods for storing complex values.

Chapter 9
CSparse

This chapter provides an overview of the use of CSparse as a stand-alone sparse matrix package available for download from www.siam.org/books/fa02. CSparse functions are divided into three sets: primary, secondary, and tertiary. The primary functions are all that an application needs to create a matrix A, solve $Ax = b$, and perform basic matrix operations. Secondary routines include the ordering and factorization methods, forward/backsolve, rank-1 update/downdate, and routines for manipulating permutation vectors. These can be useful in some applications (when a matrix is analyzed and ordered once but factorized multiple times with different numerical values but identical nonzero pattern, for example). Tertiary routines would rarely be used in any application; they are primarily meant as functions for use within CSparse itself.

Every file in CSparse starts with the statement `#include "cs.h"`, as must every program that uses CSparse. It defines the `cs` sparse matrix data structure and the prototypes for each function. The `cs.h` file is split into three sections (primary, secondary, and tertiary routines). The routines in each set are described below. Each parameter or return value of CSparse is described as one of the following:

- **in**: The parameter must be defined on input. It is not modified.
- **in/out**: The parameter must be defined on input. It is modified on output.
- **out**: The memory space for the parameter must be allocated on input. Its contents are not defined on input. It is modified and defined on output.
- **work**: The memory space for the parameter must be allocated on input. Its contents are typically not defined on input or output.
- **returns**: The return value of each CSparse function. CSparse functions that return a pointer return `NULL` if an error occurs.

If not otherwise specified, `m` is the number of rows in the sparse matrix, `n` is the number of columns, and all CSparse routines expect matrices in compressed-column form. The `order` parameter is used in several routines: `0` results in the

natural ordering; 1 is a minimum degree ordering of $A + A^T$; 2 is a minimum degree ordering of $S^T S$ where $S = A$, except rows with more than $10\sqrt{n}$ entries are removed; and 3 is a minimum degree ordering of $A^T A$.

9.1 Primary CSparse routines and definitions

cs: sparse matrix in compressed-column or triplet form

```
typedef struct cs_sparse    /* matrix in compressed-column or triplet form */
{
    int nzmax ;     /* maximum number of entries */
    int m ;         /* number of rows */
    int n ;         /* number of columns */
    int *p ;        /* column pointers (size n+1) or col indices (size nzmax) */
    int *i ;        /* row indices, size nzmax */
    double *x ;     /* numerical values, size nzmax */
    int nz ;        /* # of entries in triplet matrix, -1 for compressed-col */
} cs ;
```

cs_add: $C = \alpha A + \beta B$

```
cs *cs_add (const cs *A, const cs *B, double alpha, double beta) ;
```

Adds two sparse matrices, $C = \alpha A + \beta B$.

`A`	in	sparse matrix
`B`	in	sparse matrix
`alpha`	in	scalar
`beta`	in	scalar
	returns	`C=alpha*A+beta*B`; `NULL` on error

cs_cholsol: solve $Ax = b$ using Cholesky factorization

```
int cs_cholsol (int order, const cs *A, double *b) ;
```

Solves $Ax = b$, where A is symmetric positive definite.

`order`	in	ordering method to use (`0` or `1`)
`A`	in	sparse matrix; only upper triangular part used
`b`	in/out	size `n`; b on input, x on output
	returns	`1` if successful; `0` on error

cs_compress: triplet form to compressed-column conversion

```
cs *cs_compress (const cs *T) ;
```

Converts a triplet-form matrix `T` into a compressed-column matrix `C`. The columns of `C` are not sorted, and duplicate entries may be present in `C`.

`T`	in	sparse matrix in triplet form
	returns	`C` if successful; `NULL` on error

cs_dupl: remove duplicate entries

```
int cs_dupl (cs *A) ;
```

Removes and sums duplicate entries in a sparse matrix.

`A`	in/out	sparse matrix; duplicates summed on output
	returns	`1` if successful; `0` on error

cs_entry: add an entry to a triplet-form matrix

```
int cs_entry (cs *T, int i, int j, double x) ;
```

Memory-space and dimension of T are increased if necessary.

T	in/out	triplet matrix; new entry added on output
i	in	row index of new entry
j	in	column index of new entry
x	in	numerical value of new entry
	returns	1 if successful; 0 on error

cs_gaxpy: $y = Ax + y$

```
int cs_gaxpy (const cs *A, const double *x, double *y) ;
```

Sparse matrix times dense column vector, $y = Ax + y$.

A	in	sparse matrix
x	in	size n
y	in/out	size m
	returns	1 if successful; 0 on error

cs_load: load a matrix from a file

```
cs *cs_load (FILE *f) ;
```

Loads a triplet matrix T from a file. Each line of the file contains three values: a row index i, a column index j, and a numerical value a_{ij}. The file is zero-based.

f	in	file pointer to an open file
	returns	T if successful; 0 on error

cs_lusol: solve $Ax = b$ using LU factorization

```
int cs_lusol (int order, const cs *A, double *b, double tol) ;
```

Solves $Ax = b$, where A is square and nonsingular. The diagonal entry is selected if $a_{kk}^{[k-1]} \geq \texttt{tol} \times \max |a_{*k}^{[k-1]}|$. Partial pivoting if `tol=1`.

order	in	ordering method to use (0 to 3)
A	in	sparse matrix
b	in/out	size n; b on input, x on output
tol	in	partial pivoting tolerance
	returns	1 if successful; 0 on error

cs_multiply: $C = AB$

```
cs *cs_multiply (const cs *A, const cs *B) ;
```

Sparse matrix multiplication, $C = AB$.

A	in	sparse matrix
B	in	sparse matrix
	returns	C=A*B; NULL on error

cs_norm: matrix 1-norm

```
double cs_norm (const cs *A) ;
```

Computes the 1-norm of a sparse matrix.

`A`	in	sparse matrix
	returns	the 1-norm if successful; `-1` on error

cs_print: print a sparse matrix

```
int cs_print (const cs *A, int brief) ;
```

Prints a compressed-column or triplet-form sparse matrix.

`A`	in	sparse matrix
`brief`	in	print all of `A` if zero, a few entries otherwise
	returns	`1` if successful; `0` on error

cs_qrsol: solve a least squares or underdetermined problem

```
int cs_qrsol (int order, const cs *A, double *b) ;
```

Solves a least squares problem ($\min \|Ax-b\|_2$, where A is m-by-n with $m \geq n$), or an underdetermined system ($Ax = b$, where $m < n$).

`order`	in	ordering method to use (`0` or `3`)
`A`	in	sparse matrix
`b`	in/out	size `max(m,n)`; b (size `m`) on input, x (size `n`) on output
	returns	`1` if successful; `0` on error

cs_transpose: $C = A^T$

```
cs *cs_transpose (const cs *A, int values) ;
```

`A`	in	sparse matrix
`values`	in	pattern only if `0`, both pattern and values otherwise
	returns	`C=A'`; `NULL` on error

Primary CSparse utilities

cs_calloc: allocate and clear memory

```
void *cs_calloc (int n, size_t size) ;
```

`n`	in	number of items to allocate
`size`	in	size of each item in bytes
	returns	pointer to allocated block if successful; `NULL` on error

cs_free: free memory

```
void *cs_free (void *p) ;
```

`p`	in/out	block to free
	returns	`NULL`

cs_realloc: change size of a block of memory

```
void *cs_realloc (void *p, int n, size_t size, int *ok) ;
```

`p`	in/out	block to change
`n`	in	new size of the block in number of items
`size`	in	size of each item, in bytes
`ok`	out	`1` if successful; `0` on error
	returns	pointer to possibly moved block of memory

cs_spalloc: allocate a sparse matrix

```
cs *cs_spalloc (int m, int n, int nzmax, int values, int triplet) ;
```

Allocates a sparse matrix in either compressed-column or triplet form.

`m`	in	number of rows
`n`	in	number of columns
`nzmax`	in	maximum number of entries
`values`	in	allocate pattern only if 0, values and pattern otherwise
`triplet`	in	compressed-column if 0, triplet form otherwise
	returns	`A` if successful; `NULL` on error

cs_spfree: free a sparse matrix

```
cs *cs_spfree (cs *A) ;
```

Frees a sparse matrix, in either compressed-column or triplet form.

`A`	in/out	sparse matrix to free
	returns	`NULL`

cs_sprealloc: reallocate a sparse matrix

```
int cs_sprealloc (cs *A, int nzmax) ;
```

Changes the maximum number of entries a sparse matrix can hold.

`A`	in/out	matrix to reallocate (compressed-column or triplet)
`nzmax`	in	new maximum number of entries
	returns	1 if successful; 0 on error

cs_malloc: allocate memory

```
void *cs_malloc (int n, size_t size) ;
```

Allocates an uninitialized block of memory of size `n` items.

`n`	in	number of items to allocate
`size`	in	size of each item in bytes
	returns	pointer to allocated block if successful; `NULL` on error

9.2 Secondary CSparse routines and definitions

css: symbolic analysis

```
typedef struct cs_symbolic  /* symbolic Cholesky, LU, or QR analysis */
{
    int *pinv ;     /* inverse row perm. for QR, fill red. perm for Chol */
    int *q ;        /* fill-reducing column permutation for LU and QR */
    int *parent ;   /* elimination tree for Cholesky and QR */
    int *cp ;       /* column pointers for Cholesky, row counts for QR */
    int *leftmost ; /* leftmost[i] = min(find(A(i,:))), for QR */
    int m2 ;        /* # of rows for QR, after adding fictitious rows */
    double lnz ;    /* # entries in L for LU or Cholesky; in V for QR */
    double unz ;    /* # entries in U for LU; in R for QR */
} css ;
```

csn: numeric factorization

```
typedef struct cs_numeric   /* numeric Cholesky, LU, or QR factorization */
{
    cs *L ;         /* L for LU and Cholesky, V for QR */
    cs *U ;         /* U for LU, R for QR, not used for Cholesky */
    int *pinv ;     /* partial pivoting for LU */
    double *B ;     /* beta [0..n-1] for QR */
} csn ;
```

csd: Dulmage–Mendelsohn decomposition

```
typedef struct cs_dmperm_results    /* cs_dmperm or cs_scc output */
{
    int *p ;        /* size m, row permutation */
    int *q ;        /* size n, column permutation */
    int *r ;        /* size nb+1, block k is rows r[k] to r[k+1]-1 in A(p,q) */
    int *s ;        /* size nb+1, block k is cols s[k] to s[k+1]-1 in A(p,q) */
    int nb ;        /* # of blocks in fine dmperm decomposition */
    int rr [5] ;    /* coarse row decomposition */
    int cc [5] ;    /* coarse column decomposition */
} csd ;
```

cs_amd: approximate minimum degree ordering

```
int *cs_amd (int order, const cs *A) ;
```

Minimum degree ordering of $A + A^T$ or $A^T A$.

`order`	in	ordering method to use (0 to 3)
`A`	in	matrix to order
	returns	size `n` permutation; `NULL` on error or for natural ordering

cs_chol: Cholesky factorization

```
csn *cs_chol (const cs *A, const css *S) ;
```

Cholesky factorization $LL = PAP^T$.

`A`	in	matrix to factorize, only upper triangular part used
`S`	in	symbolic analysis from `cs_schol`
	returns	numeric factorization `N`; `NULL` on error

cs_dmperm: Dulmage–Mendelsohn decomposition

```
csd *cs_dmperm (const cs *A, int seed) ;
```

Dulmage–Mendelsohn decomposition. `seed` optionally selects a randomized algorithm.

`A`	in	matrix to order
`seed`	in	0: natural, -1: reverse, random order otherwise
	returns	Dulmage–Mendelsohn analysis `D`; `NULL` on error

cs_droptol: drop small entries

```
int cs_droptol (cs *A, double tol) ;
```

Removes entries from a matrix with absolute value $\leq$ `tol`.

`A`	in/out	matrix to remove entries from
`tol`	in	drop tolerance
	returns	1 if successful; 0 on error

cs_dropzeros: drop exact zeros

```
int cs_dropzeros (cs *A) ;
```

Removes numerically zero entries from a matrix.

A	in/out	matrix to remove entries from
	returns	1 if successful; 0 on error

cs_happly: apply a Householder reflection

```
int cs_happly (const cs *V, int i, double beta, double *x) ;
```

Applies a Householder reflection to a dense vector, $x = (I - \beta vv^T)x$.

V	in	matrix of Householder vectors
i	in	$v =$ V(:,i), the ith column of V
beta	in	the scalar β
x	in/out	the vector x of size m
	returns	1 if successful; 0 on error

cs_ipvec: $x = P^T b$

```
int cs_ipvec (const int *p, const double *b, double *x, int n) ;
```

Permutes a vector; $x = P^T b$. In MATLAB notation, x(p)=b.

p	in	permutation vector
b	in	input vector
x	out	x(p)=b, output vector
n	in	length of p, b, and x
	returns	1 if successful; 0 on error

cs_lsolve: solve a lower triangular system $Lx = b$

```
int cs_lsolve (const cs *L, double *x) ;
```

Solves a lower triangular system $Lx = b$, where x and b are dense vectors. The diagonal of L must be the first entry of each column.

L	in	lower triangular matrix
x	in/out	size n; right-hand side on input, solution on output
	returns	1 if successful; 0 on error

cs_ltsolve: solve an upper triangular system $L^T x = b$

```
int cs_ltsolve (const cs *L, double *x) ;
```

Same as cs_lsolve, except it solves $L^T x = b$ instead.

cs_lu: sparse LU factorization

```
csn *cs_lu (const cs *A, const css *S, double tol) ;
```

Sparse LU factorization of a square matrix, $PAQ = LU$.

A	in	matrix to factorize
S	in	symbolic analysis from cs_sqr
tol	in	partial pivoting threshold (1 for partial pivoting)
	returns	numeric factorization N; NULL on error

cs_permute: $C = PAQ$

```
cs *cs_permute (const cs *A, const int *p, const int *q, int values) ;
```

Permutes a sparse matrix; $C = PAQ$, or `C=A(p,q)` in MATLAB notation.

`A`	in	`m`-by-`n` matrix to permute
`p`	in	a permutation vector of length `m`
`q`	in	a permutation vector of length `n`
`values`	in	allocate pattern only if `0`, values and pattern otherwise
	returns	`C=A(p,q)`; `NULL` on error

cs_pinv: invert a permutation vector

```
int *cs_pinv (const int *p, int n) ;
```

Inverts a permutation vector. Returns `pinv[i]=k` if `p[k]=i` on input.

`p`	in	a permutation vector of length `n`
`n`	in	length of `p`
	returns	`pinv`, an integer vector of length `n`; `NULL` on error

cs_pvec: $x = Pb$

```
int cs_pvec (const int *p, const double *b, double *x, int n) ;
```

Permutes a vector; $x = Pb$. In MATLAB notation, `x=b(p)`.

`p`	in	permutation vector
`b`	in	input vector
`x`	out	`x=b(p)`, output vector
`n`	in	length of `p`, `b`, and `x`
	returns	`1` if successful; `0` on error

cs_qr: sparse QR factorization

```
csn *cs_qr (const cs *A, const css *S) ;
```

Sparse QR factorization of an m-by-n matrix A, where $m \geq n$.

`A`	in	matrix to factorize
`S`	in	symbolic analysis from `cs_sqr`
	returns	numeric factorization `N`; `NULL` on error

cs_schol: symbolic Cholesky ordering and analysis

```
css *cs_schol (int order, const cs *A) ;
```

Symbolic ordering and analysis for a Cholesky factorization.

`order`	in	ordering option (`0` or `1`)
`A`	in	matrix to factorize
	returns	`S`, symbolic analysis for `cs_chol`; `NULL` on error

cs_sqr: symbolic QR or LU ordering and analysis

```
css *cs_sqr (int order, const cs *A, int qr) ;
```

Symbolic ordering and analysis for a QR or LU factorization.

`order`	in	ordering method to use (`0` to `3`)
`A`	in	matrix to factorize
`qr`	in	analyze for QR if nonzero or LU if zero
	returns	`S`, symbolic analysis for `cs_qr` or `cs_lu`; `NULL` on error

cs_symperm: $C = PAP^T$ for a symmetric matrix A

```
cs *cs_symperm (const cs *A, const int *pinv, int values) ;
```

Symmetric permutation of a symmetric matrix A.

`A`	in	matrix to permute (only upper triangular part used)
`pinv`	in	size n; inverse permutation
`values`	in	allocate pattern only if 0, values and pattern otherwise
	returns	`C=A(p,p)`; `NULL` on error

cs_updown: rank-1 update/downdate $LL^T \pm cc^T$

```
int cs_updown (cs *L, int sigma, const cs *C, const int *parent) ;
```

Sparse Cholesky rank-1 update/downdate. The nonzero pattern $\mathcal{C}$ of c must be a subset of the nonzero pattern of column k of L, where $k = \min \mathcal{C}$.

`L`	in/out	factorization to update/downdate
`sigma`	in	+1 for update, -1 for downdate
`C`	in	the vector c
`parent`	in	the elimination tree of `L`
	returns	1 if successful; 0 on error

cs_usolve: solve an upper triangular system $Ux = b$

```
int cs_usolve (const cs *U, double *x) ;
```

Solves an upper triangular system $Ux = b$, where x and b are dense vectors. The diagonal of `U` must be the last entry of each column.

`U`	in	upper triangular matrix
`x`	in/out	size n; right-hand side on input, solution on output
	returns	1 if successful; 0 on error

cs_utsolve: solve a lower triangular system $U'x = b$

```
int cs_utsolve (const cs *U, double *x) ;
```

Same as `cs_usolve`, except it solves $U^T x = b$ instead.

