Algorithms for sparse linear systems

Nečas Center Series

Jennifer Scott, Miroslav Tůma

Algorithms for Sparse Linear Systems

OPEN ACCESS

Birkhäuser

Nečas Center Series

The Nečas Center Series aims to publish high-quality monographs, textbooks, lecture notes, habilitation and Ph.D. theses in the field of mathematics and related areas in the natural and social sciences and engineering. There is no restriction regarding the topic, although we expect that the main fields will include continuum thermodynamics, solid and fluid mechanics, mixture theory, partial differential equations, numerical mathematics, matrix computations, scientific computing and applications. Emphasis will be placed on viewpoints that bridge disciplines and on connections between apparently different fields. Potential contributors to the series are encouraged to contact the editor-in-chief and the manager of the series.

All manuscripts are peer-reviewed to meet the highest standards of scientific literature. Interested authors may submit proposals by email to the series editors or to the relevant Birkhäuser editor listed under “Contacts.”

Jennifer Scott • Miroslav Tůma

Algorithms for Sparse Linear Systems

Birkhäuser

Jennifer Scott
Department of Mathematics and Statistics
University of Reading
Reading, UK

Computational Mathematics Group
STFC Rutherford Appleton Laboratory
Harwell, UK

Miroslav Tůma
Faculty of Mathematics and Physics
Charles University
Prague, Czech Republic

ISSN 2523-3343 ISSN 2523-3351 (electronic)
Nečas Center Series
ISBN 978-3-031-25819-0 ISBN 978-3-031-25820-6 (eBook)
https://doi.org/10.1007/978-3-031-25820-6

This work was supported by University of Reading

This book is published under the imprint Birkhäuser, www.birkhauser-science.com by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The solution of linear systems of equations $Ax = b$ is a cornerstone of computational science and engineering. Being able to solve linear systems in a reliable and efficient way is of great importance and interest not only to scientists and engineers but also to a huge and varied community of people who are unaware that at the heart of the software they are using lies a linear equation solver and that this is key to its feasibility and performance. In many applications, the linear systems that must be solved are large and square and they are **sparse** (that is, many of the entries in the system matrix A are zero). **Direct methods** for solving such systems are characterized by computing a **factorization** (or **decomposition**) of A into a product of much simpler matrices in such a way that solving systems of equations with these matrices is easy and inexpensive. For example, A may be factorized into a product of triangular matrices; in principle, solving a linear system in which the system matrix is triangular is straightforward. Direct methods obtain the solution to the linear system in a finite and fixed number of steps that is independent of A and b. Because of rounding errors, the computed solution is generally not equal to the exact one but, if a direct method is well implemented, the resulting software is extremely robust and can be used as a "black box solver", with the user not needing any detailed knowledge or understanding of what is going on within the box.

By contrast, an **iterative method** (sometimes also called an indirect method) generally involves an unknown number of steps and its performance is highly problem dependent. In many cases, for the method to converge to the sought-after solution of the linear system, it is necessary to use a **preconditioner**. This has to be tailored to the system being solved. The aim is to transform the linear system into one with more favourable numerical properties so that, when applied to the transformed system, the iterative solver converges to a solution of the requested accuracy in an acceptable number of steps. The major advantage of iterative solvers over direct ones is that they require very little memory and, once the preconditioner has been constructed, most of the computational work is in the application of the preconditioner and matrix-vector products with A. For extremely large problems (for example, systems coming from discretizations of real-world three- or four-

dimensional problems), memory requirements prohibit the use of direct methods, and without suitable iterative methods the systems would be intractable.

This book presents classical techniques for matrix factorizations based on variants of Gaussian elimination that are used in sparse direct methods and discusses the construction of approximate direct and inverse factorizations that are key to developing algebraic preconditioners for use with iterative solvers. While a number of books on iterative solvers discuss the construction of simple incomplete matrix factorizations for use as preconditioners, very few attempt to unite the fields of complete and incomplete factorizations or cover contemporary approaches. To achieve this broad view, we use a single framework that emphasizes the underlying sparsity structures and highlights the importance of understanding sparse direct techniques when building algebraic preconditioners.

The book is algorithmically oriented, presenting computational schemes that are designed to provide both an understanding of sophisticated sparse factorization techniques and how they can be implemented in practice. Throughout, we include outline algorithmic descriptions and use pseudocode that is independent of any programming language. However, limitations on space mean that it is beyond the scope of the book to discuss the complex implementation details that are needed in the development of high-quality sophisticated (parallel) production software for efficiently solving sparse linear systems using modern computer architectures.

The book is aimed at students of applied mathematics and scientific computing as well as at computational scientists and software developers interested in understanding the theory and algorithms needed to tackle the challenge of solving large-scale linear systems. The presented treatment is intended to be largely self-contained, and we assume only that the reader has a basic knowledge of linear algebra and numerical mathematics.

The organization of the book is as follows. Chapter 1 provides a general introduction to sparse matrices and the challenges of solving large sparse linear systems of equations. Concepts from graph theory that are used in the development of sparse matrix algorithms are recalled in Chapter 2. The material in Chapters 1 and 2 is rather elementary, but it serves to remind the reader of important ideas and to introduce the notation and terminology that is used throughout the rest of the book. An introduction to sparse matrix factorizations, including the use of block forms, is given in Chapter 3. Then, in Chapters 4 and 5, the symbolic and numerical factorization phases of sparse Cholesky methods for solving the important class of symmetric positive definite linear systems are discussed. Sparse LU factorizations for general nonsymmetric sparse systems are described in Chapter 6. Chapter 7 is devoted to stability and pivoting strategies and includes a discussion of factorizing sparse symmetric indefinite systems. Sparse matrix ordering algorithms that are essential for the efficiency of sparse solvers are presented in Chapter 8.

The final three chapters of the book switch attention from direct methods to the study of algebraic preconditioners for use with iterative solvers. The emphasis is on employing and adapting ideas and concepts used by direct solvers in the development of effective general classes of preconditioners that can be used for tackling a wide range of problems, without relying on detailed knowledge

of the properties of the underlying application. Chapter 9 introduces algebraic preconditioners and approximate factorizations. Chapters 10 and 11 then focus on two key classes of algebraic preconditioners: incomplete factorizations and sparse approximation inverse preconditioners.

We do not attempt to cite all the vast array of publications related to sparse direct methods and algebraic preconditioners. Furthermore, we do not include proofs for all the theoretical results that we present. Rather, for each theorem, we provide one or more citations to where the reader can find a proof and/or get a better understanding of the result. In general, we include citations to the original paper/book/report (or a textbook for standard results) and, in some cases, an additional citation that is either more accessible or presents an alternative proof. In addition, at the end of each chapter, we have a short section of notes with references to key publications that give a historical perspective and/or provide further reading. It is interesting to note that a Google Scholar search in July 2022 for the term "sparse matrix" lists more than 2.7 million results, while a search for "sparse matrix decompositions" gives in excess of a million results. Although the majority may not be relevant to our areas of interest, it does indicate the wealth of the available literature as well as the importance of sparse matrix algorithms and their widespread use.

This monograph and its study of sparse linear systems represents a natural extension of our successful long-term research collaboration, combined with the research and the software development projects that we have each worked on with other researchers. Past and present colleagues at the Rutherford Appleton Laboratory that Jennifer would particularly like to acknowledge and thank for many years of collaborations and enjoyable coffee time chats are Iain Duff, Nick Gould, Jonathan Hogg, Yifan Hu, Tyrone Rees, and John Reid. Miroslav would like to express his thanks to his first major collaborator Michele Benzi, from whom he learnt a lot, to Ivan Němec, who invited him to work on codes that are now in the RFEM Structural Analysis and Engineering Software, and to his colleagues and friends in Prague, especially Zdeněk Strakoš, Miro Rozložník, Josef Málek, Petr Tichý, and Iveta Hnětynková, who created a kind and productive working environment.

We are very grateful to Hussam Al Daas, Jonathan Hogg, and Gerard Meurant for reading and commenting on all or part of a draft of the book. They spotted errors and made suggestions that led to important improvements; we really appreciate the time they spent doing this for us. We would also like to thank our institutions for opportunities to spend time in Prague, the Rutherford Appleton Laboratory and Reading working on our joint research projects. Jennifer would like to acknowledge funding over the last 30 years from the Science and Technology Facilities Council and the Engineering and Physical Sciences Research Council. And we are extremely grateful to the University of Reading for providing the funding that allows this book to be published as open access.

And, finally, we each owe a huge debt of gratitude to our families. Jennifer wishes to dedicate the book to her close family, both those who are no longer with us and those who continue to be an important part of her life, and most especially Stewart,

Emma, Simon, Mark, and Rebecca for their constant encouragement. Miroslav would like to dedicate the book to the memory of his ever-supportive parents and to thank Anna, Markéta and Martin, who have always tolerated his passion for research.

Harwell and Reading, UK — Jennifer Scott
Prague, Czech Republic — Miroslav Tůma
August 2022

Contents

Notation: Quick Reference Summary

Notational Conventions Used for Matrices and Vectors

Capital italic letters, e.g. A, L, P	Matrices
Uppercase calligraphic letters e.g. $\mathcal{I}$, $\mathcal{S}$	Sets containing indices
Lower case non-integer italic letters, e.g. p, x, y	Vectors (may also denote a scalar or function but this will be clear from the context)
Lower case integer italic letters, e.g. i, j	Integer scalars
Lower case Greek italic letters, e.g. α, β, μ_i	Real scalars
Subscripted lower case non-integer italic letters, e.g. x_i, $x_{i:j}$	Vector entries, e.g. entry i and entries i to j of x x_i may also denote column i of matrix X
Double subscripted lower case italic letters, e.g. a_{ij}	Entry in row i, column j of matrix A
Double subscripted bracketed upper case italic letters, e.g. $(A)_{ij}$	Entry in row i, column j of matrix A (alternative notation for a_{ij})

Different forms of double subscripted upper case italic letters:

$A_{ib,\,jb}$	Sub-block of matrix A in position $(ib,\, jb)$
$A_{i,:}$ or $A_{i,1:n}$	Row i of matrix A (with n columns)
$A_{:,j}$ or $A_{1:n,\,j}$	Column j of matrix A (with n rows)
$A_{i:j,k}$	Submatrix comprising rows i to j, column k
$A_{i:j,k:l}$	Submatrix comprising rows i to j, columns k to l

$A_j = A_{1:j,1:j}$	Principal leading submatrix of A of order j
$A_{\mathcal{I},\mathcal{J}}$	Submatrix of A with row and column indices in sets $\mathcal{I}$ and $\mathcal{J}$, respectively
$A_{i,\mathcal{J}}$	Entries in row i of A with column indices in set $\mathcal{J}$
$A_{\mathcal{I},j}$	Entries in column j of A with row indices in set $\mathcal{I}$
Lower case italic letters with superscript in brackets, e.g. $x^{(k)}$	Value of x at iteration k
Upper case italic letters with superscript in brackets, e.g. $A^{(k)}$	Matrix A at iteration k

Notational Conventions Used When Discussing Graphs

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$
$\mathcal{G}(A) = (\mathcal{V}(A), \mathcal{E}(A))$	Adjacency graph of matrix A
$adj_{\mathcal{G}}\{v\}$	Adjacency set of vertex v
$\mathcal{G}^{-1}(A)$	Skeleton graph of matrix A
$\mathcal{G}^k$	k-th elimination graph
$\mathcal{G}^{[k]}$	k-th quotient graph
$\mathcal{T}(A)$ or $\mathcal{T}$	Elimination tree of matrix A
$\mathcal{T}_r(i)$	i-th row subtree of $\mathcal{T}$
$\mathcal{T}(j)$	Subtree of $\mathcal{T}$ rooted at vertex j
Lower case italic letters, e.g. i, j, u, v	Graph vertices

The following are for an undirected graph $\mathcal{G}$:

$i_0 \longleftrightarrow i_1 \longleftrightarrow \ldots \longleftrightarrow i_{p-1} \longleftrightarrow i_p$	Sequence of undirected edges in $\mathcal{G}$
$(i \overset{\mathcal{G}}{\longleftrightarrow} j)$ or $(i \longleftrightarrow j)$ or (i, j)	Undirected edge between i and j in $\mathcal{G}$
$i \overset{\mathcal{G}}{\Longleftrightarrow} j$ or $i \Longleftrightarrow j$	Path from i to j in $\mathcal{G}$
$i \underset{min}{\overset{\mathcal{G}}{\Longleftrightarrow}} j$ or $i \underset{min}{\Longleftrightarrow} j$	All intermediate vertices on the path are less than $\min\{i, j\}$ (fill-path)
$i \underset{\mathcal{V}_s}{\overset{\mathcal{G}}{\Longleftrightarrow}} j$ or $i \underset{\mathcal{V}_s}{\Longleftrightarrow} j$	All intermediate vertices on the path belong to $\mathcal{V}_s$

The following are for a digraph (directed graph) $\mathcal{G}$:

$i_0 \longrightarrow i_1 \longrightarrow \ldots \longrightarrow i_{p-1} \longrightarrow i_p$	Sequence of directed edges in $\mathcal{G}$
$(i \overset{\mathcal{G}}{\longrightarrow} j)$ or $(i \longrightarrow j)$	Directed edge from i to j in $\mathcal{G}$
$i \overset{\mathcal{G}}{\Longrightarrow} j$ or $i \Longrightarrow j$	Path between i and j in $\mathcal{G}$

$i \overset{\mathcal{G}}{\underset{min}{\Longrightarrow}} j$ or $i \underset{min}{\overset{\mathcal{G}}{\Longrightarrow}} j$	All intermediate vertices on the path are less than $\min\{i, j\}$ (fill-path)
$i \overset{\mathcal{G}}{\underset{\mathcal{V}_s}{\Longrightarrow}} j$ or $i \underset{\mathcal{V}_s}{\overset{\mathcal{G}}{\Longrightarrow}} j$	All intermediate vertices on the path belong to $\mathcal{V}_s$

Specific Variables and Matrices That Are Used Throughout

A, x, b	The system matrix, solution vector, and right-hand-side vector ($Ax = b$)
A^T	The transpose of matrix A
D	Diagonal matrix with 1×1 (and possibly 2×2) blocks on the diagonal
D_A, L_A, U_A	Diagonal and strictly lower and upper triangular parts of A
I (or I_n)	Identity matrix (of order n)
L, U	Lower and upper triangular matrices; matrix factors
$\widetilde{L}, \widetilde{U}$	Approximate matrix factors
M	Preconditioner
P, Q	Row and column permutation matrices
S_r, S_c	Row and column scaling matrices
$\mathcal{S}\{A\}$ ($\mathcal{S}\{v\}$)	Sparsity pattern of matrix A (vector v)
$band(A)$	The band of a symmetrically structured matrix A
$env(A)$	The envelope of a symmetrically structured matrix A
e	Vector of all ones
e_i	i-th column of the identity matrix
f	Filled entry in a matrix factor
n	Order of A
$nz(A)$	Number of nonzero entries in A
ib, jb, kb, lb	Subscripts denoting blocks in (e.g.) A or L
nb	Number of row (and column) blocks of A in block form
$\|A\|_F$	Frobenius norm of matrix A
$\langle x, y\rangle_A$	A-inner product of vectors x and y, that is, $x^T Ay$
$\|x\|_A$	Corresponding A-norm of vector x, that is, $(x^T Ax)^{1/2}$
$\|x\|_2$	2-norm of vector x
$\kappa(C)$	Condition number of a matrix C
$\rho(C)$	Spectral radius of a matrix C
$\lambda_{min}(C), \lambda_{max}(C)$	Eigenvalues of C of smallest and largest absolute value
ρ_{growth}	Growth factor
ϵ	Machine precision
$\emptyset$	An empty set (one with no entries)

Abbreviations

AINV	Factorized approximate inverse
DAG	Directed acyclic graph
FSAI	Factorized sparse approximate inverse
IC/ ILU	Incomplete Cholesky/LU factorization
MIC/MILU	Modified incomplete Cholesky/LU factorization
PDE	Partial differential equation
SAINV	Stabilized factorized approximate inverse
SPAI	Sparse approximate inverse
SPD	Symmetric positive definite

Chapter 1
An Introduction to Sparse Matrices

Let us begin with a few words about the subject itself. What are all these research workers trying to do? Mostly, they are trying to solve $Ax = b$ . . . Amazing. Can people still find something new to say on these corny old subjects? The answer is yes . . . It is the pressure to solve bigger and more complex problems that has led people to return again and again to look in ever-increasing detail at such basic tools as a linear equations solver – Parlett (1974).

We may therefore interpret the elimination method as . . . the combination of two tricks: First, it decomposes A into a product of two [triangular] matrices . . . [and second] it forms their inverses by a simple, explicit, inductive process – Von Neumann & Goldstine (1947)

1.1 Motivation

Consider the simple matrix A on the left in Figure 1.1. Many of its entries are zero (and so are omitted). This is an example of a **sparse** matrix. The problem we are interested in is that of solving linear systems of equations $Ax = b$, where the square sparse matrix A and the vector b are given and the solution vector x is required. Such systems arise in a huge range of practical applications, including in areas as diverse as quantum chemistry, computer graphics, computational fluid dynamics, power networks, machine learning, and optimization. The list is endless and constantly growing, together with the sizes of the systems. For efficiency and to enable large systems to be solved, the sparsity of A must be exploited and operations with the zero entries avoided. To achieve this, sophisticated algorithms are required.

The majority of algorithms fall into two main categories: direct methods and iterative methods. **Direct methods** transform A using a finite sequence of elementary transformations into a product of simpler sparse matrices in such a way that solving linear systems of equations with these factor matrices is comparatively easy and inexpensive. For example, if A is symmetric, consider the Cholesky factorization $A = LL^T$, where the factor L is a lower triangular matrix (and the superscript

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_1

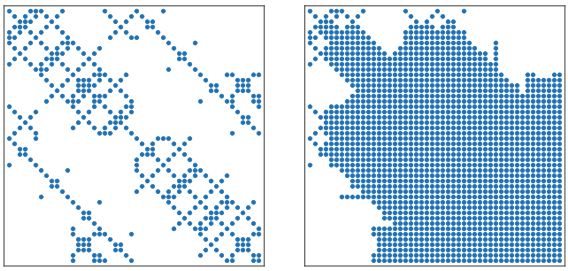


Figure 1.1 The locations of the nonzero entries in a sparse matrix from structural engineering (left) and in $L + L^T$ (right), where L is its Cholesky factor.

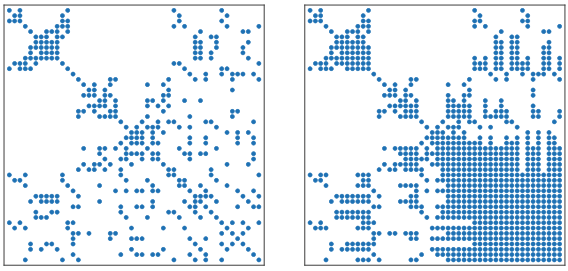


Figure 1.2 The locations of the nonzero entries in a symmetric permutation of the matrix from Figure 1.1 (left) and in $\bar{L} + \bar{L}^T$ (right), where $\bar{L}$ is the Cholesky factor of the permuted matrix.

L^T denotes the transpose of L). Solving linear systems with a triangular matrix is generally cheaper and more straightforward than for a general matrix. For the matrix in Figure 1.1, it is clear that L has filled in, that is, compared to A, it has more nonzero entries. If the amount of fill-in is too high, then the advantages of having a triangular matrix will be lost. An important question is: can we permute the rows and columns of A so as to reduce the fill-in in its factor L? One possibility is shown in Figure 1.2. Here A has been symmetrically permuted to give a matrix that has a much sparser factorization $\bar{L}\bar{L}^T$.

Having fewer entries in $\bar{L}$ reduces both the required storage and the number of operations that are needed to compute it and that must be performed when using it. This simple example suggests other possible questions, such as: how can the positions of the nonzero entries in A and in its factors be described? How can the sparsity pattern of the factors be determined from that of A? What influences the computational efficiency of matrix factorizations and other matrix transformations on contemporary computers?

Direct methods built on matrix factorizations are designed to be robust so that, properly implemented, they can be confidently used as black-box solvers for computing solutions with predictable accuracy. However, they can be expensive, requiring large amounts of memory, which increases with the size of A. By contrast, **iterative methods** compute a sequence of approximations

$$x^{(0)}, x^{(1)}, x^{(2)}, \ldots$$

that (hopefully) converge to the solution x of the linear system in an acceptable number of iterations. The number of iterations depends on the initial guess $x^{(0)}$, A and b as well as the accuracy that is wanted in x. Iterative methods use the matrix A only indirectly, through matrix–vector products, and their memory requirements are limited to a (small) number of vectors of length the order of A, making them attractive for very large problems and problems where A is not available explicitly. They can be terminated as soon as the required accuracy in the computed solution is achieved. Unfortunately, frequently convergence does not happen or the number of iterations is unacceptably large; in such cases, preconditioning is needed. The aim of preconditioning is to speed up convergence by transforming the given linear system into an equivalent system (or one from which it is easy to recover the solution of the original system) that has nicer numerical properties. For example, the transformed system could be

$$M^{-1}Ax = M^{-1}b,$$

where the matrix M is the **preconditioner** and M^{-1} denotes its inverse. Knowledge of the underlying problem, such as whether or not it arises from a partial differential equation, can help in the construction of an effective preconditioner. Otherwise, purely algebraic approaches that simply take the entries of A as input may be used. The class of **algebraic preconditioners** includes those based on incomplete (or approximate) factorizations of A. In this case, possible questions include: can some of the factor entries be discarded to obtain a sparser but approximate factor that is useful as a preconditioner? If so, which entries can be discarded? What are the implications of this on the associated computational costs?

This book uses a unified framework to address such questions for direct methods and algebraic preconditioners, examining both the theoretical and algorithmic aspects of solving large-scale linear systems of equations.

1.2 Introductory Terminology and Concepts

Our interest is in solving linear systems of equations

$$Ax = b, \tag{1.1}$$

where the matrix $A \in \mathbb{R}^{n\times n}, 1 \leq i \leq n$, is **nonsingular** and **sparse**, the right-hand side vector $b \in \mathbb{R}^n$ is given (it may be sparse or dense), and $x \in \mathbb{R}^n$ is the required solution vector. n is the **order** (or dimension) of A and the **length** of x and b. Although we focus on real A, many of the results and algorithms we present are valid for complex A.

Entries of A are referred to using the notation

$$A = (a_{ij}), \quad 1 \le i, j \le n.$$

An entry whose value is not zero (or is treated as not being equal to zero) is called a **nonzero**. Column j of A is denoted by $A_{1:n,j}$ (or $A_{:,j}$) and row i by $A_{i,1:n}$ (or $A_{i,:}$). $A_{i:j,k:l}$ denotes the $(j - i + 1) \times (l - k + 1)$ submatrix of A comprising rows i to j, columns k to l. A is **diagonal** if for all $i \neq j$, $a_{ij} = 0$; it is **lower triangular** if for all $i < j$, $a_{ij} = 0$; it is **upper triangular** if for all $i > j$, $a_{ij} = 0$. A is **unit triangular** if it is triangular and all the entries on the diagonal are equal to unity.

The matrix A is **structurally symmetric** if for all i and j for which a_{ij} is nonzero the entry a_{ji} is also nonzero. A is **symmetric** if

$$a_{ij} = a_{ji}, \quad \text{for all } i, j.$$

Otherwise, A is **nonsymmetric**. The **symmetry index** $s(A)$ of A is defined to be the number of nonzeros $a_{ij}, i \neq j$, for which a_{ji} is also nonzero divided by the total number of off-diagonal nonzeros. Small values of $s(A)$ indicate the matrix is far from symmetric, while values close to unity indicate an almost symmetric pattern. A is **symmetric positive definite** (SPD) if it is symmetric and satisfies

$$v^T A v > 0 \text{ for all nonzero } v \in \mathbb{R}^n.$$

Otherwise, A is **symmetric indefinite**. An important class of symmetric indefinite matrices are **saddle point matrices** of the form

$$A = \begin{pmatrix} G & R^T \\ R & B \end{pmatrix},$$

where $G \in \mathbb{R}^{n_1 \times n_1}$, $B \in \mathbb{R}^{n_2 \times n_2}$, $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 + n_2 = n$, G is an SPD matrix, and B is a symmetric positive semidefinite matrix (that is $v^T B v \ge 0$ for all nonzero $v \in \mathbb{R}^{n_2}$). In some applications, $B = 0$.

As we will see later, it can be useful to partition the general matrix A into blocks. We formally express the partitioning as

$$A = (A_{ib,\,jb}), \quad A_{ib,\,jb} \in \mathbb{R}^{n_i \times n_j}, \quad 1 \le ib, jb \le nb, \tag{1.2}$$

that is,

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ A_{nb,1} & A_{nb,2} & \cdots & A_{nb,nb} \end{pmatrix}.$$

We assume the square blocks $A_{jb,\,jb}$ on the diagonal are nonsingular. We say that A is **block diagonal** if $A_{ib,\,jb} = 0$ for all $ib \neq jb$. A is **block lower triangular** if $A_{1:jb-1,\,jb} = 0, 2 \leq jb \leq nb$, and it is **block upper triangular** if $A_{jb+1:nb,\,jb} = 0$, $1 \leq jb \leq nb - 1$.

Direct methods factorize the sparse matrix A into a product of other sparse matrices; what is an appropriate factorization depends on the properties of A. In this book, the focus is on the following variants of Gaussian elimination.

- For **symmetric positive definite** A, the **Cholesky factorization** $A = LL^T$, where L is a lower triangular matrix with positive diagonal entries. Observe that this can be rewritten as $A = \widehat{L}D\widehat{L}^T$, where $\widehat{L}$ is a unit lower triangular matrix and D is a diagonal matrix with positive diagonal entries. This is called the **square root-free Cholesky** factorization. If the context is clear, we will simplify the notation and use L (rather than $\widehat{L}$) for the square root-free Cholesky factor.
- For **symmetric indefinite** A, the **LDLT factorization** $A = LDL^T$, where L is a unit lower triangular matrix and D is a block diagonal matrix with blocks of size 1 or 2 on the diagonal.
- For **nonsymmetric** A, the **LU factorization** $A = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. **Gaussian elimination** is one process to put a matrix into LU form. The factorization can be rewritten as $A = LD\widehat{U}$, where $\widehat{U}$ is a unit upper triangular matrix and D is a diagonal matrix. This is called the **LDU** factorization.

As already observed, A is sparse if many of its entries are zero. Frequently, large matrices that arise in practical problems are sparse, and when solving large-scale linear systems, taking advantage of the sparsity is essential; indeed, many problems are intractable unless advantage is taken of sparsity to reduce the computational costs in terms of storage and the number of operations that must be performed. What proportion of the entries needs to be zero for the matrix to be considered as sparse is not fixed and can depend on the pattern of the entries, the operations to be performed, and the computer architecture. There have been attempts to formalize matrix sparsity more precisely. For example, a matrix of order n may be said to be sparse if it has $O(n)$ nonzeros. But here we choose not to employ a formal definition. Instead, we say that A is **sparse** if it is advantageous to exploit its zero entries. Otherwise, A is **dense**.

The **sparsity pattern** $\mathcal{S}\{A\}$ of A is the set of nonzeros, that is,

$$\mathcal{S}\{A\} = \{(i, j) \mid a_{ij} \neq 0,\ 1 \leq i, j \leq n\}.$$

The number of nonzeros in A is denoted by $nz(A)$ (or $|\mathcal{S}\{A\}|$). A is **structurally (or symbolically) singular** if there are no values of the $nz(A)$ entries of A whose row and column indices belong to $\mathcal{S}\{A\}$ for which A is nonsingular. $\mathcal{S}\{A\}$ is symmetric if for all i and j, $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$ (the values of the two entries need not be the same). If $\mathcal{S}\{A\}$ is symmetric, then A is structurally symmetric.

In some situations, sparse vectors (vectors that contain many zero entries) are considered. The sparsity pattern of a vector v of length n is given by

$$\mathcal{S}\{v\} = \{i \mid v_i \neq 0,\ 1 \leq i \leq n\},$$

and $|\mathcal{S}\{v\}|$ denotes the number of nonzeros in v. Note that here and elsewhere curly brackets {.} are used when working with sets to help distinguish sets from vectors.

We say that the matrix A is **factorizable** (or **strongly regular**) if its principal leading minors (the determinants of its principal leading submatrices) are nonzero, that is, if its LU factorization without row/column interchanges does not break down. For example, SPD matrices are factorizable. For more general A, in exact arithmetic, the following standard result holds.

Theorem 1.1 (Golub & Van Loan 1996)
If A is nonsingular, then the rows of A can be permuted so that the permuted matrix is factorizable.

The row permutations do not need to be known in advance of the factorization; rather they can be constructed as the factorization proceeds.

1.2.1 Phases of a Sparse Direct Solver

A direct method for solving the sparse system (1.1) comprises a number of distinct phases. The matrix A is factorized, and then, given the right-hand side b, the factors used to compute the solution x. There is no single direct method that performs best on all problems and all computer architectures. Instead, many different algorithms have been proposed and implemented, some focussing on special classes of problems and/or particular architectures. However, in general, most approaches split the factorization into a **symbolic phase** (also called the **analyse phase**) and a **numerical factorization phase** that computes the factors. The symbolic phase typically uses only the sparsity pattern $\mathcal{S}\{A\}$ to compute the nonzero structure of the factors of A without computing the numerical values of the nonzeros. Following the numerical factorization, the **solve phase** uses the factors to solve for a single b or for multiple right-hand sides or for a sequence of right-hand sides one-by-one.