Secondary CSparse utilities

cs_sfree: free a css symbolic analysis

```
css *cs_sfree (css *S) ;
```

`S`	in/out	symbolic analysis to free
	returns	`NULL`

cs_nfree: free a csn numeric factorization

```
csn *cs_nfree (csn *N) ;
```

`N`	in/out	numeric factorization to free
	returns	`NULL`

cs_dfree: free a csd Dulmage–Mendelsohn decomposition

```
csd *cs_dfree (csd *D) ;
```

`D`	in/out	decomposition to free
	returns	`NULL`

9.3 Tertiary CSparse routines and definitions

cs_counts: column counts for Cholesky factorization of A or A^TA

```
int *cs_counts (const cs *A, const int *parent, const int *post, int ata) ;
```

A	in	matrix to analyze
parent	in	elimination tree of A
post	in	postordering of parent
ata	in	analyze A if zero, A^TA otherwise
	returns	c, a vector of length n; NULL on error

cs_cumsum: cumulative sum of an integer vector

```
double cs_cumsum (int *p, int *c, int n) ;
```

p	out	size n+1; cumulative sum of c
c	in/out	size n; overwritten with p[0..n-1] on output
n	in	length of c; length of p is n+1
	returns	sum(c); -1 on error

cs_dfs: depth-first search of a directed graph

```
int cs_dfs (int j, cs *G, int top, int *xi, int *pstack, const int *pinv) ;
```

j	in	starting node
G	in	graph to search (G->p modified, then restored)
top	in	stack xi[top ... n-1] in use on input
xi	in/out	size n; stack containing nodes traversed
pstack	work	size n
pinv	in	mapping of rows to columns of G, ignored if NULL
	returns	new value of top; -1 on error

cs_ereach: nonzero pattern of kth row of Cholesky factor, L(k,1:k-1)

```
int cs_ereach (const cs *A, int k, const int *parent, int *s, int *w) ;
```

A	in	L is the Cholesky factor of A
k	in	find kth row of L
parent	in	elimination tree of A
s	out	size n; s[top...n-1] is nonzero pattern of L(k,1:k-1)
w	work	size n; w[0...n-1]>=0 on input, unchanged on output
	returns	top; -1 on error

cs_etree: elimination tree of A or A^TA

```
int *cs_etree (const cs *A, int ata) ;
```

A	in	matrix to analyze
ata	in	analyze A if zero, A^TA otherwise
	returns	parent of size n; NULL on error

cs_fkeep: drop entries from a sparse matrix

```
int cs_fkeep (cs *A, int (*fkeep) (int, int, double, void *), void *other) ;
```

`A`	in/out	matrix to remove entries from
`fkeep`	in	drop a_{ij} if `fkeep(i,j,aij,other)` is zero.
`other`	any	optional parameter to `fkeep`
	returns	`nz`, new number of entries in `A`; `-1` on error

cs_house: compute a Householder reflection

```
double cs_house (double *x, double *beta, int n) ;
```

Computes a Householder reflection $H = I - \beta vv^T$ so that $(Hx)_{2\ldots n} = 0$.

`x`	in/out	x on input, v on output
`beta`	out	the scalar β
`n`	in	the length of `x`
	returns	$\|x\|_2$; `-1` on error

cs_leaf: determine if j is a leaf and find least common ancestor

Determine if j is a leaf of T^i and return `q` $= lca(j_{\text{prev}}, j)$. See page 51.

cs_maxtrans: maximum matching

```
int *cs_maxtrans (const cs *A, int seed) ;
```

Finds a zero-free diagonal. `seed` optionally selects a randomized algorithm.

`A`	in	matrix to find matching for
`seed`	in	0: natural, `-1`: reverse, randomized otherwise
	returns	`jimatch`, row and column matching, size `m+n`

cs_post: postorder a tree or forest

```
int *cs_post (const int *parent, int n) ;
```

`parent`	in	defines a tree of `n` nodes
`n`	in	length of `parent`
	returns	`post[k]=i`, of size `n`; `NULL` on error

cs_randperm: random permutation

```
int *cs_randperm (int n, int seed) ;
```

`n`	in	length of `p`
`seed`	in	0: natural, `-1`: reverse, random `p` otherwise
	returns	random perm. `p`; `NULL` on error or for natural order

cs_reach: nonzero pattern of sparse triangular solve

```
int cs_reach (cs *G, const cs *B, int k, int *xi, const int *pinv) ;
```

`G`	in	graph to search (`G->p` modified, then restored)
`B`	in	right-hand side, $b = B(:,k)$
`k`	in	use `k`th column of `B`
`xi`	out	size `2*n`; output in `xi[top...n-1]`
`pinv`	in	mapping of rows to columns of `G`, ignored if `NULL`
	returns	`top`; `-1` on error

cs_scatter: scatter a sparse vector

```
int cs_scatter (const cs *A, int j, double beta, int *w, double *x, int mark,
    cs *C, int nz) ;
```

Scatters and sums a sparse vector `A(:,j)` into a dense vector, `x=x+beta*A(:,j)`.

`A`	in	the sparse vector is `A(:,j)`
`j`	in	the column of `A` to use
`beta`	in	scalar multiplied by `A(:,j)`
`w`	in/out	size `m`; node `i` is marked if `w[i]=mark`
`x`	in/out	size `m`; ignored if `NULL`
`mark`	in	mark value for `w`
`C`	in/out	pattern of `x` accumulated in `C->i`
`nz`	in	pattern of `x` placed in `C` starting at `C->i [nz]`
	returns	new value of `nz`; `-1` on error

cs_scc: strongly connected components of a square matrix

```
csd *cs_scc (cs *A) ;
```

`A`	in	matrix to analyze (`A->p` modified then restored)
	returns	strongly connected components `D`; `NULL` on error

cs_spsolve: sparse lower or upper triangular solve, $Lx = b$ or $Ux = b$

```
int cs_spsolve (cs *G, const cs *B, int k, int *xi, double *x,
    const int *pinv, int lo) ;
```

If `lo` is zero, $Ux = B_{*k}$ is solved, where $G = U$ is upper triangular. Otherwise $Lx = B_{*k}$ is solved instead, where $G = L$. Both $b = B_{*k}$ and x are sparse; $\mathcal{X}$ is the nonzero pattern of x.

`G`	in	lower or upper triangular matrix (L or U)
`B`	in	right-hand side, $b = B_{*k}$
`k`	in	use `k`th column of `B` as right-hand side
`xi`	out	size `2*n`; $\mathcal{X}$ in `xi[top...n-1]`
`x`	out	size `n`; x in `x[xi[top...n-1]]`
`pinv`	in	mapping of rows to columns of `L`, ignored if `NULL`
`lo`	in	`1` if lower triangular, `0` if upper
	returns	`top`; `-1` on error

cs_tdfs: postorder a tree

```
int cs_tdfs (int j, int k, int *head, const int *next, int *post,
    int *stack) ;
```

All arrays are of size `n`, where `n` is the number of nodes in the tree.

`j`	in	postorder the tree rooted at node `j`
`k`	in	number of nodes ordered so far
`head`	in/out	`head[i]` is first child of node `i`; `-1` on output
`next`	in	`next[i]` is next sibling of `i` or `-1` if none
`post`	in/out	postordering
`stack`	work	size `n`
	returns	new value of `k`; `-1` on error

Tertiary CSparse utilities

cs_dalloc: allocate a csd object

```
csd *cs_dalloc (int m, int n) ;
```

Allocates a csd object (a Dulmage–Mendelsohn decomposition).

m	in	number of rows of the matrix A to be analyzed
n	in	number of columns of A
	returns	Dulmage–Mendelsohn decomposition D; NULL on error

cs_ddone: return a csd Dulmage–Mendelsohn decomposition

```
csd *cs_ddone (csd *D, cs *C, void *w, int ok) ;
```

Frees internally allocated workspace and returns a csd result or NULL if an error has occurred. The result is freed if ok is zero.

D	in/out	csd result
C	in/out	temporary sparse matrix to free
w	in/out	workspace to free
ok	in	free D if zero, keep D otherwise
	returns	D; NULL on error

cs_done: free workspace and return a sparse matrix result

```
cs *cs_done (cs *C, void *w, void *x, int ok) ;
```

Frees internally allocated workspace and returns a sparse matrix result or NULL if an error has occurred. The result is freed if ok is zero.

C	in/out	sparse matrix result
w	in/out	workspace to free
x	in/out	workspace to free
ok	in	free C if zero, keep C otherwise
	returns	C; NULL on error

cs_idone: free workspace and return an int array result

```
int *cs_idone (int *p, cs *C, void *w, int ok) ;
```

Frees internally allocated workspace and returns an int array result or NULL if an error has occurred. The result is freed if ok is zero.

p	in/out	int array result
C	in/out	temporary sparse matrix to free
w	in/out	workspace to free
ok	in	free p if zero, keep p otherwise
	returns	p; NULL on error

cs_ndone: return a csn numeric factorization

```
csn *cs_ndone (csn *N, cs *C, void *w, void *x, int ok) ;
```

Frees internally allocated workspace and returns a numeric factorization result or NULL if an error has occurred. The result is freed if ok is zero.

N	in/out	numeric factorization result
C	in/out	temporary sparse matrix to free
w	in/out	workspace to free
x	in/out	workspace to free
ok	in	free N if zero, keep N otherwise
	returns	N; NULL on error

Macros

The cs.h include file starts with the following lines. The _CS_H definition ensures that cs.h can be included multiple times without causing an error. Four ANSI standard definition files are included (stdlib.h, limits.h, math.h, and stdio.h). The MATLAB mex.h file is included if CSparse is being compiled for MATLAB. Next, the CSparse version numbers, release date, and copyright are defined (for use in cs_print).

```
#ifndef _CS_H
#define _CS_H
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#ifdef MATLAB_MEX_FILE
#include "mex.h"
#endif
#define CS_VER 2                    /* CSparse Version 2.0.1 */
#define CS_SUBVER 0
#define CS_SUBSUB 1
#define CS_DATE "May 27, 2006"      /* CSparse release date */
#define CS_COPYRIGHT "Copyright (c) Timothy A. Davis, 2006"
```

Each one-line prototype described in this chapter is listed in cs.h. The cs.h file ends with the following lines, which define the CSparse macros.

```
#define CS_MAX(a,b) (((a) > (b)) ? (a) : (b))
#define CS_MIN(a,b) (((a) < (b)) ? (a) : (b))
#define CS_FLIP(i) (-(i)-2)
#define CS_UNFLIP(i) (((i) < 0) ? CS_FLIP(i) : (i))
#define CS_MARKED(w,j) (w [j] < 0)
#define CS_MARK(w,j) { w [j] = CS_FLIP (w [j]) ; }
#define CS_CSC(A) (A && (A->nz == -1))
#define CS_TRIPLET(A) (A && (A->nz >= 0))
#endif
```

9.4 Examples

The three example programs below exercise every routine and nearly every line of code in CSparse (all but out-of-memory condition handling).

cs_demo1: file I/O and basic matrix operations

The complete program cs_demo1 reads in a triplet matrix from stdin and performs various matrix operations.

```
#include "cs.h"
int main (void)
{
    cs *T, *A, *Eye, *AT, *C, *D ;
    int i, m ;
    T = cs_load (stdin) ;                /* load triplet matrix T from stdin */
    printf ("T:\n") ; cs_print (T, 0) ; /* print T */
    A = cs_compress (T) ;                /* A = compressed-column form of T */
    printf ("A:\n") ; cs_print (A, 0) ; /* print A */
    cs_spfree (T) ;                      /* clear T */
    AT = cs_transpose (A, 1) ;           /* AT = A' */
    printf ("AT:\n") ; cs_print (AT, 0) ; /* print AT */
    m = A ? A->m : 0 ;                   /* m = # of rows of A */
    T = cs_spalloc (m, m, m, 1, 1) ;     /* create triplet identity matrix */
    for (i = 0 ; i < m ; i++) cs_entry (T, i, i, 1) ;
    Eye = cs_compress (T) ;              /* Eye = speye (m) */
    cs_spfree (T) ;
    C = cs_multiply (A, AT) ;            /* C = A*A' */
    D = cs_add (C, Eye, 1, cs_norm (C)) ;   /* D = C + Eye*norm (C,1) */
    printf ("D:\n") ; cs_print (D, 0) ; /* print D */
    cs_spfree (A) ;                      /* clear A AT C D Eye */
    cs_spfree (AT) ;
    cs_spfree (C) ;
    cs_spfree (D) ;
    cs_spfree (Eye) ;
    return (0) ;
}
```

The `t1` file can be used as input to `cs_demo1`. It contains the triplet form of the matrix used in Section 2.1:

```
2 2 3.0
1 0 3.1
3 3 1.0
0 2 3.2
1 1 2.9
3 0 3.5
3 1 0.4
1 3 0.9
0 0 4.5
2 1 1.7
```

The `cs_demo1.m` script below is the MATLAB equivalent for the C program `cs_demo1`, except that the CSparse results are compared with the same operations in MATLAB. The MATLAB `load` statement can read a triplet-form matrix in the same format as the `t1` file above, except that MATLAB expects its matrices to be 1-based. MATLAB always returns matrices with sorted columns.

```
%CS_DEMO1: MATLAB version of the CSparse/Demo/cs_demo1.c program.
% Uses both MATLAB functions and CSparse mexFunctions, and compares the two
% results.  This demo also plots the results, which the C version does not do.

load ../../Matrix/t1
T = t1
A  = sparse    (T(:,1)+1, T(:,2)+1, T(:,3))
A2 = cs_sparse (T(:,1)+1, T(:,2)+1, T(:,3))
```

```
fprintf ('A difference: %g\n', norm (A-A2,1)) ;
% CSparse/Demo/cs_demo1.c also clears the triplet matrix T at this point:
% clear T
clf
subplot (2,2,1) ; cspy (A) ; title ('A', 'FontSize', 16) ;
AT = A'
AT2 = cs_transpose (A)
fprintf ('AT difference: %g\n', norm (AT-AT2,1)) ;
subplot (2,2,2) ; cspy (AT) ; title ('A''', 'FontSize', 16) ;
n = size (A,2) ;
I = speye (n) ;
C = A*AT ;
C2 = cs_multiply (A, AT)
fprintf ('C difference: %g\n', norm (C-C2,1)) ;
subplot (2,2,3) ; cspy (C) ; title ('C=A*A''', 'FontSize', 16) ;
cnorm = norm (C,1) ;
D = C + I*cnorm
D2 = cs_add (C, I, 1, cnorm)
fprintf ('D difference: %g\n', norm (D-D2,1)) ;
subplot (2,2,4) ; cspy (D) ; title ('D=C+I*norm(C,1)', 'FontSize', 16) ;
% CSparse/Demo/cs_demo1.c clears all matrices at this point:
% clear A AT C D I
% clear A2 AT2 C2 D2
```

The output of the C program `cs_demo1` is given below. Compare it with the triplet and compressed-column matrices defined in Section 2.1. Also compare it with the output of the MATLAB equivalent code above. The maximum number of entries that T can hold is 16; it was doubled four times from its original size of one entry.

```
T:
CSparse Version 2.0.1, May 27, 2006.  Copyright (c) Timothy A. Davis, 2006
triplet: 4-by-4, nzmax: 16 nnz: 10
    2 2 : 3
    1 0 : 3.1
    3 3 : 1
    0 2 : 3.2
    1 1 : 2.9
    3 0 : 3.5
    3 1 : 0.4
    1 3 : 0.9
    0 0 : 4.5
    2 1 : 1.7
A:
CSparse Version 2.0.1, May 27, 2006.  Copyright (c) Timothy A. Davis, 2006
4-by-4, nzmax: 10 nnz: 10, 1-norm: 11.1
    col 0 : locations 0 to 2
      1 : 3.1
      3 : 3.5
      0 : 4.5
    col 1 : locations 3 to 5
      1 : 2.9
      3 : 0.4
      2 : 1.7
    col 2 : locations 6 to 7
      2 : 3
      0 : 3.2
    col 3 : locations 8 to 9
```

```
      3 : 1
      1 : 0.9
AT:
CSparse Version 2.0.1, May 27, 2006.  Copyright (c) Timothy A. Davis, 2006
4-by-4, nzmax: 10 nnz: 10, 1-norm: 7.7
    col 0 : locations 0 to 1
      0 : 4.5
      2 : 3.2
    col 1 : locations 2 to 4
      0 : 3.1
      1 : 2.9
      3 : 0.9
    col 2 : locations 5 to 6
      1 : 1.7
      2 : 3
    col 3 : locations 7 to 9
      0 : 3.5
      1 : 0.4
      3 : 1
D:
CSparse Version 2.0.1, May 27, 2006.  Copyright (c) Timothy A. Davis, 2006
4-by-4, nzmax: 16 nnz: 16, 1-norm: 139.58
    col 0 : locations 0 to 3
      1 : 13.95
      3 : 15.75
      0 : 100.28
      2 : 9.6
    col 1 : locations 4 to 7
      1 : 88.62
      3 : 12.91
      0 : 13.95
      2 : 4.93
    col 2 : locations 8 to 11
      1 : 4.93
      3 : 0.68
      2 : 81.68
      0 : 9.6
    col 3 : locations 12 to 15
      1 : 12.91
      3 : 83.2
      0 : 15.75
      2 : 0.68
```

cs_demo2: matrix factorization

The cs_demo2 program reads a matrix from a file in triplet form. If it is upper (or lower) triangular, the matrix is assumed to be symmetric, and the lower (or upper) triangular part is added. The matrix is analyzed with cs_dmperm and the number of blocks, number of singletons (1-by-1 blocks), and structural rank are printed. If the matrix is symmetric, a Cholesky factorization is used to solve $Ax = b$. If the matrix is square, the system is also solved using an LU factorization. Next, a least squares problem is solved using QR factorization. Each method is tested with a range of ordering options, and the relative residual $\|Ax - b\|/(\|A\|\|x\| + \|b\|)$ and run time are printed. The cs_demo.h file contains prototypes for the cs_demo2 and

cs_demo3 programs.

```
#include "cs.h"
typedef struct problem_struct
{
    cs *A ;
    cs *C ;
    int sym ;
    double *x ;
    double *b ;
    double *resid ;
} problem ;

problem *get_problem (FILE *f, double tol) ;
int demo2 (problem *Prob) ;
int demo3 (problem *Prob) ;
problem *free_problem (problem *Prob) ;
```

The cs_demo.c file contains functions used by several demo programs.

```
#include "cs_demo.h"
#include <time.h>
/* 1 if A is square & upper tri., -1 if square & lower tri., 0 otherwise */
static int is_sym (cs *A)
{
    int is_upper, is_lower, j, p, n = A->n, m = A->m, *Ap = A->p, *Ai = A->i ;
    if (m != n) return (0) ;
    is_upper = 1 ;
    is_lower = 1 ;
    for (j = 0 ; j < n ; j++)
    {
        for (p = Ap [j] ; p < Ap [j+1] ; p++)
        {
            if (Ai [p] > j) is_upper = 0 ;
            if (Ai [p] < j) is_lower = 0 ;
        }
    }
    return (is_upper ? 1 : (is_lower ? -1 : 0)) ;
}

/* true for off-diagonal entries */
static int dropdiag (int i, int j, double aij, void *other) { return (i != j) ;}

/* C = A + triu(A,1)' */
static cs *make_sym (cs *A)
{
    cs *AT, *C ;
    AT = cs_transpose (A, 1) ;          /* AT = A' */
    cs_fkeep (AT, &dropdiag, NULL) ;    /* drop diagonal entries from AT */
    C = cs_add (A, AT, 1, 1) ;          /* C = A+AT */
    cs_spfree (AT) ;
    return (C) ;
}

/* create a right-hand side */
static void rhs (double *x, double *b, int m)
{
    int i ;
```

```
    for (i = 0 ; i < m ; i++) b [i] = 1 + ((double) i) / m ;
    for (i = 0 ; i < m ; i++) x [i] = b [i] ;
}

/* infinity-norm of x */
static double norm (double *x, int n)
{
    int i ;
    double normx = 0 ;
    for (i = 0 ; i < n ; i++) normx = CS_MAX (normx, fabs (x [i])) ;
    return (normx) ;
}

/* compute residual, norm(A*x-b,inf) / (norm(A,1)*norm(x,inf) + norm(b,inf)) */
static void print_resid (int ok, cs *A, double *x, double *b, double *resid)
{
    int i, m, n ;
    if (!ok) { printf ("    (failed)\n") ; return ; }
    m = A->m ; n = A->n ;
    for (i = 0 ; i < m ; i++) resid [i] = -b [i] ;  /* resid = -b */
    cs_gaxpy (A, x, resid) ;                        /* resid = resid + A*x  */
    printf ("resid: %8.2e\n", norm (resid,m) / ((n == 0) ? 1 :
        (cs_norm (A) * norm (x,n) + norm (b,m)))) ;
}

static double tic (void) { return (clock () / (double) CLOCKS_PER_SEC) ; }
static double toc (double t) { double s = tic () ; return (CS_MAX (0, s-t)) ; }

static void print_order (int order)
{
    switch (order)
    {
        case 0: printf ("natural    ") ; break ;
        case 1: printf ("amd(A+A')  ") ; break ;
        case 2: printf ("amd(S'*S)  ") ; break ;
        case 3: printf ("amd(A'*A)  ") ; break ;
    }
}

/* read a problem from a file */
problem *get_problem (FILE *f, double tol)
{
    cs *T, *A, *C ;
    int sym, m, n, mn, nz1, nz2 ;
    problem *Prob ;
    Prob = cs_calloc (1, sizeof (problem)) ;
    if (!Prob) return (NULL) ;
    T = cs_load (f) ;                   /* load triplet matrix T from a file */
    Prob->A = A = cs_compress (T) ;     /* A = compressed-column form of T */
    cs_spfree (T) ;                     /* clear T */
    if (!cs_dupl (A)) return (free_problem (Prob)) ; /* sum up duplicates */
    Prob->sym = sym = is_sym (A) ;      /* determine if A is symmetric */
    m = A->m ; n = A->n ;
    mn = CS_MAX (m,n) ;
    nz1 = A->p [n] ;
    cs_dropzeros (A) ;                  /* drop zero entries */
    nz2 = A->p [n] ;
```

```
    if (tol > 0) cs_droptol (A, tol) ;  /* drop tiny entries (just to test) */
    Prob->C = C = sym ? make_sym (A) : A ;  /* C = A + triu(A,1)', or C=A */
    if (!C) return (free_problem (Prob)) ;
    printf ("\n--- Matrix: %d-by-%d, nnz: %d (sym: %d: nnz %d), norm: %8.2e\n",
            m, n, A->p [n], sym, sym ? C->p [n] : 0, cs_norm (C)) ;
    if (nz1 != nz2) printf ("zero entries dropped: %d\n", nz1 - nz2) ;
    if (nz2 != A->p [n]) printf ("tiny entries dropped: %d\n", nz2 - A->p [n]) ;
    Prob->b = cs_malloc (mn, sizeof (double)) ;
    Prob->x = cs_malloc (mn, sizeof (double)) ;
    Prob->resid = cs_malloc (mn, sizeof (double)) ;
    return ((!Prob->b || !Prob->x || !Prob->resid) ? free_problem (Prob) : Prob) ;
}

/* free a problem */
problem *free_problem (problem *Prob)
{
    if (!Prob) return (NULL) ;
    cs_spfree (Prob->A) ;
    if (Prob->sym) cs_spfree (Prob->C) ;
    cs_free (Prob->b) ;
    cs_free (Prob->x) ;
    cs_free (Prob->resid) ;
    return (cs_free (Prob)) ;
}

/* solve a linear system using Cholesky, LU, and QR, with various orderings */
int demo2 (problem *Prob)
{
    cs *A, *C ;
    double *b, *x, *resid,  t, tol ;
    int k, m, n, ok, order, nb, ns, *r, *s, *rr, sprank ;
    csd *D ;
    if (!Prob) return (0) ;
    A = Prob->A ; C = Prob->C ; b = Prob->b ; x = Prob->x ; resid = Prob->resid;
    m = A->m ; n = A->n ;
    tol = Prob->sym ? 0.001 : 1 ;               /* partial pivoting tolerance */
    D = cs_dmperm (C, 1) ;                      /* randomized dmperm analysis */
    if (!D) return (0) ;
    nb = D->nb ; r = D->r ; s = D->s ; rr = D->rr ;
    sprank = rr [3] ;
    for (ns = 0, k = 0 ; k < nb ; k++)
    {
        ns += ((r [k+1] == r [k]+1) && (s [k+1] == s [k]+1)) ;
    }
    printf ("blocks: %d singletons: %d structural rank: %d\n", nb, ns, sprank) ;
    cs_dfree (D) ;
    for (order = 0 ; order <= 3 ; order += 3)   /* natural and amd(A'*A) */
    {
        if (!order && m > 1000) continue ;
        printf ("QR   ") ;
        print_order (order) ;
        rhs (x, b, m) ;                         /* compute right-hand side */
        t = tic () ;
        ok = cs_qrsol (order, C, x) ;           /* min norm(Ax-b) with QR */
        printf ("time: %8.2f ", toc (t)) ;
        print_resid (ok, C, x, b, resid) ;      /* print residual */
    }
```

```
    if (m != n || sprank < n) return (1) ;      /* return if rect. or singular*/
    for (order = 0 ; order <= 3 ; order++)      /* try all orderings */
    {
        if (!order && m > 1000) continue ;
        printf ("LU   ") ;
        print_order (order) ;
        rhs (x, b, m) ;                         /* compute right-hand side */
        t = tic () ;
        ok = cs_lusol (order, C, x, tol) ;      /* solve Ax=b with LU */
        printf ("time: %8.2f ", toc (t)) ;
        print_resid (ok, C, x, b, resid) ;      /* print residual */
    }
    if (!Prob->sym) return (1) ;
    for (order = 0 ; order <= 1 ; order++)      /* natural and amd(A+A') */
    {
        if (!order && m > 1000) continue ;
        printf ("Chol ") ;
        print_order (order) ;
        rhs (x, b, m) ;                         /* compute right-hand side */
        t = tic () ;
        ok = cs_cholsol (order, C, x) ;         /* solve Ax=b with Cholesky */
        printf ("time: %8.2f ", toc (t)) ;
        print_resid (ok, C, x, b, resid) ;      /* print residual */
    }
    return (1) ;
}
```

The `cs_demo2.c` file contains just the main program itself.

```
#include "cs_demo.h"
/* cs_demo2: read a matrix and solve a linear system */
int main (void)
{
    problem *Prob = get_problem (stdin, 1e-14) ;
    demo2 (Prob) ;
    free_problem (Prob) ;
    return (0) ;
}
```

The output of `cs_demo2` for the `bcsstk01`, `fs_183_1`, `mbeacxc`, `west0067`, and `lp_afiro` matrices is shown below. One matrix (`mbeacxc`) is actually 496-by-496, but `cs_load` returns a matrix of size 492-by-490 as determined by the largest row and column index of nonzero entries in the matrix. The matrix has a numeric and structural rank of 448, which is why the residual is `nan`.

```
--- Matrix: 48-by-48, nnz: 224 (sym: -1: nnz 400), norm: 3.57e+09
blocks: 1 singletons: 0 structural rank: 48
QR   natural    time:     0.00 resid: 2.83e-19
QR   amd(A'*A)  time:     0.00 resid: 5.19e-19
LU   natural    time:     0.00 resid: 2.63e-19
LU   amd(A+A')  time:     0.00 resid: 8.63e-20
LU   amd(S'*S)  time:     0.00 resid: 2.04e-19
LU   amd(A'*A)  time:     0.00 resid: 2.04e-19
Chol natural    time:     0.00 resid: 1.90e-19
Chol amd(A+A')  time:     0.00 resid: 2.01e-19
```

```
--- Matrix: 183-by-183, nnz: 988 (sym: 0: nnz 0), norm: 1.70e+09
zero entries dropped: 71
tiny entries dropped: 10
blocks: 38 singletons: 37 structural rank: 183
QR   natural    time:     0.01 resid: 1.09e-27
QR   amd(A'*A)  time:     0.00 resid: 5.34e-28
LU   natural    time:     0.00 resid: 3.08e-28
LU   amd(A+A')  time:     0.00 resid: 1.42e-27
LU   amd(S'*S)  time:     0.00 resid: 7.11e-28
LU   amd(A'*A)  time:     0.00 resid: 7.11e-28

--- Matrix: 492-by-490, nnz: 49920 (sym: 0: nnz 0), norm: 9.29e-01
blocks: 10 singletons: 8 structural rank: 448
QR   natural    time:     0.24 resid:      nan
QR   amd(A'*A)  time:     0.28 resid:      nan

--- Matrix: 67-by-67, nnz: 294 (sym: 0: nnz 0), norm: 6.14e+00
blocks: 2 singletons: 1 structural rank: 67
QR   natural    time:     0.00 resid: 3.42e-17
QR   amd(A'*A)  time:     0.00 resid: 1.95e-17
LU   natural    time:     0.00 resid: 3.85e-17
LU   amd(A+A')  time:     0.00 resid: 1.95e-17
LU   amd(S'*S)  time:     0.00 resid: 2.60e-17
LU   amd(A'*A)  time:     0.00 resid: 2.60e-17

--- Matrix: 27-by-51, nnz: 102 (sym: 0: nnz 0), norm: 3.43e+00
blocks: 1 singletons: 0 structural rank: 27
QR   natural    time:     0.00 resid: 9.54e-17
QR   amd(A'*A)  time:     0.00 resid: 1.89e-16
```

cs_demo3: Cholesky update/downdate

The cs_demo3 program demonstrates the use of many of the secondary CSparse routines and utilities. It computes the Cholesky factorization of a matrix and then updates and downdates it. The first two functions are in the cs_demo.c file.

```
/* free workspace for demo3 */
static int done3 (int ok, css *S, csn *N, double *y, cs *W, cs *E, int *p)
{
    cs_sfree (S) ;
    cs_nfree (N) ;
    cs_free (y) ;
    cs_spfree (W) ;
    cs_spfree (E) ;
    cs_free (p) ;
    return (ok) ;
}

/* Cholesky update/downdate */
int demo3 (problem *Prob)
{
    cs *A, *C, *W = NULL, *WW, *WT, *E = NULL, *W2 ;
    int n, k, *Li, *Lp, *Wi, *Wp, p1, p2, *p = NULL, ok ;
    double *b, *x, *resid, *y = NULL, *Lx, *Wx, s,  t, t1 ;
    css *S = NULL ;
```

```
csn *N = NULL ;
if (!Prob || !Prob->sym || Prob->A->n == 0) return (0) ;
A = Prob->A ; C = Prob->C ; b = Prob->b ; x = Prob->x ; resid = Prob->resid;
n = A->n ;
if (!Prob->sym || n == 0) return (1) ;
rhs (x, b, n) ;                                 /* compute right-hand side */
printf ("\nchol then update/downdate ") ;
print_order (1) ;
y = cs_malloc (n, sizeof (double)) ;
t = tic () ;
S = cs_schol (1, C) ;                           /* symbolic Chol, amd(A+A') */
printf ("\nsymbolic chol time %8.2f\n", toc (t)) ;
t = tic () ;
N = cs_chol (C, S) ;                            /* numeric Cholesky */
printf ("numeric  chol time %8.2f\n", toc (t)) ;
if (!S || !N || !y) return (done3 (0, S, N, y, W, E, p)) ;
t = tic () ;
cs_ipvec (S->pinv, b, y, n) ;                   /* y = P*b */
cs_lsolve (N->L, y) ;                           /* y = L\y */
cs_ltsolve (N->L, y) ;                          /* y = L'\y */
cs_pvec (S->pinv, y, x, n) ;                    /* x = P'*y */
printf ("solve    chol time %8.2f\n", toc (t)) ;
printf ("original: ") ;
print_resid (1, C, x, b, resid) ;               /* print residual */
k = n/2 ;                                       /* construct W  */
W = cs_spalloc (n, 1, n, 1, 0) ;
if (!W) return (done3 (0, S, N, y, W, E, p)) ;
Lp = N->L->p ; Li = N->L->i ; Lx = N->L->x ;
Wp = W->p ; Wi = W->i ; Wx = W->x ;
Wp [0] = 0 ;
p1 = Lp [k] ;
Wp [1] = Lp [k+1] - p1 ;
s = Lx [p1] ;
srand (1) ;
for ( ; p1 < Lp [k+1] ; p1++)
{
    p2 = p1 - Lp [k] ;
    Wi [p2] = Li [p1] ;
    Wx [p2] = s * rand () / ((double) RAND_MAX) ;
}
t = tic () ;
ok = cs_updown (N->L, +1, W, S->parent) ;   /* update: L*L'+W*W' */
t1 = toc (t) ;
printf ("update:   time: %8.2f\n", t1) ;
if (!ok) return (done3 (0, S, N, y, W, E, p)) ;
t = tic () ;
cs_ipvec (S->pinv, b, y, n) ;                   /* y = P*b */
cs_lsolve (N->L, y) ;                           /* y = L\y */
cs_ltsolve (N->L, y) ;                          /* y = L'\y */
cs_pvec (S->pinv, y, x, n) ;                    /* x = P'*y */
t = toc (t) ;
p = cs_pinv (S->pinv, n) ;
W2 = cs_permute (W, p, NULL, 1) ;               /* E = C + (P'W)*(P'W)' */
WT = cs_transpose (W2,1) ;
WW = cs_multiply (W2, WT) ;
cs_spfree (WT) ;
cs_spfree (W2) ;
```

```
    E = cs_add (C, WW, 1, 1) ;
    cs_spfree (WW) ;
    if (!E || !p) return (done3 (0, S, N, y, W, E, p)) ;
    printf ("update:   time: %8.2f (incl solve) ", t1+t) ;
    print_resid (1, E, x, b, resid) ;           /* print residual */
    cs_nfree (N) ;                              /* clear N */
    t = tic () ;
    N = cs_chol (E, S) ;                        /* numeric Cholesky */
    if (!N) return (done3 (0, S, N, y, W, E, p)) ;
    cs_ipvec (S->pinv, b, y, n) ;               /* y = P*b */
    cs_lsolve (N->L, y) ;                       /* y = L\y */
    cs_ltsolve (N->L, y) ;                      /* y = L'\y */
    cs_pvec (S->pinv, y, x, n) ;                /* x = P'*y */
    t = toc (t) ;
    printf ("rechol:   time: %8.2f (incl solve) ", t) ;
    print_resid (1, E, x, b, resid) ;           /* print residual */
    t = tic () ;
    ok = cs_updown (N->L, -1, W, S->parent) ;   /* downdate: L*L'-W*W' */
    t1 = toc (t) ;
    if (!ok) return (done3 (0, S, N, y, W, E, p)) ;
    printf ("downdate: time: %8.2f\n", t1) ;
    t = tic () ;
    cs_ipvec (S->pinv, b, y, n) ;               /* y = P*b */
    cs_lsolve (N->L, y) ;                       /* y = L\y */
    cs_ltsolve (N->L, y) ;                      /* y = L'\y */
    cs_pvec (S->pinv, y, x, n) ;                /* x = P'*y */
    t = toc (t) ;
    printf ("downdate: time: %8.2f (incl solve) ", t1+t) ;
    print_resid (1, C, x, b, resid) ;           /* print residual */
    return (done3 (1, S, N, y, W, E, p)) ;
}
```

The cs_demo3.c file contains just the main program itself.

```
#include "cs_demo.h"
/* cs_demo3: read a matrix and test Cholesky update/downdate */
int main (void)
{
    problem *Prob = get_problem (stdin, 0) ;
    demo3 (Prob) ;
    free_problem (Prob) ;
    return (0) ;
}
```

The output of cs_demo3 for the bcsstk16 matrix is shown below.

```
--- Matrix: 4884-by-4884, nnz: 147631 (sym: -1: nnz 290378), norm: 7.01e+09

chol then update/downdate amd(A+A')
symbolic chol time     0.03
numeric  chol time     0.54
solve    chol time     0.01
original: resid: 9.28e-23
update:   time:     0.00
update:   time:     0.01 (incl solve) resid: 1.05e-23
rechol:   time:     0.55 (incl solve) resid: 8.72e-24
downdate: time:     0.01
downdate: time:     0.02 (incl solve) resid: 3.60e-22
```

Chapter 10

Sparse matrices in MATLAB

Almost all MATLAB operators and functions work seamlessly on both sparse and full matrices.[15] It is possible to write an efficient MATLAB M-file that can operate on either full or sparse matrices with no changes to the code. In MATLAB, "sparse" is an attribute of the data structure used to represent a matrix.

Sparsity propagates in MATLAB; if a function or operator has sparse operands, the result is usually sparse. A fixed set of rules determines the storage class (sparse or full) of the result. In general, unary functions and operators return a result of the same storage class as the input. For example, `chol(A)` is sparse if `A` is sparse and full otherwise. The result of a binary operator (`A+B`, for example) is sparse if both `A` and `B` are sparse and full if both `A` and `B` are full. If the operands are mixed, the result is usually full, unless the operation preserves sparsity (`[A B]`, `[A;B]`, and `A.*B` are sparse if either `A` or `B` are sparse, for example). The submatrix `A(i,j)` has the same type as `A`, unless it is a scalar (in which case `A(i,j)` is full). Submatrix assignment (`A(i,j) = ...`) leaves the storage class of `A` unchanged.

10.1 Creating sparse matrices

There are many ways of creating a sparse matrix in MATLAB, most of which rely on the MATLAB `sparse` function. Usually the fastest method is to create the triplet form first and then use `sparse`, much like how `cs_entry`, `cs_compress`, and `cs_dupl` can be used. Avoid using `for` loops. This example creates an n^2-by-n^2 matrix, corresponding to the second difference operator applied to an n-by-n mesh.

```
function A = mesh2d2 (n)
% create an n-by-n 2D mesh for the 2nd difference operator
nn = 1:n^2 ;
ii = [nn-n ; nn-1 ; nn ; nn+1 ; nn+n] ;
jj = repmat (nn, 5, 1) ;
xx = repmat ([-1 -1 4 -1 -1]', 1, n^2) ;
keep = find (ii >= 1 & ii <= n^2 & jj >= 1 & jj <= n^2) ;
A = sparse (ii (keep), jj (keep), xx (keep)) ;
```

[15] Since MATLAB refers to matrices as either `sparse` or `full`, that nomenclature is used here.