The fill-in in the matrix factors can render a direct method infeasible. Thus the symbolic phase typically incorporates finding a permutation (ordering) of the rows and columns of A to limit fill-in. There are many different ways to look for fill-reducing orderings; this is discussed in Chapter 8. Once the permutation has been selected, the symbolic phase determines the sparsity pattern of the factors of the permuted matrix and other key properties such as the number of entries in each row and column of the factors. This is achieved using the close relationships between matrices and graphs, which we review in Chapter 2. A symbolic factorization can also be used in algorithms that construct approximate factorizations by dropping

nonzeros from A and factoring the resulting sparser matrix. These approximate factors can be employed as preconditioners for an iterative method.

Historically, the symbolic phase was much faster than the factorization phase, but considerable effort has gone into parallelizing the factorization so that the gap between the times for the two phases has narrowed. Indeed, the ordering part of the symbolic phase can dominate the total solution time. To prevent the symbolic phase from becoming a computational bottleneck, it needs to use efficient implementations of sophisticated algorithms. By setting up the data structures needed for computing and holding the factors, the symbolic factorization contributes to the efficiency of the subsequent numerical factorization in terms of time and memory. In many applications (for instance, when solving nonlinear equations), it is necessary to solve a series of problems in which the numerical values of the entries of A change but $\mathcal{S}\{A\}$ does not. In this case, the symbolic phase can generally be performed just once and its cost amortized across the numerical factorizations.

1.2.2 Comments on the Computational Environment

The von Neumann architecture—the fundamental architecture upon which nearly all digital computers have been based—involves the union of a central processing unit (CPU) and the memory, interconnected via input/output (I/O) mechanisms, as depicted in Figure 1.3. Despite being extremely simple, this sequential model remains useful, although nowadays the role of the CPU is undertaken by a mixture of powerful processors, co-processors, cores, GPUs, and so on, and current computer architectures employ complex memory hierarchies. Performing arithmetic operations on the processing units is much faster than communication-based operations. Moreover, improvements in the speed of the processing units outpace those in the memory-based hardware. Moore's law is an example of an experimentally derived observation of this kind.

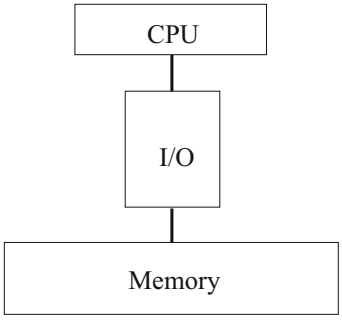


Figure 1.3 A simple uniprocessor von Neumann computer model.

Two important milestones in processor development have been **multiple functional units** that compute identical numerical operations in parallel and **data pipelining** (also called **vectorization**) that enables the efficient processing of vectors and matrices. Vectorization is often supported by additional hardware and software tools (for instance, **instruction pipelining**) and by memory components such as **registers** and by memory architectures with multiple layers, including small but fast memories called **caches**. Superscalar processors that enable the **overlapping** of identical (or different) arithmetic operations during runtime have been a standard component of computers since the 1990s. The ever-increasing heterogeneity of processing units and their hardware environment inside computers has led to significant effort being invested to support code implementations. For example, expressing the code via units of scheduling and execution called **threads**.

A key objective of many numerical linear algebra algorithms is reducing time to solution. This is usually bound by one of the following.

- Compute throughput, that is, the number of arithmetic operations that can be performed per cycle.
- Memory throughput, that is, the number of operands than can be fetched from memory/cache and/or registers each cycle.
- Latency, which is the time from initiating a compute instruction or memory request before it is completed and the result available for use in the next computation.

Depending on which of these is the constraining factor, a given algorithm is said to be compute-bound, memory-bound, or latency-bound. Latency can often be hidden by performing non-dependent operations arising from a different part of a vector or matrix while waiting for a result, and as such is most typically a constraining factor for small problems or, more rarely, in the execution of complex algorithms on less powerful processors where resource limitation (for example, the number of registers) prevents such approaches.

On modern machines, the memory throughput is normally much lower than that required to keep all functional units busy without significant reuse of operands, and this is generally true at all levels of cache. It can be useful to consider an algorithm's compute intensity, that is, the ratio of the number of operations to the number of operands read from memory. Most chips are designed such that dense matrix–matrix multiply, which typically performs n^3 operations on n^2 data (with ratio k for a blocked algorithm with block size k), can run at full compute throughput, while matrix–vector multiply performs n^2 operations on n^2 data (ratio 1) and is limited by the memory throughput. The development of basic linear algebra subroutines (**BLAS**) for performing common linear algebra operations on dense matrices was partially motivated by obtaining a high ratio. In the late 1980s, matrix–matrix operations (implemented by Level 3 BLAS) became a must once computers were able to store matrix blocks with accompanying processor instructions inside registers and fast caches. Matrix–matrix operations are able to take advantage of the fact that data that are reused within a small amount of time or are stored in close memory locations (temporal and spatial locality) are processed efficiently.

Consequently, employing Level 3 BLAS when designing and implementing matrix algorithms (for both sparse and dense matrices) can improve performance compared to using Level 1 and Level 2 BLAS.

There are other important motivations behind using the BLAS. In particular, they facilitate software development by providing standardized codes for performing common vector and matrix operations that are robust, efficient, and portable. Machine-specific optimized BLAS libraries are available for a wide variety of computer architectures, and because of the importance and widespread use of the BLAS, new implementations are provided by computer vendors as architectures change.

In this book, we discuss the design of algorithms that aim to achieve computational efficiency through exploiting data locality and using established matrix block and vector operations as fundamental building blocks. We assume an idealized computer model, not a specific architecture or language.

1.2.3 Finite Precision Arithmetic

When designing numerical algorithms, it is important to consider how the numerical operations are performed and the effects of computational errors. Finite precision arithmetic underlies all computations that are performed numerically. Historically, computer arithmetic varied greatly between different computer manufacturers, and this was a source of many problems when attempting to write software that could be easily ported between computers. Variations were reduced significantly in 1985 with the development of the Institute for Electrical and Electronic Engineering (IEEE) standard for computer floating-point arithmetic. The IEEE standard is now widely used, and the majority of contemporary computers represent real numbers using binary floating-point arithmetic that expresses real numbers as

$$a = \pm d_1.d_2 \ldots d_t \times 2^k,$$

where k is an integer and $d_i \in \{0, 1\}, 1 \leq i \leq t$, with $d_1 = 1$ unless $d_2 = d_3 = \ldots = d_t = 0$. The number of digits t is 24 in single precision and 53 in double precision. The exponent k lies in the range $-126 \leq k \leq 127$ in single precision and $-1022 \leq k \leq 1023$ in double precision. Floating-point operations can be written as

$$fl(a\ op\ b) = (a\ op\ b)(1 + \delta), \qquad |\delta| \leq \epsilon,$$

where *op* is a mathematical operation (such as $=, +, -, \times, /, \surd$) and $(a\ op\ b)$ is the exact result of the operation, and ϵ is the **machine precision** (or unit roundoff). $2 \times \epsilon$ is the smallest floating-point number that when added to the floating-point number 1.0 produces a result that is different from 1.0. For IEEE single precision arithmetic, ϵ is $2^{-24} \approx 10^{-7}$ and for double precision $\epsilon = 2^{-53} \approx 10^{-16}$. Any operation on floating-point numbers should be thought of as introducing a relative error of

absolute value at most ϵ. When the results of such operations are fed into other operations to form an algorithm, these errors propagate through the calculations. The two main sources of computational errors that are consequences of floating-point arithmetic are rounding errors and truncation errors. Certain operations can amplify the errors and lead to catastrophic failure when algorithms that are exact in conventional arithmetic are executed in floating-point arithmetic. Such algorithms are said to be **numerically unstable**; for sparse linear systems, this is discussed in Chapter 7.

1.2.4 Bit Compatibility

For sequential solvers, achieving bit compatibility (in the sense that two runs on the same machine using the same binary and identical input data should produce identical output) is not a problem. But enforcing bit compatibility can limit dynamic parallelism, and when designing parallel sparse solvers, the objective of efficiency potentially conflicts with that of bit compatibility. Bit compatibility is essential for some users because of regulatory requirements (for example, within the nuclear or financial industries) or to build trust in their software from nontechnical users (who may find the non-reproducibility of results worrying or unacceptable). For others, it is just a desirable feature for debugging purposes. Often linear solves occur at the core of much more complicated codes that typically feature heuristics that can be sensitive to very small changes in the linear solutions found.

The critical issue is the way in which N numbers (or, more generally, matrices) are assembled, that is,

$$sum = \sum_{j=1}^{N} C_j,$$

where the C_j are computed using one or more processors. The assembly is commutative but, because of the potential rounding of the intermediate results, is not associative so that the result sum depends on the order in which the C_j are assembled. A straightforward approach to achieving bit compatibility is to enforce a defined order on each assembly operation, independent of the number of processors, but this may adversely limit the scope for parallelism.

1.2.5 Complexity of Algorithms

The computational complexity of a numerical algorithm is typically based on estimating asymptotically the number of integer or floating-point operations or the memory usage. Computational complexity is expressed as a function of the algorithm's input parameters (typically the problem size) and is concerned with

how fast that function grows. Only the highest order terms are considered: scalar factors and lower order terms are ignored. For simplicity, consider a single input parameter. A real function $y(d)$ of a nonnegative real d satisfies $y = O(g)$ if there exist positive constants c and d_0 such that

$$|y(d)| \leq cg(d) \text{ for all } d \geq d_0.$$

$O(g)$ bounds y asymptotically from above. As a simple illustration, consider the quadratic function in d

$$y(d) = \alpha d^2 + \beta d - \gamma, \qquad \alpha \neq 0.$$

In this case, $y(d) = O(d^2)$, and the coefficient of the highest asymptotic term is α. In some cases, a function can also be asymptotically bounded from below. However, we will only use the $O(.)$ notation because it is more important for sparse matrix algorithms to specify upper bounds than to discuss special cases that may imply lower bounds.

Computational complexity can estimate quantities related to the worst-case behaviour of an algorithm or its average behaviour. When considering complexity based on operation counts, as a result of using a unit-cost random-access computer model, it is common to assume the operations have a unit cost. But in practice there can be a significant difference between the cost of operations, such as addition and subtraction, and operations with integer operands or operations using different precisions. Division and square root operations can be significantly more expensive than multiply/add operations; the difference is highly dependent on the computing platform. Thus, unit cost can be a significant simplification, and counting floating-point operations is arguably of limited value in assessing the performance of different algorithms on modern computers. Nevertheless, sparse matrix algorithms that are $O(n^3)$ are considered to be computationally too expensive: the goal when designing algorithms is that they should be of linear (or close to linear) in the input, that is, linear in n or $nz(A)$. Linear complexity is often achieved in the symbolic phase of a sparse direct solver, but the complexity of the numerical factorization phase is typically higher and may determine the size of the linear systems that can be solved using a sparse direct method. However, for modern computer architectures, the number of floating-point operations is not necessarily a good indicator of the time required to solve the linear system. Indeed, parallel implementations of algorithms that perform more operations than the minimum needed can lead to reductions in the runtime because costly data movements and synchronizations can be limited by, for example, duplicating operations on multiple processors.

As computers have become more powerful (in terms of both the computational speed and the available memory), the size of the linear systems that can be solved using a (parallel) dense method that ignores sparsity in A has steadily increased; nowadays linear systems with n of the order 10^5 can potentially be tackled using a dense solver (although if A is sparse, the operation count and solution time will generally be greatly reduced by using algorithms that limit operations on zeros).

Many practical applications lead to systems where A is sparse and n is significantly larger than this. The size of systems that can be solved using a sparse direct method has also steadily increased over the years, and the algorithms they use have become ever more sophisticated so that it is commonplace to solve systems of order greater than 10^7. But the complexity does limit the problem size, and for very large systems, an iterative solver is often the only option.

In computer science, complexity theory introduces additional concepts and distinguishes between problems for which algorithms of polynomial complexity exist and those where a hypothesis is that only algorithms of super polynomial complexity exist. Without going into detail, we refer to problems in this latter class as being **combinatorially hard**.

1.3 Sparse Matrices and Their Representation in a Computer

To implement sparse matrix algorithms on a computer requires special **data structures** and **storage schemes** that allow matrices and vectors to be stored, retrieved, manipulated, and updated. There are many ways to do this; key to them all is that they must be compact and avoid storing and manipulating numerically zero entries.

1.3.1 Sparse Vector Storage

A sparse vector can be stored using a real array for the nonzero values together with an integer array containing the indices of these entries, as demonstrated by the following example.

Example 1.1 Let v be the sparse row vector

$$v = \begin{pmatrix} 1. & -2. & 0. & -3. & 0. & 5. & 3. & 0. \end{pmatrix}. \tag{1.3}$$

The real array `valV` that stores the nonzero values and corresponding integer array of their indices `indV` is of length $|\mathcal{S}\{v\}| = 5$ and is as follows:

Subscripts	1	2	3	4	5
`valV`	1.	−2.	−3.	5.	3.
`indV`	1	2	4	6	7

Alternatively, a **linked list** can be used. While modern programming languages often support linked lists directly as an abstract data structure, in sparse matrix algorithms it is usual to implement them explicitly using arrays together with an integer that points to the first entry (the header pointer). Each entry is associated

with a link that points to the next entry or is null if the entry is the last in the list. The links can be adjusted so that the values are scanned in a different order without moving the physical locations. Storing the vector (1.3) as a linked list is illustrated in Example 1.2. Here v is stored in two different ways, emphasizing that the order of the entries is determined by the links, not by the physical locations of the entries.

Example 1.2 Two possible ways of storing the sparse vector (1.3) using linked lists.

Subscripts	1	2	3	4	5
Values	1.	−2.	−3.	5.	3.
Indices	1	2	4	6	7
Links	2	3	4	5	0
Header	1				

Subscripts	1	2	3	4	5
Values	5.	3.	1.	−2.	−3.
Indices	6	7	1	2	4
Links	2	0	4	5	1
Header	3				

There are two important reasons for using linked lists. Firstly, it is straightforward to add extra entries, and secondly, entries can be removed without any data movement. This is illustrated in Example 1.3. Linked lists are an example of a **dynamic** structure.

Example 1.3 On the left, an entry −4 has been added to the sparse vector (1.3) in position 5, and, on the right, the entry −2 in position 2 has been removed. $*$ indicates the entry is not accessed. The links that have changed are in bold.

Subscripts	1	2	3	4	5	6
Values	1.	−2.	−3.	5.	3.	−4.
Indices	1	2	4	6	7	5
Links	2	3	4	5	**6**	**0**
Header	1					

Subscripts	1	2	3	4	5
Values	1.	$*$	−3.	5.	3.
Indices	1	$*$	4	6	7
Links	**3**	$*$	4	5	0
Header	1				

1.3.2 Sparse Matrix Storage

The vector data structures can be generalized to sparse matrices. The simplest way to store a sparse matrix is using **coordinate** (or **triplet**) format. The individual entries of A are held as triplets (i, j, a_{ij}), where i is the row index and j is the column index of the entry $a_{ij} \neq 0$. Three arrays (one real and two integer) each of length $nz(A)$ are needed. Although this form is easy to create, it is not efficient for manipulating sparse matrices (for example, just adding two sparse matrices with different sparsity structures presents difficulties).

The **CSR (Compressed Sparse Row)** format is widely used. The column indices of the entries of A are held by rows in an integer array (which we will call `colindA`) of length $nz(A)$, with those in row 1 followed by those in row 2, and so on (with no space between rows). Often, within each row, the entries are held by

increasing column index. A real array `valA` of the same length holds the values of the corresponding entries of A in the same order. A third array `rowptrA` of length $n+1$ is such that its i-th entry points to the position of the start of row i ($1 \le i \le n$) of A within `colindA` and `valA`, and `rowptrA`$(n+1)$ is set to $nz(A)+1$.

CSC (Compressed Sparse Columns) format is defined analogously by holding the entries by columns, rather than by rows. If A is symmetric, only the lower (or upper) triangular part is generally stored. If the matrix values are not stored, the arrays `rowptrA` and `colindA` represent the graph $\mathcal{G}(A)$, which we discuss in the next chapter.

Example 1.4 Let A be the sparse matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 3. & & & -2. & \\ & 1. & & & 4. \\ -1. & & 3. & & 1. \\ & & & 1. & \\ & 7. & & & 6. \end{pmatrix} \end{array}. \tag{1.4}$$

Coordinate format represents A as follows. Note that the entries are in no particular order.

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowindA`	3	2	3	4	1	1	2	5	3	5
`colindA`	3	2	1	4	4	1	5	5	5	2
`valA`	3.	1.	−1.	1.	−2.	3.	4.	6.	1.	7.

CSR format represents A as follows. Here the entries within each row are in order of increasing column index. This additional condition is often but not always used.

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowptrA`	1	3	5	8	9	11				
`colindA`	1	4	2	5	1	3	5	4	2	5
`valA`	3.	−2.	1.	4.	−1.	3.	1.	1.	7.	6.

The CSR and CSC formats are **static** data structures. While reading A is straightforward, it can be difficult to make modifications, for instance, adding a new entry at a specified location. Removing an entry is also problematic. The value of the entry could be set to zero, but if a significant number of entries are set to zero, this may not be efficient because, when A is used, operations are performed on zeros and more memory than is necessary is used. Adding and deleting entries are possible if the sparse rows or columns are stored using linked lists.

Example 1.5 The matrix in (1.4) can be held as a collection of columns, each in a linked list, as follows. Here the array `colA_head` holds header pointers, with the i-th entry pointing to the location of the first entry in column i.

Subscripts	1	2	3	4	5	6	7	8	9	10
rowindA	3	2	3	4	1	1	2	5	3	5
valA	3.	1.	−1.	1.	−2.	3.	4.	6.	1.	7.
link	0	10	0	0	4	3	9	0	8	0
colA_head	6	2	1	5	7					

For column 4, colA_head(4) = 5, rowindA(5) = 1 and valA(5) = −2, so the first entry in column 4 is $a_{14} = -2$. Next, link(5) = 4, rowindA(4) = 4, and valA(4) = 1, so the second entry in column 4 is $a_{44} = 1$. Because link(4) = 0, there are no more entries in the column. If we want to add an entry to the (3, 4) position while retaining the order of the entries within column 4, then we do this by setting valA(11) to hold the new entry, and rowindA(11) = 3, link(5) = 11, and link(11) = 4 (the original value of link(5)). The resulting link array is shown below, with the entries that have changed given in bold.

Subscripts	1	2	3	4	5	6	7	8	9	10	11
link	0	10	0	0	**11**	3	9	0	8	0	**4**

A disadvantage of linked list storage is that it prohibits the fast access to rows (or columns) of the matrix that is needed for efficient processing on contemporary computers that use vectorization and/or work with matrix blocks. Consequently, CSR or CSC formats are commonly used in sparse direct methods.

Static data structures are efficient for sparse matrix factorizations if the sparsity structures of the factors are known before the factorization begins. However, it is often the case that new nonzero entries need to be added and/or others need to be removed, and it is not necessarily possible to predict the required space in advance. A storage scheme that has some space to embed new nonzeros is the **DS (Dynamic Sparse)** format. It stores the nonzeros of both the rows and columns of A in real arrays valAR and valAC, with the corresponding row and column indices held in integer arrays rowindA and colindA. Pointers to the start of each row and column are stored in the integer arrays rowptrA and colptrA, as in the CSR and CSC formats. In addition, the lengths of the compressed rows and columns (which are called row and column segments) are stored separately. In some situations, it can be sufficient to hold only the row (or the column) information (DSR and DSC formats). The following example illustrates the DS format.

Example 1.6 Consider again the matrix given by (1.4). The DS format represents A using two sets of arrays. The first four store the matrix by rows, and the second four store it by columns. The entries are in no particular order in both sets of arrays. The arrays rlength and clength hold the numbers of entries in the rows and columns, respectively. Free space between segments can be used to store new nonzero entries, and it is this that makes the storage scheme efficient, provided the number of changes to the matrix structure during the factorization is limited.

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`rowptrA`	1	5	8	12	14										
`colindA`	1	4			2	5		1	3	5		4		2	5
`valAR`	3.	−2.			1.	4.		−1.	3.	1.		1.		7.	6.
`rlength`	2	2	3	1	2										
`colptrA`	1	4	6	9	12										
`rowindA`	1	3		2	5	3			1	4		2	3	5	
`valAC`	3.	−1.		1.	7.	3.			−2.	1.		4.	1.	6.	
`clength`	2	2	1	2	3										

Blocked formats may be used to accelerate multiplication between a sparse matrix and a dense vector. Iterative methods typically require that the same sparse matrix is multiplied by vectors many times before a solution is found. The matrix can be put into a block storage format once, and then the cost of finding the blocks and converting the matrix format can be offset by the savings that result from repeatedly multiplying the matrix. The **Variable Block Row (VBR)** format groups together similar adjacent rows and columns. The numbers of such rows and columns can be different in each dimension, resulting in variable sized blocks. For a large sparse block-structured matrix, using a VBR format potentially reduces the amount of integer storage, and the block representation enables numerical algorithms to perform the kernel matrix operations more efficiently on the block entries. However, only heuristic algorithms are available for determining the groupings of the rows and columns.

The data structure of the VBR format uses six arrays. Integer arrays `rptr` and `cptr` hold the index of the first row in each block row and the index of the first column in each block column, respectively. In many cases, the block row and column partitionings are conformal, and only one of these arrays is needed. The real array `valA` contains the entries of the matrix block-by-block in column-major order. The integer array `indx` holds pointers to the beginning of each block entry within `valA`. The index array `bindx` holds the block column indices of the block entries of the matrix, and finally, the integer array `bptr` holds pointers to the start of each row block in `bindx`.

Example 1.7 Let A be the sparse matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} 1. & 2. & & & & 3. & & \\ 4. & 5. & & & & 6. & & \\ & & 7. & 8. & 9. & 10. & & \\ 11. & 12. & & & & & 15. & 16. \\ & 13. & & & & & 17. & \\ 14. & & & & & & & 18. \\ & & 19. & & 20. & & & \\ & & 21. & 22. & & & & \end{pmatrix} \end{array}.$$

Here the row blocks comprise rows 1:2, 3, 4:6, and 7:8. The column blocks comprise columns 1:2, 3:5, 6, 7:8. The VBR format stores A as follows.

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
`rptr`	1	3	4	7	9																	
`cptr`	1	3	6	7	9																	
`valA`	1.	4.	2.	5.	3.	6.	7.	8.	9.	10.	11.	14.	12.	13.	15.	17.	16.	18.	19.	21.	22.	20.
`indx`	1	5	7	10	11	15	19															
`bindx`	1	3	2	3	1	4	2															
`bptr`	1	3	5	7																		

1.4 Notes and References

There are some excellent textbooks that provide in-depth coverage of numerical linear algebra for dense matrices (such as Golub & Van Loan, 1996; Demmel, 1997; Trefethen & Bau, 1997, and Strang, 2007). Although sparse direct methods have been a constant subject for research since the 1960s and despite their importance and widespread use, there has only ever been a handful of books focusing on them. The most recent are Davis (2006) and Duff et al. (2017), but see also Tewarson (1973), George & Liu (1981), Pissanetzky (1984), and Zlatev (1991). In addition, Meurant (1999) covers both direct and iterative methods. The books by Björck (1996, 2015) and Wendland (2017) are also relevant.

We focus on factorizations based on Gaussian elimination, but another important class of direct methods are those based on orthogonal factorizations, most notably QR factorizations of the form $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. These methods are generally more expensive than those that use LU factorizations (in terms of operation counts, the density of the factors, and the time required to solve the linear system), but they can offer advantages in terms of numerical stability. We refer the reader to the book by Davis (2006) for a study of such approaches.

Over the last fifty years, in addition to the huge quantity of journal articles relating to specific aspects of sparse direct methods, a number of useful survey and overview papers have been published. These not only summarize important aspects of sparse direct methods but provide interesting historical perspectives on the theoretical, algorithmic, and software developments in the field. Early surveys include Tewarson (1970), Reid (1974), Duff (1977, 1981), while the comprehensive survey of Demmel et al. (1993) sums up early developments in parallel sparse direct solvers. Gould et al. (2007) look specifically at software that implements sparse direct methods, while the excellent survey of Davis et al. (2016) includes many further references to review papers and early conference proceedings where some of the key ideas related to sparse direct methods were first introduced. A short overview of modern sparse elimination methods is given by Bollhöfer et al. (2020).

A wide range of books devoted to iterative methods for solving large-scale linear systems have been written, for example, Axelsson (1994), Greenbaum (1997), Saad (2003b), van der Vorst (2003), Olshanskii & Tyrtyshnikov (2014), Meurant & Duintjer Tebbens (2020), Bai & Pan (2021), and Ciaramella & Gander (2022).

There are many references to contemporary computational environments. To understand the basic principles and connection of computations with basic linear algebra subroutines (BLAS), a good starting point is Dongarra et al. (1998), while contributions in van der Vorst & Van Dooren (2015) provide a general resource on parallel computation in numerical linear algebra. Specific features of finite precision arithmetic in this field are clearly and thoroughly explained in Higham (2002). For the complexity of algorithms as well as for much of the terminology related to the sparse data structures used in this book, we refer to Tarjan (1983); we also recommend Cormen et al. (2009) or Skiena (2020).

Texts providing details of the storage formats that are primarily for sparse direct methods include Pissanetzky (1984), Østerby & Zlatev (1983) (this discusses, in particular, dynamic data structures; see also the technical report of Duff, 1980). Storage schemes used in connection to preconditioned iterative methods are considered in Saad (2003b). VBR and other sparse storage formats are described, for example, in the SPARSKIT library documentation of Saad (1994b). Buluç et al. (2011) provide a good review and evaluation of storage formats for sparse matrices and their impact on primitive operations.

Chapter 2
Sparse Matrices and Their Graphs

The choice of data structure is one of the most important steps in algorithm design and implementation. Sparse matrix algorithms are no exception. The representation of a sparse matrix not only determines the efficiency of the algorithm, but also influences the algorithm design process –Buluç et al. (2011).

Every sparse matrix problem is a graph problem and every graph problem is a sparse matrix problem –Gilbert et al. (2006).

Many sparse matrix algorithms exploit the close relationship between matrices and graphs. We make no assumption regarding the reader's prior knowledge of graph theory. The purpose of this chapter is to summarize basic concepts from graph theory that will be exploited later and to establish the notation and terminology that will be used throughout.

2.1 Introduction to Graphs

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a finite set $\mathcal{V}$ of **vertices** (or **nodes**), and a set $\mathcal{E}$ of **edges** defined as pairs of distinct vertices. When there is no distinction between the pairs of vertices (u, v) and (v, u), the edges are represented by unordered pairs, and the graph is **undirected**. If, however, the pairs are ordered, the graph is a **directed graph**, or a **digraph**. Examples of simple graphs are given in Figures 2.1 and 2.2.

A labelling (or ordering) of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n vertices is a bijection of $\{1, 2, \ldots, n\}$ onto $\mathcal{V}$. The integer i $(1 \leq i \leq n)$ assigned to a vertex in $\mathcal{V}$ is called the **label** (or simply the **number**) of that vertex. Our standard choice of vertices will be $\mathcal{V} = \{1, \ldots, n\}$ so that the vertices are directly identified by their labels.

$\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ is a **subgraph** of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if and only if $\mathcal{V}_s \subseteq \mathcal{V}$ and $\mathcal{E}_s \subseteq \mathcal{E}$ and $(u_s, v_s) \in \mathcal{E}_s$ implies $u_s, v_s \in \mathcal{V}_s$. The subgraph is an **induced subgraph** if $\mathcal{E}_s$ contains all the edges in $\mathcal{E}$ that have both u and v in $\mathcal{V}_s$. Two graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_2

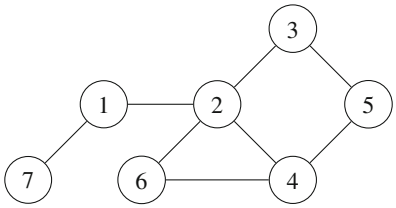


Figure 2.1 An example of an undirected graph.

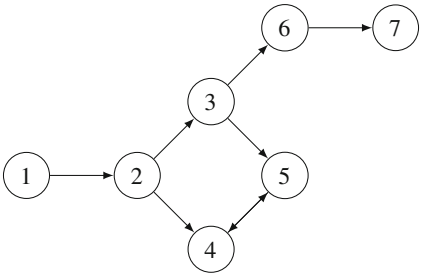


Figure 2.2 An example of a directed graph (digraph). The arrows indicate the direction of an edge. There are an edge $(4 \rightarrow 5)$ and an edge $(5 \rightarrow 4)$.

and $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ are **isomorphic** if there is a bijection $g : \mathcal{V} \rightarrow \mathcal{V}_s$ that preserves adjacency, that is, $(u, v) \in \mathcal{E}$ if and only if $(g(u), g(v)) \in \mathcal{E}_s$.