If `for` loops are unavoidable, at least try to create subsets of more than one entry at a time. This function computes the same matrix as `mesh2d2`. Preallocating a matrix at its final size is better than repeatedly appending new entries to it.

```
function A = mesh2d1 (n)
% create an n-by-n 2D mesh for the 2nd difference operator
ii = zeros (5*n^2, 1) ;     % preallocate ii, jj, and xx
jj = zeros (5*n^2, 1) ;
xx = zeros (5*n^2, 1) ;
k = 1 ;
for j = 0:n-1
    for i = 0:n-1
        s = j*n+i + 1 ;
        ii (k:k+4) = [(j-1)*n+i j*n+(i-1) j*n+i j*n+(i+1) (j+1)*n+i ] + 1 ;
        jj (k:k+4) = [s s s s s] ;
        xx (k:k+4) = [-1 -1 4 -1 -1] ;
        k = k + 5 ;
    end
end
% remove entries beyond the boundary
keep = find (ii >= 1 & ii <= n^2 & jj >= 1 & jj <= n^2) ;
A = sparse (ii (keep), jj (keep), xx (keep)) ;
```

Here is an example that creates a random finite-element matrix.

```
function A = frand (n,nel,s)
% A = frand (n,nel,s) creates an n-by-n sparse matrix consisting of nel finite
% elements, each of which are of size s-by-s with random symmetric nonzero
% pattern, plus the identity matrix.
ss = s^2 ;
nz = nel*ss ;
ii = zeros (nz,1) ;
jj = zeros (nz,1) ;
xx = zeros (nz,1) ;
k = 1 ;
for e = 1:nel
    i = 1 + fix (n * rand (s,1)) ;
    i = repmat (i, 1, s) ;
    j = i' ;
    x = rand (s,s) ;
    ii (k:k+ss-1) = i (:) ;
    jj (k:k+ss-1) = j (:) ;
    xx (k:k+ss-1) = x (:) ;
    k = k + ss ;
end
A = sparse (ii,jj,xx,n,n) + speye (n) ;
```

The `cs_frand` function is the CSparse version of `frand`.

```
cs *cs_frand (int n, int nel, int s)
{
    int ss = s*s, nz = nel*ss, e, i, j, *P ;
    cs *A, *T = cs_spalloc (n, n, nz, 1, 1) ;
    if (!T) return (NULL) ;
    P = cs_malloc (s, sizeof (int)) ;
    if (!P) return (cs_spfree (T)) ;
    for (e = 0 ; e < nel ; e++)
```

```
    {
        for (i = 0 ; i < s ; i++) P [i] = rand () % n ;
        for (j = 0 ; j < s ; j++)
        {
            for (i = 0 ; i < s ; i++)
            {
                cs_entry (T, P [i], P [j], rand () / (double) RAND_MAX) ;
            }
        }
    }
    for (i = 0 ; i < n ; i++) cs_entry (T, i, i, 1) ;
    A = cs_compress (T) ;
    cs_spfree (T) ;
    return (cs_dupl (A) ? A : cs_spfree (A)) ;
}
```

The *worst* way to create a sparse matrix is with a statement `A(i,j) = ...`, where `i` and `j` are scalars. Below are four methods of creating the same matrix `A`. The first method is an example of what *not* to do.

```
% method 1: A(i,j) = ...
rand ('state', 0) ;
A = sparse (n,n) ;
for k = 1:nz
    % compute some arbitrary entry and add it into the matrix
    i = 1 + fix (n * rand (1)) ;
    j = 1 + fix (n * rand (1)) ;
    x = rand (1) ;
    A (i,j) = A (i,j) + x ; % VERY slow, esp. if A(i,j) not already nonzero!
end
```

A better method, if the matrix is to be constructed one entry at a time, is to preallocate a triplet form, fill it, and then convert it to a sparse form.

```
% method 2: triplet form, one entry at a time
rand ('state', 0) ;
ii = zeros (nz, 1) ;    % preallocate ii, jj, and xx
jj = zeros (nz, 1) ;
xx = zeros (nz, 1) ;
for k = 1:nz
    % compute some arbitrary entry and add it into the matrix
    ii (k) = 1 + fix (n * rand (1)) ;
    jj (k) = 1 + fix (n * rand (1)) ;
    xx (k) = rand (1) ;
end
A = sparse (ii,jj,xx) ;
```

If the number of entries is unknown, the size of the triplet matrix can be increased as needed, just like `cs_entry`.

```
% method 3: triplet form, one entry at a time, pretend nz is unknown
rand ('state', 0) ;
len = 16 ;
ii = zeros (len, 1) ;
jj = zeros (len, 1) ;
xx = zeros (len, 1) ;
```

```
for k = 1:nz
    % compute some arbitrary entry and add it into the matrix
    if (k > len)
        % double the size of ii,jj,xx
        len = 2*len ;
        ii (len) = 0 ;
        jj (len) = 0 ;
        xx (len) = 0 ;
    end
    ii (k) = 1 + fix (n * rand (1)) ;
    jj (k) = 1 + fix (n * rand (1)) ;
    xx (k) = rand (1) ;
end
A = sparse (ii (1:k), jj (1:k), xx (1:k)) ;
```

Of course, the best method is to avoid `for` loops completely.

```
% method 4: avoid the for loop
rand ('state', 0) ;
e = rand (3, nz) ;
e (1,:) = 1 + fix (n * e (1,:)) ;
e (2,:) = 1 + fix (n * e (2,:)) ;
A = sparse (e (1,:), e (2,:), e (3,:)) ;
```

Each of the above four methods constructs the same matrix `A`. The first is exceedingly slow, taking $O(|A|^2)$ time. Methods 2 and 3 take about the same time, but method 4 is much faster (methods 2, 3, and 4 all take $O(|A|)$ time, however).

Finally, never create a full matrix `A` and then convert it to sparse. For example, `A=sparse(eye(n))` takes $O(n^2)$ time and memory, but `A=speye(n)` takes only $O(n)$ time and memory. The difference is dramatic for large `n`.

10.2 Sparse matrix functions and operators

The following is a list of the primary functions and operators for sparse matrices in MATLAB 7.2.

referencing and assigning submatrices:	
`subsref`	`...=A(i,j)` extracts a sparse submatrix of `A`. Matrix permutation is an important special case. `C=A(p,q)` is the same as $C = PAQ$ if `p` and `q` are permutations of `1:m` and `1:n`, respectively, that represent the permutation matrices P and Q (where `[m n]=size(A)`).
`subasgn`	`A(i,j)=...` modifies a sparse submatrix of `A`.
`subsindex`	`...=A(k)`, one-dimensional indexing.
elementary sparse matrices:	
`spdiags`	A generalization of `diag` that constructs a matrix from diagonals or extracts diagonals of a matrix.
`speye`	`I=speye(n)` is the sparse identity matrix; `I=speye(m,n)` is `m`-by-`n`.
`sprand`	`A=sprand(m,n,d)` is a random `m`-by-`n` sparse matrix with about `d*m*n` nonzero entries with values uniformly distributed in the range $[0, 1]$. `A=sprand(S)` is a random matrix with the same pattern as `S`.

elementary sparse matrices, continued:	
`sprandn`	Identical to `sprand` but with normally distributed random entries.
`sprandsym`	`A=sprandsym(n,d)` is a symmetric random matrix with about dn^2 entries. `A=sprandsym(n,d,rc,kind)` is also positive definite with a reciprocal condition number of `rc` (`kind` selects the method).

full to sparse conversion:	
`find`	`[i j x]=find(A)` extracts the triplet form of a matrix `A`; it is the opposite of `sparse`. With one output, `i=find(A)` returns the one-dimensional indices `i` of the nonzero entries in `A`.
`full`	`C=full(A)` converts a sparse matrix `A` into a full matrix `C`.
`sparse`	`A=sparse(C)` converts a full matrix `C` into sparse matrix `A`. `A=sparse(i,j,x,m,n,s)` converts a triplet form into an `m`-by-`n` sparse matrix `A` with space for `s` entries. `A=sparse(i,j,x,m,n)` uses `s=length(x)`. `A=sparse(i,j,x)` uses `m=max(i)` and `n=max(j)`, just like `cs_sparse`. `A=sparse(m,n)` is a sparse `m`-by-`n` matrix with all zero entries.
`spconvert`	`A=spconvert(x)` is `A=sparse(x(:,1),x(:,2),x(:,3))` if `x` has 3 columns. `A` is complex if `size(x,2)=4`.

working with sparse matrices:	
`issparse`	`issparse(A)` is 1 if `A` is sparse and 0 otherwise.
`nnz`	`nnz(A)` is the number of nonzeros in `A` (`A->p[A->n]` in CSparse).
`nonzeros`	`x` from `[i,j,x]=find(A)` (`A->x` in CSparse).
`nzmax`	`nzmax(A)` is the maximum number of nonzeros `A` can hold; this is increased when necessary (`A->nzmax` in CSparse).
`spalloc`	`A=spalloc(m,n,nzmax)` allocates a zero `m`-by-`n` sparse matrix with space for `nzmax` entries.
`spfun`	Applies a function to the entries of a matrix. `C=spfun(@f,A)` is a matrix with the same nonzero pattern as `A`, where `C(i,j)=f(A(i,j))` (zero entries are removed from `C`).
`spones`	`C=spones(A)` is a binary sparse matrix with the pattern of `A`.
`spparms`	Parameters for sparse methods. Try `spparms('spumoni',2)`.
`spy`	`spy(A)` plots a picture of a sparse matrix.

ordering methods:	
`amd`	Approximate minimum degree algorithm used in backslash, `lu`, and `chol`. `p=amd(A)` orders `A+A'` so that `chol(A(p,p))` tends to be sparser than `chol(A)`.
`colamd`	Column approximate minimum degree ordering used in backslash and `lu`. `q=colamd(A)` finds a column ordering so that `lu(A(:,q))` tends to be sparser than `lu(A)`. Dense rows are removed from `A` prior to ordering `A` (like `cs_amd(A,2)`). For QR factorization, use `q=colamd(A,[n m])`, where `[m n]=size(A)`, so that dense rows in `A` are not ignored (like `cs_amd(A,3)`). This finds a column ordering `q` so that `qr(A(:,q))` tends to be sparser than `qr(A)`.
`colperm`	Sorts columns by increasing degree.

ordering methods, continued:	
`dmperm`	`[p,q,r,s]=dmperm(A)` is the Dulmage–Mendelsohn decomposition of `A`, where `C=A(p,q)` is in block upper triangular form. The kth block is `C(r(k):r(k+1)-1,s(k):s(k+1)-1)`.
`randperm`	`p=randperm(n)` is a random permutation of `1:n`.
`symamd`	Approximate minimum degree, based on `colamd`; `p=symamd(A)`, is like `p=amd(A)`, just slower. `symamd` orders the matrix with pattern `tril(A)+tril(A)'`.
`symrcm`	`p=symrcm(A)` finds a reverse Cuthill–McKee ordering so that the entries of `C(p,p)` tend to be close to the diagonal, where `C=A+A'`.

linear algebra:	
`cholinc`	Incomplete Cholesky factorization. See Problem 4.13.
`condest`	`c=condest(A)` is a lower bound for `cond(full(A),1)` that is much less expensive to compute.
`eigs`	`e=eigs(A)` returns the six largest eigenvalues of a square matrix `A`, using a method that is less expensive than `e=eig(A)`. `[V,e]=eigs(A)` also returns the eigenvectors `V`. Many more options are available. Based on ARPACK.
`luinc`	Incomplete LU factorization. See Problem 6.13.
`normest`	`s=normest(A)` is an estimate of the 2-norm `norm(full(A))` that is much less expensive to compute.
`spaugment`	`S=spaugment(A,c)` is the augmented system `S=[c*I A;A' 0]`.
`sprank`	`s=sprank(A)` is the structural rank of `A` (the size of the maximum matching).
`svds`	`s=svds(A)` finds the six largest singular values of `A`. Many more options are available.

sparse factorization methods:	
`lu`	LU factorization. `[L,U,P]=lu(A)` returns `L*U=P*A`, using the left-looking GPLU algorithm (much like `cs_lu`) with no fill-reducing ordering. `[L2,U]=lu(A)` is the same, except `L2=P'*L` is returned. `[L,U,P,Q]=lu(A)` uses UMFPACK, where `Q` is a fill-reducing ordering and `P` is for both threshold partial pivoting and reducing fill-in. A second input to `lu` controls the threshold.
`qr`	Givens-rotation-based QR factorization. `R=qr(A)` computes just `R` and not `Q`; this uses much less memory than `[Q R]=qr(A)`. `R=qr(A,0)` returns `R` with just `min(size(A))` rows. `[x,R]=qr(A,b)` solves the least squares problem to minimize $\|Ax - b\|_2$. No column ordering is used to reduce `nnz(R)`; use `q=colamd(A,[n m])` and `qr(A(:,q))` to reduce fill-in.
`chol`	Cholesky factorization of `A`, using CHOLMOD. `R=chol(A)` returns an upper triangular matrix where `R'*R=A`. No fill-reducing ordering is used; use `p=amd(A)` or `p=symamd(A)` and `chol(A(p,p))` to reduce fill-in.

operators (highlights only; all MATLAB operators work for sparse matrices):	
\	Backslash, or `mldivide`. See Section 8.5.
/	Slash, or `mrdivide`. See Section 8.5.
'	`C=A'` is the transpose of `A` (complex conjugate transpose if `A` is complex).
+	`C=A+B` adds two matrices. `C` is sparse if `A` and `B` are sparse.
-	`C=A-B` subtracts two matrices. `C` is sparse if `A` and `B` are sparse.
*	`C=A*B` multiplies two matrices. `C` is sparse if `A` and `B` are sparse.
;	`C=[A;B]` concatenates `A` and `B` vertically; `A` and `B` must have the same number of columns. `C` is sparse if `A` or `B` are sparse.
,	`C=[A,B]` concatenates `A` and `B` horizontally; `A` and `B` must have the same number of rows. `C` is sparse if `A` or `B` are sparse.

iterative methods:

Iterative methods exploit sparsity when solving $Ax = b$ by not factorizing A. They typically rely on repeated matrix-vector multiplications. Methods in MATLAB include `bicg`, `bicgstab`, `cgs`, `gmres`, `lsqr`, `minres`, `pcg`, `qmr`, and `symmlq`.

tree and graph operations:	
`etree`	`parent=etree(A)` is the elimination tree of `triu(A)+triu(A)'`. `[parent post]=etree(A)` also returns the elimination tree postordering. `etree(A,'col')` finds the elimination tree of `A'*A`.
`etreeplot`	`etreeplot(A)` plots a picture of the elimination tree of `A+A'`.
`gplot`	`gplot(A,xy)` plots a picture of the undirected graph of `A+A'`, where the `n`-by-2 matrix `xy` gives the x-y coordinates of each node.
`symbfact`	Symbolic Cholesky factorization. `c=symbfact(A)` is a vector of column counts of the Cholesky factor `L=chol(A)'`, where only `triu(A)` is accessed. `symbfact(A,'col')` analyzes `A'*A` but does not form it. Additional outputs are `[c h parent post R]=symbfact(...)`, where `h` is the height of the elimination tree, `parent` is the tree, `post` is the postordering of the tree, and `R` is a binary matrix with the same pattern `chol(A)`.
`treelayout`	`[x,y,h]=treelayout(parent)` finds x-y coordinates for the nodes of a tree, and the height `h` of the tree, for use in `treeplot`.
`treeplot`	`treeplot(parent)` plots a picture of the elimination tree.

functions that partially work on sparse matrices or that have sparse substitutes:	
`cholupdate`	Rank-1 update/downdate of full Cholesky factorization. Use CHOLMOD or `cs_updown` for the sparse case.
`cond`	Use `condest` instead.
`eig`	Use `eigs` instead or `d=eig(A)` for sparse symmetric `A`.
`norm`	Works for 1-norm, ∞-norm, vector 2-norm, and Frobenius norm. Use `normest(A)` to estimate the 2-norm of a sparse matrix `A`.
`poly`	Only works if `A` is symmetric.
`svd`	Use `svds` instead.

functions and features that do not work on sparse matrices:

Functions and features of MATLAB that do not work at all for sparse matri-

ces include N-dimensional arrays for $N > 2$, different types (only double and complex double are available), `airy`, `bessel`, `besselj`, `bessely`, `besseli`, `besselk`, `besselh`, `betainc`, `bitand`, `bitcmp`, `bitor`, `bitxor`, `bitset`, `bitget`, `bitshift`, `complex`, `condeig`, `conv2`, `convn`, `deconv`, `erf`, `erfc`, `erfcx`, `fft`, `fft2`, `fftn`, `filter`, `filter2`, `funm`, `gamma`, `gammaln`, `gsvd`, `hess`, `histc`, `ifft`, `ifftn`, `linsolve`, `logm`, `lsqnonneg`, `null`, `ordeig`, `ordqz`, `ordschur`, `orth`, `qz`, `pinv`, `psi`, `rank`, `rcond`, `reallog`, `realpow`, `realsqrt`, `residue`, `rsf2csf`, `schur`, `sqrtm`, `ss2zp`, `subspace`, `surfnorm`, and `tzero`.

The following accept sparse inputs but produce full outputs: `ellipj`, `ellipke`, `erfcinv`, `erfinv`, `expint`, `gammainc`, `legendre`, `polyeig`, `polyval`, and `polyvalm`.

10.3 CSparse MATLAB interface

The MATLAB mexFunction interface for CSparse allows all of CSparse to be used within MATLAB. MATLAB already includes most of the functionality of CSparse (and much more) with the exception of the rank-1 sparse Cholesky update/downdate. Most CSparse functions are about as fast, or faster, than built-in functions in MATLAB 7.2 with notable exceptions of `chol` and `lu` because they rely on dense matrix kernels. See `CSparse/MATLAB/cs_install.m` to compile and install CSparse for use in MATLAB.

`cs_add`: sparse matrix addition

Usage: `C = cs_add(A,B,alpha,beta)`

MATLAB equivalent: `C = alpha*A+beta*B`

Adds two sparse matrices. `alpha` and `beta` default to 1 if not present.

See also `cs_multiply`, `cs_gaxpy`, `plus`, `minus`.

`cs_amd`: approximate minimum degree ordering

Usage: `p = cs_amd(A,order)`

MATLAB equivalent: `p = amd(A)`, `p = colamd(A)`, or `p = symamd(A)`

Approximate minimum degree ordering. The `order` parameter is optional. The default is `order=1`, which orders `A+A'`. `order=2` orders `S'*S`, where `S = A` except that dense rows are removed from `S`. `order=3` orders `A'*A`.

See also `amd`, `colamd`, `symamd`.

`cs_chol`: sparse Cholesky factorization

Usage: `[L,p] = cs_chol(A,drop)`

MATLAB equivalent: `p=amd(A) ; L=chol(A(p,p)')'`, or `L=chol(A')'`

Factorizes `A` or `A(p,p)` into its Cholesky factorization `L*L'`. `drop` is optional; if zero, numerically zero entries are not dropped from `L`. The default is to drop these entries (`drop = 1`). With one output, no fill-reducing ordering is used.