In an undirected graph, two vertices u and v in $\mathcal{V}$ are said to be **adjacent** (or **neighbours**) if $e = (u, v) \in \mathcal{E}$; the edge e is **incident** to the vertex u and to the vertex v. We also use the notation $(u \longleftrightarrow v)$ for an edge (or $(u \overset{\mathcal{G}}{\longleftrightarrow} v)$ to emphasize the edge belongs to the graph $\mathcal{G}$). The **degree** $deg_{\mathcal{G}}(u)$ of $u \in \mathcal{V}$ is the number of vertices in $\mathcal{V}$ that are adjacent to u, and the **adjacency set** $adj_{\mathcal{G}}\{u\}$ is the set of these adjacent vertices (thus $|adj_{\mathcal{G}}\{u\}| = deg_{\mathcal{G}}(u)$). If $\mathcal{V}_s$ is a subset of the vertices, then the adjacency set $adj_{\mathcal{G}}\{\mathcal{V}_s\}$ is the set of vertices in $\mathcal{V} \setminus \mathcal{V}_s$ that are adjacent to at least one vertex in $\mathcal{V}_s$. A subgraph is a **clique** when every pair of vertices is adjacent. In the example in Figure 2.1, $deg_{\mathcal{G}}(2) = 4$ and $adj_{\mathcal{G}}\{2\} = \{1, 3, 4, 6\}$. The induced subgraph with vertices $\mathcal{V}_s = \{2, 4, 6\}$ is a clique.

In a digraph, we use the notation $(u \rightarrow v)$ or $(u \overset{\mathcal{G}}{\longrightarrow} v)$ for a directed edge. There can be an edge $(u \rightarrow v)$ but no edge $(v \rightarrow u)$. The adjacency set of u can be split into two parts

$$adj^+_{\mathcal{G}}\{u\} = \{v \mid (u \rightarrow v) \in \mathcal{E}\} \quad \text{and} \quad adj^-_{\mathcal{G}}\{u\} = \{v \mid (v \rightarrow u) \in \mathcal{E}\}.$$

In the example given in Figure 2.2, $adj^+_{\mathcal{G}}\{2\} = \{3, 4\}$ and $adj^-_{\mathcal{G}}\{2\} = 1$.

2.2 Walks, Paths, Cycles, and DAGs

A sequence of k edges in an undirected graph $\mathcal{G}$

$$u_0 \longleftrightarrow u_1 \longleftrightarrow \ldots \longleftrightarrow u_{k-1} \longleftrightarrow u_k$$

is called a **walk** of length k. If $\mathcal{G}$ is a digraph, then the sequence

$$u_0 \longrightarrow u_1 \longrightarrow \ldots \longrightarrow u_{k-1} \longrightarrow u_k$$

is a **directed walk**. The vertices u_0 and u_k are connected by the walk, and for $k > 0$, u_k is said to be **reachable** from u_0; the set of vertices that are reachable from u_0 is denoted by $\mathcal{R}each(u_0)$. The walk is **closed** if $u_0 = u_k$; a closed walk is called a **cycle**. Graphs that do not contain cycles are **acyclic**. A (directed) **trail** is a (directed) walk in which all the edges are distinct and a (directed) **path** is a (directed) trail in which all the vertices (and therefore also all the edges) are distinct. The **distance** between two vertices is the number of edges in the shortest path connecting them (this is also called the **length** of the path). In Figure 2.2, there is a path of length 4 from vertex 1 to vertex 7 but no path from vertex 7 to vertex 1.

In the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a path between a pair of its vertices with labels i and j is denoted by

$$i \overset{\mathcal{G}}{\Longleftrightarrow} j$$

or, if it is clear which graph the path is in, by

$$i \Longleftrightarrow j.$$

If all intermediate vertices on the path are less than $\min\{i, j\}$, then the path is called a **fill-path** and is denoted by

$$i \underset{min}{\overset{\mathcal{G}}{\Longleftrightarrow}} j \quad \text{or} \quad i \underset{min}{\Longleftrightarrow} j.$$

If all intermediate vertices on the path belong to a subset $\mathcal{V}_s$, then the path is denoted by

$$i \underset{\mathcal{V}_s}{\overset{\mathcal{G}}{\Longleftrightarrow}} j \quad \text{or} \quad i \underset{\mathcal{V}_s}{\Longleftrightarrow} j.$$

If $\mathcal{G}$ is a digraph, the double-sided arrow symbols are replaced by one-sided ones $\Longrightarrow$ in the direction of the edges. For example,

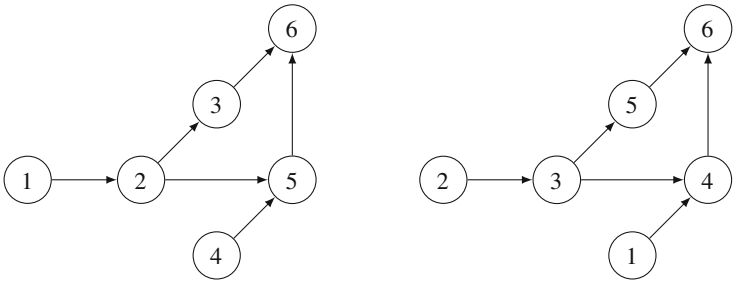


Figure 2.3 An example of a DAG with two different topological orderings (see Section 4.4).

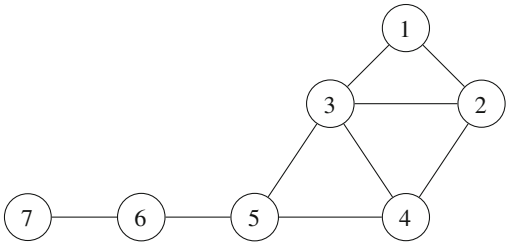


Figure 2.4 An example of an undirected graph to illustrate reachability. If $\mathcal{V}_s = \{4, 5\}$, then $\mathcal{R}each(2, \mathcal{V}_s) = \{1, 3, 6\}$ and $\mathcal{R}each(6, \mathcal{V}_s) = \{2, 3, 7\}$.

$$i \overset{\mathcal{G}}{\Longrightarrow} j, \quad i \Longrightarrow j, \quad i \underset{min}{\Longrightarrow} j \quad \text{and} \quad i \underset{\mathcal{V}_s}{\Longrightarrow} j.$$

A very important special case of a digraph is one with no cycles. A directed acyclic graph is called as **DAG**. In a DAG, if there is a path $u \Longrightarrow v$ of nonzero length, then u is called an **ancestor** of v and v is said to be a **descendant** of u. Figure 2.3 depicts a DAG with two different orderings. For the labelling of the vertices on the left, vertices 2, 3, 5, and 6 are descendants of vertex 1, but only vertices 5 and 6 are descendants of vertex 4. Note that if the direction of each edge in a DAG is reversed, the resulting graph is also a DAG.

The notion of a **reachable set** is useful for the study of Gaussian elimination. Given a graph and a subset $\mathcal{V}_s$ of its vertices, if u and v are two distinct vertices that do not belong to $\mathcal{V}_s$, then v is reachable from u through $\mathcal{V}_s$ if u and v are connected by a path that is either of length 1 or is composed entirely of vertices that belong to $\mathcal{V}_s$ (except for the endpoints u and v). Given $\mathcal{V}_s$ and $u \notin \mathcal{V}_s$, the reachable set $\mathcal{R}each(u, \mathcal{V}_s)$ is the set of all vertices that are reachable from u through $\mathcal{V}_s$. Note that if $\mathcal{V}_s$ is empty or u does not belong to $adj_{\mathcal{G}}(\mathcal{V}_s)$, then $\mathcal{R}each(u, \mathcal{V}_s) = adj_{\mathcal{G}}(u)$. A simple example is given in Figure 2.4.

2.3 Trees, Components, and Connectivity

An undirected graph is **connected** if every pair of vertices is connected by a path. A connected acyclic graph is called a **tree**, that is, a tree is an undirected graph in which any two vertices are connected by exactly one path. Every tree has at least two vertices of degree 1. Such vertices are called **leaf** vertices. A graph is a **forest** if it consists of a disjoint union of trees. This is illustrated in Figure 2.5.

If $\mathcal{G}$ is connected, then a **spanning tree** of $\mathcal{G}$ is a subgraph of $\mathcal{G}$ that is a tree containing every vertex of $\mathcal{G}$. In general, a graph may have several spanning trees, but a graph that is not connected does not contain a spanning tree.

The concept of connectivity can be extended to the general case. A digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is **strongly connected** if for every pair of vertices $u, v \in \mathcal{V}$ there is a path from u to v and a path from v to u.

An **equivalence relation** defined for a collection of pairs of members of a set is a relation that satisfies three simple properties: reflexivity, symmetry, and transitivity. A key property of an equivalence relation on a set is that it induces a partitioning of the set. Strong connectivity is an equivalence relation on $\mathcal{V}$. It induces a partitioning $\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_s$ such that each $\mathcal{V}_i$ $(1 \leq i \leq s)$ is strongly connected and is maximal with this property: no additional vertices from $\mathcal{G}$ can be included in $\mathcal{V}_i$ without breaking its strong connectivity. The $\mathcal{V}_i$ are called **strongly connected components** (or sometimes just **strong components**) of $\mathcal{G}$.

Any undirected tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ can be converted into a **directed rooted tree** $\mathcal{T}' = (\mathcal{V}, \mathcal{E}')$ by specifying a **root** vertex r. Note that r can be chosen arbitrarily: any choice gives a directed rooted tree. An edge $(u, v) \in \mathcal{E}$ becomes a directed edge $(u \to v) \in \mathcal{E}'$ if there is a path from u to r such that the first edge of this path is from u to v. Given r, this directed path is unique. We illustrate this transformation in Figure 2.6. v is called the **parent** of u if the directed edge $(u \to v) \in \mathcal{E}'$; u is said to be a **child** of v (two or more child vertices are referred to as **children**). Two vertices in a rooted tree are **siblings** if they have the same parent. Leaf vertices have no children. A rooted tree is a special case of a DAG.

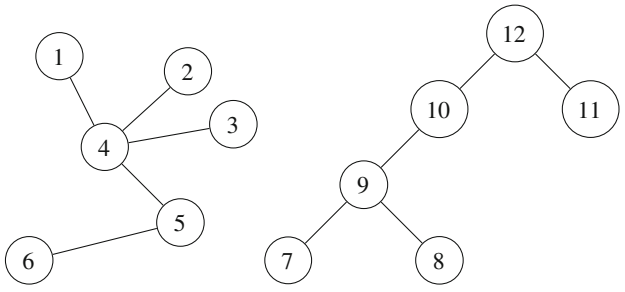


Figure 2.5 An example of an undirected graph with 12 vertices that is a forest (it consists of two disjoint trees). Vertices 1, 2, 3, 6, 7, 8, and 11 are leaf vertices.

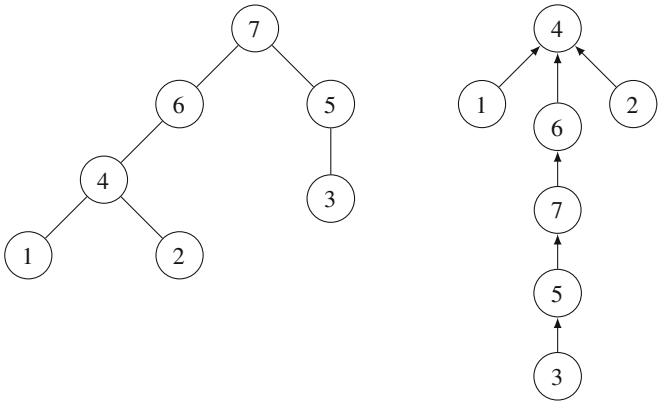


Figure 2.6 An example of an undirected tree $\mathcal{T}$ (left) and the rooted tree $\mathcal{T}'$ (right) obtained from $\mathcal{T}$ by choosing the root $r = 4$. The arrows indicate the direction of the edges.

2.4 Adjacency Graphs

Adjacency graphs provide a link between sparse matrices and graphs. If A is a sparse matrix of order n, then an **adjacency graph** $\mathcal{G}(A) = (\mathcal{V}(A), \mathcal{E}(A))$ (often written simply as $\mathcal{G}$) with n vertices $\mathcal{V}(A) = \{1, \ldots, n\}$ can be associated with it. If A is structurally symmetric, then the edge set is

$$\mathcal{E}(A) = \left\{(i, j) \mid a_{ij} \neq 0,\ i \neq j\right\}.$$

A digraph can be associated with a nonsymmetric A by setting

$$\mathcal{E}(A) = \{(i \to j) \mid a_{ij} \neq 0,\ i \neq j\}.$$

Each diagonal nonzero a_{ii} corresponds to a loop or self-edge. They are generally omitted from $\mathcal{G}$, and many algorithms that use $\mathcal{G}$ implicitly assume that the diagonal entries of A are present. Figure 2.7 depicts the sparsity patterns of two simple sparse matrices and their graphs. To capture not only the sparsity pattern of A but also the values of the entries, $\mathcal{G}$ can be transformed into a **weighted** graph using a mapping $\mathcal{E}(A) \to \mathbb{R}$ and/or $\mathcal{V}(A) \to \mathbb{R}$.

A special case is the directed graph associated with a triangular matrix. If L is a lower triangular matrix and U is an upper triangular matrix, then the directed graphs $\mathcal{G}(L)$ and $\mathcal{G}(U)$ have edge sets

$$\mathcal{E}(L) = \{(i \to j) \,|\, l_{ij} \neq 0,\ i > j\} \quad \text{and} \quad \mathcal{E}(U) = \{(i \to j) \,|\, u_{ij} \neq 0,\ i < j\}. \tag{2.1}$$

It is sometimes convenient to use $\mathcal{G}(L^T)$ in which the direction of the edges is reversed

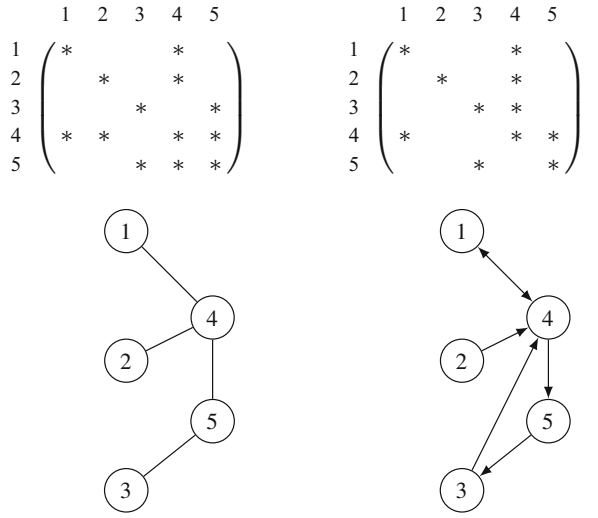


Figure 2.7 An example of a structurally symmetric sparse matrix and its undirected graph (left) and a nonsymmetric sparse matrix and its digraph (right). Arrows indicate the direction of the edges in the digraph.

$$\mathcal{E}(L^T) = \{(j \to i) \,|\, l_{ij} \neq 0,\ i > j\}. \tag{2.2}$$

It is straightforward to see that $\mathcal{G}(L)$, $\mathcal{G}(L^T)$, and $\mathcal{G}(U)$ are DAGs; they are sometimes referred to as elimination DAGs.

2.5 Matrix Permutations and Orderings

In sparse matrix algorithms, permutations are important transformations. A **permutation matrix** P is a square matrix that has exactly one entry equal to unity in each row and column, and all remaining entries are zeros (that is, it is a permutation of the identity matrix). Premultiplying a matrix by P reorders the rows and postmultiplying by P reorders the columns. P can be represented by an integer-valued **permutation vector** p, where p_i is the column index of the unity within the i-th row of P. For example,

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad p = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

The graph of a matrix A is unchanged if a symmetric permutation $A' = PAP^T$ is performed, only the labelling (that is, the ordering) of the vertices changes, and

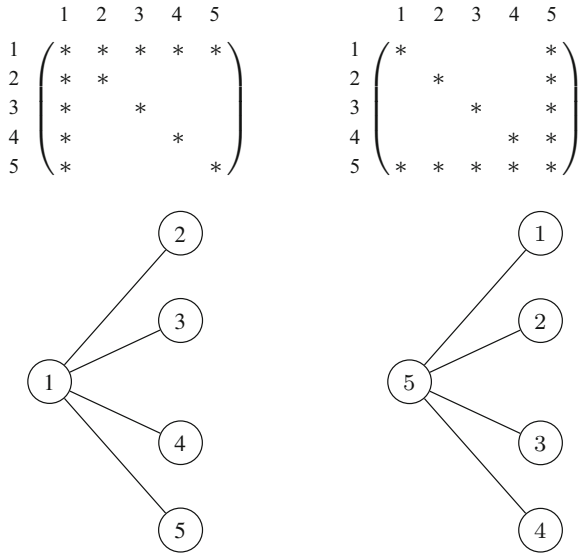


Figure 2.8 An example of an arrowhead matrix and its undirected graph (left) and a symmetrically permuted arrowhead matrix and its undirected graph (right).

thus relabelling $\mathcal{G}(A)$ can be used to permute A. This invariance property is key in sparse matrix algorithms. As an example, consider the arrowhead matrix A and its graph $\mathcal{G}(A)$ given in Figure 2.8. The symmetrically permuted matrix A' and $\mathcal{G}(A')$ are also shown, with P chosen such that the first row and column of A are the last row and column of A'.

The digraph $\mathcal{G}$ of a general matrix A is not invariant under nonsymmetric permutations PAQ, with $Q \neq P^T$. A **topological ordering** of $\mathcal{G}$ is a labelling of its vertices such that for every edge $(i \rightarrow j)$, vertex i precedes vertex j (i.e., $i < j$). It can be shown that a topological ordering is possible if and only if $\mathcal{G}$ has no directed cycles, that is, it is a DAG. Any DAG has at least one topological ordering. The non-uniqueness of topological orderings of a DAG is shown in Figure 2.3.

2.6 Lists, Stacks and Queues

Sparse matrix algorithms frequently require the storage and manipulation of lists. A **list** is an ordered sequence of arbitrary elements

$$(u_0, u_1, \ldots, u_{k-1}, u_k), \tag{2.3}$$

u_0 is the **head** of the list, and u_k is its **tail**. An empty list is denoted by ().

A **stack** is a list in which elements can only be added to or removed from the head. A pointer locates the head of the stack. Let $S = (u_0, u_1, \ldots, u_{k-1}, u_k)$ be a stack. $push(S, v)$ denotes adding v onto the stack by incrementing the pointer by one, giving $(v, u_0, \ldots u_k)$. $pop(S, u_0)$ denotes the stack $(u_1, \ldots u_k)$ that results from decreasing the pointer by one (removing u_0 from the head). A **queue** is a list in which elements can be added to the tail (appended) or removed (popped) from the head. Consider the queue $\mathcal{Q} = (u_0, u_1, \ldots, u_{k-1}, u_k)$. The append operation $append(\mathcal{Q}, u_{k+1})$ results in the queue $(u_0, \ldots u_k, u_{k+1})$, and the pop operation $pop(\mathcal{Q}, u_0)$ results in the queue $(u_1, \ldots u_k)$.

2.7 Graph Searches

Many sparse matrix reordering algorithms involve searching the adjacency graph $\mathcal{G}(A)$. The sequence in which the vertices are visited can be used, for example, to reorder the graph and hence permute the matrix. Given a start vertex, a **graph search** (also called a **graph traversal**) performs a step-by-step exploration of the vertices and edges of $\mathcal{G}(A)$, generating sets of visited vertices and explored edges. Let $\mathcal{V}_v$ be the set of visited vertices and $\mathcal{V}_n$ be the set of vertices that have not yet been visited. Following some chosen rule, the search step selects an unexplored edge such that one of its vertices belongs to $\mathcal{V}_v$. If the other vertex belongs to $\mathcal{V}_n$, then this vertex is moved into $\mathcal{V}_v$, and the edge is flagged as explored. The explored edge may be directed or undirected; in an undirected graph, the edge (u, v) formally corresponds to the pair of edges $(u \rightarrow v)$ and $(v \rightarrow u)$.

2.7.1 Breadth-First Search

Starting from a chosen start vertex s, a **breadth-first search** (BFS) explores all the vertices adjacent to s. It then explores all the vertices whose distance from s is 2, and then 3, and so on (that is, sibling vertices are visited before child vertices); a queue is used in its implementation. The search terminates when there are no unexplored edges (u, v) with $u \in \mathcal{V}_v$ and $v \in \mathcal{V}_n$ that are reachable from s. A simple example with $s = 1$ is given in Figure 2.9. All the vertices that are at the same distance from s are said to belong to the same **level** of the graph. At each level, the order in which the vertices are visited is not fixed.

2.7.2 Depth-First Search

A **depth-first search** (DFS) of a graph $\mathcal{G}$ visits child vertices before visiting sibling vertices; that is, it traverses the depth of a path before exploring its breadth. Starting

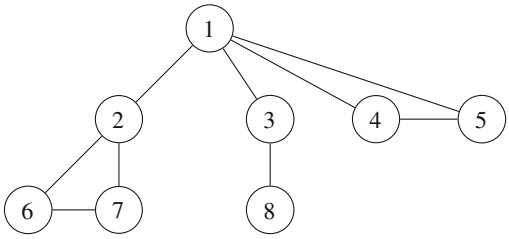


Figure 2.9 An illustration of a BFS of a connected undirected graph, with the labels indicating the order in which the vertices are visited. Vertices 2, 3, 4, 5 are all at distance 1 from s and so belong to the first level; vertices 6, 7, 8 belong to the second level.

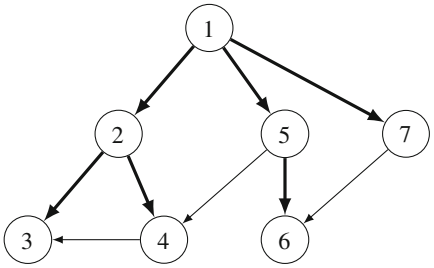


Figure 2.10 An illustration of a DFS of a connected directed graph. The labels indicate the order in which the vertices are visited. The edges of the DFS spanning tree are in bold.

from a chosen vertex s, the set of vertices that are visited are those vertices u for which a directed path from s to u exists in $\mathcal{G}$. This will give different results depending on s and how ties are broken. In the example given in Figure 2.10, the search works from left to right. Like the BFS, all vertices in $\mathcal{R}each(s)$ are visited. The edges that are traversed form a DFS spanning tree. In general, visiting all the edges of a graph results in a DFS forest that consists of exactly one DFS spanning tree for each connected component of the original graph. Thus the DFS can be used to compute connected components (see Algorithm 3.6).

There are a number of ways to construct the output vertex order for a DFS. In a **preorder** list, the vertices are returned in the order in which they are added into $\mathcal{V}_v$, while in a **postorder** list, the vertices are in the order in which they are last visited during the DFS algorithm (note that the reverse of a postordering is not the same as preordering). For the example in Figure 2.10, the vertices are added into $\mathcal{V}_v$ in the order 1, 2, 3, 4, 5, 6, 7, and this is the preorder list. The sequence in which the DFS visits the vertices is 1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 1, 7, 1. In this sequence, vertex 3 is the first vertex to appear for the last time so the postordering starts with vertex 3. The next vertex to appear for the last time is vertex 4, followed by vertex 2, and so on, resulting in the postorder list 3, 4, 2, 6, 5, 7, 1.

Algorithm 2.1 presents a DFS and outputs both the preorder and postorder lists. The call **dfs_step** is made exactly once for each vertex v. Observe that if there is a path from vertex v to vertex w in the search tree, then v is labelled ahead of w in the preorder list and w is labelled ahead of v in postorder list.

ALGORITHM 2.1 Find preorder and postorder lists using a DFS
Input: Directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.
Output: Preorder list $preorder$ and postorder list $postorder$.

```
1:  V_v = ∅, preorder = () and postorder = ()
2:  for all v ∈ V do
3:      if v ∉ V_v then
4:          push(preorder, v)                    ▷ Add v onto the preorder stack
5:          V_v = V_v ∪ {v}                      ▷ Add v to the set of visited vertices
6:          dfs_step(v)
7:      end if
8:  end for
9:  recursive function (dfs_step(v))
10:     for all (v → w) ∈ E do
11:         if w ∉ V_v then
12:             push(preorder, w)                ▷ Add w onto the preorder stack
13:             V_v = V_v ∪ {w}                  ▷ Add w to the set of visited vertices
14:             dfs_step(w)                      ▷ recursive search
15:         end if
16:     end for
17:     push(postorder, v)                       ▷ Add v onto the postorder stack
18: end recursive function
```

2.8 Notes and References

Graph theory has become an important mathematical tool in a wide variety of subjects, as well as being a mathematical discipline in its own right. There are many introductory textbooks. For example, the first four chapters of Wilson (1996) provide a basic foundation course, including definitions and examples of graphs, and the graduate-level textbook Bondy & Murty (2008) presents a coherent introduction to graph theory. The introductions to graphs given in computer science monographs such as Cormen et al. (2009) and Skiena (2020) are also ideal for our purposes.

Many papers that present sparse matrix algorithms employ graph concepts. Significant contributions include Parter (1961), Rose (1973), Rose et al. (1976), and Rose & Tarjan (1978). Important ideas first appeared in the published proceedings of some of the early conferences that focussed on sparse matrix computations, including Reid (1971), Rose & Willoughby (1972), Duff (1981), and Evans (1985). Much of the fundamental work from the 1960s and 1970s is given in the book by Tewarson (1973) and summarized later by Pissanetzky (1984). The general texts on sparse factorizations by George & Liu (1981), Davis (2006), and Duff et al. (2017) provide further sources of references and examples; see also Kepner & Gilbert (2011).

Discussions of data structures and graph searches can be found in Aho et al. (1983) and Tarjan (1983). The systematic analysis of the depth-first search algorithm

is given in Tarjan (1972), but backtracking techniques on which this search is based were used even earlier in artificial intelligence and combinatorial optimization.

Chapter 3
Introduction to Matrix Factorizations

If numerical analysts understand anything, surely it must be Gaussian elimination. This is the oldest and truest of numerical algorithms . . . This algorithm has been so successful that to many of us, Gaussian elimination and $Ax = b$ are more or less synonymous. – Trefethen (1985).

Gaussian elimination is the standard method for solving a system of linear equations. As such, it is one of the most ubiquitous numerical algorithms and plays a fundamental role in scientific computation. – Higham (2011)

This chapter introduces the basic concepts of Gaussian elimination and its formulation as a matrix factorization that can be expressed in a number of mathematically equivalent but algorithmically different ways.

Using unweighted graphs to capture the sparsity structures of matrices during Gaussian elimination is simplified by assuming that the result of adding, subtracting, or multiplying two nonzeros is nonzero. It follows that if $A = LU$ and $\mathcal{E}_L$ denotes the set of (directed) edges of the digraph $\mathcal{G}(L)$, then for $i > j$

$$a_{ij} \neq 0 \;\text{ implies }\; (i \to j) \in \mathcal{E}_L.$$

This is the **non-cancellation assumption**. It allows the following observation.

Observation 3.1 *The sparsity structures of the LU factors of A satisfy*

$$\mathcal{S}\{A\} \subseteq \mathcal{S}\{L + U\}.$$

*That is, the factors may contain entries that lie outside the sparsity structure of A. Such entries are termed **filled entries**, and together the filled entries are called the **fill-in**. The graph obtained from $\mathcal{G}(A)$ by adding the fill-in is called the **filled graph**.*

Numerical cancellations in LU factorizations rarely happen, and in general, they are difficult to predict, particularly in floating-point arithmetic. Thus, such

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_3

accidental zeros are not normally exploited in implementations, and we will ignore the possibility of their occurrence.

3.1 Gaussian Elimination: An Overview

The traditional way of describing Gaussian elimination is based on the systematic column-by-column annihilation of the entries in the lower triangular part of A. Assuming A is factorizable, this can be written formally as sequential multiplications by **column elimination matrices** that yield the **elimination sequence**

$$A = A^{(1)}, A^{(2)}, \ldots, A^{(n)} \tag{3.1}$$

of partially eliminated matrices as follows:

$$A^{(1)} \to A^{(2)} = C_1 A^{(1)} \to A^{(3)} = C_2 C_1 A^{(1)} \to \ldots \to A^{(n)} = C_{n-1} \ldots C_2 C_1 A^{(1)}.$$

The unit lower triangular matrices C_i $(1 \le i \le n-1)$ are the column elimination matrices. Elementwise, assuming $a_{11} = a_{11}^{(1)} \neq 0$, the first step $C_1 A^{(1)} = A^{(2)}$ is

$$\begin{pmatrix} 1 & & & \\ -a_{21}^{(1)}/a_{11}^{(1)} & 1 & & \\ -a_{31}^{(1)}/a_{11}^{(1)} & & 1 & \\ \vdots & & & 1 \\ -a_{n1}^{(1)}/a_{11}^{(1)} & & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \ldots & a_{2n}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & \ldots & a_{3n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \ldots & a_{nn}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ldots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & \ldots & a_{3n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \ldots & a_{nn}^{(2)} \end{pmatrix},$$

and provided $a_{22}^{(2)} \neq 0$, the second step $C_2 A^{(2)} = A^{(3)}$ is

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & -a_{32}^{(2)}/a_{22}^{(2)} & 1 & & \\ & \vdots & & 1 & \\ & -a_{n2}^{(2)}/a_{22}^{(2)} & & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ldots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & \ldots & a_{3n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \ldots & a_{nn}^{(2)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & \ldots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ldots & \ldots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \ldots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \ldots & a_{nn}^{(3)} \end{pmatrix}.$$

The k-th **partially eliminated matrix** is $A^{(k)}$. The **active entries** in $A^{(k)}$ are denoted by $a_{ij}^{(k)}$, $1 \le k \le i, j \le n$ (in the sparse case, many of the entries are zero), and the $(n-k+1) \times (n-k+1)$ submatrix of $A^{(k)}$ containing the active entries is termed its **active submatrix**. The graph associated with the active submatrix is the

k-th **elimination graph** and is denoted by $\mathcal{G}^k$. If $\mathcal{S}\{A\}$ is nonsymmetric, then $\mathcal{G}^k$ is a digraph.

The inverse of each C_k is the unit lower triangular matrix that is obtained by changing the sign of all the off-diagonal entries, and because the product of unit lower triangular matrices is a unit lower triangular matrix, it is clear that provided $a_{kk}^{(k)} \neq 0\ (1 \leq k < n)$

$$A = A^{(1)} = C_1^{-1} C_2^{-1} \ldots C_{n-1}^{-1} A^{(n)} = LU,$$

where the unit lower triangular matrix L is the product $C_1^{-1} C_2^{-1} \ldots C_{n-1}^{-1}$ and $U = A^{(n)}$ is an upper triangular matrix. The subdiagonal entries of L are the negative of the subdiagonal entries of the matrix $C_1 + C_2 + \ldots + C_{n-1}$. If A is a symmetric positive definite (SPD) matrix, then setting $U = DL^T$, the LU factorization can be written as

$$A = LDL^T,$$

which is the square root-free Cholesky factorization. Alternatively, it can be expressed as the Cholesky factorization

$$A = (LD^{1/2})(LD^{1/2})^T,$$

where the lower triangular matrix $LD^{1/2}$ has positive diagonal entries.

The process of performing an LU factorization can be rewritten in the **generic form** given in Algorithm 3.1. Here each l_{ik} is called a **multiplier**, and the $a_{kk}^{(k)}$ are called **pivots**. The assumption that A is factorizable implies $a_{kk}^{(k)} \neq 0$ for all k. Algorithm 3.1 comprises three nested loops. There are six ways of assigning the indices to the loops, with the loops having different ranges. The performance of the variants can differ significantly depending on the computer architecture. The key difference is the way the data are accessed from the factorized part of matrix and

ALGORITHM 3.1 Generic LU factorization
Input: Factorizable matrix A.
Output: LU factorization $A = LU$.

```
1: for ———— do
2:     for ———— do
3:         for ———— do
4:             l_ik = a_ik^(k) / a_kk^(k)
5:             a_ij^(k+1) = a_ij^(k) − l_ik a_kj^(k)
6:         end for
7:     end for
8: end for
```

applied to the part that is not yet factorized. But in exact arithmetic, they result in the same L and U, which allows any of them to be used to demonstrate theoretical properties of LU factorizations. To identify the variants, names that derive from the order in which the indices are assigned to the loops can be used. The kij and kji variants are called **submatrix LU factorizations**. The schemes jik and jki compute the factors by columns and are called **column factorizations**. The final two are **row factorizations** because they proceed by rows. A row factorization can be considered as a column LU factorization applied to A^T.

3.1.1 Submatrix LU Factorizations

Each outermost step of the submatrix LU variants computes one row of U and one column of L. The first step ($k = 1$) is

$$C_1 A = \begin{pmatrix} 1 & \\ -A_{2:n,1}/a_{11} & I \end{pmatrix} \begin{pmatrix} a_{11} & A_{1,2:n} \\ A_{2:n,1} & A_{2:n,2:n} \end{pmatrix} = \begin{pmatrix} a_{11} & A_{1,2:n} \\ & S \end{pmatrix},$$

where the $(n-1) \times (n-1)$ active submatrix

$$S = A_{2:n,2:n} - A_{2:n,1} A_{1,2:n}/a_{11} = A_{2:n,2:n} - L_{2:n,1} U_{1,2:n}$$

is the **Schur complement** of A with respect to a_{11}. If A is factorizable, then so too is S and the process can be repeated.

More generally, the operations performed at each step k correspond to a sequence of rank-one updates. The resulting Schur complement can be written in terms of entries of the matrices from the elimination sequence and entries of the computed factors. After $k-1$ steps ($1 < k \le n$), the $(n-k+1) \times (n-k+1)$ Schur complement of A with respect to its $(k-1) \times (k-1)$ principal leading submatrix is the active submatrix of the partially eliminated matrix $A^{(k)}$ given by

$$\begin{aligned} S^{(k)} &= \begin{pmatrix} a_{kk} & \dots & a_{kn} \\ \vdots & \ddots & \vdots \\ a_{nk} & \dots & a_{nn} \end{pmatrix} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} u_{jk} & \dots & u_{jn} \end{pmatrix} \\ &= A_{k:n,k:n} - \sum_{j=1}^{k-1} L_{k:n,j} U_{j,k:n} \\ &= \begin{pmatrix} a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ \vdots & \ddots & \vdots \\ a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix} = A_{k:n,k:n}^{(k)}. \end{aligned} \tag{3.2}$$

If A is SPD, then the Cholesky and LDLT factorizations that are special cases of the submatrix approach are termed **right-looking** (fan-out) factorizations.

3.1.2 Column LU Factorizations

In the column LU factorization, the outermost index in Algorithm 3.1 is j. For $j = 1$, $l_{11} = 1$, and the off-diagonal entries in column 1 of L are obtained by dividing the corresponding entries in column 1 of A by $u_{11} = a_{11}$. Assume $j - 1$ columns $(1 < j \leq n)$ of L and U have been computed. The partial column factorization can be expressed as

$$\begin{pmatrix} L_{1:j-1,1:j-1} \\ L_{j:n,1:j-1} \end{pmatrix} U_{1:j-1,1:j-1} = \begin{pmatrix} A_{1:j-1,1:j-1} \\ A_{j:n,1:j-1} \end{pmatrix}.$$

Column j of U and then column j of L are computed using the identities

$$U_{1:j-1,j} = L^{-1}_{1:j-1,1:j-1} A_{1:j-1,j}, \quad u_{jj} = a_{jj} - L_{j,1:j-1} U_{1:j-1,j},$$

and

$$l_{jj} = 1,\ L_{j+1:n,j} = (A_{j+1:n,j} - L_{j+1:n,1:j-1} U_{1:j-1,j})/u_{jj}.$$

Thus the strictly upper triangular part of column j of U is determined by solving the triangular system

$$L_{1:j-1,1:j-1} U_{1:j-1,j} = A_{1:j-1,j},$$

and the strictly lower triangular part of column j of L is computed as a linear combination of column $A_{j+1:n,j}$ of A and previously computed columns of L.

If A is symmetric and the pivots can be used in the order $1, 2, \ldots$ without modification, then there is the following link between its column LU and LDLT factorizations.

Observation 3.2 *The j-th diagonal entry d_{jj} $(1 \leq j \leq n)$ of the LDLT factorization of the symmetric matrix A is*

$$d_{jj} = u_{jj} = a_{jj} - \sum_{k=1}^{j-1} d_{kk} l_{jk}^2.$$

The L factor is the same as is computed by the column LU factorization; its computation can be written as

ALGORITHM 3.2 Basic column LU factorization with partial pivoting
Input: Nonsingular nonsymmetric matrix A.
Output: LU factorization $PA = LU$, where P is a row permutation matrix.

1: Interchange rows of A so that $|a_{11}| = \max\{|a_{i1}| \,|\, 1 \le i \le n\}$
2: $l_{11} = 1,\ u_{11} = a_{11},\ L_{2:n,1} = A_{2:n,1}/a_{11}$
3: **for** $j = 2 : n$ **do**
4: Solve $L_{1:j-1,1:j-1}U_{1:j-1,j} = A_{1:j-1,j}$
5: $z_{1:n-j+1} = A_{j:n,j} - L_{j:n,1:j-1}U_{1:j-1,j}$
6: Apply row interchanges to z, A and L so that $|z_1| = \max\{|z_i| \,|\, 1 \le i \le n - j + 1\}$.
7: $l_{jj} = 1$, $u_{jj} = z_1$ and $L_{j+1:n,j} = z_{2:n-j+1}/z_1$
8: **end for**

$$d_{jj}L_{j+1:n,j} = A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k}\, d_{kk}\, l_{jk}.$$

The U factor is equal to DL^T. Computing L and D in this way is called the ***left-looking*** *(fan-in) factorization.*

So far, we have assumed that A is factorizable. If A is nonsingular, then there exists a row permutation matrix P such that PA is factorizable (Theorem 1.1), and if there are zeros on the diagonal, then the rows can always be permuted to achieve a nonzero diagonal. Consider the simple 2×2 matrix A and its LU factorization

$$A = \begin{pmatrix} \delta & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \delta^{-1} & 1 \end{pmatrix} \begin{pmatrix} \delta & 1 \\ & 1 - \delta^{-1} \end{pmatrix}.$$

If $\delta = 0$, this factorization does not exist, and if δ is very small, then the entries in the factors involving δ^{-1} are very large. But interchanging the rows of A, we have

$$PA = \begin{pmatrix} 1 & 1 \\ \delta & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \delta & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ & 1 - \delta \end{pmatrix},$$

which is valid for all $\delta \neq 1$. Algorithm 3.2 presents a basic column LU factorization scheme for nonsingular A. The interchanging of rows at each elimination step to select the entry of largest absolute value in its column as the next pivot is called **partial pivoting**. It avoids small pivots and results in an LU factorization of a row permuted matrix PA in which the absolute value of each entry of L is at most 1. In practice, partial pivoting (or another pivoting strategy) is incorporated into all LU factorization variants. Pivoting strategies are discussed in Chapter 7.

3.1.3 *Factorizations by Bordering*

The generic LU factorization scheme does not cover all possible approaches. An alternative is **factorization by bordering**. Set all diagonal entries of L to 1, and assume the first $k-1$ rows of L and first $k-1$ columns of U ($1 < k \leq n$) have been computed (that is, $L_{1:k-1,1:k-1}$ and $U_{1:k-1,1:k-1}$). At step k, the factors must satisfy

$$A_{1:k,1:k} = \begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix}.$$

Equating terms, the lower triangular part of row k of L and the upper triangular part of column k of U are obtained by solving

$$L_{k,1:k-1}U_{1:k-1,1:k-1} = A_{k,1:k-1},$$
$$L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}.$$

The diagonal entry u_{kk} is then given by

$$u_{kk} = a_{kk} - L_{k,1:k-1}U_{1:k-1,k} \quad (\text{with } u_{11} = a_{11}).$$

3.2 Fill-in in Sparse Gaussian Elimination

Here we give some simple results that describe fill-in in the matrix factors; strategies to limit fill-in will be presented in Chapter 8. We start by looking at the rules that establish the positions of the entries in the factors. Assume $\mathcal{S}\{A\}$ is symmetric, and consider the elimination graph $\mathcal{G}^k$ at step k. Its vertices are the $n-k+1$ uneliminated vertices. Its edge set contains the edges in $\mathcal{G}(A)$ connecting these vertices and additional edges corresponding to filled entries produced during the first $k-1$ elimination steps. The sequence of graphs $\mathcal{G}^1 \equiv \mathcal{G}(A), \mathcal{G}^2, \ldots$ is generated recursively using **Parter's rule**:

To obtain the elimination graph $\mathcal{G}^{k+1}$ from $\mathcal{G}^k$, delete vertex k and add all possible edges between vertices that are adjacent to vertex k in $\mathcal{G}^k$.

Denoting $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{E}^k)$ and $\mathcal{G}^{k+1} = (\mathcal{V}^{k+1}, \mathcal{E}^{k+1})$, this can be written as

$$\mathcal{V}^{k+1} = \mathcal{V}^k \setminus \{k\}, \ \mathcal{E}^{k+1} = \mathcal{E}^k \cup \{(i,j) \,|\, i,j \in adj_{\mathcal{G}^k}\{k\}\} \setminus \{(i,k) \,|\, i \in adj_{\mathcal{G}^k}\{k\}\}.$$

If $\mathcal{S}\{A\}$ is nonsymmetric, then the elimination graphs are digraphs and Parter's rule generalizes as follows:

To obtain the elimination graph $\mathcal{G}^{k+1}$ from $\mathcal{G}^k$, delete vertex k and add all edges $(i \xrightarrow{\mathcal{G}^{k+1}} j)$ such that $(i \xrightarrow{\mathcal{G}^k} k)$ and $(k \xrightarrow{\mathcal{G}^k} j)$.

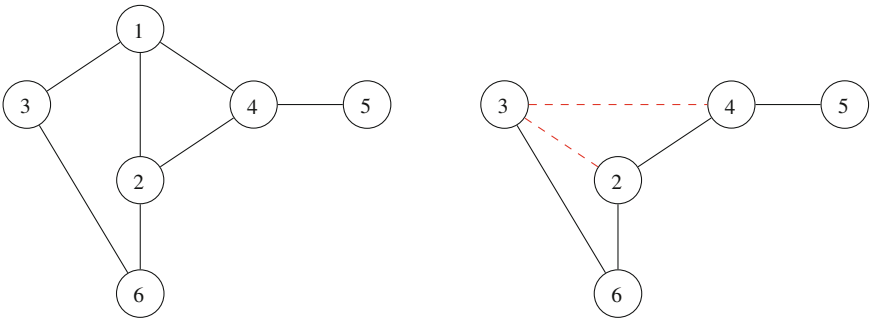


Figure 3.1 Illustration of Parter's rule. The original undirected graph $\mathcal{G} = \mathcal{G}^1$ and the elimination graph $\mathcal{G}^2$ that results from eliminating vertex 1 are shown on the left and right, respectively. The red dashed lines denote fill edges. The vertices $\{2, 3, 4\}$ become a clique.

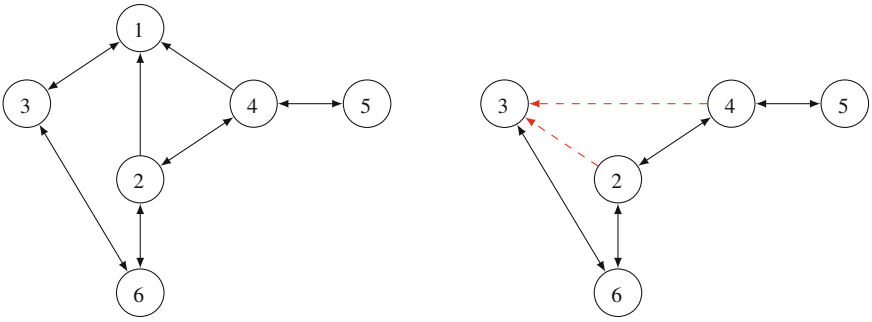


Figure 3.2 Illustration of Parter's rule for a nonsymmetric $\mathcal{S}\{A\}$. The original digraph $\mathcal{G} = \mathcal{G}^1$ and the directed elimination graph $\mathcal{G}^2$ that results from eliminating vertex 1 are shown on the left and right, respectively. The red dashed lines denote fill edges.

Simple examples are given in Figures 3.1 and 3.2.

In terms of graph theory, if $\mathcal{S}\{A\}$ is symmetric, then Parter's rule says that the adjacency set of vertex k becomes a clique when k is eliminated. Thus, Gaussian elimination systematically generates cliques. As the elimination process progresses, cliques grow or more than one clique join to form larger cliques, a process known as **clique amalgamation**. A clique with m vertices has $m(m-1)/2$ edges, but it can be represented by storing a list of its vertices, without any reference to edges. This enables important savings in both storage and data movement to be achieved during the symbolic phase of a direct solver.

The repeated application of Parter's rule specifies all the edges in $\mathcal{G}(L + L^T)$:

(i, j) is an edge of $\mathcal{G}(L + L^T)$ if and only if (i, j) is an edge of $\mathcal{G}(A)$ or (i, k) and (k, j) are edges of $\mathcal{G}(L + L^T)$ for some $k < i, j$.

This generalizes to a nonsymmetric matrix A and its LU factorization:

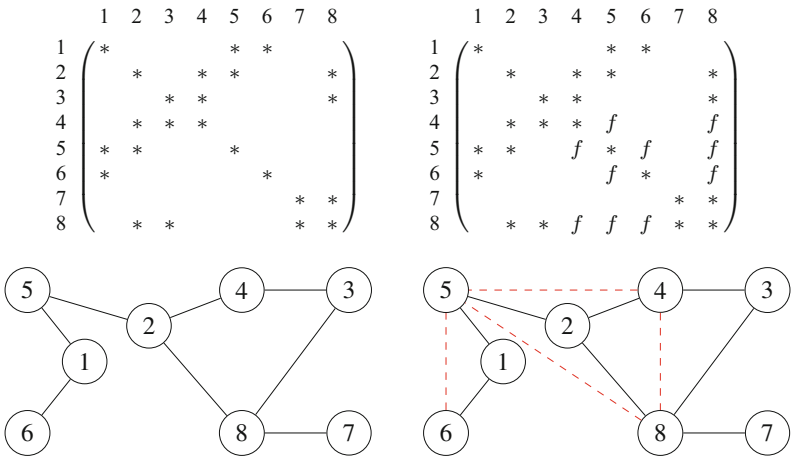


Figure 3.3 Example to illustrate fill-in during the factorization of a symmetric matrix, with the eliminations performed in the natural order. $\mathcal{S}\{A\}$ and $\mathcal{S}\{L + L^T\}$ are on the left and right, respectively, with the corresponding undirected graphs $\mathcal{G}(A)$ and $\mathcal{G}(L + L^T)$. Filled entries in $L + L^T$ are denoted by f. The red dashed lines in the filled graph $\mathcal{G}(L + L^T)$ correspond to filled entries.

$(i \rightarrow j)$ is an edge of the digraph $\mathcal{G}(L+U)$ if and only if $(i \rightarrow j)$ is an edge of the digraph $\mathcal{G}(A)$ or $(i \rightarrow k)$ and $(k \rightarrow j)$ are edges of $\mathcal{G}(L + U)$ for some $k < i, j$.

Parter's rule is a local rule that uses the dependency on nonzeros obtained in previous steps of the factorization. The following result, which uses the path notation of Section 2.2, fully characterizes the nonzero entries in the factors using only paths in $\mathcal{G}(A)$.

Theorem 3.1 (Rose et al. 1976; Rose & Tarjan 1978)

(a) Let $\mathcal{S}\{A\}$ be symmetric and $A = LL^T$. Then $(L + L^T)_{ij} \neq 0$ if and only if there is a fill-path $i \overset{\mathcal{G}(A)}{\underset{min}{\Longleftrightarrow}} j$.

(b) Let $\mathcal{S}\{A\}$ be nonsymmetric and $A = LU$. Then $(L + U)_{ij} \neq 0$ if and only if there is a fill-path $i \overset{\mathcal{G}(A)}{\underset{min}{\Longrightarrow}} j$.

The fill-paths may not be unique.

Figure 3.3 illustrates Theorem 3.1 for symmetric $\mathcal{S}\{A\}$. There is a filled entry in position $(8, 6)$ of L because there is a fill-path $8 \overset{\mathcal{G}(A)}{\underset{min}{\Longleftrightarrow}} 6$ given by the sequence of (undirected) edges $8 \longleftrightarrow 2 \longleftrightarrow 5 \longleftrightarrow 1 \longleftrightarrow 6$.

Corollary 3.2 characterizes edges of $\mathcal{G}^k$ in terms of reachable sets in the original graph $\mathcal{G}(A)$.

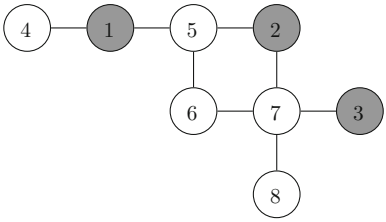


Figure 3.4 An example to illustrate reachable sets in $\mathcal{G}(A)$. The grey vertices 1, 2, and 3 are eliminated in the first three elimination steps ($\mathcal{V}^4 = \{1, 2, 3\}$).

Corollary 3.2 (Rose et al., 1976; George & Liu, 1980b)
Assume $\mathcal{S}\{A\}$ is symmetric. Let $\mathcal{V}^k$ be the set of $k-1$ vertices of $\mathcal{G}(A)$ that have already been eliminated, and let v be a vertex in the elimination graph $\mathcal{G}^k$. Then the set of vertices adjacent to v in $\mathcal{G}^k$ is the set $\mathcal{R}each(v, \mathcal{V}^k)$ of vertices reachable from v through $\mathcal{V}^k$ in $\mathcal{G}(A)$.

Proof The proof is by induction on k. The result holds trivially for $k = 1$ because $\mathcal{R}each(v, \mathcal{V}^1) = adj_{\mathcal{G}(A)}\{v\}$. Assume the result holds for $\mathcal{G}^1, \ldots, \mathcal{G}^k$ with $k \geq 1$, and let v be a vertex in the graph $\mathcal{G}^{k+1}$ that is obtained after eliminating v_k from $\mathcal{G}^k$. If v is not adjacent to v_k in $\mathcal{G}^k$, then $\mathcal{R}each(v, \mathcal{V}^{k+1}) = \mathcal{R}each(v, \mathcal{V}^k)$. Otherwise, if v is adjacent to v_k in $\mathcal{G}^k$, then $adj_{\mathcal{G}^{k+1}}\{v\} = \mathcal{R}each(v, \mathcal{V}^k) \cup \mathcal{R}each(v_k, \mathcal{V}^k)$. In both cases, Parter's rule implies that the new adjacency set is exactly equal to the vertices that are reachable from v through $\mathcal{V}^{k+1}$, that is, $\mathcal{R}each(v, \mathcal{V}^{k+1})$. □

Figure 3.4 depicts a graph $\mathcal{G}(A)$. The adjacency sets of the vertices in $\mathcal{G}^4$ that result from eliminating vertices $\mathcal{V}^4 = \{1, 2, 3\}$ are $adj_{\mathcal{G}^4}\{4\} = \mathcal{R}each(4, \mathcal{V}^4) = \{5\}$, $adj_{\mathcal{G}^4}\{5\} = \mathcal{R}each(5, \mathcal{V}^4) = \{4, 6, 7\}$, $adj_{\mathcal{G}^4}\{6\} = \mathcal{R}each(6, \mathcal{V}^4) = \{5, 7\}$, $adj_{\mathcal{G}^4}\{7\} = \mathcal{R}each(7, \mathcal{V}^4) = \{5, 6, 8\}$, and $adj_{\mathcal{G}^4}\{8\} = \mathcal{R}each(8, \mathcal{V}^4) = \{7\}$.

We remark that neither the local characterization of filled entries using Parter's rule nor Theorem 3.1 provides a direct answer as to whether a certain edge belongs to $\mathcal{G}(L+L^T)$ (or $\mathcal{G}(L+U)$); without performing the eliminations, they do not tell us whether a given entry of a factor of A is nonzero. Such questions are addressed by deeper theoretical and algorithmic results that are presented in subsequent chapters.

3.3 Triangular Solves

Once an LU factorization has been computed, the solution x of the linear system $Ax = b$ is computed by solving the lower triangular system

$$Ly = b, \tag{3.3}$$

followed by the upper triangular system

$$Ux = y. \tag{3.4}$$

Solving a system with a triangular matrix and dense right-hand side vector is straightforward. The solution of (3.3) can be computed using **forward substitution** in which the component y_1 is determined from the first equation, substitute it into the second equation to obtain y_2, and so on. Once y is available, the solution of (3.4) can be obtained by **back substitution** in which the last equation is used to obtain x_n, which is then substituted into equation $n-1$ to obtain x_{n-1}, and so on. Algorithm 3.3 is a simple lower triangular solve for dense b. If L is unit lower triangular, step 3 is not needed.

ALGORITHM 3.3 Forward substitution: lower triangular solve $Ly = b$ with b dense
Input: Lower triangular matrix L with nonzero diagonal entries and dense right-hand side b.
Output: The dense solution vector y.

```
1: Initialise y = b
2: for j = 1 : n do
3:     y_j = y_j / l_jj
4:     for i = j + 1 : n do
5:         if l_ij ≠ 0 then
6:             y_i = y_i − l_ij y_j
7:         end if
8:     end for
9: end for
```

When b is sparse, the solution y is also sparse. In particular, if in Algorithm 3.3 $y_k = 0$, then the outer loop with $j = k$ can be skipped. Furthermore, if $b_1 = b_2 = \ldots = b_k = 0$ and $b_{k+1} \neq 0$, then $y_1 = y_2 = \ldots = y_k = 0$. Scanning y to check for zeros adds $O(n)$ to the complexity. But if the set of indices $\mathcal{J} = \{j \mid y_j \neq 0\}$ is known beforehand, then Algorithm 3.3 can be replaced by Algorithm 3.4. A possible way to determine $\mathcal{J}$ is discussed later (Theorem 5.2).

Note that the combined effect of forward substitution (3.3) followed by back substitution (3.4) often results in the final solution vector x being dense. This is the case if $y_n \neq 0$ and U has an entry in each off-diagonal row i ($1 \leq i < n$).

3.4 Reducibility and Block Triangular Forms

The performance of algorithms for computing factorizations of sparse matrices can frequently be significantly enhanced by first permuting A to have a block form or by

ALGORITHM 3.4 Forward substitution: lower triangular solve $Ly = b$ with b sparse
Input: Lower triangular matrix L with nonzero diagonal entries, sparse vector b and the set $\mathcal{J}$ of indices j such that $y_j \neq 0$.
Output: The sparse solution vector y.

```
1: Initialise y = b
2: for j ∈ J do                          ▷ Take indices from J in increasing order
3:     y_j = y_j / l_jj
4:     for i = j + 1 : n do
5:         if l_ij ≠ 0 then
6:             y_i = y_i − l_ij y_j
7:         end if
8:     end for
9: end for
```

partitioning A into blocks. Permuting to block form is closely connected to matrix reducibility. A is said to be **reducible** if there is a permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{p_1,p_1} & A_{p_1,p_2} \\ 0 & A_{p_2,p_2} \end{pmatrix},$$

where A_{p_1,p_1} and A_{p_2,p_2} are nontrivial square matrices (that is, they are of order at least 1). If A is not reducible, it is **irreducible**. If A is structurally symmetric, then $A_{p_1,p_2} = 0$ and PAP^T is block diagonal. The following example illustrates that a one-sided permutation can transform an irreducible matrix A into a reducible matrix AQ.

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & \\ 1 & & \end{pmatrix}, \quad Q = \begin{pmatrix} & & 1 \\ & 1 & \\ 1 & & \end{pmatrix}, \quad AQ = \begin{pmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{pmatrix}.$$

A matrix A is said to be a **Hall matrix** (or has the **Hall property**) if every set of k columns has nonzeros in at least k rows ($1 \leq k \leq n$). A is a **strong Hall matrix** (or has the **strong Hall property**) if every set of k columns ($1 \leq k < n$) has nonzeros in at least $k + 1$ rows. The strong Hall property trivially implies the Hall property. The Hall property applies to rectangular $m \times n$ matrices with $m \geq n$. If A is square, then A has the strong Hall property if and only if the directed graph $\mathcal{G}(A)$ is strongly connected.

The following theorem is an important consequence of reducibility.

Theorem 3.3 (Brualdi & Ryser 1991)
Given a nonsingular nonsymmetric matrix A, there exists a permutation matrix P such that

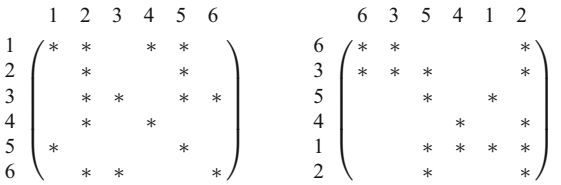


Figure 3.5 The sparsity patterns of A (left) and the upper block triangular form PAP^T with two blocks $A_{ib,ib}$, $i = 1, 2$, of orders 2 and 4 (right).

$$PAP^T = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ 0 & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nb,nb} \end{pmatrix}, \tag{3.5}$$

where the square matrices $A_{ib,ib}$ on the diagonal are irreducible. The set $\{A_{ib,ib} \mid 1 \le ib \le nb\}$ is uniquely determined (but the blocks may appear on the diagonal in a different order). The order of the rows and columns within each $A_{ib,ib}$ may not be unique.

The **upper block triangular form** (3.5) is also known as the **Frobenius normal form**. It is said to be nontrivial if $nb > 1$, and this is the case if A does not have the strong Hall property. An example of a matrix that can be symmetrically permuted to block triangular form with $nb = 2$ is given in Figure 3.5.

In practice, many of the blocks in (3.5) are either sparse or zero blocks. Assuming the blocks $A_{ib,ib}$ on the diagonal are all nonsingular, an LU factorization of each can be computed independently. These can then be used to solve the permuted system $PAP^T y = c$ as a sequence of nb smaller problems, as outlined in Algorithm 3.5. The solution of the original system $Ax = b$ follows by setting $c = Pb$ and $x = P^T y$. Because the algorithms used to transform A into a block triangular form are typically graph-based (and do not use the numerical values of the entries of A), pivoting needs to be incorporated within the factorization of the diagonal blocks. Algorithm 3.5 employs partial pivoting for this.

The transversal of a matrix A is the set of its nonzero diagonal elements. A has a **full** or **maximum transversal** if all its diagonal entries are nonzero. There exist permutation matrices P and Q such that PAQ has a full transversal matrix if and only if A has the Hall property. Moreover, if A is nonsingular, then it can be nonsymmetrically permuted to have a full transversal. However, the converse is clearly not true (for example, a matrix with all its entries equal to one has a full transversal, but it is singular). Permuting A to have a full transversal will be discussed in Section 6.3.

If A has a full transversal, then there exists a permutation matrix P_s such that $P_s A P_s^T$ has the form (3.5). In other words, once A has a full transversal, a symmetric permutation is sufficient to obtain the form (3.5). Finding P_s is identical

ALGORITHM 3.5 Solve a sparse linear system in upper block triangular form
Input: Upper block triangular matrix (3.5) and a conformally partitioned right-hand side vector c.
Output: The conformally partitioned solution vector y.