See also `cs_amd`, `cs_updown`, `chol`, `amd`, `symamd`.

`cs_cholsol`: solve $Ax = b$ using a sparse Cholesky factorization

Usage: `x = cs_cholsol(A,b,order)`

MATLAB equivalent: `x = A\b`

Solves $Ax = b$ using a sparse Cholesky factorization. `b` must be a full vector. If `order` is present, `cs_amd(A,order)` is used.

See also `cs_chol`, `cs_amd`, `cs_lusol`, `cs_qrsol`, `mldivide`.

`cs_counts`: column counts for sparse Cholesky factorization

Usage: `c = cs_counts(A,mode)`

MATLAB equivalent: `c = symbfact(A)` or `c = symbfact(A,'col')`

Returns a vector of the column counts of `L`, `c = sum(spones(chol(A)'))`, except that it is computed more efficiently. `c = cs_counts(A,'col')` computes the counts for the factorization of `A'*A`.

See also `symbfact`.

`cs_dmperm`: maximum matching or Dulmage–Mendelsohn decomposition

Usage: `[p,q,r,s,cc,rr] = cs_dmperm(A)` or `p = cs_dmperm(A)`

MATLAB equivalent: `[p,q,r,s] = dmperm(A)` or `p = dmperm(A)`

`p = cs_dmperm(A)` finds a maximum matching `p` such that `p(j) = i` if column `j` is matched to row `i` or `-1` if column `j` is unmatched. If `A` is square and full structural rank, `p` is a row permutation and `A(p,:)` has a zero-free diagonal. The structural rank of `A` is `sprank(A) = sum(p>0)`.

`[p,q,r,s,cc,rr] = cs_dmperm(A)` finds the Dulmage–Mendelsohn decomposition of `A`. `p` and `q` are permutation vectors. `cc` and `rr` are vectors of length 5. `C = A(p,q)` is split into a 4-by-4 set of coarse blocks

$$C = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ & & A_{23} & A_{24} \\ & & & A_{34} \\ & & & A_{44} \end{bmatrix},$$

where A_{12}, A_{23}, and A_{34} are square with zero-free diagonals. The columns of A_{11} are the unmatched columns, and the rows of A_{44} are the unmatched rows. Any of these blocks can be empty. In the coarse decomposition, the (i,j)th block is `C(rr(i):rr(i+1)-1,cc(j):cc(j+1)-1)`. In terms of a linear system, `[A11 A12]` is the undetermined part of the system (it is always rectangular and with more columns than rows or 0-by-0), A_{23} is the well-determined part of the system (it is always square), and `[A34 ; A44]` is the overdetermined part of the system (it is always rectangular with more rows than columns or 0-by-0). The structural rank of `A` is `rr(4)-1`. The A_{23} submatrix is further subdivided into block upper triangular form via the fine decomposition (the strongly connected components of A_{23}).

`C(r(i):r(i+1)-1,s(j):s(j+1)-1)` is the (i,j)th block of the fine decomposition. The $(1,1)$ block is the rectangular block `[A11 A12]`, unless this block is 0-by-0. The (b,b) block is the rectangular block `[A34 ; A44]`, unless this block is 0-by-0, where `b = length(r)-1`. All other diagonal blocks are submatrices of A_{23} and are square with a zero-free diagonal.

A second argument provides a seed for a randomized maximum matching.

See also `cs_dmspy`, `cs_dmsol`, `dmperm`, `sprank`, `cs_randperm`.

cs_dmsol: solve $Ax = b$ using a Dulmage–Mendelsohn decomposition

See Section 8.4 for a complete description of the `cs_dmsol` M-file.

cs_dmspy: plot a Dulmage–Mendelsohn decomposition

Usage: `[p,q,r,s,cc,rr] = cs_dmspy(A,res)`

MATLAB equivalent: `[p,q,r,s] = dmperm(A) ; spy(A(p,q))`

Plots a picture of the Dulmage–Mendelsohn decomposition of a matrix. It first computes `[p,q,r,s,cc,rr] = cs_dmperm(A)`, does `cspy(A(p,q))`, and then draws boxes around the coarse and fine decompositions. A second input argument (`cs_dmspy(A,res)`) changes the resolution of the image to `res`-by-`res` (default resolution is 256). If `res` is zero, `spy` is used instead of `cspy`.

See also `cs_dmperm`, `cs_dmsol`, `dmperm`, `sprank`, `spy`, `cspy`.

cs_droptol: remove small entries from a sparse matrix

Usage: `C = cs_droptol(A,tol)`

MATLAB equivalent: `A = A.*(abs(A)>tol)`

Removes small entries from a sparse matrix (those with magnitude $\leq$ `tol`).

cs_esep: find an edge separator

See Section 7.6 for a complete description of the `cs_esep` M-file.

cs_etree: elimination tree of `A` or `A'*A`

Usage: `[parent,post] = cs_etree(A,mode)`

MATLAB equivalent: `[parent,post] = etree(A)`

`parent = cs_etree(A)` returns the elimination tree of `A`. With a second input, `parent = cs_etree(A,'col')` returns the elimination tree of `A'*A`. For the symmetric case (`cs_etree(A)`), only `triu(A)` is used.

`[parent,post] = cs_etree(...)` also returns a postorder of the tree.

See also `etree`, `treeplot`.

cs_gaxpy: sparse matrix times vector

Usage: `z = cs_gaxpy(A,x,y)`

MATLAB equivalent: `z = A*x+y`

`x` and `y` must be full vectors.

See also `plus`, `mtimes`.

cs_lsolve: solve a sparse lower triangular system, $Lx = b$

Usage: `x = cs_lsolve(L,b)`

MATLAB equivalent: `x = L\b`

Solves a sparse lower triangular system. `L` must be lower triangular with a zero-free diagonal. `b` can be a full or sparse vector.

See also `cs_ltsolve`, `cs_usolve`, `cs_utsolve`, `mldivide`.

cs_ltsolve: solve a sparse upper triangular system, $L^T x = b$

Usage: `x = cs_ltsolve(L,b)`
MATLAB equivalent: `x = L'\b`
Solves a sparse upper triangular system. `L` must be lower triangular with a zero-free diagonal. `b` must be a full vector.
See also `cs_lsolve`, `cs_usolve`, `cs_utsolve`, `mldivide`.

cs_lu: sparse LU factorization

Usage: `[L,U,p,q] = cs_lu(A,tol)`
MATLAB equivalent: `[L,U,P,Q] = lu(A,tol)`
`[L,U,p] = cs_lu(A)` factorizes `A(p,:)` into `L*U`.
`[L,U,p] = cs_lu(A,tol)` factorizes `A(p,:)` into `L*U`. Entries on the diagonal are given preference in partial pivoting.
`[L,U,p,q] = cs_lu(A)` factorizes `A(p,q)` into `L*U`, using a fill-reducing ordering `q = cs_amd(A,2)`. Normal partial pivoting is used.
`[L,U,p,q] = cs_lu(A,tol)` factorizes `A(p,q)` into `L*U`, using a fill-reducing ordering `q = cs_amd(A)`. Entries on the diagonal are given preference in partial pivoting. With a pivot tolerance `tol`, the entries in `L` have magnitude `1/tol` or less. `tol = 1` is normal partial pivoting (but with the `q = cs_amd(A)` ordering). `tol = 0` ensures `p = q`. `0<tol<1` is relaxed partial pivoting; the diagonal is selected if it is at least `tol*max(abs(A(:,k)))`.
See also `cs_amd`, `lu`, `umfpack`, `amd`, `colamd`.

cs_lusol: solve $Ax = b$ using LU factorization

Usage: `x = cs_lusol(A,b,order,tol)`
MATLAB equivalent: `x = A\b`
`x = cs_lusol(A,b)` computes `x=A\b`, where A is sparse and square, and `b` is a full vector. The ordering `cs_amd(A,2)` is used. `x = cs_lusol(A,b,1)` also computes `x=A\b` but uses the `cs_amd(A)` ordering with diagonal preference (default `tol=0.001`). If `order` is present, `cs_amd(A,order)` is used.
See also `cs_lu`, `cs_amd`, `cs_cholsol`, `cs_qrsol`, `mldivide`.

cs_make: compiles CSparse for use in MATLAB

Usage: `cs_make`
See also `mex`. Type `help cs_make` in MATLAB for more information, including instructions on how to add new mexFunctions to CSparse.

cs_multiply: sparse matrix multiply

Usage: `C = cs_multiply(A,B)`
MATLAB equivalent: `C = A*B`
See also `cs_gaxpy`, `cs_add`, `mtimes`.

cs_nd: generalized nested dissection

See Section 7.6 for a complete description of the `cs_nd` M-file.

cs_nsep: find a node separator

See Section 7.6 for a complete description of the cs_nsep M-file.

cs_permute: permute a sparse matrix

Usage: `C = cs_permute(A,p,q)`
MATLAB `C = A(p,q)`
See also `cs_symperm`, `subsref`.

cs_print: print a sparse matrix

Usage: `cs_print(A,brief)`
MATLAB equivalent: `A` with no semicolon
`cs_print(A)` prints a sparse matrix. `cs_print(A,1)` prints just a few entries.

cs_qleft: apply Householder vectors on the left

See Section 5.3 for a description of the `cs_qleft` M-file.

cs_qr: sparse QR factorization

Usage: `[V,beta,p,R,q] = cs_qr(A)`
MATLAB equivalent: `q=colamd(A,[n m]) ; [Q,R] = qr(A(:,q))`
`[V,beta,p,R] = cs_qr(A)` computes the QR factorization of `A(p,:)`.
`[V,beta,p,R,q] = cs_qr(A)` computes the QR factorization of `A(p,q)`. The fill-reducing ordering `q` is found via `q = cs_amd(A,3)`.
`A` must be `m`-by-`n` with $m \geq n$. If `A` is structurally rank deficient, additional empty rows may have been added to `V` and `R`. The orthogonal factor `Q` can be obtained via `Q = cs_qright(V,Beta,p,speye(size(V,1)))`.
See also `cs_amd`, `cs_qr`, `cs_qright`, `cs_dmperm`, `qr`, `colamd`.

cs_qright: apply Householder vectors on the right

See Section 5.3 for a description of the `cs_qright` M-file.

cs_qrsol: solve a sparse least squares problem

Usage: `x = cs_qrsol(A,b,order)`
MATLAB `x = A\b`
`x = cs_qrsol(A,b)` solves the overdetermined least squares problem to find `x` that minimizes `norm(A*x-b)`, where `b` is a full vector. `A` is `m`-by-`n` with $m \geq n$. If `order` is present, `cs_amd(A,order)` is used. If `A` has fewer rows than columns, an underdetermined problem is solved.
See also `cs_qr`, `cs_amd`, `cs_lusol`, `cs_cholsol`, `mldivide`.

cs_randperm: random permutation

Usage: `p = cs_randperm(n,seed)`
MATLAB equivalent: `p = randperm(n)`

cs_scc: strongly connected components of a square sparse matrix

Usage: `[p,r] = cs_scc(A)`

MATLAB equivalent: `[p,q,r,s] = dmperm(A+speye(size(A,1)))`

`[p,r] = cs_scc(A)` finds a permutation `p` so that `A(p,p)` is permuted into block upper triangular form. The diagonal of `A` is ignored.

See also `cs_dmperm`, `dmperm`.

cs_sep: convert an edge separator into a node separator

See Section 7.6 for a complete description of the `cs_sep` M-file.

cs_sparse: convert a triplet form into a MATLAB sparse matrix

Usage: `A = cs_sparse(i,j,x)`

MATLAB equivalent: `A = sparse(i,j,x)`

`A = cs_sparse(i,j,x)` is identical to `A = sparse(i,j,x)`, except that `x` must be real, and the lengths of `i`, `j`, and `x` must be the same. The MATLAB `sparse` function has many more features than `cs_sparse`.

See also `sparse`, `spconvert`.

cs_sqr: symbolic QR or LU ordering and analysis

Usage: `[vnz,rnz,parent,c,leftmost,p,q] = cs_sqr(A)`

MATLAB equivalent: `[c,parent] = symbfact (A, 'col')`

`[vnz,rnz,parent,c,leftmost,p] = cs_sqr(A)` computes the symbolic QR factorization of `A(p,:)`. `[vnz,rnz,parent,c,leftmost,p,q] = cs_sqr(A)` computes the symbolic QR factorization of `A(p,q)`. The fill-reducing ordering `q` is found via `q=cs_amd(A,3)`. `vnz` is the number of entries in the matrix of Householder vectors, `V`. `rnz` is the number of entries in `R`. `parent` is the elimination tree. `c(i)` is the number of entries in `R(i,:)`. `leftmost(i)` is `min(find(A(i,q)))`. `p` is the row permutation used to ensure `R` has a symbolically zero-free diagonal. `q` is the fill-reducing ordering if requested.

See also `cs_amd`, `cs_qr`.

cs_symperm: symmetric permutation of a symmetric matrix

Usage: `C = cs_symperm(A,p)`

MATLAB equivalent: `C = triu(A)+triu(A,1)' ; C = triu(C(p,p))`

`C = cs_symperm(A,p)` computes `C = A(p,p)` but accesses only the upper triangular part of `A` and returns `C` upper triangular (`A` and `C` are symmetric with just their upper triangular parts stored). `A` must be square.

See also `cs_permute`, `subsref`, `triu`.

cs_transpose: transpose a sparse matrix

Usage: `C = cs_transpose(A)`

MATLAB equivalent: `C = A'`

See also `transpose`, `ctranspose`.

cs_updown: rank-1 sparse Cholesky update/downdate

Usage: `L = cs_updown(L,c,parent,sigma)`

MATLAB equivalent: `L = sparse(choldupate(full(L),c,sigma))`

`L=cs_updown(L,c,parent)` computes the rank-1 update `L=chol(L*L'+c*c')'`, where `parent` is the elimination tree of `L`. `c` must be a sparse column vector, and `find(c)` must be a subset of `find(L(:,k))`, where `k=min(find(c))`.

`L=cs_updown(L,c,parent,'-')` is the downdate `L=chol(L*L'-c*c')'`.

`L=cs_updown(L,c,parent,'+')` is the update `L=chol(L*L'+c*c')'`.

Updating/downdating is much faster than refactorization with `cs_chol` or `chol`. `L` must not have an entries dropped due to numerical cancellation.

See also `cs_etree`, `cs_chol`, `etree`, `cholupdate`, `chol`.

`cs_usolve`: solve a sparse upper triangular system $Ux = b$

Usage: `x = cs_usolve(U,b)`

MATLAB equivalent: `x = U\b`

Solves a sparse upper triangular system. `U` must be upper triangular with a zero-free diagonal. `b` can be a full or sparse vector.

See also `cs_lsolve`, `cs_ltsolve`, `cs_utsolve`, `mldivide`.

`cs_utsolve`: solve a sparse lower triangular system $U'x = b$

Usage: `x = cs_utsolve(U,b)`

MATLAB equivalent: `x = U'\b`

Solves a sparse lower triangular system. `U` must be upper triangular with a zero-free diagonal. `b` must be a full vector.

See also `cs_lsolve`, `cs_ltsolve`, `cs_usolve`, `mldivide`.

`cspy`: plot a sparse matrix in color

Usage: `cspy(A,res)`

`cspy(A)` plots a sparse matrix, in color, with a default resolution of 256-by-256. `cspy(A,res)` changes the resolution to `res`. Entries with tiny absolute value are light tan. Entries with large magnitude are black. Entries in the midrange (the median of the $\log_{10}$ of the nonzero values, $\pm$ one standard deviation) range from light green to deep blue. With no inputs, the color legend of `cspy` is plotted. `[s,M,H] = cspy(A)` returns the scale factor `s`, the image `M`, and colormap `H`.

See also `cs_dmspy`, `spy`.

10.4 Examples

The `cs_demo1`, `cs_demo2`, and `cs_demo3` M-files in the `MATLAB/Demo` directory are MATLAB equivalents of the C demo programs of the same name. They access CSparse via a set of mexFunctions. These demos also plot their results with `cspy`.