1: **for** $ib = 1 : nb$ **do** ▷ LU factorizations of the $A_{ib,ib}$ blocks can be performed in parallel
2: Compute $P_{ib}A_{ib,ib} = L_{ib}U_{ib}$ ▷ Sparse LU factorization with partial pivoting
3: **end for**
4: Solve $L_{nb}U_{nb}\, y_{nb} = P_{nb}c_{nb}$ ▷ Perform forward and back substitutions
5: **for** $ib = nb - 1 : 1$ **do**
6: **for** $jb = ib + 1 : nb$ **do**
7: $c_{ib} = c_{ib} - A_{ib,jb}y_{jb}$ ▷ Sparse matrix-vector operation (skip if $A_{ib,jb} = 0$)
8: **end for**
9: Solve $L_{ib}U_{ib}\, y_{ib} = P_{ib}c_{ib}$ ▷ Perform forward and back substitutions
10: **end for**

to finding the strongly connected components (SCCs) of the digraph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ (Section 2.3). To find the SCCs, $\mathcal{V}$ is partitioned into non-empty subsets $\mathcal{V}_i$ with each vertex belonging to exactly one subset. Each vertex i in the **quotient graph** corresponds to a subset $\mathcal{V}_i$, and there is an edge in the quotient graph with endpoints i and j if $\mathcal{E}$ contains at least one edge with one endpoint in $\mathcal{V}_i$ and the other in $\mathcal{V}_j$. The **condensation** (or component graph) of a digraph is a quotient graph in which the SCCs form the subsets of the partition, that is, each SCC is contracted to a single vertex. This reduction provides a simplified view of the connectivity between components. An example is given in Figure 3.6. It has five SCCs: $\{p, q, r\}$, $\{s, t, u\}$, $\{v\}$, $\{w\}$, and $\{x\}$.

The following result gives the relationship between SCCs and DAGs.

Theorem 3.4 (Sharir 1981; Cormen et al. 2009)
The condensation $\mathcal{G}_C$ of a digraph is a DAG (directed acyclic graph).

Because any DAG can be topologically ordered, $\mathcal{G}_C = (\mathcal{V}_C, \mathcal{E}_C)$ can be topologically ordered, and if $\mathcal{V}_i$ and $\mathcal{V}_j$ are contracted to s_i and s_j and $(s_i \longrightarrow s_j) \in \mathcal{E}_C$, then $s_i < s_j$. It follows that to permute A to block triangular form it is sufficient to find the SCCs of $\mathcal{G}(A)$. That is, topologically ordering the vertices of the condensation $\mathcal{G}_C$ induced by the SCCs is the quotient graph that implies the block triangular form. There are many ways to find SCCs, one of which is Tarjan's algorithm (Algorithm 3.6). The key idea here is that vertices of an SCC form a subtree in the DFS spanning tree of the graph. The algorithm performs depth-first searches, keeping track of two properties for each vertex v: when v was first

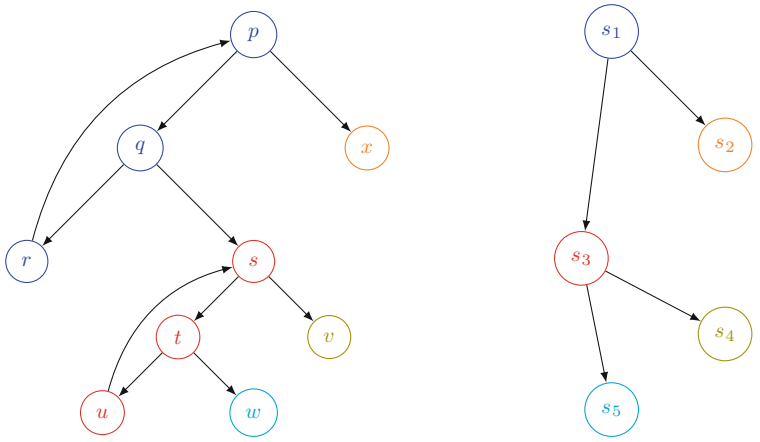


Figure 3.6 An illustration of the strong components of a digraph. On the left, the five SCCs are denoted using different colours and on the right is the condensation DAG $\mathcal{G}_C$ formed by the SCCs.

encountered (held in $invorder(v)$) and the lowest numbered vertex that is reachable from v (called the low-link value and held in $lowlink(v)$). It pushes vertices onto a stack as it goes and outputs a SCC when it finds a vertex for which $invorder(v)$ and $lowlink(v)$ are the same. The value $lowlink(v)$ is computed during the DFS from v, as this finds the vertices that are reachable from v.

In Algorithm 3.6, the variable $index$ is the DFS vertex number counter that is incremented when an unvisited vertex is visited. S is the vertex stack. It is initially empty and is used to store the history of visited vertices that are not yet committed to an SCC. Vertices are added to the stack in the order in which they are visited. The outermost loop of the algorithm visits each vertex that has not yet been visited, ensuring vertices that are not reachable from the starting vertex are eventually visited. The recursive function **scomp_step** performs a single DFS, finding all descendants of vertex v, and reporting all SCCs for that subgraph. When a vertex v finishes recursing, if $lowlink(v) = invorder(v)$, then it is the root vertex of an SCC comprising all of the vertices above it on the stack. The algorithm pops the stack up to and including v; these popped vertices form an SCC. The algorithm is linear in the number of edges and vertices, that is, it is of complexity $O(|\mathcal{V}|+|\mathcal{E}|)$.

3.5 Block Partitioning

In this section, we assume that $\mathcal{S}\{A\}$ is symmetric and $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is the adjacency graph of A.

ALGORITHM 3.6 Tarjan's algorithm to find the strongly connected components (SCCs) of a digraph
Input: Digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$.
Output: Strongly connected components of $\mathcal{G}$, determined one-by-one.

```
1: V_v = ∅, S = (), index = 0,                ▷ Each vertex is initially unvisited
2: for each v ∈ V do
3:     if v ∉ V_v then
4:         scomp_step(v)
5:     end if
6: end for
7: recursive function (scomp_step(v))
8:     V_v = V_v ∪ {v}                        ▷ Add v to the set of visited vertices
9:     index = index + 1                      ▷ Set the index for v to smallest unused index
10:    invorder(v) = index, lowlink(v) = index
11:    push(S, v)                             ▷ Put v on the stack
12:    Set v = head(S)                        ▷ v is the current head of S.
13:    for each (v → w) ∈ E do                ▷ Look in the adjacency list of v
14:        if w ∉ V_v then                    ▷ w not yet been visited; recurse on it
15:            scomp_step(w)
16:            lowlink(v) = min(lowlink(v), lowlink(w))
17:        else if w ∈ S then                 ▷ w is in the stack and hence in current SCC
18:            lowlink(v) = min(lowlink(v), invorder(w))
19:        end if
20:    end for
21:    if lowlink(v) = invorder(v) then
22:        pop all vertices down to v from S to obtain a new SCC
23:    end if
24: end recursive function
```

3.5.1 Block Structure Based on Supervariables

Sets of columns of A frequently have identical sparsity patterns. For instance, when A arises from a finite element discretization, the columns corresponding to variables that belong to the same set of finite elements have the same pattern, and this occurs as a result of each node of the finite element mesh having multiple degrees of freedom associated with it. This repetition of the sparsity patterns can be used to substantially enhance performance.

Adjacent vertices u and v in an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ are said to be **indistinguishable** if they have the same neighbours, that is, $adj_{\mathcal{G}}\{u\} \cup \{u\} =$

$adj_{\mathcal{G}}\{v\} \cup \{v\}$. A set of mutually indistinguishable vertices is called an **indistinguishable vertex set**. If $\mathcal{U} \subseteq \mathcal{V}$ is an indistinguishable vertex set, then $\mathcal{U}$ is **maximal** if $\mathcal{U} \cup \{w\}$ is not indistinguishable for any $w \in \mathcal{V} \setminus \mathcal{U}$.

Indistinguishability is an equivalence relation on $\mathcal{V}$, and maximal indistinguishable vertex sets represent its classes. This implies a partitioning of $\mathcal{V}$ into $nsup \geq 1$ non-empty disjoint subsets

$$\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_{nsup}. \tag{3.6}$$

An indistinguishable vertex set can be represented by a single vertex, called a **supervariable**.

If the vertices belonging to each subset $\mathcal{V}_1, \ldots, \mathcal{V}_{nsup}$ are numbered consecutively, with those in $\mathcal{V}_i$ preceding those in $\mathcal{V}_{i+1}$ ($1 \leq i < nsup$), and if P is the permutation matrix corresponding to this ordering, then the permuted matrix PAP^T has a block structure in which the blocks are dense (with the possible exception of the diagonal entries, which can be zero); the dimensions of the blocks are equal to the sizes of the indistinguishable sets.

One approach for identifying supervariables is outlined in Algorithm 3.7. Initially, all the vertices are placed in a single vertex set (that is, into a single supervariable). This is split into two supervariables by taking the first vertex $j = 1$ and moving vertices in the adjacency set of j into a new vertex set (a new supervariable). Each vertex j is considered in turn, and each vertex set $\mathcal{V}_{sv}$ that contains a vertex in $adj_{\mathcal{G}}\{j\} \cup j$ is split into two by moving the vertices in $adj_{\mathcal{G}}\{j\} \cup j$ that belong to $\mathcal{V}_{sv}$ into a new vertex set. Note that as a result of the splitting and moving of vertices, a vertex set can become empty, in which case it is discarded. Once the supervariables have been determined, the permuted matrix PAP^T can be condensed to a matrix of order equal to $nsup$; the corresponding graph is called the **supervariable** graph. If the average number of variables in each supervariable is k, using the supervariable graph will reduce the amount of integer data that is read during the symbolic phase by a factor of about k^2.

As an illustration, consider the following 5×5 matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & * & & & * \\ * & * & & & * \\ & & * & * & * \\ & & * & * & * \\ * & * & * & * & * \end{pmatrix} \end{array}.$$

Initially, 1, 2, 3, 4, 5 are put into a single vertex set $\mathcal{V}_1$. Consider $j = 1$. Vertices $i = 1, 2$ and 5 belong to $adj_{\mathcal{G}}\{1\} \cup \{1\}$; they are moved from $\mathcal{V}_1$ into a new vertex set. There is no further splitting of the vertex sets for $j = 2$. For $j = 3$, $adj_{\mathcal{G}}\{3\} \cup \{3\} = \{3, 4, 5\}$. Vertices $i = 3$ and 4 are moved from $\mathcal{V}_1$ into a new vertex set. $\mathcal{V}_1$ is now empty and can be discarded. Vertex $i = 5$ is moved from the vertex set that holds vertices 1 and 2 into a new vertex set. For $j = 4$ and 5, no additional splitting is performed. Thus, three supervariables are found, namely $\{1, 2\}$, $\{3, 4\}$, and $\{5\}$.

ALGORITHM 3.7 Find the supervariables of an undirected graph
Input: Graph $\mathcal{G}$ of a symmetrically structured matrix.
Output: Partitioning of $\mathcal{V}$ into indistinguishable vertex sets.

$\mathcal{V}_1 = \{1, 2, \ldots, n\}$
for $j = 1 : n$ **do**
  for $i \in adj_{\mathcal{G}}\{j\} \cup j$ **do**
    Find sv such that $i \in \mathcal{V}_{sv}$
    if this is the first occurrence of sv for the current index j **then**
      Establish a new vertex set $\mathcal{V}_{nsv}$ and move i from $\mathcal{V}_{sv}$ to $\mathcal{V}_{nsv}$
    else
      Move i from $\mathcal{V}_{sv}$ to $\mathcal{V}_{nsv}$
    end if
    Discard $\mathcal{V}_{sv}$ if it is empty
  end for
end for

3.5.2 *Block Structure Using Symbolic Dot Products*

An alternative way to find a block structure uses symbolic dot products between the rows of the matrix. While fully dense blocks can be found this way, it can also be used to determine an approximate block structure in which blocks are classified as dense or sparse based on a chosen threshold; this can be useful in preconditioning iterative methods. Although we assume that $\mathcal{S}\{A\}$ is symmetric, modifications can extend the approach to general nonsymmetric A.

Rewrite A as row vectors

$$A = \left(a_1^T, \ldots, a_n^T\right)^T, \text{ where } a_i^T = A_{i,1:n},$$

and consider $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$. A partition $\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_{nb}$ is constructed using row products $a_i^T a_k$ between different rows of A. These express the level of orthogonality between the rows; if $a_i^T a_k$ is small, then i and k are assigned to different vertex sets. Algorithm 3.8 treats all entries of A as unity, and the symbolic row products can be considered as a generalization of the angles between rows expressed by their cosines, hence the notation *cosine* for the vector that stores these products. The vertex sets are described using the vector *adjmap*. On output, if $adjmap(i_1) = adjmap(i_2)$, then vertices i_1 and i_2 belong to the same vertex set. Symmetry of $\mathcal{S}\{A\}$ simplifies the computation of the symbolic row products because for row i only $k > i$ is considered, that is, only the symbolic row products that correspond to one triangle of $A^T A$ are checked.

The procedure outlined in Algorithm 3.8 and illustrated in Figure 3.7 is controlled by a threshold parameter $\tau \in (0, 1]$. j is added to the subset to which i

ALGORITHM 3.8 Find approximately indistinguishable vertex sets in an undirected graph

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of a symmetrically structured matrix A, the number nz_i of entries in row i of A ($1 \le i \le n$), and a threshold parameter $\tau \in (0, 1]$.

Output: Partitioning of $\mathcal{V}$ into nb disjoint approximately indistinguishable vertex sets.

```
1: nb = 0, adjmap(1 : n) = 0, cosine(1 : n) = 0
2: for i = 1 : n do
3:     if adjmap(i) = 0 then
4:         nb = nb + 1                                   ▷ Start a new set
5:         adjmap(i) = ib
6:         for (i, j) ∈ E do                  ▷ Corresponds to an entry in A_{i,1:n}
7:             for (k, j) ∈ E with k > i do   ▷ Both rows i and k have an entry in
                                                 column j
8:                 if adjmap(k) = 0 then      ▷ k has not been yet added to some
                                                 partitioning set
9:                     cosine(k) = cosine(k) + 1   ▷ Increase partial dot product
10:                end if
11:            end for
12:            for k with cosine(k) ≠ 0 do
13:                if cosine(k)^2 ≥ τ^2 * nz_i * nz_k then   ▷ Test similarity of row
                                                               patterns
14:                    adjmap(k) = nb
15:                end if
16:                cosine(k) = 0
17:            end for
18:        end for
19:    end if
20: end for
```

belongs if the cosine of the angle between them exceeds τ. If $\tau < 1$, the block structure depends on the order in which the rows are processed, while $\tau = 1$ gives the exact indistinguishable vertex sets because, in this case, the row patterns being compared must be the identical for the rows to be assigned to the same set.

3.6 Notes and References

A standard description of LU factorizations based on the generic scheme given in Algorithm 3.1 can be found in the classical book by Ortega (1988b); this includes the

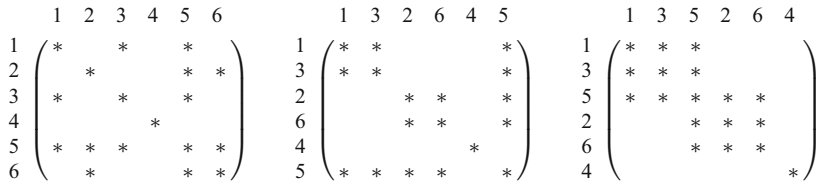


Figure 3.7 An example to illustrate Algorithm 3.8. The original matrix is given (left) together with the permuted matrix with indistinguishable vertex sets $\mathcal{V} = \{1, 3\} \cup \{2, 6\} \cup \{4\} \cup \{5\}$ obtained using $\tau = 1$ (centre) and the permuted matrix with approximately indistinguishable vertex sets $\mathcal{V} = \{1, 3, 5\} \cup \{2, 6\} \cup \{4\}$ obtained using $\tau = 0.5$ (right). The threshold $\tau = 0.5$ results in putting row 5 into the same set as row 1, making the vertex sets only approximately indistinguishable. The permuted matrix on the right has an approximate block form.

symmetric case and discusses early parallelization issues (which are also considered in the review of Dongarra et al. (1984)). A more algorithmically oriented approach is given in Golub & Van Loan (1996). For the column variant with partial pivoting, we recommend the detailed description of the sparse case in Gilbert & Peierls (1988). Many results for sparse LU factorizations are surveyed by Gilbert & Ng (1993) and Gilbert (1994). Pothen & Toledo (2004) consider both symmetric and nonsymmetric matrices in their survey of graph models of sparse elimination. The review by Davis et al. (2016) provides many further references.

Parter (1961) presents Parter's rule, and its nonsymmetric version is given in Haskins & Rose (1973). Building on the paper of Rose et al. (1976), Rose & Tarjan (1978) were the first to methodically consider the symbolic structure of Gaussian elimination for nonsymmetric matrices. Related work is included in the seminal paper on Cholesky factorizations by Liu (1986). Fill-in rules in the general context of Schur complements in LU factorizations can be found in Eisenstat & Liu (1993b).

Classical and detailed treatments of triangular solves that also cover sparse issues are given in the papers Brayton et al. (1970), Gilbert & Peierls (1988), and Gilbert (1994). For reducibility theory that is closely connected to the general theory of matrices, see Brualdi & Ryser (1991), which includes, for example, a proof of Theorem 3.4.

Algorithm 3.6 for computing strongly connected components of a digraph is introduced in Tarjan (1972); see also Sharir (1981) and Duff & Reid (1978) for an early implementation.

For identifying supervariables, Algorithm 3.7 follows Reid & Scott (1999), but see also Ashcraft (1995) and Hogg & Scott (2013a) (the latter presents an efficient variant that employs a stack). The approximate block partitioning of Section 3.5.2 is from the paper by Saad (2003a), which also describes some modifications of the basic approach; more sophisticated schemes with overlapping blocks are given in Fritzsche et al. (2013).

Chapter 4
Sparse Cholesky Solver: The Symbolic Phase

The modern view of numerical linear algebra as being to a large extent the study and systematic use of matrix decompositions has certainly been influenced by Cholesky's posthumously published work – Benzi (2017).

This chapter focuses on the symbolic phase of a sparse Cholesky solver. The sparsity pattern $\mathcal{S}\{A\}$ of the symmetric positive definite (SPD) matrix A is used to determine the nonzero structure of the Cholesky factor L without computing the numerical values of the nonzeros. The subsequent numerical factorization is discussed in the next chapter. Because the symbolic phase works only with $\mathcal{S}\{A\}$ (the values of the entries of A are not considered), it is also used for symmetric indefinite matrices and sometimes within LU factorizations of symmetrically structured nonsymmetric problems. It is implicitly assumed that all the diagonal entries of A are included in $\mathcal{S}\{A\}$ (even if they are zero). During the factorization phase, it may be necessary to amend the data structures to allow for indefiniteness. This makes the factorization of indefinite matrices potentially more expensive and more complex; this is considered further in Chapter 7.

A fundamental difference between dense and sparse Cholesky factorizations is that, in the latter, each column of L depends on only a subset of the previous columns. The elimination tree is a data structure that describes the dependencies among the columns of A during its factorization. A key result that assists in the understanding of sparse Cholesky factorizations is that the sparsity pattern of column j of L is the union of the pattern of column j of the lower triangular part of A and the patterns of the children of j in the elimination tree; this is shown in Section 4.3. Furthermore, the fact that disjoint parts of the elimination tree can be factored independently offers the potential for high-level tree-based parallelism that does not exist for dense matrices.

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_4

4.1 Column Replication Principle

We begin by looking at how the sparsity pattern of a computed column of L influences the patterns of subsequent Schur complements. From (3.2), the Schur complement $S^{(k)}$ can be written as

$$S^{(k)} = A_{k:n,k:n} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} l_{kj} & \dots & l_{nj} \end{pmatrix}. \tag{4.1}$$

Consider column j of L ($1 \le j \le k-1$), and let $l_{ij} \neq 0$ for some $i > j$. The involvement of l_{ij} in the outer product in (4.1) allows the following observation.

Observation 4.1 *For any $i > j \ge 1$ such that $l_{ij} \neq 0$*

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\}. \tag{4.2}$$

This is called the **column replication principle** *because the pattern of column j of L (rows i to n) is replicated in the pattern of column i of L.*

Denote the row index of the first subdiagonal nonzero entry in column j of L by $parent(j)$, that is,

$$parent(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}. \tag{4.3}$$

If there is no such entry, set $parent(j) = 0$. The row index $parent(parent(j))$ is denoted by $parent^2(j)$, and so on. Applying column replication recursively implies the sparsity pattern of column j of L is replicated in that of column $parent(j)$, which in turn is replicated in the pattern of column $parent^2(j)$, and so on. This is illustrated in Figure 4.1. Here $j = 1$, and because the first subdiagonal entry in column 1 is in row 3, $parent(1) = 3$. Likewise, $parent(3) = parent^2(1) = 5$.

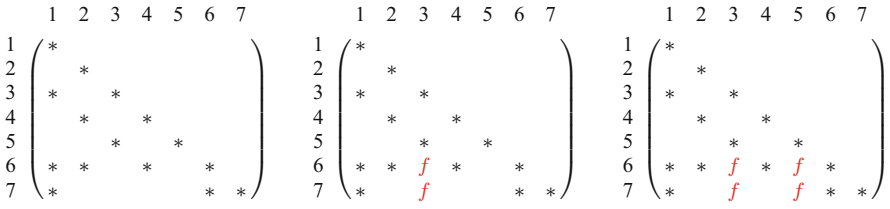


Figure 4.1 An illustration of column replication. On the left are the entries in L before step 1 of a Cholesky factorization (that is, the entries in the lower triangular part of A); in the centre, we show the replication of the nonzeros from column 1 in the pattern of column $parent(1) = 3$ (red entries f); on the right, we show the subsequent replication in column $parent^2(1) = 5$.

The following result shows that, provided A is irreducible, the mapping $parent(j)$ has nonzero values given by (4.3) for all $j < n$.

Theorem 4.1 (Liu 1986) *If A is SPD and irreducible, then in each column j ($1 \leq j < n$) of its Cholesky factor L there exists an entry $l_{ij} \neq 0$ with $i > j$.*

Proof From Parter's rule, each step of the Cholesky factorization corresponds to adding new edges into the graph of the corresponding Schur complement. If A is irreducible, then the graphs corresponding to the Schur complements are connected. Consequently, for any vertex j ($1 \leq j < n$) in any of these graphs, there is at least one vertex i with $i > j$ to which j is connected. This corresponds to the nonzero entry in column j of L. □

With the convention $parent^1(j) = parent(j)$, the next theorem shows that if entry l_{ij} of L is nonzero, then $parent^t(j) = i$ for some $t \geq 1$, and there is an entry in row i of L in each of the columns in the replication sequence $j,\ parent^1(j),\ parent^2(j),\ \ldots,\ parent^t(j)$.

Theorem 4.2 (Liu 1990; George 1998) *Let A be SPD, and let L be its Cholesky factor. If $l_{ij} \neq 0$ for some $j < i \leq n$, then there exists $t \geq 1$ such that $parent^t(j) = i$ and $l_{ik} \neq 0$ for $k = j,\ parent^1(j),\ parent^2(j),\ \ldots,\ parent^t(j)$.*

Proof If $i = parent^1(j)$, the result is immediate. Otherwise, there exists an index k, $j < k < i$ of a subdiagonal entry in column j of L such that $k = parent^1(j)$. Column replication implies $l_{ik} \neq 0$. Applying an inductive argument to l_{ik}, the result follows after a finite number of steps. □

If there is a sequence of nonzeros in a row of L, it is natural to ask where the sequence begins. It is straightforward to see if there is no $k \geq 1$ such that $a_{ik} \neq 0$, no replication of nonzeros can start in row i. The main result on the replication of nonzeros of A is summarized as Theorem 4.3.

Theorem 4.3 (Liu 1986) *Let A be SPD, and let L be its Cholesky factor. If $a_{ij} = 0$ for some $1 \leq j < i \leq n$, then there is a filled entry $l_{ij} \neq 0$ if and only if there exist $k < j$ and $t \geq 1$ such that $a_{ik} \neq 0$ and $parent^t(k) = j$.*

4.2 Elimination Trees

The discussion of column replication is significantly simplified using elimination trees. The **elimination tree** (or **etree**) $\mathcal{T}(A)$ (or simply $\mathcal{T}$) of an SPD matrix has vertices $1, 2, \ldots, n$ and an edge between each pair $(j, parent(j))$, where $parent(j)$ is given by (4.3); j is a root vertex of the tree if $parent(j) = 0$. The edges of $\mathcal{T}$ are considered to be directed from a child to its parent, that is,

$$\mathcal{E}(\mathcal{T}) = \{(j \longrightarrow i) \mid i = parent(j)\}.$$

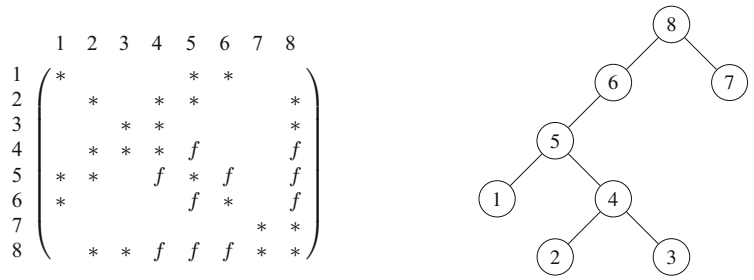


Figure 4.2 An illustration of a sparse matrix A with a symmetric sparsity pattern and its elimination tree $\mathcal{T}(A)$. The root vertex is 8. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.

If $\mathcal{T}$ has a single component, then the root vertex is n. Despite the terminology, the elimination tree need not be connected and in general is a **forest**. For simplicity, in our discussions, we assume $\mathcal{T}$ has a single component, and we say that $\mathcal{T}$ is described by the vector $parent$.

An example of a matrix and its elimination tree is given in Figure 4.2. Here and elsewhere, following conventional notation, directional arrows are omitted from the tree plot.

Concepts such as child, leaf, ancestor, and descendant vertices introduced in Section 2.3 for directed rooted trees can be applied to $\mathcal{T}$. Additionally, $anc_{\mathcal{T}}\{j\}$ and $desc_{\mathcal{T}}\{j\}$ are defined to be the sets of ancestors and descendants of vertex j in $\mathcal{T}$. We denote by $\mathcal{T}(j)$ the **subtree** of $\mathcal{T}$ induced by j and $desc_{\mathcal{T}}\{j\}$; j is the root vertex of $\mathcal{T}(j)$. The **size** $|\mathcal{T}(j)|$ is the number of vertices in the subtree. A **pruned subtree** of $\mathcal{T}(j)$ is the connected subgraph induced by j and a subset of $desc_{\mathcal{T}}\{j\}$. That is, for any vertex i in a pruned subtree of $\mathcal{T}(j)$, all the ancestors of i belong to the pruned subtree. A pruned subtree of $\mathcal{T}$ shares the mapping $parent$ with $\mathcal{T}$.

The following observation is straightforward.

Observation 4.2 *If $i \in anc_{\mathcal{T}}\{j\}$ for some $j \neq i$, then $i > j$.*

The connection between the mapping $parent$ and the sets of ancestors and descendants is emphasized by the next observation.

Observation 4.3 *If i and j are vertices of the elimination tree $\mathcal{T}$ with $j < i \leq n$, then*

$$i \in anc_{\mathcal{T}}\{j\} \text{ if and only if } j \in desc_{\mathcal{T}}\{i\} \text{ if and only if } parent^t(j) = i \text{ for some } t \geq 1.$$

The results in Section 4.1 can be expressed using rooted trees. Consider, for example, Theorem 4.2. Instead of stating that there exists $t \geq 1$ such that $parent^t(j) = i$, we can write that $i \in anc_{\mathcal{T}}\{j\}$. Rewriting Theorem 4.3 as the following corollary provides a clear characterization of the sparsity patterns of the rows of L.

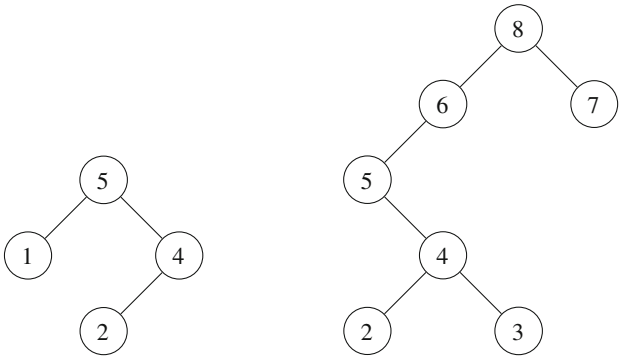


Figure 4.3 The row subtree $\mathcal{T}_r(5)$ of the elimination tree $\mathcal{T}$ from Figure 4.2 (left). Vertex 3 has been pruned because $a_{35} = 0$. The row subtree $\mathcal{T}_r(8)$ (right) differs from $\mathcal{T} = \mathcal{T}(A)$ because vertex 1 has been pruned ($a_{18} = 0$).

Corollary 4.4 (Liu 1986) *Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. If i and j are vertices of $\mathcal{T}$ with $j < i \leq n$ and $a_{ij} = 0$, then $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $j \in anc_{\mathcal{T}}(k)$ and $a_{ik} \neq 0$.*

The subtree of $\mathcal{T}$ with vertices that correspond to the nonzeros of row i of L is called the i-th **row subtree** and is denoted by $\mathcal{T}_r(i)$. Formally, it is a pruned subtree of $\mathcal{T}$ induced by the union of the vertex set

$$\{i\} \cup \{k \mid a_{ik} \neq 0 \text{ and } k < i\}$$

with all vertices on the directed paths in $\mathcal{T}$ from k to i, that is, with all their ancestors from $\mathcal{T}_r(i)$. The root vertex is i, and the leaf vertices are a subset of the column indices in the i-th row of the lower triangular part of A. Figure 4.3 illustrates row subtrees for the matrix and elimination tree from Figure 4.2. Note that row subtrees are connected subgraphs of $\mathcal{T}$, even if $\mathcal{T}$ is not connected. If $\mathcal{T}$ can be found without determining the pattern of L, then $\mathcal{T}_r(i)$ can be used to derive the sparsity pattern of row i of L, without having to store each entry explicitly.

Theorem 4.5 characterizes the ancestors of a given vertex j using paths in $\mathcal{G}(A)$. The proof helps clarify the relationship between $\mathcal{T}$ and paths in $\mathcal{G}(A)$.

Theorem 4.5 (Schreiber 1982; Liu 1986) *If i and j are vertices in the elimination tree $\mathcal{T}$ with $j < i \leq n$, then $i \in anc_{\mathcal{T}}\{j\}$ if and only if there exists a path*

$$j \overset{\mathcal{G}(A)}{\underset{\{1,\ldots,i\}}{\Longrightarrow}} i. \tag{4.4}$$

Proof Assume $i \in anc_{\mathcal{T}}\{j\}$. Then there is a path $j \overset{\mathcal{T}}{\Longrightarrow} i$ of length $l \geq 1$. Each edge of this path belongs to $\mathcal{G}(L)$ and corresponds either to an edge in $\mathcal{G}(A)$ or to a fill-path in $\mathcal{G}(A)$. Connecting these paths together gives (4.4).

Conversely, if the path (4.4) exists, induction on its length can be used to prove the result. If the path is of length 1, then the result holds because i and j are connected in $\mathcal{G}(A)$ by an edge. Consequently, from Theorem 4.2, i is an ancestor of j. Now assume that the result is true for all paths of length less than l ($l > 1$), and consider a path of length l. Let m be the largest vertex on this path. If $m < j$, then (4.4) is a fill-path connecting i and j and, therefore, $i \in anc_{\mathcal{T}}\{j\}$. Otherwise, for $m \geq j$, the assumption implies $i \in anc_{\mathcal{T}}\{m\} \cup \{m\}$ and $m \in anc_{\mathcal{T}}\{j\} \cup \{j\}$, that is, $i \in anc_{\mathcal{T}}\{j\}$. □

Given a vertex j in $\mathcal{T}$, the following corollary indicates how to find $parent(j)$ (if it exists). If the set of ancestors of j is non-empty, then the lowest numbered one is its parent.

Corollary 4.6 (Liu 1986, 1990) *Vertex i is the parent of vertex j in $\mathcal{T}$ if and only if i is the lowest numbered vertex satisfying $j < i \leq n$ for which there is a path (4.4).*

The existence of (4.4) is equivalent to requiring i and j belong to the same component of the graph $\mathcal{G}(A_{1:i,1:i})$ corresponding to the $i \times i$ principal leading submatrix $A_{1:i,1:i}$ of A. Figure 4.4 depicts $\mathcal{G}(A)$ for the matrix A given in Figure 4.2. Consider vertex 4. Its set of ancestors for which paths from Theorem 4.5 exist comprises vertices 5, 6, and 8. Vertex 7 is not an ancestor of 4 because there is no path from 7 to 4 in the graph $\mathcal{G}(A_{1:7,1:7})$. Among the ancestors of 4, vertex 5 fulfils the condition from Corollary 4.6 and is thus the parent of 4.

$\mathcal{T} = \mathcal{T}(A)$ can be constructed by stepwise extensions of the elimination trees of the principal leading submatrices of A. Assume we have $\mathcal{T}(A_{1:i-1,1:i-1})$ and we want to construct $\mathcal{T}(A_{1:i,1:i})$. Initialize $\mathcal{T}(A_{1:i,1:i}) = \mathcal{T}(A_{1:i-1,1:i-1})$. If there are no entries in row i of A to the left of the diagonal, then there is nothing to do, and only an isolated vertex i is added. Otherwise, i is the root of the row subtree $\mathcal{T}_r(i)$ and an ancestor of some vertex j in $\mathcal{T}$. The ancestors k of j with $k < i$ are in $\mathcal{T}(A_{1:i-1,1:i-1})$. Because row subtrees are connected subgraphs of $\mathcal{T}$, a directed path in $\mathcal{T}(A_{1:i,1:i})$ with $parent^t(j) = i$ exists for some $t \geq 1$. The search for this path starts from $jroot = j$ and continues, while $parent(jroot) \neq 0$ and $parent(jroot) \neq i$, using a sequence of assignments $jroot = parent(jroot)$. It terminates once $parent(jroot) = i$ or i is found to have already been added when

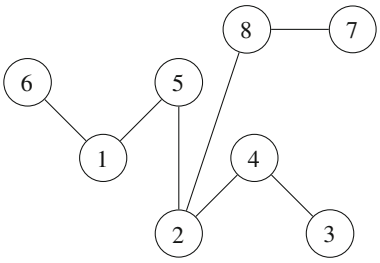


Figure 4.4 The graph $\mathcal{G}(A)$ of the matrix from Figure 4.2 illustrating Theorem 4.5 and Corollary 4.6.

tracing the path from another entry j' in row i. The construction of $\mathcal{T}$ is summarized in Algorithm 4.1.

ALGORITHM 4.1 Construction of an elimination tree
Input: A with a symmetric sparsity pattern and its undirected graph $\mathcal{G}$.
Output: Elimination tree $\mathcal{T}$ described by the vector $parent$.

```
1: for i = 1 : n do                                   ▷ Loop over the rows of A
2:     parent(i) = 0                                  ▷ Initialisation
3:     for j ∈ adj_G{i} and j < i do                  ▷ Loop over the below diagonal entries in row i
4:         jroot = j
5:         while parent(jroot) ≠ 0 and parent(jroot) ≠ i do   ▷ Find the current root
6:             jroot = parent(jroot)
7:         end while
8:         if parent(jroot) = 0 then
9:             parent(jroot) = i                      ▷ Make i the parent of jroot
10:        end if
11:    end for
12: end for
```

The most expensive part of Algorithm 4.1 is the **while** loop that searches for subtree roots. Because the directed path from j to its root $parent^t(j)$ is unique, shortcuts can be incorporated; this is called **path compression**. Having found a directed path from j to k, subsequent searches can be made more efficient by introducing a vector $ancestor$ and setting $ancestor(j) = k$. The modified algorithm is outlined in Algorithm 4.2. It maintains two structures using the current values of $parent$ and $ancestor$. The tree described by $ancestor$ is termed the **virtual tree**.

Figure 4.5 shows a matrix for which path compression makes constructing $\mathcal{T}$ significantly more efficient. For this example, $\mathcal{T}$ is determined by the mapping $parent(6) = 0$; $parent(i) = i + 1$ for $i = 1, \ldots, 5$. The complexity of Algorithm 4.1 is $O(n^2)$, but for this example the complexity of Algorithm 4.2 is $O(n)$. Formally, the complexity of Algorithm 4.2 is $O(nz(A) \log_2(n))$, where $nz(A)$ is the number of nonzeros of A, but the logarithmic factor is rarely reached. Additional modifications can reduce the theoretical complexity to $O(nz(A)\, g(nz(A), n))$, where $g(nz(A), n)$ is a very slowly increasing function called the functional inverse of Ackermann's function. This means that, in practice, the complexity of constructing $\mathcal{T}$, and hence of obtaining an implicit representation of $\mathcal{S}\{L\}$, is close to linear in $nz(A)$ (which in general is much smaller than $nz(L)$).

ALGORITHM 4.2 Construction of an elimination tree using path compression
Input: A with a symmetric sparsity pattern and its undirected graph $\mathcal{G}$.
Output: Elimination tree $\mathcal{T}$ described by the vector $parent$.

```
1:  for i = 1 : n do                                   ▷ Loop over the rows of A
2:      parent(i) = 0, ancestor(i) = 0                 ▷ Initialisation
3:      for j ∈ adj_G{i} and j < i do                  ▷ Loop over the below diagonal entries in row i
4:          jroot = j
5:          while ancestor(jroot) ≠ 0 and ancestor(jroot) ≠ i do
6:              l = ancestor(jroot)
7:              ancestor(jroot) = i                    ▷ Path compression to accelerate future searches
8:              jroot = l
9:          end while
10:         if ancestor(jroot) = 0 then
11:             ancestor(jroot) = i  and  parent(jroot) = i
12:         end if
13:     end for
14: end for
```

$$\begin{pmatrix} * & * & * & * & * & * \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & \\ * & & & & & * \end{pmatrix}$$

Figure 4.5 A sparse matrix for which computing the elimination tree using Algorithm 4.2 is much more efficient than using Algorithm 4.1.

The following simple theorem states that there is no edge in $\mathcal{G}(L + L^T)$ between vertices belonging to subtrees of $\mathcal{T}$ with different vertex sets. If there was such an edge (s, t), then from Theorem 4.2, one of the vertices s and t must be an ancestor of the other, which is a contradiction. The importance of this result is that it implies that for any such pairs of vertices the corresponding column sparsity patterns in L can be computed in parallel.

Theorem 4.7 (Liu 1990) *Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. Let $\mathcal{T}(i)$ and $\mathcal{T}(j)$ be two vertex-disjoint subtrees of $\mathcal{T}$. Then for all $s \in \mathcal{T}(i)$ and $t \in \mathcal{T}(j)$, the entry l_{st} of L is zero.*

4.3 Sparsity Pattern of L

The explicit structure of L is not always required; sometimes only the numbers of nonzeros in each row and column of L are needed. For example, when comparing the amount of fill-in in the factors for different initial orderings of A, allocating factor storage, finding relaxed supernodes (see Section 4.6), and determining load balance and synchronization events in parallel factorizations.

Let $row_L\{i\}$ denote the sparsity pattern of the off-diagonal part of row i of L, that is,

$$row_L\{i\} = \mathcal{S}\{L_{i,1:i-1}\} = \{j \mid j < i,\ l_{ij} \neq 0\}, \quad 1 \leq i \leq n.$$

The number of entries in L is

$$nz(L) = \sum_{i=1}^{n} |row_L\{i\}| + n.$$

Corollary 4.4 implies $row_L\{i\}$ is given by the vertices of the row subtree $\mathcal{T}_r(i)$. This suggests Algorithm 4.3. Here the vector $mark$ is used to flag vertices so as to avoid including them more than once within a row subtree. The complexity of the algorithm is $O(nz(L))$.

ALGORITHM 4.3 Computation of the row sparsity patterns of the Cholesky factor L
Input: A with a symmetric sparsity pattern, its undirected graph $\mathcal{G}$ and elimination tree $\mathcal{T}$ described by the vector $parent$.
Output: Row sparsity patterns $row_L\{i\}$ of the Cholesky factor L of A ($1 \leq i \leq n$).

```
1:  for i = 1 : n do                                  ▷ Loop over the rows of A
2:      row_L{i} = ∅                                  ▷ Initialisation
3:      mark(i) = i
4:      for k ∈ adj_G{i} and k < i do                 ▷ Loop over the below diagonal entries in
                                                        row i
5:          j = k
6:          while mark(j) ≠ i do                      ▷ Column j not yet encountered in row i
7:              mark(j) = i                           ▷ Flag j as encountered in row i
8:              row_L{i} = row_L{i} ∪ {j}             ▷ Add j to the sparsity pattern of row i
9:              j = parent(j)                         ▷ Move up the elimination tree
10:         end while
11:     end for
12: end for
```

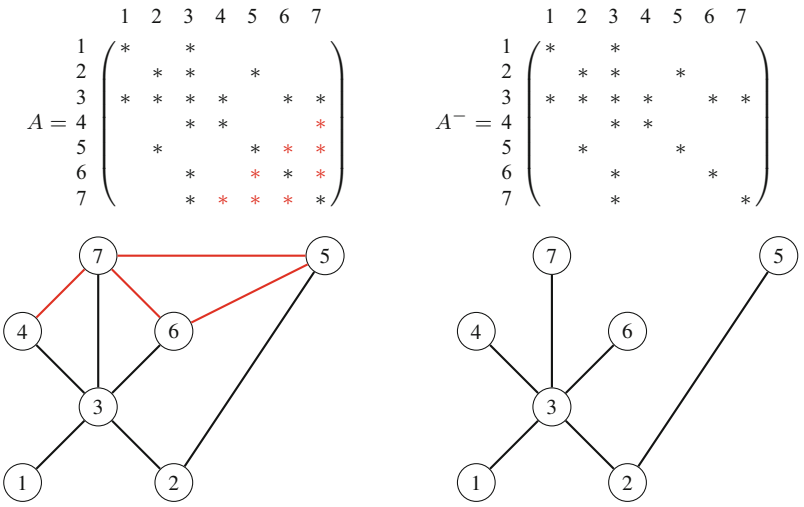


Figure 4.6 An illustration of the sparsity pattern of A and its graph $\mathcal{G}(A)$ (left) and the sparsity pattern of the corresponding skeleton matrix A^- and graph $\mathcal{G}(A^-)$ (right). The entries in A and edges of $\mathcal{G}(A)$ that do not belong to the skeleton matrix and graph are depicted in red.

Efficiency can be improved by employing the **skeleton graph** $\mathcal{G}(A^-)$ that is obtained from $\mathcal{G}(A)$ by removing every edge (i, j) for which $j < i$ and j is not a leaf vertex of $\mathcal{T}_r(i)$. $\mathcal{G}(A^-)$ is the smallest subgraph of $\mathcal{G}(A)$ with the same filled graph as $\mathcal{G}(A)$. The corresponding matrix is the **skeleton matrix**. An example is given in Figure 4.6. The complexity of constructing the elimination tree using the skeleton matrix and its graph $\mathcal{G}(A^-)$ is $O(nz(A^-)\, g(nz(A^-), n))$, where $nz(A^-)$ is the number of entries in the skeleton matrix. Because $nz(A^-)$ is often significantly smaller than $nz(A)$, an implementation that processes $\mathcal{G}(A^-)$ rather than $\mathcal{G}(A)$ can be substantially faster.

Analogously to the row sparsity patterns, let $col_L\{j\}$ denote the sparsity pattern of the off-diagonal part of column j of L, that is,

$$col_L\{j\} = \mathcal{S}(L_{j+1:n,j}) = \{i \mid i > j,\ l_{ij} \neq 0\}, \quad 1 \leq j \leq n.$$

The column replication principle can be written as

$$col_L\{j\} \subseteq col_L\{parent(j)\} \cup parent(j).$$

Theorem 4.8 describes $col_L\{j\}$ using the vertices of the subtree $\mathcal{T}(j)$.

Theorem 4.8 (George & Liu 1980c, 1981) *The column sparsity pattern $col_L\{j\}$ of the Cholesky factor L of the matrix A is equal to the adjacency set of vertices of the subtree $\mathcal{T}(j)$ in $\mathcal{G}(A)$, that is,*

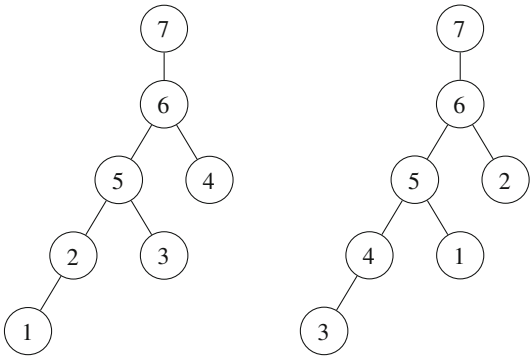


Figure 4.7 Two topological orderings of an elimination tree.

$$col_L\{j\} = adj_{\mathcal{G}(A)}\{\mathcal{T}(j)\}. \tag{4.5}$$

Proof If $i \in col_L\{j\}$, then $j \in row_L\{i\}$, and Theorem 4.3 implies $j \in anc_{\mathcal{T}}\{k\}$ for some k such that $a_{ik} \neq 0$. That is, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$. Conversely, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$ implies that in row i the entry in column j of L is nonzero. Thus, $j \in row_L\{i\}$, and hence, $i \in col_L\{j\}$. □

Algorithm 4.3 can be used to compute the column counts and the column sparsity patterns because when j is added to $row_L\{i\}$ at line 8, i can be added to $col_L\{j\}$. This does not generally obtain the column sparsity patterns sequentially. To derive an approach that does compute them sequentially, rewrite (4.5) as follows:

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \bigcup_{\{k \mid k \in \mathcal{T}(j) \setminus \{j\}\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}.$$

Using the column replication, this can be significantly simplified

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \bigcup_{\{k \mid j = parent(k)\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}. \tag{4.6}$$

This is used to obtain Algorithm 4.4, which constructs the sparsity pattern of each column j of L as the union of the sparsity pattern of column j of A ($adj_{\mathcal{G}(A)}\{j\}$) and the patterns of the children of j in $\mathcal{T}(A)$. Here $child\{j\}$ denotes the set of children of j. Because any child k of j satisfies $k < j$, the j-th outer step has the information needed to compute the sparsity pattern described by (4.6). Observe that $\mathcal{T}(A)$ does not need to be input.

ALGORITHM 4.4 Determining the sparsity patterns of each column of L
Input: A with symmetric sparsity pattern and its undirected graph $\mathcal{G}$.
Output: Column sparsity patterns $col_L\{j\}$ of the Cholesky factor L of A ($1 \leq j \leq n$).

```
1: for j = 1 : n do                                  ▷ Loop over the columns of L
2:     child{j} = ∅                                  ▷ Initialisation
3:     col_L{j} = adj_G{j} \ {1, ..., j − 1}
4:     for k ∈ child{j} do                           ▷ Unifying child structures in (4.6)
5:         col_L{j} = col_L{j} ∪ col_L{k} \ {j}
6:     end for
7:     if col_L{j} ≠ ∅ then
8:         l = min{i | i ∈ col_L{j}}
9:         child{l} = child{l} ∪ {j}                 ▷ Parent of j detected using Corollary 4.6
10:    end if
11: end for
```

4.4 Topological Orderings

The outer loop in Algorithm 4.4 does not have to be performed in the strict order $j = 1, \ldots, n$. What is necessary is that for each step j, the column sparsity pattern for each child of j has already been computed. An ordering of the vertices in a tree (and, more generally, in a DAG) is a topological ordering if, for all i and j, $j \in desc_{\mathcal{T}}\{i\}$ implies $j < i$ (Section 2.2). Observation 4.2 confirms that the ordering of vertices in the elimination tree $\mathcal{T}$ is a topological ordering. A new topological ordering of $\mathcal{T}$ defines a relabelling of its vertices corresponding to a symmetric permutation of A. This is illustrated in Figure 4.7. The sparsity patterns of the Cholesky factors of A and PAP^T can be different, but the following result shows that the amount of fill-in is the same.

Theorem 4.9 (Liu 1990) *Let $\mathcal{S}\{A\}$ be symmetric. If P is the permutation matrix corresponding to a topological ordering of the elimination tree $\mathcal{T}$ of A, then the filled graphs of A and PAP^T are isomorphic.*

There are many topological orderings of $\mathcal{T}$. One class is obtained using the depth-first search given by Algorithm 2.1. This searches all the components of $\mathcal{T}$ starting at their root vertices. In this case, once vertex i has been visited, all the vertices of the subtree $\mathcal{T}(i)$ are visited immediately after i and i is labelled as the last vertex of $\mathcal{T}(i)$. A topological ordering of $\mathcal{T}$ is a **postordering** if the vertex set of any subtree $\mathcal{T}(i)$ ($i = 1, \ldots, n$) is a contiguous sublist of $1, \ldots, n$. Unless additional rules on how vertices are selected are imposed, a postordering is generally not unique, as demonstrated in Figure 4.8. One possible postordering is defined in Algorithm 2.1. In this case, there is some freedom in the depth-first search to choose from the vertices that have not been visited, resulting in different postorderings.

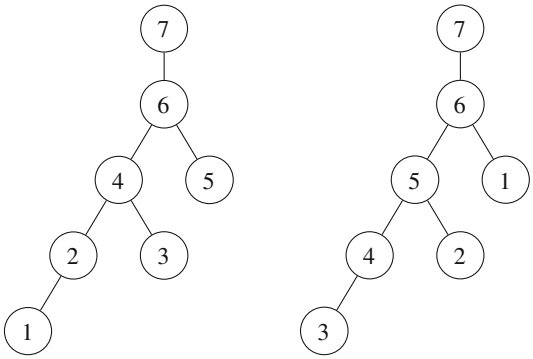


Figure 4.8 An example to illustrate the non-uniqueness of postorderings of an elimination tree.

4.5 Leaf Vertices of Row Subtrees

Leaf vertices of row subtrees play a key role in graph algorithms related to sparse Cholesky factorizations. They can be used to find the skeleton matrix described in Section 4.3, and they are important in parallel processing based on fundamental supernodes (see Section 4.6.1). Theorem 4.10 describes the relation between standard subtrees of $\mathcal{T}$ and row subtrees obtained by pruning (Section 4.2). This pruning is determined by the leaf vertices of row subtrees.

Theorem 4.10 (Liu 1986) *Let the elimination tree $\mathcal{T}$ of A be postordered. Let the column indices of the nonzeros in the strictly lower triangular part of row i of A be $c_1, \ldots, c_s$ with $s \geq 1$ and $0 < c_1 < \ldots < c_s < i$. Then c_t is a leaf vertex of the row subtree $\mathcal{T}_r(i)$ if and only if*

$$t = 1 \;\; or \;\; (1 < t \leq s \;\; and \;\; c_{t-1} \notin \mathcal{T}(c_t)).$$

Proof c_1 is always a leaf vertex of $\mathcal{T}_r(i)$. If this is not the case, then there exists a directed path from some vertex k, $k \neq c_1$ to i via c_1 such that $k \in \mathcal{T}_r(i)$ and $a_{ik} \neq 0$. Postordering of $\mathcal{T}$ implies $k < c_1$. This is a contradiction because c_1 is the index of the first nonzero in row i.

Consider now $t > 1$. Assume that $c_{t-1} \in \mathcal{T}(c_t)$ and that c_t is a leaf vertex of $\mathcal{T}_r(i)$. Row replication (Theorem 4.2) implies any $k \in anc_{\mathcal{T}}\{c_{t-1}\} \cup \{c_{t-1}\}$ such that $c_{t-1} \leq k < i$ satisfies $l_{ik} \neq 0$. Because $\mathcal{T}$ is postordered, $c_{t-1} \leq k \leq c_t$, and there is at least one $k < c_t$ satisfying this inequality. It follows that $k = c_{t-1}$. Because k belongs to $\mathcal{T}_r(i)$, c_t cannot be a leaf vertex of $\mathcal{T}_r(i)$, which is a contradiction.

Conversely, assume for $t > 1$ that $c_{t-1} \notin \mathcal{T}(c_t)$ and c_t is not a leaf vertex of $\mathcal{T}_r(i)$. From the second part of the assumption and the fact that $c_t \in \mathcal{T}_r(i)$, it follows that there is at least one leaf vertex $k < i$ of $\mathcal{T}_r(i)$ from which there is a directed path to i via c_t. Thus $k < c_t$. From the definition of the postordering of $\mathcal{T}$, all vertices l with $k < l \leq c_t$ are vertices of $\mathcal{T}(c_t)$. Vertex c_{t-1} must be among them and $c_{t-1} \in \mathcal{T}(c_t)$. This contradiction completes the proof. □

ALGORITHM 4.5 Find the sizes of subtrees $\mathcal{T}(i)$ of $\mathcal{T}$
Input: Elimination tree $\mathcal{T}$ described by the vector $parent$.
Output: Subtree sizes $|\mathcal{T}(i)|$ $(1 \le i \le n)$.

```
1: |T(1 : n)| = 1
2: for i = 1 : n − 1 do
3:     k = parent(i)
4:     |T(k)| = |T(k)| + |T(i)|
5: end for
```

Corollary 4.11 (Liu 1986) *Under the assumptions of Theorem 4.10, c_t is a leaf vertex of $\mathcal{T}_r(i)$ if and only if*

$$t = 1 \ \text{ or } \ (1 < t \le s \ \text{ and } \ c_{t-1} < c_t - |\mathcal{T}(c_t)| + 1).$$

Subtree sizes can be computed using Algorithm 4.5. Correctness of Algorithm 4.5 is guaranteed because $parent$ defines a topological ordering of $\mathcal{T}$.

Theorem 4.12 relaxes the condition that the entries in the rows of A are sorted by increasing column indices. This allows the leaf vertices of the row subtrees to be determined by columns.

Theorem 4.12 (Liu et al. 1993) *Consider the elimination tree $\mathcal{T}$ of A. Vertex j is a leaf vertex of some row subtree of $\mathcal{T}$ if and only if there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \le n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$.*

Proof Assume that for some $i \in anc_{\mathcal{T}}\{j\}$ vertex j is a leaf vertex of $\mathcal{T}_r(i)$. That is, $i \in adj_{\mathcal{G}(A)}\{j\}, i > j$. Suppose there exists k in $\mathcal{T}(j)\setminus\{j\}$ such that $i \in adj_{\mathcal{G}(A)}\{k\}$. Then all the ancestors of k, $k \le i$, in particular j, belong to $\mathcal{T}_r(i)$ and j cannot be a leaf vertex of $\mathcal{T}_r(i)$. This is a contradiction.

Conversely, assume that j is not a leaf vertex of any row subtree of $\mathcal{T}$ and that there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \le n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$. Because j is not a leaf vertex of any such $\mathcal{T}_r(i)$, Theorem 4.3 implies that there exists $k \in \mathcal{T}(j) \setminus \{j\}$ such that $a_{ik} \ne 0$, which gives a contradiction and completes the proof. □

To find leaf vertices of row subtrees of $\mathcal{T}$, Algorithm 4.6 uses a marking scheme based on Theorem 4.12 and exploits the postordering of $\mathcal{T}$. The auxiliary vector $prev\_nonz$ stores the column indices of the most recently encountered entries in the rows of the strictly lower triangular part of A.

4.6 Supernodes and the Assembly Tree

Because of column replication, the columns of L generally become denser as the Cholesky factorization proceeds. Exploiting this density can significantly enhance

ALGORITHM 4.6 Find leaf vertices of row subtrees of $\mathcal{T}$
Input: A with a symmetric sparsity pattern and a corresponding postordered elimination tree $\mathcal{T}$.
Output: Logical vector $isleaf$ with entries set to true for leaf vertices of row subtrees.

```
1: isleaf(1 : n) = false,  prev_nonz(1 : n) = 0
2: Compute |T(1 : n)|                                  ▷ Use Algorithm 4.5
3: for j = 1 : n do                                    ▷ Loop over the columns of A
4:     for i such that i > j and a_ij ≠ 0 do           ▷ Row index in strictly lower
                                                         triangular part of A
5:         k = prev_nonz(i)                            ▷ Column index of most recently seen entry in row i
6:         if k < j − |T(j)| + 1 then
7:             isleaf(j) = true                        ▷ j is a leaf vertex by Corollary 4.11
8:         end if
9:         prev_nonz(i) = j                            ▷ Flag j as the most recently seen entry in row i
10:    end for
11: end for
```

the performance of the numerical factorization in terms of both computation time and memory requirements. For this, we require the concept of supernodes. The idea is to group together columns with the same sparsity structure, so that they can be treated as a dense matrix for storage and computation. Let $1 \le s, t \le n$ with $s + t - 1 \le n$. A set of contiguously numbered columns of L with indices $S = \{s, s + 1, \ldots, s + t - 1\}$ is a **supernode** of L if

$$col_L\{s\} \cup \{s\} = col_L\{s + t - 1\} \cup \{s, \ldots, s + t - 1\}, \tag{4.7}$$

and S cannot be extended for $s > 1$ by adding $s - 1$ or for $s + t - 1 < n$ by adding $s + t$. Because S cannot be extended, it is a **maximal** subset of column indices. In graph terminology, a supernode is a **maximal clique** of contiguous vertices of $\mathcal{G}(L + L^T)$. A supernode may contain a single vertex. Figure 4.9 illustrates the supernodes in a Cholesky factor of order 8.

The **supernodal elimination** or **assembly tree** is defined to be the reduction of the elimination tree that contains only supernodes. Each vertex of the elimination tree is associated with one elimination, and a single integer (the index of its parent) is needed. Associated with each vertex of the assembly tree is an index list of the row indices of the nonzeros in the columns of the supernode. These implicitly define the sparsity pattern of L. An example that demonstrates the difference between the elimination and assembly trees is given in Figure 4.10. Here the elimination tree is postordered, and there are 5 supernodes: {1, 2}, 3, 4, 5, {6, 7, 8, 9}. For supernode 1 that comprises columns 1 and 2, the row index list is {1, 2, 8, 9}.

$$L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} * & & & & & & & \\ * & * & & & & & & \\ & & * & & & & & \\ & & * & * & & & & \\ * & * & * & * & * & & & \\ & & & & * & * & & \\ * & * & & & * & * & * & \\ & & * & * & * & * & * & * \end{pmatrix} \end{array}$$

Figure 4.9 An example to illustrate supernodes in L. The first supernode comprises columns 1 and 2, the second columns 3 and 4, and the third columns 5–8.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & & & & & & * & * \\ * & * & & & & & & * & * \\ & & * & * & & & & * & \\ & & * & * & & & * & * & \\ & & & & * & * & * & & \\ & & & & * & * & f & * & * \\ & & & * & * & f & * & f & * \\ * & * & * & * & & * & f & * & f \\ * & * & & & & * & * & f & * \end{pmatrix} \end{array}$$

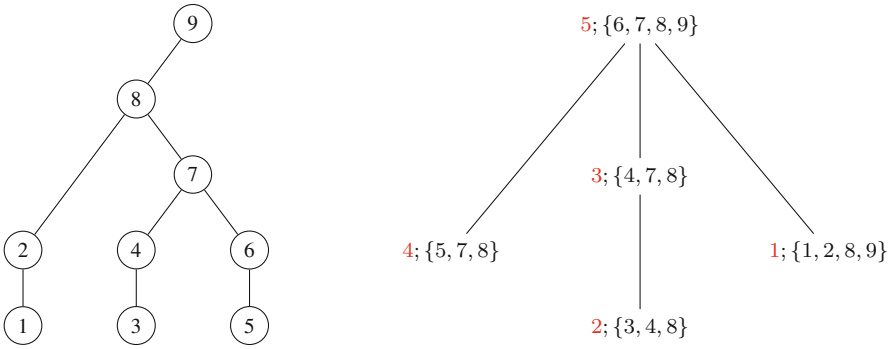


Figure 4.10 A sparse matrix and its postordered elimination tree (left) and postordered assembly tree (right). Filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f. For the assembly tree, the vertices are in red and the index lists associated with each vertex are given.