A mexFunction interfaces a C or Fortran program to MATLAB. Once compiled, it acts just like an M-file. Its name is always "`mexFunction`" and it always has the same parameters. C- and Fortran-callable MATLAB functions with the prefix `mx` provide access to MATLAB data structures, while functions with `mex` prefixes operate in the MATLAB environment. Below is a sample mexFunction in CSparse that interfaces the `cs_chol` function to MATLAB (the file `cs_chol_mex.c`). It calls `cs_mex_get_sparse` to convert a MATLAB matrix `A` into a CSparse matrix `A`. The

matrix is analyzed and factorized with `cs_schol` and `cs_chol`. The `drop` parameter determines whether or not numerically zero entries should be dropped from the matrix (they must be kept for `cs_updown` to work properly). `cs_mex_put_sparse` returns `L` to the MATLAB caller. If two output parameters have been provided, then the permutation `p` is computed and returned to MATLAB via `cs_mex_put_int`.

```
#include "cs_mex.h"
/* cs_chol: sparse Cholesky factorization */
void mexFunction (int nargout, mxArray *pargout [ ], int nargin,
    const mxArray *pargin [ ])
{
    cs Amatrix, *A ;
    int order, n, drop, *p ;
    css *S ;
    csn *N ;
    if (nargout > 2 || nargin < 1 || nargin > 2)
        mexErrMsgTxt ("Usage: [L,p] = cs_chol(A,drop)") ;
    A = cs_mex_get_sparse (&Amatrix, 1, 1, pargin [0]) ;        /* get A */
    n = A->n ;
    order = (nargout > 1) ? 1 : 0 ;                 /* determine ordering */
    S = cs_schol (order, A) ;                       /* symbolic Cholesky */
    N = cs_chol (A, S) ;                            /* numeric Cholesky */
    if (!N) mexErrMsgTxt ("cs_chol failed: not positive definite\n") ;
    drop = (nargin == 1) ? 1 : mxGetScalar (pargin [1]) ;
    if (drop) cs_dropzeros (N->L) ;                 /* drop zeros if requested*/
    pargout [0] = cs_mex_put_sparse (&(N->L)) ;     /* return L */
    if (nargout > 1)
    {
        p = cs_pinv (S->pinv, n) ;                  /* p=pinv' */
        pargout [1] = cs_mex_put_int (p, n, 1, 1) ; /* return p */
    }
    cs_nfree (N) ;
    cs_sfree (S) ;
}
```

`nargin` and `nargout` are the number of input and output parameters. `pargout` and `pargin` are arrays of pointers to the input and output parameters. The `cs_chol` mexFunction makes use of a set of utility routines called `cs_mex_*` shared by all CSparse mexFunctions, in the `cs_mex.c` file, listed below.

```
#include "cs_mex.h"
/* check MATLAB input argument */
void cs_mex_check (int nel, int m, int n, int square, int sparse, int values,
    const mxArray *A)
{
    int nnel, mm = mxGetM (A), nn = mxGetN (A) ;
    if (values)
    {
        if (mxIsComplex (A)) mexErrMsgTxt ("matrix must be real") ;
        if (!mxIsDouble (A)) mexErrMsgTxt ("matrix must be double") ;
    }
    if (sparse && !mxIsSparse (A)) mexErrMsgTxt ("matrix must be sparse") ;
    if (!sparse && mxIsSparse (A)) mexErrMsgTxt ("matrix must be full") ;
    if (nel)
    {
        /* check number of elements */
```

```
        nnel = mxGetNumberOfElements (A) ;
        if (m >= 0 && n >= 0 && m*n != nnel) mexErrMsgTxt ("wrong length") ;
    }
    else
    {
        /* check row and/or column dimensions */
        if (m >= 0 && m != mm) mexErrMsgTxt ("wrong dimension") ;
        if (n >= 0 && n != nn) mexErrMsgTxt ("wrong dimension") ;
    }
    if (square && mm != nn) mexErrMsgTxt ("matrix must be square") ;
}

/* get a MATLAB sparse matrix and convert to cs */
cs *cs_mex_get_sparse (cs *A, int square, int values, const mxArray *Amatlab)
{
    cs_mex_check (0, -1, -1, square, 1, values, Amatlab) ;
    A->m = mxGetM (Amatlab) ;
    A->n = mxGetN (Amatlab) ;
    A->p = mxGetJc (Amatlab) ;
    A->i = mxGetIr (Amatlab) ;
    A->x = values ? mxGetPr (Amatlab) : NULL ;
    A->nzmax = mxGetNzmax (Amatlab) ;
    A->nz = -1 ;    /* denotes a compressed-col matrix, instead of triplet */
    return (A) ;
}

/* return a sparse matrix to MATLAB */
mxArray *cs_mex_put_sparse (cs **Ahandle)
{
    cs *A ;
    mxArray *Amatlab ;
    A = *Ahandle ;
    Amatlab = mxCreateSparse (0, 0, 0, mxREAL) ;
    mxSetM (Amatlab, A->m) ;
    mxSetN (Amatlab, A->n) ;
    mxSetNzmax (Amatlab, A->nzmax) ;
    cs_free (mxGetJc (Amatlab)) ;
    cs_free (mxGetIr (Amatlab)) ;
    cs_free (mxGetPr (Amatlab)) ;
    mxSetJc (Amatlab, A->p) ;           /* assign A->p pointer to MATLAB A */
    mxSetIr (Amatlab, A->i) ;
    mxSetPr (Amatlab, A->x) ;
    mexMakeMemoryPersistent (A->p) ;    /* ensure MATLAB does not free A->p */
    mexMakeMemoryPersistent (A->i) ;
    mexMakeMemoryPersistent (A->x) ;
    cs_free (A) ;                       /* frees A struct only, not A->p, etc */
    *Ahandle = NULL ;
    return (Amatlab) ;
}

/* get a MATLAB dense column vector */
double *cs_mex_get_double (int n, const mxArray *X)
{
    cs_mex_check (0, n, 1, 0, 0, 1, X) ;
    return (mxGetPr (X)) ;
}
```

```
/* return a double vector to MATLAB */
double *cs_mex_put_double (int n, const double *b, mxArray **X)
{
    double *x ;
    int k ;
    *X = mxCreateDoubleMatrix (n, 1, mxREAL) ;      /* create x */
    x = mxGetPr (*X) ;
    for (k = 0 ; k < n ; k++) x [k] = b [k] ;       /* copy x = b */
    return (x) ;
}

/* get a MATLAB flint array and convert to int */
int *cs_mex_get_int (int n, const mxArray *Imatlab, int *imax, int lo)
{
    double *p ;
    int i, k, *I = cs_malloc (n, sizeof (int)) ;
    cs_mex_check (1, n, 1, 0, 0, 1, Imatlab) ;
    p = mxGetPr (Imatlab) ;
    *imax = 0 ;
    for (k = 0 ; k < n ; k++)
    {
        i = p [k] ;
        I [k] = i - 1 ;
        if (i < lo) mexErrMsgTxt ("index out of bounds") ;
        *imax = CS_MAX (*imax, i) ;
    }
    return (I) ;
}

/* return an int array to MATLAB as a flint row vector */
mxArray *cs_mex_put_int (int *p, int n, int offset, int do_free)
{
    mxArray *X = mxCreateDoubleMatrix (1, n, mxREAL) ;
    double *x = mxGetPr (X) ;
    int k ;
    for (k = 0 ; k < n ; k++) x [k] = (p ? p [k] : k) + offset ;
    if (do_free) cs_free (p) ;
    return (X) ;
}
```

The `cs_chol.m` M-file provides documentation. Typing `help cs_chol` in the MATLAB Command Window prints the comments in the first part of `cs_chol.m`.

```
function [L,p] = cs_chol (A,drop)
%CS_CHOL sparse Cholesky factorization.
%   L = cs_chol(A) is the same as L = chol(A)', using triu(A).
%   [L,p] = cs_chol(A) first orders A with p=cs_amd(A), so that L*L' = A(p,p).
%   A second optional input argument controls whether or not numerically zero
%   entries are removed from L.  cs_chol(A) and cs_chol(A,1) drop them;
%   cs_chol(A,0) keeps them.  They must be kept for cs_updown to work properly.
%
%   See also CS_AMD, CS_UPDOWN, CHOL, AMD, SYMAMD.

%   Copyright 2006, Timothy A. Davis.
%   http://www.cise.ufl.edu/research/sparse
error ('cs_chol mexFunction not found') ;
```

The `mex` and `mx` functions used by CSparse are listed below.

`mex.h`	required mexFunction include file
`mexErrMsgTxt`	print error message and abort the mexFunction
`mexFunction`	required name of a mexFunction
`mexMakeMemoryPersistent`	ensures memory persists after mexFunction completes
`mxArray`	a MATLAB array
`mxCalloc`	MATLAB equivalent of the C `calloc` function
`mxCreateDoubleMatrix`	create a MATLAB full matrix
`mxCreateDoubleScalar`	create a MATLAB scalar (a 1-by-1 matrix)
`mxCreateSparse`	create a MATLAB sparse matrix
`mxFree`	MATLAB equivalent of the C `free` function
`mxGetJc`	get the column pointers of a MATLAB sparse matrix
`mxGetIr`	get the row indices of a MATLAB sparse matrix
`mxGetPr`	get the numerical values of a MATLAB sparse matrix
`mxGetM`	get number of rows of a MATLAB matrix
`mxGetN`	get number of columns of a MATLAB matrix
`mxGetNumberOfElements`	get number of entries of a MATLAB matrix
`mxGetNzmax`	get the maximum number of entries of a MATLAB matrix
`mxGetScalar`	get the value of a MATLAB scalar
`mxGetString`	get a MATLAB string
`mxIsChar`	true if a MATLAB matrix is a string
`mxIsComplex`	true if a MATLAB matrix is complex
`mxIsDouble`	true if a MATLAB matrix is double
`mxIsSparse`	true if a MATLAB matrix is sparse
`mxMalloc`	MATLAB equivalent of the C `malloc` function
`mxRealloc`	MATLAB equivalent of the C `realloc` function
`mxSetJc`	set the column pointers of a MATLAB sparse matrix
`mxSetIr`	set the row indices of a MATLAB sparse matrix
`mxSetPr`	set the numerical values of a MATLAB sparse matrix
`mxSetM`	set number of rows of a MATLAB matrix
`mxSetN`	set number of columns of a MATLAB matrix
`mxSetNzmax`	set the maximum number of entries of a MATLAB matrix

10.5 Further reading

Gilbert, Moler, and Schreiber introduced sparse matrices into MATLAB [105], including the first implementation of the sparse backslash. Additional sparse matrix functions of Amestoy, Davis, and Duff (AMD [1, 2]), Davis and Duff (UMFPACK [27, 28, 31, 32]), Davis, Gilbert, Larimore, and Ng (COLAMD [33, 34]), Davis, Hager, Chen, and Rajamanickam (CHOLMOD [30]), and Lehoucq, Sorensen, and Yang (ARPACK [144, 145, 188]) have been included. `condest` is based on Higham and Tisseur's [136] method, a generalization of Hager's 1-norm estimator [123]. Penny Anderson, Bobby Cheng, and Pat Quillen have written many of the sparse matrix methods in MATLAB. For more information on MATLAB, see Higham and Higham [133] or Davis and Sigmon [38]. Duff [47] discusses how random matrices (`sprand`, `sprandn`, and `sprandsym`) can give misleading results when factorized.

Exercises

10.1. Compare the performance (speed and accuracy) of the CSparse mexFunctions and MATLAB, using a range of large sparse matrices from real applications.

Appendix A

Basics of the C programming language

The following is a brief overview of the features of C used in CSparse, including a description of how each feature translates into MATLAB. See [141] for a full yet concise description of the C programming language.

Variables

Six of C's basic variable types are used in CSparse:

`int`	an integer
`unsigned int`	an integer that is always positive
`double`	a double-precision floating-point value
`size_t`	an integer large enough to hold a pointer
`char`	a character
`void`	an object of no specific type used for pointers

In MATLAB, variables do not need to be declared (except with the rarely used `global` statement). They must be declared in C. For example, the following C statements declare integer scalars `i` and `j` and a double-precision scalar `x`. Declarations can include initializations as well. C also includes a `complex` type.

```
int i ;
double x ;
int j = 0 ;
```

In C, an array is represented by a *pointer* to an address in memory containing the first element of the array. The following is also an example of a `/* comment */` in C.

```
int *s ;            /* declares a pointer to an int */
double *x ;         /* declares a pointer to a double */
void *p ;           /* declares a pointer to "void" */
```

Statements and Operators

Logical operators are slightly different in C and MATLAB. In C, ! is the logical negation operator, which is the same as the tilde operator (~) in MATLAB. The not-equals operator is != in C and ~= in MATLAB. In both languages, logical *or* (||) and logical *and* (&&) are the same. The caret (^) operator in C is the exclusive-or (`bitxor` in MATLAB). Logical statements are evaluated left to right; evaluation terminates as soon as the result is known.

A pointer whose value is `NULL` is special (`NULL` is zero). It points to nothing. In both C and MATLAB, true is nonzero and false is zero. Thus if `x` is a pointer, the expression `!x` is true if `x` is a `NULL` pointer and false otherwise.

In C, single statements must always be terminated with a semicolon. A compound statement can be used in C wherever a single statement can be used; it starts with { and ends with }. The latter is analogous to `end` in MATLAB. Assignments can be used as expressions in C but not in MATLAB. For example, `x=y=0 ;` in C sets both `x` and `y` to zero. The comma operator can be used to separate a pair of expressions. Both expressions are evaluated, and the result is the value of the right operand. It is used only in `for` loops in CSparse. C does not have matrix operators. The ternary operator `?:` is unique to C. In C, the statement

```
x = (y > 0) ? z : w ;
```

is equivalent to the MATLAB statements

```
if (y > 0)
    x = z ;
else
    x = w ;
end
```

If `p` is a pointer, `*p` denotes the value stored at that memory location; that is, the unary * operator *dereferences* a C pointer to obtain the value stored at the memory location referred to by the pointer. The expression `&x` is a pointer to the variable `x`. Arithmetic can be performed on C pointers. If `x` is a pointer to an array of size 10, `x[i]` is the `i`th entry in the array, where `i` can be in the range `0` to `9`. In MATLAB, the array is `x(1)` through `x(10)`. Pointer arithmetic is always done in units of the size of the type of variable pointed to by the pointer. Thus, `*(x+i)` is identical to `x[i]`. CSparse uses very little pointer arithmetic of the form `x+i`, except when laying out subarrays from a larger workspace. It uses the form `x[i]` extensively.

The ++ and -- operators are unique to C. As stand-alone statements, `x++` and `++x` are both the same as `x = x + 1`. When used in expression, `x++` increments `x` after its value is used in the expression, and `++x` increments `x` before it is used. Thus,

```
y = x++ ;
```

is the same (in both C and MATLAB) as

```
y = x ;
x = x + 1 ;
```

and

```
y = ++x ;
```

is the same (in both C and MATLAB) as

```
x = x + 1 ;
y = x ;
```

Likewise, the C statement

```
x [k++] = i ;
```

is the same as the C statements

```
x [k] = i ;
k = k + 1 ;
```

C has a suite of assignment operators that modify the left-hand side. The following table shows five of these and their equivalents using the regular assignment in C.

`x += 2`	`x = x + 2`
`x -= 2`	`x = x - 2`
`x /= 2`	`x = x / 2`
`x *= 2`	`x = x * 2`
`x %= 2`	`x = x % 2`

The `%` operator in C is the `rem` function in MATLAB, except that the meaning of `a % b` when either `a` or `b` are negative is machine dependent (it is used in CSparse only for positive numbers).

Variables can be *typecast* into values of a different type. For example, to convert an `int` to a `double`,

```
x = (double) i ;
```

This conversion is done automatically by the assignment `x=i` and when variables of one type are passed to a function expecting another type, so it is rarely needed.

Control structures

The `while` loop is almost the same in C and MATLAB. These two code fragments are the same, the first in C and the second in MATLAB:

```
while (x < 10)
{
    x = x + i ;
}
```

```
while (x < 10)
    x = x + i ;
end
```

Both C and MATLAB have a `for` loop, but they differ in how they work. These two code fragments are the same in C and MATLAB, respectively.

```
for (i = 0 ; i < n ; i++)
{
    x = x + i ;
}
```

```
for i = 0:n-1
    x = x + i ;
end
```

The C `for` loop has four components. In general,

```
for ( initialization ; condition ; post )
{
    body
}
```

is identical to

```
initialization
while ( condition )
{
    body
    post
}
```

Thus, these two loops are identical in C:

```
for (i = 0 ; i < n ; i++)
{
    x = x + i ;
}
```

```
i = 0 ;
while (i < n)
{
    x = x + i ;
    i = i + 1 ;
}
```

Any of the four components of a `for` loop can be empty. Thus `for (;;) ;` is an infinite loop. The `continue` and `break` statements are identical in C and MATLAB; the former causes the next iteration of the nearest enclosing `for` or `while` loop to begin, the latter terminates the nearest enclosing loop immediately.

Functions

Parameters to C and MATLAB functions are both passed by value; a C function and a MATLAB M-file function cannot modify their input parameters. However, a pointer can be passed by value to a C function, and the *contents* of what that pointer points to can be modified. Most CSparse functions do not modify their inputs. The `const` keyword when used in a function header or prototype declares that the function does not modify the contents of an array (or, equivalently, what a pointer argument points to).

A C function can return only a single value to its caller. This value can be a pointer. For example, the `cs_malloc(n,b)` function returns a pointer to a newly allocated (but uninitialized) memory space of size large enough to hold an array of `n` items each of `b` bytes. The `sizeof` operator is applied to a C type and returns the number of bytes in an object of that type (normally, `sizeof(int)` is four, and `sizeof(double)` is eight). The C `return` statement is like the MATLAB `return`, except that it also defines the value returned to the caller. The following C function and MATLAB function are the same, except MATLAB always uses double-precision floating-point values to represent its integers (called a *flint* in MATLAB).

```
int *myfunc (int n)
{
    int *p, i ;
    p = cs_malloc (n, sizeof (int)) ;
    for (i = 0 ; i < n ; i++) p [i] = i ;
    return (p) ;
}
```

In MATLAB, the same function is

```
function p = myfunc (n)
p = 0:n-1 ;
```

These functions are called in the same way in C and MATLAB: `p=myfunc(n);`. C functions that are private to a specific source code file are declared `static`. They can be called only by other functions in that same file, just like a nested or private function in MATLAB. A *prototype* is a statement declaring the name, parameters (and their type), and return value of a function. These are normally placed in an *include file* (described below), so that they can be incorporated into any code that calls the function. Prototypes for functions returning an `int` are not strictly required. However, if the declaration and use of the function are different and no prototype is present, the results are unpredictable. Prototypes should always be used in well-written C code. MATLAB has no prototypes but checks each usage of a function as it is called at run time. The prototype for `myfunc` is

```
int *myfunc (int n) ;
```

Both C and MATLAB can work with pointers to functions (called function handles in MATLAB). Consider the `cs_fkeep` function with prototype:

```
int cs_fkeep (cs *A, int (*fkeep) (int, int, double, void *), void *other) ;
```

Its second argument is a pointer to a function with four parameters (two `int`'s, a `double`, and a pointer to `void`). The function `cs_fkeep` calls the `fkeep` function for each entry in the matrix. An example of the use of `cs_fkeep` is in the `cs_droptol` function. Note that `cs_droptol` passes a pointer to `cs_tol` to `cs_fkeep`.

```
static int cs_tol (int i, int j, double aij, void *tol)
{
    return (fabs (aij) > *((double *) tol)) ;
}
int cs_droptol (cs *A, double tol)
{
    return (cs_fkeep (A, &cs_tol, &tol)) ;    /* keep all large entries */
}
```

Data structures

Both C and MATLAB can create a compound object called a `struct`. In C, these must be declared statically. They can be dynamically created in MATLAB. The following code fragments are identical. In C, the `mystuff` type is first defined as a structure containing a scalar integer and a pointer to a double. It is then used in a declaration statement to define `f` of type `mystuff`.

```
typedef struct mystuffstruct
{
    int i ;
    double *x ;
} mystuff ;
mystuff f ;
f.i = 3 ;
f.x = cs_calloc (4, sizeof (double)) ;
```

In MATLAB, the `struct` function may be used, or the components of a structure may be defined one at a time:

```
g = struct ('i', 3, 'x', zeros (4,1)) ;
f.i = 3 ;
f.x = zeros (4,1) ;
```

In the C example above, `f` is declared as an object of type `mystuff`. A pointer to an object of this type can also be defined. Accessing the contents of a `struct` with a pointer is different, using the `->` operator. The fragment below is identical to the C code above, except that it uses a pointer instead. The `->` operator is used exclusively in CSparse, instead of the dot (`.`). In C, if `p` is a pointer to a `struct` containing the member `i`, the expressions `p->i` and `(*p).i` are identical.

```
typedef struct mystuffstruct
{
    int i ;
    double *x ;
} mystuff ;
mystuff *p ;
p = cs_malloc (1, sizeof (mystuff)) ;
p->i = 3 ;
p->x = cs_calloc (4, sizeof (double)) ;
```

Examples

Consider the following statement from `cs_pvec`:

```
for (k = 0 ; k < n ; k++) x [k] = b [p ? p [k] : k] ;
```

A more leisurely way of expressing this is

```
for (k = 0 ; k < n ; k++)
{
    if (p != NULL)
    {
```

```
        x [k] = b [p [k]] ;
    }
    else
    {
        x [k] = b [k] ;
    }
}
```

Both examples shown above compile into code that is equally fast. The former is just more concise. The statements from `cs_transpose`

```
Ci [q = w [Ai [p]]++] = j ; /* place A(i,j) as entry C(j,i) */
if (Cx) Cx [q] = Ax [p] ;
```

are a more concise form of the following:

```
i = Ai [p] ;
q = w [i] ;
w [i] = w [i] + 1 ;
Ci [q] = j ;
if (Cx != NULL)
{
    Cx [q] = Ax [p] ;
}
```

The left-to-right evaluation of logical statements in C is exploited by this example from `cs_multiply`.

```
if (nz + m > C->nzmax && !cs_sprealloc (C, 2*(C->nzmax)+m))
{
    return (cs_done (C, w, x, 0)) ;             /* out of memory */
}
```

If `nz+m` is less than the space available in `C` (`C->nzmax`), the first operand of `&&` is false, and the second expression is not evaluated. Thus, `cs_sprealloc` is called only if `nz+m > C->nzmax`. If this first expression is true but `cs_sprealloc` returns false, then it failed to allocate sufficient memory. In this case, `cs_done` frees all workspace and the result `C` and returns `NULL` to signify that it ran out of memory.