Supernodes can be characterized by the following result on the column counts of L, from which we see that supernodes can be found using column counts rather than the column sparsity patterns that appear in (4.7).

Theorem 4.13 (Liu et al. 1993) *The set of columns of L with indices $S = \{s, s + 1, \ldots, s + t - 1\}$ is a supernode of L if and only if it is a maximal set of contiguous columns such that $s + i - 1$ is a child of $s + i$ for $i = 1, \ldots, t - 1$ and*

$$|\, col_L\{s\}\,| = |\, col_L\{s + t - 1\}\,| + t - 1. \tag{4.8}$$

Proof Let S be a supernode. For $i, j \in S$ with $i > j$, we have $i \in col_L\{j\}$. This implies that in the postordered elimination tree the vertex $i = j + 1$ is the parent of j for $j = s, \ldots, s + t - 2$. Moreover, from Observation 4.2, for any $i, j \in S$ with $i > j, i \in col_L\{j\}$ implies $col_L\{j\} \setminus \{1, \ldots, i\} \subseteq col_L\{i\}$. Therefore,

$$|\, col_L\{s + i\}\,| \geq |\, col_L\{s + i - 1\}\,| - 1, \quad i = 1, \ldots, t - 1, \tag{4.9}$$

with equality if and only if

$$col_L\{s + i\} = col_L\{s + i - 1\} \setminus \{s + i\},$$

that is, if S is a supernode.

Conversely, assume S is a maximal set of contiguous columns such that, for $i = 1, \ldots, t - 1$, $s + i - 1$ is a child of $s + i$ and S satisfies (4.8). Because of column replication, such a sequence of parent and child vertices must satisfy (4.9) with equality if and only if (4.7) is satisfied. It follows that S is a supernode. □

Supernodes enhance the efficiency of sparse factorizations and sparse triangular solves because they enable floating-point operations to be performed on dense submatrices rather than on individual nonzeros, thus improving memory hierarchy utilization and allowing the use of highly efficient dense linear algebra kernels (such as Level 3 BLAS kernels). Because the rows and columns of a supernode have a common sparsity structure, this only needs to be stored once, reducing indirect addressing. Supernodes help to increase the granularity of tasks, which is useful for improving the computation to overhead ratio in a parallel implementation. Fill-in results in supernodes near the root of the assembly tree often being much larger than those close to the leaf vertices.

Observe that the columns within a supernode are numbered consecutively, but they can be numbered within the supernode in any order without changing the number of nonzeros in L (assuming the corresponding rows are permuted symmetrically). On some architectures, particularly those using GPUs, this freedom can be exploited to improve the factorization efficiency. Essentially, it is desirable to order the columns within a supernode such that the entries of L form fewer but less fragmented dense blocks.

Some applications, such as power grid analysis, in which the basis of the linear system is not a finite element or finite difference discretization of a physical domain, can lead to sparse matrices that incur very little fill-in during factorization. The supernodes can then be very small, and the costs associated with identifying them may not be offset by the increase in performance resulting from the potential for block operations. However, as supernodes can offer such significant performance gains, it can be advantageous to merge (small) supernodes that have similar (but not exactly the same) nonzero patterns, despite this increasing the overall fill-in and operation count. This process is termed **supernode amalgamation**, and the resultant nodes are often referred to as **relaxed supernode**.

4.6.1 Fundamental Supernodes

In practice, fundamental supernodes are easier to work with in the numerical factorization. Let $1 \le s, t \le n$ with $s + t - 1 \le n$. A maximal set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a **fundamental supernode** if for any successive pair $i-1$ and i in the list, $i-1$ is the only child of i in $\mathcal{T}$ and $col_L\{i\} \cup \{i\} = col_L\{i-1\}$. s is termed the starting vertex. An example is given in Figure 4.11. The difference between the sets of supernodes and fundamental supernodes is normally not large, with the latter having (slightly) more blocks in the resulting partitioning of L. Note that fundamental supernodes are independent of the choice of the postordering of $\mathcal{T}$. Theorem 4.14 describes the relationship between fundamental supernodes and the leaf vertices of row subtrees of $\mathcal{T}$. In particular, it characterizes starting vertices of the fundamental supernodes. The leaf vertices of $\mathcal{T}$ are trivially starting vertices of fundamental supernodes. But, possibly surprisingly, so too are the leaf vertices of row subtrees.

Theorem 4.14 (Liu et al. 1993) *Assume $\mathcal{T}$ is postordered. Vertex s is the starting vertex of a fundamental supernode if and only if it has at least two child vertices in $\mathcal{T}$ or it is a leaf vertex of a row subtree of $\mathcal{T}$.*

Proof If s has at least two child vertices then, from the definition of a fundamental supernode, it must be the starting vertex of a fundamental supernode. Assume that, for some $i > s$, s is a leaf vertex of $\mathcal{T}_r(i)$. If s is also a leaf vertex of $\mathcal{T}$, then s is a starting vertex of a supernode. The remaining case is s having only one child. Because $\mathcal{T}$ is postordered, this child must be $s-1$. Theorem 4.3 then implies $a_{is} \neq 0$ and $a_{i,s-1} = 0$, that is, $i \in col_L\{s\}$ and $i \notin col_L\{s-1\}$. It follows that

$$\mathcal{S}\{L_{s-1:n,s-1}\} \subsetneqq \mathcal{S}\{L_{s:n,s}\} \cup \{s-1\},$$

and vertices s and $s-1$ cannot belong to the same supernode. Hence, s is the starting vertex of a new fundamental supernode.

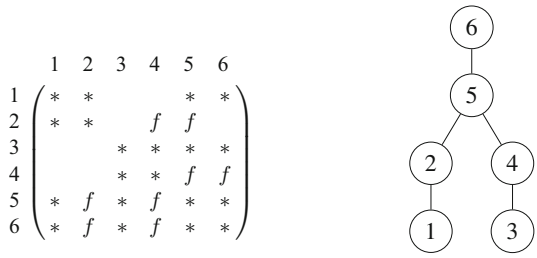


Figure 4.11 A matrix A and its postordered elimination tree $\mathcal{T}$ for which the set of supernodes $\{1, 2\}$ and $\{3, 4, 5, 6\}$ and the set of fundamental supernodes $\{1, 2\}$, $\{3, 4\}$ and $\{5, 6\}$ are different. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.

Conversely, assume that s is the starting vertex of a fundamental supernode S. If s has no child vertices or at least two child vertices, the result follows. If s has exactly one child vertex, postordering implies this child is $s-1$. Because S is maximal, there exists i such that $i \notin col_L\{s-1\}$ and $i \in col_L\{s\}$ (otherwise S could be extended by adding $s-1$). Hence, s is a leaf vertex of $\mathcal{T}_r(i)$. □

Because fundamental supernodes are characterized by their starting vertices, they can be found by modifying Algorithm 4.6 to incorporate marking leaf vertices of the row subtrees and vertices with at least two child vertices. Once the elimination tree has been computed, the complexity is $O(n+nz(A))$. The computation can be made even more efficient by using the skeleton graph $\mathcal{G}(A^-)$.

4.7 Notes and References

The excellent monographs by Tewarson (1973), George & Liu (1981), and Davis (2006) represent milestones in the development of contemporary symbolic factorization algorithms and their implementation. A complementary way to follow many of the developments is by looking at the early software (and accompanying user documentation), such as YSMP (Eisenstat et al., 1982) and SPARSPAK (George & Ng, 1984). In addition, there are several influential survey articles focusing on sparse Cholesky algorithms and emphasizing the crucial role of the elimination tree, for example, Liu (1990), George (1998); see also Bollhöfer & Schenk (2006), Hogg & Scott (2013a) and the more recent comprehensive survey of Davis et al. (2016). The latter provides a general overview of much of the research related to sparse direct methods and includes pointers to many specialized references.

There are a large number of journal articles that provide a fuller understanding of the theory and algorithms employed in symbolic factorizations. Schreiber (1982) defines the elimination tree of a sparse symmetric matrix. The seminal paper of Liu (1986) describes elimination tree construction, while for an extensive overview of the roles of elimination trees and topological orderings as well as the determination of the column sparsity patterns of the factor L, we refer to Liu (1990). If only row and column counts of L are needed, the fastest known algorithms are described in Gilbert et al. (1994). This paper also refers to another admirable paper of Liu et al. (1993) that describes the efficient computation of fundamental supernodes based on the leaf vertices of row subtrees of the elimination tree.

A key driver behind research into efficient (in terms of time and memory) sparse Cholesky algorithms has always been the development of computational codes. Many currently available packages implement not only sparse Cholesky factorizations but also more general LDLT factorizations of sparse symmetric indefinite matrices. The software is necessarily highly sophisticated and is therefore generally accompanied by technical reports and/or journal publications that explain the data structures and choices that were made in the algorithm and software design as well as providing details of the different options that are offered (examples include Duff (2004), Reid & Scott (2009), Hogg et al. (2010)).

Chapter 5
Sparse Cholesky Solver: The Factorization Phase

The adoption of Cholesky's method owes not a little to the publicity given to it shortly after the end of World War II by British mathematicians and computer pioneers, including Alan Turing, Leslie Fox, Jim Wilkinson, and especially John Todd – Benzi (2017).

Achieving high performance for sparse direct solvers in general, and sparse Cholesky factorization, in particular, is a very well researched topic – Rennich et al. (2016)

Having considered the symbolic phase of a sparse Cholesky solver in the previous chapter, the focus of this chapter is the subsequent numerical factorization phase. If A is a symmetric positive definite (SPD) matrix, then it is factorizable (strongly regular) and (in exact arithmetic) its Cholesky factorization $A = LL^T$ exists. LDLT factorizations of general symmetric indefinite matrices are considered in Chapter 7.

5.1 Dense Cholesky Factorizations

Because efficient implementations of sparse Cholesky factorizations rely heavily on exploiting dense blocks, we first consider algorithms for the Cholesky factorization of dense matrices that can be applied to such blocks. Algorithm 5.1 is a basic left-looking algorithm. It is an in-place algorithm because L can overwrite the lower triangular part of A (thus reducing memory requirements if A is no longer required).

Writing A in the block form (1.2), the computation can be reorganized to give Algorithm 5.2. This allows the exploitation of Level 3 BLAS for the computationally intensive components (dense matrix-matrix multiplies and dense triangular solves). Here A has nb block columns, which are referred to as **panels**. Step 6 can be performed using Algorithm 5.1.

Algorithms 5.1 and 5.2 are left-looking. This means that the updates are not applied immediately. Instead, all updates from previous (block) columns are applied together to the current (block) column before it is factorized. In a right-looking

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_5

ALGORITHM 5.1 In-place dense left-looking Cholesky factorization
Input: Dense SPD matrix A.
Output: Factor L such that $A = LL^T$.

1: **for** $j = 1 : n$ **do**
2: $L_{j:n,j} = A_{j:n,j}$ ▷ Only the lower triangular part of A is required
3: **for** $k = 1 : j - 1$ **do**
4: $L_{j:n,j} = L_{j:n,j} - L_{j:n,k}\, l_{jk}$ ▷ Update column j using previous columns
5: **end for**
6: $l_{jj} = (l_{jj})^{1/2}$ ▷ Overwrite the diagonal entry with its square root
7: $L_{j+1:n,j} = L_{j+1:n,j} / l_{jj}$ ▷ Scale off-diagonal entries in column j
8: **end for**

ALGORITHM 5.2 In-place dense left-looking panel Cholesky factorization
Input: Dense SPD matrix A in the form (1.2) with nb panels.
Output: Factor L such that $A = LL^T$.

1: **for** $jb = 1 : nb$ **do**
2: $L_{jb:nb,jb} = A_{jb:nb,jb}$
3: **for** $kb = 1 : jb - 1$ **do**
4: $L_{jb:nb,jb} = L_{jb:nb,jb} - L_{jb:nb,kb}\, L_{jb,kb}^T$ ▷ Update block column jb
5: **end for**
6: Compute in-place factorization of $L_{jb,jb}$ ▷ Overwrite $L_{jb,jb}$ with its Cholesky factor
7: $L_{jb+1:nb,jb} = L_{jb+1:nb,jb}\, L_{jb,jb}^{-T}$ ▷ Dense triangular solve
8: **end for**

approach (Algorithm 5.3), outer product updates are applied to the part of the matrix that has not yet been factored as they are generated.

The large panel updates can be split into operations involving only blocks. This is shown in Algorithm 5.4 for the right-looking approach.

The panel and block descriptions of the factorization enable efficient parallelization. The three main block operations, which are called tasks, are **factorize**(jb), **solve**(ib, jb), and **update**(ib, jb, kb). There are the following dependencies between the tasks.

factorize(jb) depends on **update**(jb, kb, jb) for all $kb = 1, \ldots, jb - 1$.
solve(ib, jb) depends on **update**(ib, kb, jb) for all $kb = 1, \ldots, jb - 1$, and **factorize**(jb).
update(ib, jb, kb) depends on **solve**(ib, kb), **solve**(jb, kb).

ALGORITHM 5.3 In-place dense right-looking panel Cholesky factorization
Input: Dense SPD matrix A in the form (1.2) with nb panels.
Output: Factor L such that $A = LL^T$.

for $jb = 1 : nb$ **do**
  $L_{jb:nb,jb} = A_{jb:nb,jb}$
end for
for $jb = 1 : nb$ **do**
  Compute in-place factorization of $L_{jb,jb}$ ▷ Overwrite $L_{jb,jb}$ with its Cholesky factor
  $L_{jb+1:nb,jb} = L_{jb+1:nb,jb}\, L_{jb,jb}^{-T}$ ▷ Dense triangular solve
  for $kb = jb + 1 : nb$ **do**
    $L_{kb:nb,kb} = L_{kb:nb,kb} - L_{kb:nb,jb}\, L_{kb,jb}^{T}$
  end for
end for

ALGORITHM 5.4 In-place dense right-looking block Cholesky factorization
Input: Dense SPD matrix A in the form (1.2) with $nb \times nb$ blocks.
Output: Factor L such that $A = LL^T$.

for $jb = 1 : nb$ **do**
  $L_{jb:nb,jb} = A_{jb:nb,jb}$
end for
for $jb = 1 : nb$ **do**
  Compute in-place factorization of $L_{jb,jb}$ ▷ Task **factorize**(jb)
  for $ib = jb + 1 : nb$ **do**
    $L_{ib,jb} = L_{ib,jb}\, L_{jb,jb}^{-T}$ ▷ Task **solve**(ib, jb)
    for $kb = jb + 1 : ib$ **do**
      $L_{ib,kb} = L_{ib,kb} - L_{ib,jb}\, L_{kb,jb}^{T}$ ▷ Task **update**(ib, jb, kb)
    end for
  end for
end for

A dependency graph can be used to schedule the tasks. Its vertices correspond to tasks and dependencies between tasks are represented as directed edges. The result is a directed acyclic graph (DAG). A task is ready for execution if and only if all tasks with incoming edges to it have completed. DAG-driven linear algebra uses either a static or dynamic schedule based on these graphs to implement the tasks in a parallel environment. In practice, it is not necessary to explicitly compute the

task DAG: it can be constructed on-the-fly taking into account the dependencies. The task DAG allows a lot of flexibility in the order in which tasks are carried out: the left- and right-looking approaches correspond to particular restricted orderings of the tasks.

5.2 Introduction to Sparse Cholesky Factorizations

There are several classes of algorithms that implement sparse Cholesky factorizations. Their major differences relate to how they schedule the computations. This affects the use of dense kernels, the amount of memory required during the factorization as well as the potential for parallel implementations. As in the dense case, the factorization is split into tasks that involve computations on and between dense submatrices and the precedence relations among them can be captured by a task graph.

We start by extending the dense Cholesky factorizations to the sparse case in a straightforward way. In practice, it is essential for efficiency to exploit the supervariables of A and the supernodes of L. Thus, while for simplicity of the descriptions and notation, we refer to rows and columns of A and L, these typically represent block rows and block columns and, as in the above discussion of the dense block factorization algorithm, the entries of A and L are then submatrices.

The entries of L satisfy the relationship

$$L_{j+1:n,j} = \left(A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k} l_{jk} \right) / l_{jj} \quad \text{with} \quad l_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2},$$

and from this we deduce the following result.

Theorem 5.1 (Liu 1990) *The numerical values of the entries in column $j > k$ of L depend on the numerical values in column k of L if and only if $l_{jk} \neq 0$.*

The theoretical background of the previous chapter based on the elimination tree $\mathcal{T}$ enables the dependencies in Theorem 5.1 to be searched for efficiently. In particular, $\mathcal{T}$ allows the row (or column) counts of L to be computed and they can be used to allocate storage for L. It can also be used to find supernodes and the resulting (block) elimination tree can then be employed to determine the (block) column structure of L. In practice, it can be beneficial to split large supernodes into smaller panels to better conform to computer caches.

Algorithms 5.5 and 5.6 are simplified sparse left- and right-looking Cholesky factorization algorithms that are straightforward sparse variants of Algorithms 5.1 and 5.4, respectively (the latter with $nb = n$, that is, without considering blocks). Here, we assume that the sparsity pattern of L has already been determined in the symbolic phase and static storage formats based, for example, on compressed columns and/or rows are used.

ALGORITHM 5.5 Simplified sparse left-looking Cholesky factorization
Input: SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.
Output: Factor L such that $A = LL^T$.

```
1:  l_ij = a_ij for all (i, j) ∈ S{L}        ▷ Filled entries in L are initialised to zero
2:  for j = 1 : n do
3:      for k ∈ {k < j | l_jk ≠ 0} do
4:          for i ∈ {i ≥ j | l_ik ≠ 0} do
5:              l_ij = l_ij − l_ik l_jk
6:          end for
7:      end for
8:      l_jj = (l_jj)^{1/2}
9:      for i ∈ {i > j | l_ij ≠ 0} do
10:         l_ij = l_ij / l_jj
11:     end for
12: end for
```

ALGORITHM 5.6 Simplified sparse right-looking Cholesky factorization
Input: SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.
Output: Factor L such that $A = LL^T$.

```
1:  l_ij = a_ij for all (i, j) ∈ S{L}        ▷ Filled entries in L are initialised to zero
2:  for j = 1 : n do
3:      l_jj = (l_jj)^{1/2}
4:      for i ∈ {i > j | l_ij ≠ 0} do
5:          l_ij = l_ij / l_jj
6:      end for
7:      for k ∈ {k > j | l_kj ≠ 0} do
8:          for i ∈ {i ≥ k | l_ij ≠ 0} do
9:              l_ik = l_ik − l_ij l_kj
10:         end for
11:     end for
12: end for
```

An alternative for sparse matrices held in row-wise format is to compute L one row at a time. This is sometimes called an **up-looking** factorization because rows 1 to $i-1$ are employed to compute row i ($i > 1$). The approach is asymptotically optimal in the work performed and for highly sparse matrices it is potentially extremely efficient because the entries of A are used in the natural order in which they are stored. However, it is difficult to incorporate high level BLAS.

The following relation holds for the i-th row of L

$$L_{i,1:i-1}^T = L_{1:i-1,1:i-1}^{-1} A_{1:i-1,i} \quad \text{with} \quad l_{ii}^2 = a_{ii} - L_{i,1:i-1} L_{i,1:i-1}^T.$$

The application of $L_{1:i-1,1:i-1}^{-1}$ can be implemented by solving the triangular system

$$L_{1:i-1,1:i-1} y = A_{1:i-1,i},$$

and setting $L_{i,1:i-1}^T = y$. The following result can be used to determine the sparsity pattern of y.

Theorem 5.2 (Gilbert 1994) *Consider a sparse lower triangular matrix L and the DAG $\mathcal{G}(L^T)$ with vertex set $\{1, 2, \ldots, n\}$ and edge set $\{(j \longrightarrow i) \mid l_{ij} \neq 0\}$. The sparsity pattern $\mathcal{S}\{y\}$ of the solution y of the system $Ly = b$ is the set of all vertices reachable in $\mathcal{G}(L^T)$ from $\mathcal{S}\{b\}$.*

Proof From Algorithm 3.4 and assuming the non-cancellation assumption, we see that (a) if $b_i \neq 0$, then $y_i \neq 0$ and (b) if for some $j < i$, $y_j \neq 0$ and $l_{ij} \neq 0$, then $y_i \neq 0$. These two conditions can be expressed as a graph transversal problem in $\mathcal{G}(L^T)$. (a) adds all vertices in $\mathcal{S}\{b\}$ to the set of visited vertices and (b) states that if vertex j has been visited, then all its neighbours in $\mathcal{G}(L^T)$ are added to the set of visited vertices. Thus $\mathcal{S}\{y\} = \mathcal{R}each(\mathcal{S}\{b\}) \cup \mathcal{S}\{b\}$. □

Figure 5.1 illustrates the sparsity patterns of a lower triangular matrix L and vector b together with $\mathcal{G}(L^T)$. The vertices that are reachable from $\mathcal{S}\{b\} = \{2, 4\}$ are 5 and 6 and thus $\mathcal{S}\{y\} = \{2, 4, 5, 6\}$.

Algorithm 5.7 outlines a sparse row Cholesky factorization that is based on the repeated solution of triangular linear systems. Theorem 5.2 can be used to determine the sparsity pattern of row i at Step 3, that is, by finding all the vertices that are reachable in $\mathcal{G}(L_{1:j-1,1:j-1}^T)$ from the set $\{i \mid a_{ij} \neq 0,\ i < j\}$. A depth-first search

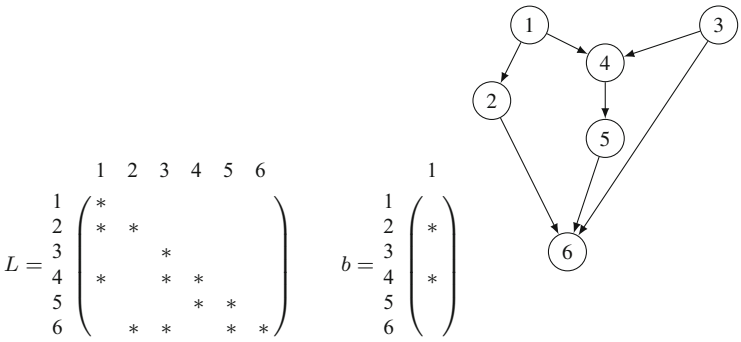


Figure 5.1 An example to illustrate L, b and $\mathcal{G}(L^T)$.

ALGORITHM 5.7 Sparse up-looking Cholesky factorization
Input: SPD matrix A.
Output: Factor L such that $A = LL^T$.

1: $l_{11} = (a_{11})^{1/2}$
2: **for** $i = 2 : n$ **do**
3: Find $\mathcal{S}\{L_{i,1:i-1}\}$ ▷ Sparsity pattern of row i
4: $L^T_{i,1:i-1} = L^{-1}_{1:i-1,1:i-1} A_{1:i-1,i}$ ▷ Sparse triangular solve
5: $l_{ii} = a_{ii} - L_{i,1:i-1} L^T_{i,1:i-1}$
6: $l_{ii} = (l_{ii})^{1/2}$
7: **end for**

of $\mathcal{G}(L^T_{1:j-1,1:j-1})$ determines the vertices in the row sparsity patterns in topological order, and performing the numerical solves in that order correctly preserves the numerical dependencies. Alternatively, because nonzeros of $L_{i,1:i-1}$ correspond to the vertices in the i-th row subtree $\mathcal{T}_r(i)$ that are not equal to i, another option is to find the row subtrees using $\mathcal{T}(A)$.

5.3 Supernodal Sparse Cholesky Factorizations

The simplified schemes form the basis of sophisticated supernodal algorithms that are designed to be efficient in parallel computational environments. Consider the right-looking variant and recall that a supernode consists of one or more consecutive columns of L with the same sparsity pattern. These nonzeros are stored as a dense trapezoidal matrix (only the lower triangular part of the block on the diagonal needs to be stored and the rows of zeros in the columns of the supernode are not held). This is termed a **nodal matrix** (see Figure 5.2).

Once a supernode is ready to be factorized, a dense Cholesky factorization of the block on the diagonal of the nodal matrix is performed (one of the approaches of Section 5.1 can be used). Then a triangular solve is performed with the computed factor and the rectangular part of the nodal matrix. The next step is to iterate over ancestors of the supernode in the assembly tree. For each parent, the rows of the current supernode corresponding to the parent's columns are identified, and then the outer product of those rows and the part of the supernode below those columns formed (update operations). The resulting matrix can be held in a temporary buffer. The rows and columns of this buffer are matched against indices of the ancestors and are added to them in a sparse scatter operation. For efficiency, the updates may use panels so that the temporary buffer remains in cache.

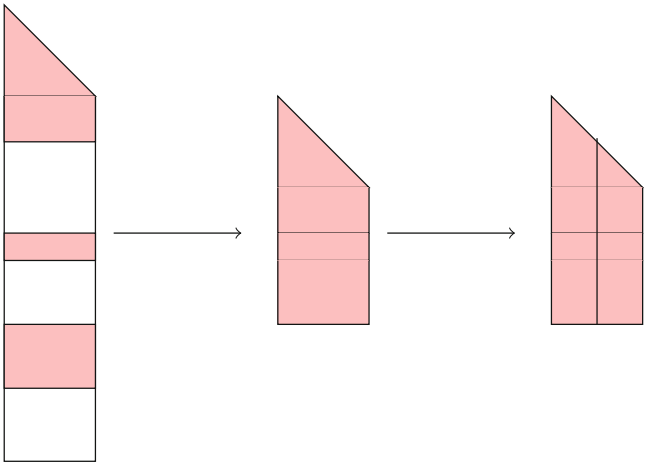


Figure 5.2 An illustration of a supernode (left), the corresponding nodal matrix (centre), and the nodal matrix with two panels (right). The shaded lower triangular part of the block on the diagonal and the shaded block rows are treated as dense.

5.3.1 DAG-Based Approach

The DAG-based approach can also be extended to the sparse case. Each nodal matrix is subdivided into blocks. The factorization is split into tasks in which a single block is revised. The key difference compared to the dense case is that it is necessary to distinguish between two types of update operations: **update_internal** performs the update between blocks in the same nodal matrix and **update_between** performs the update when the blocks belong to different nodal matrices. Thus the sparse Cholesky factorization is split into the following tasks; the first two are illustrated in Figure 5.3. In this example, the nodal matrix has two block columns that do not contain the same number of columns.

factorize_block(L_{diag}) Computes the dense Cholesky factor L_{diag} of the block on the diagonal (leftmost plot). If the block is trapezoidal, the factorization is followed by a triangular solve of its rectangular part $L_{rect} = L_{rect} L_{diag}^{-T}$ (centre plot).

solve_block(L_{dest}) Performs a triangular solve of an off-diagonal block L_{dest} of the form $L_{dest} = L_{dest} L_{diag}^{-T}$ (rightmost plot).

update_internal(L_{dest} , L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_{dest}, L_r and L_c belong to the same nodal matrix.

update_between(L_{dest} , L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_r and L_c belong to the same nodal matrix and L_{dest} belongs to a different nodal matrix.

Again, the tasks are partially ordered and a task DAG is used to capture the dependencies. For example, the updating of a block of a nodal matrix from a block

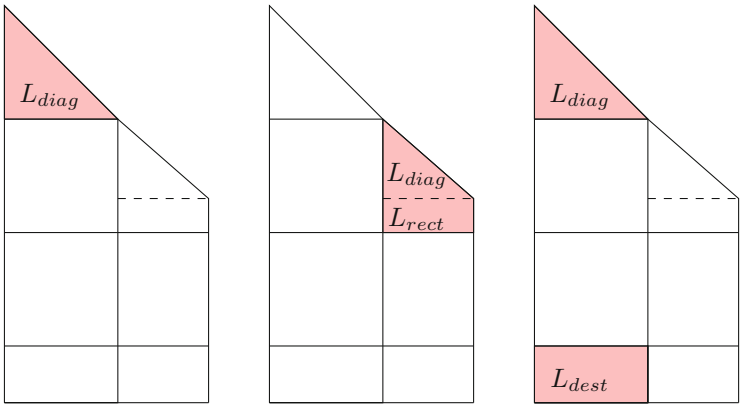


Figure 5.3 An illustration of a blocked nodal matrix with two block columns. The first block on the diagonal is triangular and the second one is trapezoidal. The task **factorize_block** is illustrated on the left and in the centre; the task **solve_block** is illustrated on the right.

column of L that is associated with a descendant of the supernode has to wait until all the relevant rows of the block column are available. At each stage of the factorization, tasks will be executing (in parallel) while others are held (in a stack or pool of tasks) ready for execution.

5.4 Multifrontal Method

The **multifrontal** method is an alternative way to compute a sparse Cholesky factorization. To discuss this popular approach, we use the following result that determines which rows and columns influence particular Schur complements using the terminology of the elimination tree.