C library functions

The following C library functions are used in CSparse:

`fabs(x)`	absolute value of `x`
`sqrt(x)`	square root of `x`
`malloc(n)`	allocates a block of `n` bytes of memory
`calloc(n,b)`	allocates a block of `n` items each of size `b` and sets it to zero
`free(p)`	frees a block of memory
`realloc(p,n)`	changes the size of a block of memory to `n` bytes
`printf`	just like `fprintf` in MATLAB
`fscanf`	just like `fscanf` in MATLAB
`clock`	returns CPU time used (used only in the CSparse demos)
`qsort`	sorts an array
`rand`	random number generator
`srand`	set random number seed

C preprocessor

The C preprocessor is a text-only preprocessing step, applied to a program before it is compiled. Preprocessor statements start with #, usually in the first column. The statements used in CSparse are listed below.

`#include`	includes a file
`#define`	defines a macro or token
`#ifdef`	true if the token is defined
`#ifndef`	true if the token is not defined
`#else`	the "else" part to an `ifdef` or `ifndef`
`#endif`	the "endif" part to an `ifdef` or `ifndef`

The `#include` statement has one of the forms

```
#include <file.h>
#include "file.h"
```

The only difference between the two is where the C compiler looks for the file called `file.h`. The first one looks in a sequence of predetermined locations (dependent on your compiler and operating system). The second (`"file.h"`) looks first in the same place as the current source file and, failing that, looks in the same place as the `<file.h>` form. This file is copied into the source code that has the `#include` statement before it is compiled.

The `#define` statement can define a token or a macro. A token is a single word that has no parameters. For example, the word `NULL` is defined as `0`, or `((void *) 0)`, in the `<stdio.h>` or `<stdlib.h>` file. The `#define` statement is used in CSparse to define the memory management routines CSparse should use. If CSparse is being compiled in a MATLAB mexFunction, the token `MATLAB_MEX_FILE` is defined, and the MATLAB memory management routines are used (`mxMalloc`, `mxCalloc`, `mxFree`, and `mxRealloc`) instead of their standard C counterparts (`malloc`, `calloc`, `free`, and `realloc`). Consider three macros that are defined in the `cs.h` file:

```
#define CS_MAX(a,b) (((a) > (b)) ? (a) : (b))
#define CS_MIN(a,b) (((a) < (b)) ? (a) : (b))
#define CS_FLIP(i) (-(i)-2)
```

`CS_MAX` and `CS_MIN` are easier-to-read versions of the ternary `?:` operator and compute the maximum and minimum of `a` and `b`, respectively. The `CS_FLIP(i)` macro computes the simple function `-(i)-2`, so named because it "flips" an integer about the pivotal integer `-1`, somewhat analogous to flipping the sign-bit of a number. More precisely, `CS_FLIP(-1)` is `-1`, and for all integers (ignoring overflow) `CS_FLIP(CS_FLIP(i))` equals `i`.

Bibliography

[1] P. R. AMESTOY, T. A. DAVIS, AND I. S. DUFF, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905. (Cited on pp. 131, 186.)

[2] ——, *Algorithm 837: AMD, an approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 381–388. (Cited on pp. 131, 186.)

[3] P. R. AMESTOY AND I. S. DUFF, *Vectorization of a multiprocessor multifrontal code*, Intl. J. Supercomp. Appl., 3 (1989), pp. 41–59. (Cited on p. 143.)

[4] P. R. AMESTOY, I. S. DUFF, AND J.-Y. L'EXCELLENT, *Multifrontal parallel distributed symmetric and unsymmetric solvers*, Comput. Methods Appl. Mech. Engrg., 184 (2000), pp. 501–520. (Cited on pp. 94, 143.)

[5] P. R. AMESTOY, I. S. DUFF, J.-Y. L'EXCELLENT, AND J. KOSTER, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41. (Cited on pp. 94, 143.)

[6] P. R. AMESTOY, I. S. DUFF, AND C. PUGLISI, *Multifrontal QR factorization in a multiprocessor environment*, Numer. Linear Algebra Appl., 3 (1996), pp. 275–300. (Cited on pp. 82, 143.)

[7] P. R. AMESTOY, A. GUERMOUCHE, J.-Y. L'EXCELLENT, AND S. PRALET, *Hybrid scheduling for the parallel solution of linear systems*, Parallel Comput., 32 (2006), pp. 136–156. (Cited on pp. 94, 143.)

[8] E. ANDERSON, Z. BAI, C. BISCHOF, S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, AND D. SORENSEN, *LAPACK Users' Guide*, SIAM, Philadelphia, 3rd ed., 1999. (Cited on p. 67.)

[9] M. ARIOLI, J. W. DEMMEL, AND I. S. DUFF, *Solving sparse linear systems with sparse backward error*, SIAM J. Matrix Anal. Appl., 10 (1989), pp. 165–190. (Cited on pp. 141, 144.)

[10] C. Ashcraft, *Compressed graphs and the minimum degree algorithm*, SIAM J. Sci. Comput., 16 (1995), pp. 1404–1411. (Cited on p. 143.)

[11] C. Ashcraft and R. Grimes, *SPOOLES: An object-oriented sparse matrix library*, in Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, 1999. (Cited on p. 143.)

[12] C. Ashcraft, R. Grimes, and J. G. Lewis, *Accurate symmetric indefinite linear equation solvers*, SIAM J. Matrix Anal. Appl., 20 (1998), pp. 513–561. (Cited on p. 143.)

[13] C. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon, *Progress in sparse matrix methods for large linear systems on vector supercomputers*, Intl. J. Supercomp. Appl., 1 (1987), pp. 10–30. (Cited on p. 143.)

[14] S. T. Barnard, A. Pothen, and H. D. Simon, *A spectral algorithm for envelope reduction of sparse matrices*, Numer. Linear Algebra Appl., 2 (1995), pp. 317–334. (Cited on p. 132.)

[15] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1993. (Cited on p. 6.)

[16] C. H. Bischof, C.-T. Pan, and P. T. P. Tang, *A Cholesky up- and downdating algorithm for systolic and SIMD architectures*, SIAM J. Sci. Comput., 14 (1993), pp. 670–676. (Cited on p. 67.)

[17] Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996. (Cited on p. 6.)

[18] J. R. Bunch and L. Kaufmann, *Some stable methods for calculating inertia and solving symmetric linear systems*, Math. Comp., 31 (1977), pp. 163–179. (Cited on p. 141.)

[19] J. R. Bunch, L. Kaufmann, and B. N. Parlett, *Decomposition of a symmetric matrix*, Numer. Math., 27 (1976), pp. 95–110. (Cited on p. 141.)

[20] N. A. Carlson, *Fast triangular factorization of the square root filter*, AIIA Journal, 11 (1973), pp. 1259–1265. (Cited on p. 67.)

[21] W. M. Chan and A. George, *A linear time implementation of the reverse Cuthill-McKee algorithm*, BIT, 20 (1980), pp. 8–14. (Cited on p. 132.)

[22] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert, *Predicting fill for sparse orthogonal factorization*, J. Assoc. Comput. Mach., 33 (1986), pp. 517–532. (Cited on pp. 72, 81.)

[23] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990. (Cited on pp. 6, 35, 132.)

[24] H. L. Crane, N. E. Gibbs, W. G. Poole, Jr., and P. K. Stockmeyer, *Algorithm 508: Matrix bandwidth and profile reduction*, ACM Trans. Math. Software, 2 (1976), pp. 375–377. (Cited on p. 132.)

[25] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th Conference of the ACM, 1969, Brandon Press, Princeton, NJ, pp. 157–172. (Cited on p. 132.)

[26] A. C. Damhaug and J. K. Reid, *MA46: a Fortran code for direct solution of sparse unsymmetric linear systems of equations from finite-element applications*. Technical report RAL-TR-96-010, Rutherford Appleton Laboratory, Didcot, UK, 1996. (Cited on p. 143.)

[27] T. A. Davis, *Algorithm 832: UMFPACK V4.3, an unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 196–199. (Cited on pp. 94, 186.)

[28] ——, *A column pre-ordering strategy for the unsymmetric-pattern multifrontal method*, ACM Trans. Math. Software, 30 (2004), pp. 165–195. (Cited on pp. 94, 143, 186.)

[29] ——, *Algorithm 849: A concise sparse Cholesky factorization package*, ACM Trans. Math. Software, 31 (2005), pp. 587–591. (Cited on pp. 58, 66, 67, 143.)

[30] ——, *CHOLMOD users' guide*, www.cise.ufl.edu/research/sparse/cholmod, University of Florida, Gainesville, FL, 2005. (Cited on pp. 24, 132, 143, 186.)

[31] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 140–158. (Cited on pp. 94, 143, 143, 186.)

[32] ——, *A combined unifrontal/multifrontal method for unsymmetric sparse matrices*, ACM Trans. Math. Software, 25 (1999), pp. 1–19. (Cited on pp. 94, 143, 186.)

[33] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, *Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 377–380. (Cited on pp. 131, 186.)

[34] ——, *A column approximate minimum degree ordering algorithm*, ACM Trans. Math. Software, 30 (2004), pp. 353–376. (Cited on pp. 131, 186.)

[35] T. A. Davis and W. W. Hager, *Modifying a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 606–627. (Cited on p. 67.)

[36] ——, *Multiple-rank modifications of a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 997–1013. (Cited on p. 67.)

[37] ——, *Row modifications of a sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 26 (2005), pp. 621–639. (Cited on p. 67.)

[38] T. A. Davis and K. Sigmon, *MATLAB Primer*, Chapman & Hall/CRC Press, Boca Raton, FL, 7th ed., 2005. (Cited on pp. 6, 186.)

[39] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997. (Cited on p. 6.)

[40] 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. Appl., 20 (1999), pp. 720–755. (Cited on p. 143.)

[41] J. W. Demmel, J. R. Gilbert, and X. S. Li, *An asynchronous parallel supernodal algorithm for sparse Gaussian elimination*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 915–952. (Cited on p. 143.)

[42] F. Dobrian, G.-K. Kumfert, and A. Pothen, *The design of sparse direct solvers using object-oriented techniques*, in Advances in Software Tools for Scientific Computing, Springer-Verlag, Berlin, 2000, pp. 89–131. (Cited on p. 143.)

[43] D. S. Dodson, R. G. Grimes, and J. G. Lewis, *Algorithm 692: Model implementation and test package for the sparse basic linear algebra subprograms*, ACM Trans. Math. Software, 17 (1991), pp. 264–272. (Cited on p. 24.)

[44] ——, *Sparse extensions to the Fortran basic linear algebra subprograms*, ACM Trans. Math. Software, 17 (1991), pp. 253–262. (Cited on p. 24.)

[45] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979. (Cited on p. 67.)

[46] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of level-3 basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17. (Cited on p. 67.)

[47] I. S. Duff, *On the number of nonzeros added when Gaussian elimination is performed on sparse random matrices*, Math. Comp., 28 (1974), pp. 219–230. (Cited on p. 186.)

[48] ——, *Algorithm 575: Permutations for a zero-free diagonal*, ACM Trans. Math. Software, 7 (1981), pp. 387–390. (Cited on p. 132.)

[49] ——, *On algorithms for obtaining a maximum transversal*, ACM Trans. Math. Software, 7 (1981), pp. 315–330. (Cited on p. 132.)

[50] ——, *Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 270–280. (Cited on p. 143.)

[51] ——, *MA57—a code for the solution of sparse symmetric definite and indefinite systems*, ACM Trans. Math. Software, 30 (2004), pp. 118–144. (Cited on p. 143.)

[52] I. S. Duff, A. M. Erisman, and J. K. Reid, *On George's nested dissection method*, SIAM J. Numer. Anal., 13 (1976), pp. 686–695. (Cited on p. 132.)

[53] ——, *Direct Methods for Sparse Matrices*, Oxford University Press, New York, 1986. (Cited on pp. 6, 94, 131.)

[54] I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, and K. Turner, *The factorization of sparse symmetric indefinite matrices*, IMA J. Numer. Anal., 11 (1991), pp. 181–204. (Cited on p. 143.)

[55] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14. (Cited on p. 24.)

[56] I. S. Duff, M. A. Heroux, and R. Pozo, *An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum*, ACM Trans. Math. Software, 28 (2002), pp. 239–267. (Cited on p. 24.)

[57] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 889–901. (Cited on pp. 94, 132, 133.)

[58] ——, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996. (Cited on pp. 94, 132, 133.)

[59] I. S. Duff and J. K. Reid, *Algorithm 529: Permutations to block triangular form*, ACM Trans. Math. Software, 4 (1978), pp. 189–192. (Cited on p. 132.)

[60] ——, *An implementation of Tarjan's algorithm for the block triangularization of a matrix*, ACM Trans. Math. Software, 4 (1978), pp. 137–147. (Cited on p. 132.)

[61] ——, *Some design features of a sparse matrix code*, ACM Trans. Math. Software, 5 (1979), pp. 18–35. (Cited on pp. 94, 131, 143.)

[62] ——, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Trans. Math. Software, 9 (1983), pp. 302–325. (Cited on pp. 94, 131, 143.)

[63] ——, *The multifrontal solution of unsymmetric sets of linear equations*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 633–641. (Cited on pp. 94, 143.)

[64] ——, *MA47, a Fortran code for the direct solution of indefinite sparse symmetric linear systems.* Technical report RAL-95-001, Rutherford Appleton Laboratory, Didcot, UK, 1995. (Cited on p. 143.)

[65] ——, *The design of MA48: A code for the direct solution of sparse unsymmetric linear systems of equations*, ACM Trans. Math. Software, 22 (1996), pp. 187–226. (Cited on p. 143.)

[66] ——, *Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems*, ACM Trans. Math. Software, 22 (1996), pp. 227–257. (Cited on p. 143.)

[67] I. S. Duff and J. A. Scott, *The design of a new frontal code for solving sparse, unsymmetric systems*, ACM Trans. Math. Software, 22 (1996), pp. 30–45. (Cited on p. 143.)

[68] ——, *A frontal code for the solution of sparse positive-definite symmetric systems arising from finite-element applications*, ACM Trans. Math. Software, 25 (1999), pp. 404–424. (Cited on p. 143.)

[69] ——, *A parallel direct solver for large sparse highly unsymmetric linear systems*, ACM Trans. Math. Software, 30 (2004), pp. 95–117. (Cited on p. 143.)

[70] I. S. Duff and C. Voemel, *Algorithm 818: A reference model implementation of the sparse BLAS in Fortran 95*, ACM Trans. Math. Software, 28 (2002), pp. 268–283. (Cited on p. 24.)

[71] I. S. Duff and T. Wiberg, *Remarks on implementation of $O(n^{1/2}\tau)$ assignment algorithms*, ACM Trans. Math. Software, 14 (1988), pp. 267–287. (Cited on p. 132.)

[72] A. L. Dulmage and N. S. Mendelsohn, *Two algorithms for bipartite graphs*, J. Soc. Indust. Appl. Math., 11 (1963), pp. 183–194. (Cited on p. 130.)

[73] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale sparse matrix package, I: The symmetric codes*, Internat. J. Numer. Methods Engrg., 18 (1982), pp. 1145–1151. (Cited on pp. 66, 131.)

[74] S. C. Eisenstat and J. W. H. Liu, *Exploiting structural symmetry in a sparse partial pivoting code*, SIAM J. Sci. Comput., 14 (1993), pp. 253–257. (Cited on p. 94.)

[75] ——, *The theory of elimination trees for sparse unsymmetric matrices*, SIAM J. Matrix Anal. Appl., 26 (2005), pp. 686–705. (Cited on p. 94.)

[76] 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 (1981), pp. 225–237. (Cited on pp. 66, 131.)

[77] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference, Las Vegas, NV, 1982, pp. 175–181. (Cited on p. 132.)

[78] M. FIEDLER, *Algebraic connectivity of graphs*, Czechoslovak Math J., 23 (1973), pp. 298–305. (Cited on p. 132.)

[79] T. FRÄNGSMYR, ED., *Les Prix Nobel*, Nobel Foundation, Stockholm, 1991. (Cited on p. 131.)

[80] C. FU, X. JIAO, AND T. YANG, *Efficient sparse LU factorization with partial pivoting on distributed memory architectures*, IEEE Trans. Parallel Distrib. Systems, 9 (1998), pp. 109–125. (Cited on p. 143.)

[81] A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363. (Cited on p. 132.)

[82] ——, *An automatic one-way dissection algorithm for irregular finite-element problems*, SIAM J. Numer. Anal., 17 (1980), pp. 740–751. (Cited on p. 132.)

[83] A. GEORGE AND M. T. HEATH, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra Appl., 34 (1980), pp. 69–83. (Cited on pp. 72, 81.)

[84] A. GEORGE, M. T. HEATH, AND E. NG, *Solution of sparse underdetermined systems of linear equations*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 988–997. (Cited on p. 82.)

[85] A. GEORGE AND J. W. H. LIU, *The design of a user interface for a sparse matrix package*, ACM Trans. Math. Software, 5 (1979), pp. 139–162. (Cited on pp. 131, 143.)

[86] ——, *An implementation of a pseudo-peripheral node finder*, ACM Trans. Math. Software, 5 (1979), pp. 284–295. (Cited on p. 132.)

[87] ——, *A minimal storage implementation of the minimum degree algorithm*, SIAM J. Numer. Anal., 17 (1980), pp. 282–299. (Cited on p. 131.)

[88] ——, *An optimal algorithm for symbolic factorization of symmetric matrices*, SIAM J. Comput., 9 (1980), pp. 583–593. (Cited on p. 66.)

[89] ——, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice–Hall, Englewood Cliffs, NJ, 1981. (Cited on pp. 6, 52, 66, 143.)

[90] ——, *Householder reflections versus Givens rotations in sparse orthogonal decomposition*, Linear Algebra Appl., 88 (1987), pp. 223–238. (Cited on p. 81.)

[91] ——, *The evolution of the minimum degree ordering algorithm*, SIAM Rev., 31 (1989), pp. 1–19. (Cited on p. 131.)

[92] A. GEORGE, J. W. H. LIU, AND E. G. NG, *Row ordering schemes for sparse Givens transformations: I. Bipartite graph model*, Linear Algebra Appl., 61 (1984), pp. 55–81. (Cited on p. 81.)

[93] ——, *Row ordering schemes for sparse Givens transformations: II. Implicit graph model*, Linear Algebra Appl., 75 (1986), pp. 203–223. (Cited on p. 81.)

[94] ——, *Row ordering schemes for sparse Givens transformations: III. Analyses for a model problem*, Linear Algebra Appl., 75 (1986), pp. 225–240. (Cited on p. 81.)

[95] ——, *A data structure for sparse QR and LU factorizations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 100–121. (Cited on pp. 72, 80, 81.)

[96] A. George and E. Ng, *On row and column orderings for sparse least square problems*, SIAM J. Numer. Anal., 20 (1983), pp. 326–344. (Cited on p. 81.)

[97] ——, *An implementation of Gaussian elimination with partial pivoting for sparse systems*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 390–409. (Cited on pp. 72, 83, 94.)

[98] ——, *Symbolic factorization for sparse Gaussian elimination with partial pivoting*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 877–898. (Cited on p. 94.)

[99] N. E. Gibbs, *Algorithm 509: A hybrid profile reduction algorithm*, ACM Trans. Math. Software, 2 (1976), pp. 378–387. (Cited on p. 132.)

[100] N. E. Gibbs, W. G. Poole, Jr., and P. K. Stockmeyer, *A comparison of several bandwidth and reduction algorithms*, ACM Trans. Math. Software, 2 (1976), pp. 322–330. (Cited on p. 132.)

[101] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 62–79. (Cited on pp. 6, 17, 66, 83, 84, 135.)

[102] J. R. Gilbert, X. S. Li, E. G. Ng, and B. W. Peyton, *Computing row and column counts for sparse QR and LU factorization*, BIT, 41 (2001), pp. 693–710. (Cited on pp. 66, 81.)

[103] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 334–352. (Cited on p. 94.)

[104] J. R. Gilbert, G. L. Miller, and S.-H. Teng, *Geometric mesh partitioning: Implementation and experiments*, SIAM J. Sci. Comput., 19 (1998), pp. 2091–2110. (Cited on p. 132.)

[105] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: Design and implementation*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356. (Cited on pp. 24, 35, 141, 143, 186.)

[106] J. R. Gilbert and E. G. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, in Graph Theory and Sparse Matrix Computations, A. George, J. R. Gilbert, and J. W. H. Liu, eds., vol. 56 of IMA Vol. Math. Appl., Springer-Verlag, New York, 1993, pp. 107–139. (Cited on pp. 81, 83, 84, 94.)

[107] 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. Appl., 15 (1994), pp. 1075–1091. (Cited on pp. 66, 81.)

[108] ——, *Separators and structure prediction in sparse orthogonal factorization*, Linear Algebra Appl., 262 (1997), pp. 83–97. (Cited on p. 81.)

[109] J. R. GILBERT AND T. PEIERLS, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 862–874. (Cited on pp. 30, 35, 94, 143.)

[110] P. E. GILL, G. H. GOLUB, W. MURRAY, AND M. A. SAUNDERS, *Methods for modifying matrix factorizations*, Math. Comp., 28 (1974), pp. 505–535. (Cited on p. 67.)

[111] M. I. GILLESPIE AND D. D. OLESKY, *Ordering Givens rotations for sparse QR factorization*, SIAM J. Matrix Anal. Appl., 16 (1994), pp. 1024–1041. (Cited on p. 81.)

[112] W. GIVENS, *Computation of plane unitary rotations transforming a general matrix to triangular form*, J. Soc. Indust. Appl. Math., 6 (1958), pp. 26–50. (Cited on p. 81.)

[113] G. H. GOLUB, *Numerical methods for solving linear least squares problems*, Numer. Math., 7 (1965), pp. 206–216. (Cited on p. 70.)

[114] G. H. GOLUB AND C. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 3rd ed., 1996. (Cited on pp. 6, 72, 81.)

[115] K. GOTO AND R. VAN DE GEIJN, *On reducing TLB misses in matrix multiplication.* Technical report TR-2002-55, University of Texas, Austin, TX, 2002. (Cited on p. 67.)

[116] N. I. M. GOULD, Y. HU, AND J. A. SCOTT, *A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations*, www.numerical.rl.ac.uk/reports/reports.html. Technical report RAL-2005-005, Rutherford Appleton Laboratory, Didcot, UK, 2005, ACM Trans. Math. Software, to appear. (Cited on pp. 6, 67, 134.)

[117] A. GREENBAUM, *Iterative Methods for Solving Linear Systems*, SIAM, Philadelphia, 1997. (Cited on p. 6.)

[118] A. GUPTA, *Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices*, SIAM J. Matrix Anal. Appl., 24 (2002), pp. 529–552. (Cited on pp. 94, 143.)

[119] A. GUPTA, G. KARYPIS, AND V. KUMAR, *Highly scalable parallel algorithms for sparse matrix factorization*, IEEE Trans. Parallel Distrib. Systems, 8 (1997), pp. 502–520. (Cited on pp. 94, 143, 143.)

[120] F. G. GUSTAVSON, *Finding the block lower triangular form of a sparse matrix*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976, pp. 275–290. (Cited on p. 132.)

[121] ——, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Software, 4 (1978), pp. 250–269. (Cited on p. 24.)

[122] S. M. HADFIELD, *On the LU Factorization of Sequences of Identically Structured Sparse Matrices within a Distributed Memory Environment*, Ph.D. thesis, University of Florida, Gainesville, FL, 1994. (Cited on p. 94.)

[123] W. W. HAGER, *Condition estimates*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 311–316. (Cited on pp. 96, 186.)

[124] ——, *Updating the inverse of a matrix*, SIAM Rev., 31 (1989), pp. 221–239. (Cited on p. 67.)

[125] ——, *Minimizing the profile of a symmetric matrix*, SIAM J. Sci. Comput., 23 (2002), pp. 1799–1816. (Cited on p. 132.)

[126] W. W. HAGER, S. C. PARK, AND T. A. DAVIS, *Block exchange in graph partitioning*, in Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems, P. M. Pardalos, ed., Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000, pp. 299–307. (Cited on p. 132.)

[127] D. R. HARE, C. R. JOHNSON, D. D. OLESKY, AND P. VAN DEN DRIESSCHE, *Sparsity analysis of the QR factorization*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 655–669. (Cited on p. 81.)

[128] M. T. HEATH, *Numerical methods for large sparse linear least squares problems*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 497–513. (Cited on p. 81.)

[129] M. T. HEATH AND P. RAGHAVAN, *A Cartesian parallel nested dissection algorithm*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 235–253. (Cited on pp. 132, 143.)

[130] ——, *Performance of a fully parallel sparse solver*, Intl. J. Supercomp. Appl., 11 (1997), pp. 49–64. (Cited on p. 143.)

[131] B. HENDRICKSON AND R. LELAND, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM J. Sci. Comput., 16 (1994), pp. 452–469. (Cited on p. 132.)

[132] P. HÉNON, P. RAMET, AND J. ROMAN, *PaStiX: A high-performance parallel direct solver for sparse symmetric positive definite systems*, Parallel Comput., 28 (2002), pp. 301–321. (Cited on p. 143.)

[133] D. J. HIGHAM AND N. J. HIGHAM, *MATLAB Guide*, SIAM, Philadelphia, 2nd ed., 2005. (Cited on pp. 6, 186.)

[134] N. J. HIGHAM, *Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation*, ACM Trans. Math. Software, 14 (1988), pp. 381–396. (Cited on p. 96.)

[135] ——, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 2nd ed., 2002. (Cited on pp. 6, 81, 141, 144.)

[136] N. J. HIGHAM AND F. TISSEUR, *A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra*, SIAM J. Matrix Anal. Appl., 21 (2000), pp. 1185–1201. (Cited on pp. 96, 186.)

[137] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231. (Cited on p. 132.)

[138] A. S. HOUSEHOLDER, *Unitary triangularization of a nonsymmetric matrix*, J. Assoc. Comput. Mach., 5 (1958), pp. 339–342. (Cited on p. 81.)

[139] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392. (Cited on p. 132.)

[140] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell System Tech. J., 49 (1970), pp. 291–307. (Cited on p. 132.)

[141] B. W. KERNIGHAN AND D. M. RITCHIE, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 2nd ed., 1988. (Cited on pp. 6, 187.)

[142] G. KUMFERT AND A. POTHEN, *Two improved algorithms for reducing the envelope and wavefront*, BIT, 37 (1997), pp. 559–590. (Cited on p. 132.)

[143] K. S. KUNDERT, *Sparse matrix techniques*, in Circuit Analysis, Simulation and Design: General Aspects of Circuit Analysis and Design, A. E. Ruehli, ed., vol. 3, part 1 of Advances in CAD for VLSI, North–Holland, New York, 1986, pp. 281–324. (Cited on p. 143.)

[144] R. B. LEHOUCQ AND D. C. SORENSEN, *Deflation techniques for an implicitly restarted Arnoldi iteration*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 789–821. (Cited on p. 186.)

[145] R. B. LEHOUCQ, D. C. SORENSEN, AND C. YANG, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, SIAM, Philadelphia, 1998. (Cited on p. 186.)

[146] J. G. LEWIS, *Algorithm 582: The Gibbs-Poole-Stockmeyer and Gibbs-King algorithms for reordering sparse matrices*, ACM Trans. Math. Software, 8 (1982), pp. 190–194. (Cited on p. 132.)

[147] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Math. Software, 29 (2003), pp. 110–140. (Cited on pp. 94, 94, 143.)

[148] J. W. H. Liu, *A compact row storage scheme for Cholesky factors using elimination trees*, ACM Trans. Math. Software, 12 (1986), pp. 127–148. (Cited on pp. 40, 41, 66.)

[149] ——, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 1190–1211. (Cited on p. 81.)

[150] ——, *The role of elimination trees in sparse factorization*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 134–172. (Cited on pp. 6, 44, 66.)

[151] ——, *A generalized envelope method for sparse factorization by rows*, ACM Trans. Math. Software, 17 (1991), pp. 112–129. (Cited on p. 66.)

[152] ——, *The multifrontal method for sparse matrix solution: Theory and practice*, SIAM Rev., 34 (1992), pp. 82–109. (Cited on pp. 66, 94.)

[153] W.-H. Liu and A. H. Sherman, *Comparative analysis of the Cuthill–McKee and the reverse Cuthill–McKee ordering algorithms for sparse matrices*, SIAM J. Numer. Anal., 13 (1976), pp. 198–213. (Cited on p. 132.)

[154] S.-M. Lu and J. L. Barlow, *Multifrontal computation with the orthogonal factors of sparse matrices*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 658–679. (Cited on p. 82.)

[155] H. M. Markowitz, *The elimination form of the inverse and its application to linear programming*, Management Sci., 3 (1957), pp. 255–269. (Cited on p. 131.)

[156] P. Matstoms, *Sparse QR factorization in MATLAB*, ACM Trans. Math. Software, 20 (1994), pp. 136–159. (Cited on p. 82.)

[157] C. Moler, *Numerical Computing with MATLAB*, SIAM, Philadelphia, 2004. (Cited on p. 6.)

[158] E. G. Ng and B. W. Peyton, *A tight and explicit representation of Q in sparse QR factorization*, IMA Preprint Series 981, University of Minnesota, Minneapolis, MN, 1992. (Cited on p. 81.)

[159] ——, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., 14 (1993), pp. 1034–1056. (Cited on p. 143.)

[160] ——, *Some results on structure prediction in sparse QR factorization*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 443–459. (Cited on p. 81.)

[161] E. G. Ng and P. Raghavan, *Performance of greedy ordering heuristics for sparse Cholesky factorization*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 902–914. (Cited on p. 132.)

[162] S. OLIVEIRA, *Exact prediction of QR fill-in by row-merge trees*, SIAM J. Sci. Comput., 22 (2001), pp. 1962–1973. (Cited on p. 81.)

[163] C.-T. PAN, *A modification to the LINPACK downdating algorithm*, BIT, 30 (1990), pp. 707–722. (Cited on p. 67.)

[164] B. N. PARLETT, *The Symmetric Eigenvalue Problem*, SIAM, Philadelphia, 1997. (Cited on p. 6.)

[165] S. PARTER, *The use of linear graphs in Gauss elimination*, SIAM Rev., 3 (1961), pp. 119–130. (Cited on p. 39.)

[166] F. PELLEGRINI, J. ROMAN, AND P. R. AMESTOY, *Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering*, Concurrency: Pract. Exp., 12 (2000), pp. 68–84. (Cited on p. 132.)

[167] D. J. PIERCE AND J. G. LEWIS, *Sparse multifrontal rank revealing QR factorization*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 159–180. (Cited on pp. 82, 143.)

[168] A. POTHEN, *Predicting the structure of sparse orthogonal factors*, Linear Algebra Appl., 194 (1993), pp. 183–204. (Cited on p. 81.)

[169] A. POTHEN AND C. FAN, *Computing the block triangular form of a sparse matrix*, ACM Trans. Math. Software, 16 (1990), pp. 303–324. (Cited on p. 132.)

[170] A. POTHEN, H. D. SIMON, AND K.-P. LIOU, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452. (Cited on p. 132.)

[171] P. RAGHAVAN, *Domain-separator codes for the parallel solution of sparse linear systems.* Technical report CSE-02-004, Penn State, University Park, PA, 2002. (Cited on p. 143.)

[172] J. K. REID AND J. A. SCOTT, *Ordering symmetric sparse matrices for small profile and wavefront*, Internat. J. Numer. Methods Engrg., 45 (1999), pp. 1737–1755. (Cited on p. 132.)

[173] ——, *Implementing Hager's exchange methods for matrix profile reduction*, ACM Trans. Math. Software, 28 (2002), pp. 377–391. (Cited on p. 132.)

[174] D. J. ROSE AND R. E. TARJAN, *Algorithmic aspects of vertex elimination on directed graphs*, SIAM J. Appl. Math., 34 (1978), pp. 176–197. (Cited on p. 94.)

[175] D. J. ROSE, R. E. TARJAN, AND G. S. LUEKER, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266–283. (Cited on pp. 37, 94.)

[176] E. Rothberg and S. C. Eisenstat, *Node selection strategies for bottom-up sparse matrix ordering*, SIAM J. Matrix Anal. Appl., 19 (1998), pp. 682–695. (Cited on p. 132.)

[177] V. Rotkin and S. Toledo, *The design and implementation of a new out-of-core sparse Cholesky factorization method*, ACM Trans. Math. Software, 30 (2004), pp. 19–46. (Cited on p. 143.)

[178] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2nd ed., 2003. (Cited on pp. 6, 68, 96.)

[179] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Future Generation Comp. Sys., 20 (2001), pp. 475–487. (Cited on pp. 94, 143.)

[180] O. Schenk, K. Gärtner, and W. Fichtner, *Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors*, BIT, 40 (2000), pp. 158–176. (Cited on pp. 94, 143.)

[181] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Trans. Math. Software, 8 (1982), pp. 256–276. (Cited on pp. 39, 52, 66.)

[182] J. A. Scott, *The design of a portable parallel frontal solver for chemical process engineering problems*, Computers in Chem. Eng., 25 (2001), pp. 1699–1709. (Cited on p. 143.)

[183] ——, *A parallel frontal solver for finite element applications*, Internat. J. Numer. Methods Engrg., 50 (2001), pp. 1131–1144. (Cited on p. 143.)

[184] ——, *Parallel frontal solvers for large sparse linear systems*, ACM Trans. Math. Software, 29 (2003), pp. 395–417. (Cited on p. 143.)

[185] K. Shen, T. Yang, and X. Jiao, *S+: Efficient 2D sparse LU factorization on parallel machines*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 282–305. (Cited on p. 143.)

[186] A. H. Sherman, *Algorithms for sparse Gaussian elimination with partial pivoting*, ACM Trans. Math. Software, 4 (1978), pp. 330–338. (Cited on p. 143.)

[187] S. W. Sloan, *An algorithm for profile and wavefront reduction of sparse matrices*, Internat. J. Numer. Methods Engrg., 23 (1986), pp. 239–251. (Cited on p. 132.)

[188] D. C. Sorensen, *Implicit application of polynomial filters in a k-step Arnoldi method*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 357–385. (Cited on p. 186.)

[189] G. W. Stewart, *The effects of rounding error on an algorithm for downdating a Cholesky factorization*, J. Inst. Math. Appl., 23 (1979), pp. 203–213. (Cited on p. 67.)

[190] ——, *Matrix Algorithms, Volume I: Basic Decompositions*, SIAM, Philadelphia, 1998. (Cited on pp. 6, 67, 81.)

[191] ——, *Matrix Algorithms, Volume II: Eigensystems*, SIAM, Philadelphia, 2001. (Cited on p. 6.)

[192] ——, *Building an old-fashioned sparse solver*, www.cs.umd.edu/~stewart. Technical report CS-TR-4527/UMIACS-TR-2003-95, University of Maryland Computer Science Department, College Park, MD, 2003. (Cited on p. 6.)

[193] G. STRANG, *Introduction to Linear Algebra*, Wellesley-Cambridge Press, Wellesley, MA, 3rd ed., 2003. (Cited on p. 6.)

[194] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160. (Cited on p. 132.)

[195] ——, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225. (Cited on p. 67.)

[196] W. F. TINNEY AND J. W. WALKER, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proc. IEEE, 55 (1967), pp. 1801–1809. (Cited on p. 131.)

[197] C. WALSHAW, M. CROSS, AND M. EVERETT, *Parallel dynamic graph partitioning for adaptive unstructured meshs*, J. Parallel Distrib. Comput., 47 (1997), pp. 102–108. (Cited on p. 132.)

[198] S. WOLFRAM, *The Mathematica Book*, Wolfram Media, Champaign, IL, 5th ed., 2003. (Cited on p. 143.)

[199] M. YANNAKAKIS, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Disc. Meth., 2 (1981), pp. 77–79. (Cited on p. 130.)

[200] Z. ZLATEV, J. WASNIEWSKI, AND K. SCHAUMBURG, *Y12M: Solution of Large and Sparse Systems of Linear Algebraic Equations*, Lect. Notes Comput. Sci. 121, Springer-Verlag, Berlin, 1981. (Cited on p. 143.)

This page intentionally left blank

Index