Theorem 5.3 (Liu 1990) *Let A be SPD and let $\mathcal{T}$ be its elimination tree. The numerical values of entries in column k of the Cholesky factor L of A only affect the numerical values of entries in column i of L for $i \in anc_{\mathcal{T}}\{k\}$ $(1 \leq k < i \leq n-1)$.*

Proof From (4.1), setting $S^{(1)} = A$, for $k \geq 2$ the $(n-k+1) \times (n-k+1)$ Schur complement $S^{(k)}$ can be expressed as

$$S^{(k)} = S^{(k-1)}_{k:n,k:n} - \begin{pmatrix} l_{k,k-1} \\ \vdots \\ l_{n,k-1} \end{pmatrix} \begin{pmatrix} l_{k,k-1} & \dots & l_{n,k-1} \end{pmatrix} = S^{(k-1)}_{k:n,k:n} - L_{k:n,k-1} L^T_{k:n,k-1}. \tag{5.1}$$

Theorem 4.2 implies that all nonzero off-diagonal entries l_{ik} in column k of L explicitly used in the update (5.1) are such that $i \in anc_{\mathcal{T}}\{k\}$. Considering the

Cholesky factorization as a sequence of Schur complement updates, only columns i with $i \in anc_{\mathcal{T}}\{k\}$ can be influenced numerically by the Schur complement update in the k-th step of the factorization, and the result follows. □

The computation of subsequent Schur complements by adding individual updates as in (5.1) is straightforward; the multifrontal method employs further modifications and enhancements of this basic concept. First, because the vertices of $\mathcal{T}$ are topologically ordered, the order in which the updates are applied progresses up the tree from the leaf vertices to the root vertex. This allows the computation of $S^{(k)}$ to be rewritten as

$$S^{(k)} = A_{k:n,k:n} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} L_{k:n,j} L_{k:n,j}^T,$$

emphasizing the role of $\mathcal{T}$. In place of Schur complements, the multifrontal method uses frontal matrices connected to subtrees of $\mathcal{T}$. Assume $k, k_1, \ldots, k_r$ are the row indices of the nonzeros in column k of L. The **frontal matrix** F_k of the k-th subtree $\mathcal{T}(k)$ of $\mathcal{T}$ is the dense $(r+1) \times (r+1)$ matrix defined by

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \ldots & a_{kk_r} \\ a_{k_1k} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_rk} & 0 & \ldots & 0 \end{pmatrix} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} \begin{pmatrix} l_{kj} \\ l_{k_1j} \\ \vdots \\ l_{k_rj} \end{pmatrix} \begin{pmatrix} l_{kj} & l_{k_1j} & \ldots & l_{k_rj} \end{pmatrix}. \tag{5.2}$$

One step of the Cholesky factorization of F_k can be written as

$$F_k = \begin{pmatrix} l_{kk} & 0 & \ldots & 0 \\ l_{k_1k} & & & \\ \vdots & & I & \\ l_{k_rk} & & & \end{pmatrix} \begin{pmatrix} 1 & & & \\ & & & \\ & & V_k & \\ & & & \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1k} & \ldots & l_{k_rk} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix} \tag{5.3}$$

$$= \begin{pmatrix} l_{kk} \\ l_{k_1k} \\ \vdots \\ l_{k_rk} \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1k} & \ldots & l_{k_rk} \end{pmatrix} + \begin{pmatrix} 0 & & & \\ & & & \\ & & V_k & \\ & & & \end{pmatrix}, \tag{5.4}$$

where V_k is termed a **generated element** (it is also sometimes called an **update matrix** or a **contribution block**). The name "generated element" is because the multifrontal method has its origins in the simpler **frontal method**, which uses a single frontal matrix. The frontal method was originally proposed for problems arising in finite element problems to avoid the need to explicitly construct the system matrix A; it was later generalized to non-element problems. It works with a single frontal matrix and has less scope for parallelisation compared to the multifrontal method; it is no longer widely used.

Equating the last r rows and columns in (5.2) and (5.4) yields

$$V_k = -\sum_{j \in \mathcal{T}(k)} \begin{pmatrix} l_{k_1 j} \\ \vdots \\ l_{k_r j} \end{pmatrix} \begin{pmatrix} l_{k_1 j} & \dots & l_{k_r j} \end{pmatrix}. \tag{5.5}$$

Assume that c_j $(j = 1, \dots, s)$ are the children of k in $\mathcal{T}$. The set $\mathcal{T}(k) \setminus \{k\}$ is the union of disjoint sets of vertices in the subtrees $\mathcal{T}(c_j)$. Each of these subtrees is represented in the overall update by the generated element (5.5). Thus, F_k can be written in an recursive form using the generated elements of the children of k as follows

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \dots & a_{kk_r} \\ a_{k_1 k} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_r k} & 0 & \dots & 0 \end{pmatrix} \boxplus V_{c_1} \boxplus \dots \boxplus V_{c_s}. \tag{5.6}$$

Here, the operation $\boxplus$ denotes the addition of matrices that have row and column indices belonging to subsets of the same set of indices (in this case, $k, k_1, \dots, k_r$); entries that have the same row and column indices are summed. This is referred to as the **extend-add operator**.

Adding a row and column of A and the generated elements into a frontal matrix is called the **assembly**. A variable is **fully summed** if it is not involved in any rows and columns of A that have still to be assembled or in a generated element. Once a variable is fully summed, it can be eliminated. A key feature of the multifrontal method is that the frontal matrices and the generated elements are compressed and stored without zero rows and columns as small dense matrices. Integer arrays are used to maintain a mapping of the local contiguous indices of the frontal matrices to the global indices of A and its factors. Symmetry allows only the lower triangular part of these matrices to be held. Algorithm 5.8 outlines the basic multifrontal method.

ALGORITHM 5.8 Basic multifrontal Cholesky factorization

Input: SPD matrix A and its elimination tree.
Output: Factor L such that $A = LL^T$.

1: **for** $k = 1 : n$ **do**
2: Assemble the frontal matrix F_k using (5.6) ▷ Only the lower triangle is needed
3: Perform a partial Cholesky factorization of F_k using (5.3) to obtain column k of L and the generated element V_k
4: **end for**

ALGORITHM 5.9 Multifrontal Cholesky factorization using the assembly tree
Input: SPD matrix A and its assembly tree.
Output: Factor L such that $A = LL^T$.

1: $nelim = 0$ ▷ $nelim$ is the number of eliminations performed
2: **for** $kb = 1 : nsup$ **do** ▷ $nsup$ is the number of supernodes
3: Assemble the frontal matrix F_{kb}; let l be the number of fully summed variables in F_{kb}
4: Perform a block partial Cholesky factorization of F_{kb} to obtain columns $nelim + 1$ to $nelim + l$ of L and the generated element V_{kb}
5: $nelim = nelim + l$
6: **end for**

We have the following observation.

Observation 5.1 *Each generated element V_k is used only once to contribute to a frontal matrix $F_{parent(k)}$. Furthermore, the index list for the frontal matrix F_k is the set of row indices of the nonzeros in column k of the Cholesky factor L.*

In practical implementations, efficiency is improved by using the assembly tree (Section 4.6) because it allows more than one elimination to be performed at once. This is outlined in Algorithm 5.9. Here kb is used to index the frontal matrix on the kb-th step ($1 \le kb \le nsup$).

As an example, consider the matrix and its assembly tree given in Figure 4.10. The $nsup = 5$ supernodes are $\{1, 2\}, 3, 4, 5, \{6, 7, 8, 9\}$ and so variables 1 and 2 can be eliminated together on the first step. Assembling rows/columns 1 and 2 of the original matrix, the frontal matrix F_1 and generated element V_1 have the structure

$$F_1 = \begin{array}{c} \\ 1 \\ 2 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & & \\ * & * & & \end{pmatrix} \end{array}, \quad V_1 = \begin{array}{c} \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 8 & 9 \end{array} \\ \begin{pmatrix} f & f \\ f & f \end{pmatrix} \end{array},$$

where f denotes fill-in entries (only the lower triangular entries are stored in practice). Similarly,

$$F_2 = \begin{array}{c} \\ 3 \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 4 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \end{pmatrix} \end{array}, \quad V_2 = \begin{array}{c} \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 8 \end{array} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{array}.$$

The frontal matrix F_3 and generated element V_3 are given by

$$F_3 = \begin{array}{c} \\ 4 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array} \leftrightarrow V_2, \quad V_3 = \begin{array}{c} \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array}.$$

Then

$$F_4 = \begin{array}{c} \\ 5 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 5 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array}, \quad V_4 = \begin{array}{c} \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array},$$

and, finally, with $kb = 5$ we have

$$F_5 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & & * & * \\ & * & & * \\ * & & * & \\ * & * & & * \end{pmatrix} \end{array} \leftrightarrow V_4 \leftrightarrow V_3 \leftrightarrow V_1.$$

An important implementation detail is how and where to store the generated elements. The partial factorization of F_{kb} at supernode kb can be performed once the partial factorizations at all the vertices belonging to the subtree of the assembly tree with root vertex kb are complete. If the vertices of the assembly tree are ordered using a depth-first search, the generated elements required at each stage are the most recently computed ones amongst those that have not yet been assembled. This makes it convenient to use a stack. This affects the order in which the variables are eliminated but in exact arithmetic, the results are identical.

Nevertheless, the memory demands of the multifrontal method can be very large. Not only is it dependent on the initial ordering of A but the ordering of the children of a vertex in the assembly tree can significantly affect the required stack size. Some implementations target limiting stack storage requirements. An attractive feature of the multifrontal method is that the generated elements can be held using auxiliary storage (in files on disk) to restrict the in-core memory requirements, allowing larger problems to be solved than would otherwise be possible.

5.5 Parallelism Within Sparse Cholesky Factorizations

Sparse Cholesky factorizations use supernodes and task graphs (the assembly tree for the multifrontal method) to control the computation. The number of rows and columns in a supernode typically increases away from the leaf vertices and towards

the root of the task graph because a supernode accumulates fill-in from its ancestors in the task graph. As a result, tasks that are relatively close to the root tend to have more work associated with them. On the other hand, the width of the task graph shrinks close to the root. In other words, a typical task graph for sparse matrix factorization tends to have a large number of small independent tasks close to the leaf vertices, but a small number of large tasks close to the root. An ideal parallelization strategy that would match the characteristics of the problem is as follows. Initially, assign the relatively plentiful independent tasks at or near the leaf vertices to parallel threads or processes. This is called **task** or **tree level** parallelism; it is influenced by the ordering of A. As tasks complete, other tasks become available and are scheduled similarly. This continues while there are enough independent tasks to keep all the threads or processes busy. When the number of available parallel tasks becomes too small, the only way to keep the latter busy is to assign more than one to a task. This is termed **node level** parallelism. The number of threads or processes working on individual tasks should increase as the number of parallel tasks decreases. Eventually, all threads or processes are available to work on the root task. The computation corresponding to the root task is equivalent to factoring a dense matrix of the size of the root supernode.

The multifrontal method is often the formulation of choice for highly parallel implementations of sparse matrix factorizations. This is because of its natural data locality (most of the work of the factorization is performed in the dense frontal matrices) and the ease of synchronization that it permits. In general, each supernode is updated by multiple other supernodes and it can potentially update many other supernodes during the course of the factorization. If implemented naively, all these updates may require excessive locking and synchronization in a shared-memory environment or generate excessive message-traffic in a distributed environment. In the multifrontal method, the updates are accumulated and channelled along the paths from the leaf vertices of the assembly tree to its root vertex. This gives a manageable structure to the potentially haphazard interaction among the tasks.

In Section 1.2.4, bit compatibility was discussed. While different orderings of the children of a vertex in the assembly tree do not affect the total number of floating-point operations that are performed in the multifrontal method, in finite-precision arithmetic changing the order of the assemblies into the frontal matrices can lead to slightly different results. Given that the number of children is typically small and that large matrices can be partitioned such that summations can be safely performed in parallel, the overhead in the multifrontal method of enforcing a defined order of the summation is relatively small. By contrast, in the supernodal approach, for each data block a number of matrices equal to the block dependencies are summed. Given the relatively large numbers (several thousand) for many nodes, an enforced order may be detrimental to efficiency.

5.6 Notes and References

Exploiting panels and blocks in both left- and right-looking Cholesky factorization algorithms is extremely important. The development of sparse supernodal factorizations for uniprocessors and multiprocessors in the 1990s is discussed by Ng & Peyton (1993a,b); Rothberg & Gupta (1993) presents an early comparison of various types of block Cholesky factorizations. PaStiX of Hénon et al. (2002) is a parallel left-looking supernodal solver that is primarily designed for positive definite systems. Rotkin & Toledo (2004) introduce a hybrid left-looking/right-looking algorithm and Rozin & Toledo (2005) show that no sparse numerical factorization is uniformly better than the others. An up-looking approach, which is fast in practice for very sparse matrices, is employed in the widely used CHOLMOD solver of Chen et al. (2008). The package `HSL_MA87` implements a sparse DAG-based Cholesky factorization for shared-memory architectures; further details of the approach can be found in Hogg et al. (2010).

The multifrontal algorithm has its origins in the simpler frontal method of Irons (1970), which was developed by the civil engineering community from the 1960s onwards to solve the linear systems that arise within finite element methods. At a time when the main memory of even the most powerful computers was extremely limited, the frontal method was heavily influenced by the need to minimize the memory requirements of the linear solver. It was initially designed for SPD banded linear systems and was subsequently extended to nonsymmetric problems by Hood (1976) and to the symmetric indefinite case by Reid (1981); Duff (1984) generalizes the approach to non-element problems. The frontal method proceeds by alternating the assembly of the finite elements into a single dense frontal matrix with the elimination and update of variables. Once variables have been eliminated they are no longer needed during the factorization and so they are removed from the frontal matrix and stored elsewhere (for example, not in main memory but on an external disk) until needed during the solve phase. This frees up space to accommodate the next element to be assembled. Because the frontal method does not use the assembly tree, the frontal matrix can be much larger than those in the multifrontal method, leading to higher operation counts but also allowing the use of BLAS with larger block sizes. Efficient implementations were developed up until the late 1990s. For example, by Duff & Scott (1996, 1999), who provide a package `MA62` for SPD problems in element form that employs a single array of length n, exploits Level 3 BLAS, and holds the computed factors on disk; a coarse-grained parallel version is also available, see Duff & Scott (1994) and Scott (2001).

The frontal method and the work of Speelpenning (1978) on the so-called generalized element method led to the development by Duff & Reid (1983) of the multifrontal method for solving general symmetric systems (including systems in element form). A detailed matrix-based explanation is given in Liu (1992). The method is implemented in some of the most important sparse direct solvers. The MUMPS (2022) package, which has been actively developed over many years, provides a state-of-the-art distributed memory general-purpose multifrontal solver

that uses shared-memory parallelism within each MPI process. Other important parallel multifrontal solvers are HSL_MA97 (Hogg & Scott, 2013b) and WSMP (2020), while the serial package MA57 of Duff (2004) (which superseded the original and perhaps most well-known multifrontal solver MA27 of Duff & Reid, (1983)) remains very popular. An attractive feature of HSL_MA97 is that it computes bit-compatible solutions. HSL_MA77 of Reid & Scott (2009) is designed to minimize memory requirements by allowing the factors and the multifrontal stack to be efficiently held outside of main memory (an option that is also offered by MUMPS). In common with earlier frontal solvers, HSL_MA77 allows the user to input the system matrix in element form (that is, A is not explicitly assembled for problems coming from finite element applications but is input one element at a time).

The use of GPUs is well-suited to a multifrontal or supernodal factorization because these approaches rely on regular block computations within dense submatrices. Implementing the multifrontal method (including for symmetric indefinite matrices) on GPU architectures is discussed in Hogg et al. (2016), while Lacoste et al. (2012) and Rennich et al. (2016) present GPU-accelerated supernodal factorizations. Discussion of the use of GPUs within direct solvers is included in the comprehensive survey of Davis et al. (2016).

Chapter 6
Sparse LU Factorizations

The closer one looks, the more subtle and remarkable Gaussian elimination appears – Trefethen (1985)

Gaussian elimination is living mathematics. It has mutated successfully for the last two hundred years to meet changing social needs – Grcar (2011)

This chapter considers the LU factorization of a general nonsymmetric nonsingular sparse matrix A. In practice, numerical pivoting for stability and/or ordering of A to limit fill-in in the factors is often needed and the computed factorization is then of a permuted matrix PAQ. Pivoting is discussed in Chapter 7 and ordering algorithms in Chapter 8.

6.1 Sparse LU Factorizations and Their Graph Models

In Chapter 4, graphs were used to describe structural changes during a sparse Cholesky factorization. In particular, the elimination tree was shown to play a key role and, in the previous chapter, the use of DAGs was discussed. For general matrices, there are a number of ways that graphs can be employed.

6.1.1 Use of Elimination DAGs

The first graph model uses the elimination DAGs associated with L and U that were defined in (2.1)–(2.2). The following observation, which is illustrated in Figure 6.1, generalizes Observation 4.1 to nonsymmetric matrices.

Observation 6.1 *If $i > j$ and $u_{ji} \neq 0$, then the* **column replication principle** *states*

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_6

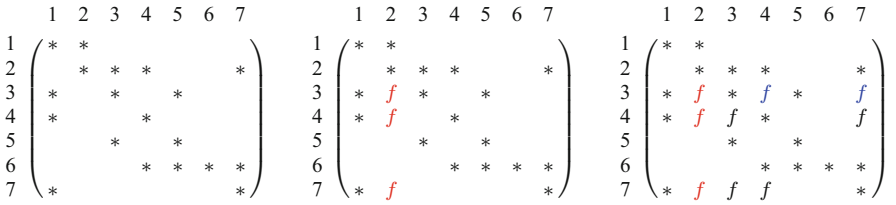


Figure 6.1 An illustration of the column and row replication principles of sparse LU factorizations. The matrix A is on the left. In the centre, we show in red the filled entries in L resulting from the replication of the first column in the second column because $u_{12} \neq 0$. On the right, we show in blue the filled entries in U resulting from the replication of the second row in the third row because $l_{32} \neq 0$. Other filled entries resulting from subsequent steps of the factorization are denoted in black.

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\},$$

that is, the pattern of column j of L (rows i to n) is replicated in the pattern of column i of L. Analogously, if $i > j$ and $l_{ij} \neq 0$, then the **row replication principle** *states*

$$\mathcal{S}\{U_{j,i:n}\} \subseteq \mathcal{S}\{U_{i,i:n}\},$$

that is, the pattern of row j of U (columns i to n) is replicated in the pattern of row i of U.

Algorithm 6.1 outlines a basic sparse LU factorization. Here it is assumed that A is factorizable so that pivoting is not needed. The remainder of this chapter looks at techniques that can be used to develop the approach into an efficient one.

The following theorem formulates the recursive column replication and the replication of nonzeros along rows of L using directed paths in $\mathcal{G}(U)$; an analogous result holds for the rows of U and directed paths in $\mathcal{G}(L^T)$.

Theorem 6.1 (Gilbert & Liu 1993) *Assume that for some $k < j$ there is a directed path $k \overset{\mathcal{G}(U)}{\Longrightarrow} j$. Then*

$$\mathcal{S}\{L_{j:n,k}\} \subseteq \mathcal{S}\{L_{j:n,j}\}. \tag{6.1}$$

Moreover, if $l_{ik} \neq 0$ for some $i > j$, then $l_{is} \neq 0$ for all vertices s on this path.

The next two theorems generalize Theorem 4.3 to A being a general nonsymmetric matrix.

Theorem 6.2 (Gilbert & Liu 1993) *If $a_{ij} = 0$ and $i > j$, then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $a_{ik} \neq 0$ and there is a directed path $k \overset{\mathcal{G}(U)}{\Longrightarrow} j$.*

ALGORITHM 6.1 Basic sparse LU factorization
Input: Nonsymmetric and factorizable matrix $A = L_A + D_A + U_A$.
Output: LU factorization $A = LU$.

```
1: L = I + L_A                          ▷ Identity plus strictly lower triangular part of A
2: U = D_A + U_A                        ▷ Diagonal plus strictly upper triangular part of A
3: for k = 1 : n − 1 do
4:     for i ∈ {i > k | l_ik ≠ 0} do
5:         l_ik = l_ik / u_kk
6:         U_{i,i:n} = U_{i,i:n} − U_{k,i:n} l_ik        ▷ Update row i of U
7:     end for
8:     for j ∈ {j > k | u_kj ≠ 0} do
9:         L_{j+1:n,j} = L_{j+1:n,j} − L_{j+1:n,k} u_kj  ▷ Update column j of L
10:    end for
11: end for
```

Theorem 6.3 (Gilbert & Liu 1993) *If $a_{ij} = 0$ and $i < j$, then there is a filled entry $u_{ij} \neq 0$ if and only if there exists $k < i$ such that $a_{kj} \neq 0$ and there is a directed path $k \xrightarrow{\mathcal{G}(L^T)} i$.*

Theorems 6.2 and 6.3 are demonstrated in Figure 6.2. Consider the directed path $1 \to 3 \to 5 \to 6$ in $\mathcal{G}(U)$. Existence of this path implies the fill-in in L, first in

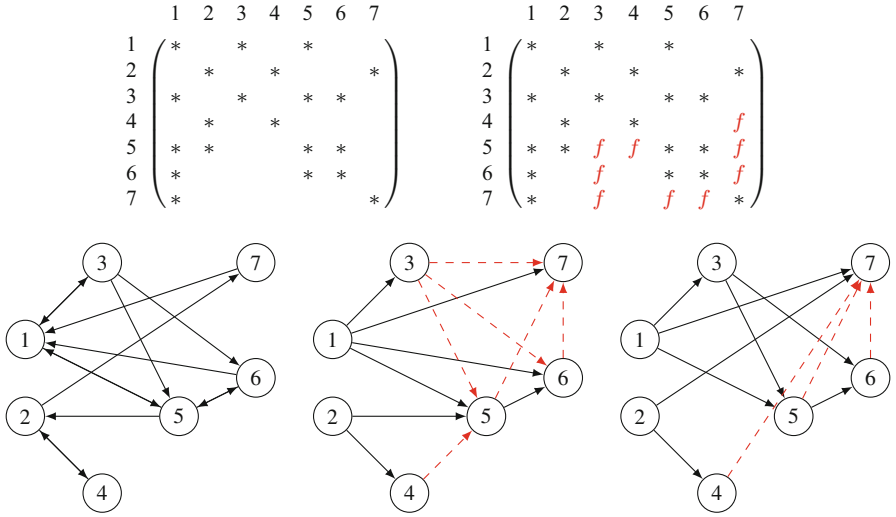


Figure 6.2 The sparsity patterns of A (left) and $L+U$ (right) together with the graphs $\mathcal{G}(A)$ (left), $\mathcal{G}(L^T)$ (centre) and $\mathcal{G}(U)$ (right). The filled entries are denoted by f and the corresponding edges are the red dashed lines.

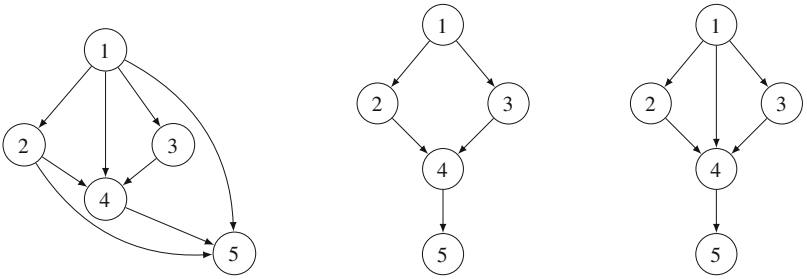


Figure 6.3 Example to show the transitive reduction of a DAG. $\mathcal{G}$ is on the left, its transitive reduction $\mathcal{G}^0$ is in the centre, and one possible $\mathcal{G}'$ that is equireachable with $\mathcal{G}$ is on the right.

column 3, then in columns 5 and 6. Similarly, the directed path $2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ in $\mathcal{G}(L^T)$ implies fill-in at positions (4, 7), (5, 7) and (6, 7) in U.

6.1.2 Transitive Reduction and Equireachability

To employ $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$ in efficient algorithms, they need to be simplified. One possibility is to use transitive reductions that are sparser and preserve reachability within the graphs. A subgraph $\mathcal{G}^0 = (\mathcal{V}, \mathcal{E}^0)$ is a **transitive reduction** of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if the following conditions hold:

($T1$) there is a path from vertex i to vertex j in $\mathcal{G}$ if and only if there is a path from i to j in $\mathcal{G}^0$ (reachability condition), and
($T2$) there is no subgraph with vertex set $\mathcal{V}$ that satisfies ($T1$) and has fewer edges (minimality condition).

A transitive reduction is unique for a DAG, as shown in the following theorem and illustrated in Figure 6.3.

Theorem 6.4 (Aho et al. 1972) *Let $\mathcal{G}$ be a DAG. The transitive reduction $\mathcal{G}^0$ of $\mathcal{G}$ is unique and is the subgraph that has an edge for every path in $\mathcal{G}$ and has no proper subgraph with this property.*

If $\mathcal{S}\{A\}$ is symmetric, then, as illustrated in Figure 6.4, the role of the transitive reduction is played by the elimination tree.

Theorem 6.5 (Liu 1990; Eisenstat & Liu 2005a) *If A is symmetrically structured, then the transitive reduction of the DAG $\mathcal{G}(L^T)$ (= $\mathcal{G}(U)$) is the elimination tree $\mathcal{T}(A)$.*

Obtaining the exact transitive reduction of a DAG can be expensive. Instead, approximate reductions that drop the minimality condition may be computed. A directed graph $\mathcal{G}'$ with the same vertex set as $\mathcal{G}$ that satisfies condition ($T1$) is said

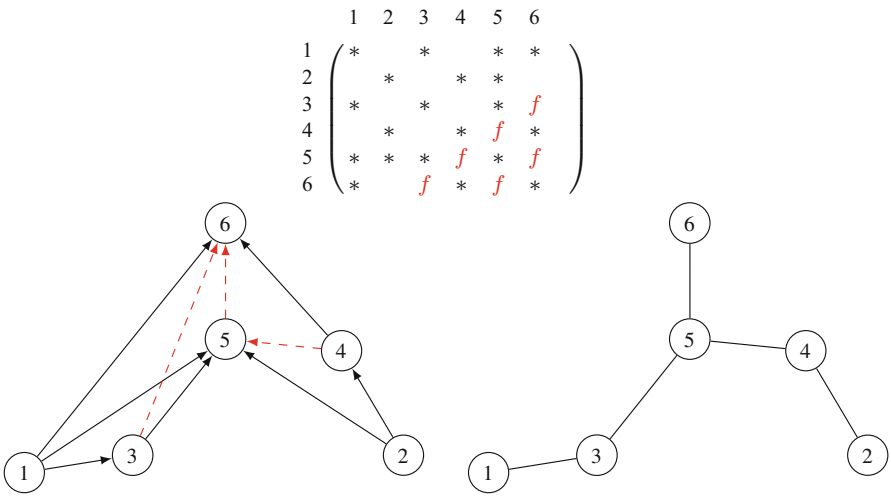


Figure 6.4 The sparsity patterns of $L+U$ of a symmetrically structured A together with the DAG $\mathcal{G}(L^T)$ (left) and the elimination tree $\mathcal{T}(A)$ (right). The filled entries are denoted by f and the corresponding edges are the red dashed lines. It is straightforward to see that $\mathcal{T}(A)$ is obtained as the transitive reduction of $\mathcal{G}(L^T)$.

to be **equireachable** with $\mathcal{G}$. The next result is a simplification of Theorem 6.1; an analogous result holds for the sparsity patterns of the rows of U.

Theorem 6.6 (Gilbert & Liu 1993) *Assume $\mathcal{G}'$ is equireachable with $\mathcal{G}(U)$ and for some $k < j$ there is a directed path $k \overset{\mathcal{G}'}{\Rightarrow} j$. Then (6.1) holds. Moreover, if $l_{ik} \neq 0$ for some $i > j$, then $l_{is} \neq 0$ for all vertices s on the directed path.*

Equireachability enables sparse triangular linear systems to be solved more efficiently. In Chapter 5, Theorem 5.2 describes how to obtain the sparsity pattern $\mathcal{J}$ of the solution of a lower triangular system using paths in $\mathcal{G}(L^T)$. This graph can be replaced by any graph that is equireachable with $\mathcal{G}(L^T)$. Equireachability also allows Theorems 6.2 and 6.3 to be rewritten using paths in a graph $\mathcal{G}'$ that is equireachable with $\mathcal{G}$.

Theorem 6.7 (Gilbert & Liu 1993) *If $a_{ij} = 0$ and $i > j$, then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $a_{ik} \neq 0$ and a directed path $k \overset{\mathcal{G}'(U)}{\Longrightarrow} j$, where $\mathcal{G}'(U)$ is equireachable with $\mathcal{G}(U)$.*

Theorem 6.8 (Gilbert & Liu 1993) *If $a_{ij} = 0$ and $i < j$, then there is a filled entry $u_{ij} \neq 0$ if and only if there exists $k < i$ such that $a_{kj} \neq 0$ and a directed path $k \overset{\mathcal{G}'(L^T)}{\Longrightarrow} i$, where $\mathcal{G}'(L^T)$ is equireachable with $\mathcal{G}(L^T)$.*

Figure 6.5 depicts $\mathcal{G}(U)$ and $\mathcal{G}'(U)$ for the matrix in Figure 6.2.

A description of the sparsity patterns of the columns of L can be obtained from the Schur complement (3.2) as follows:

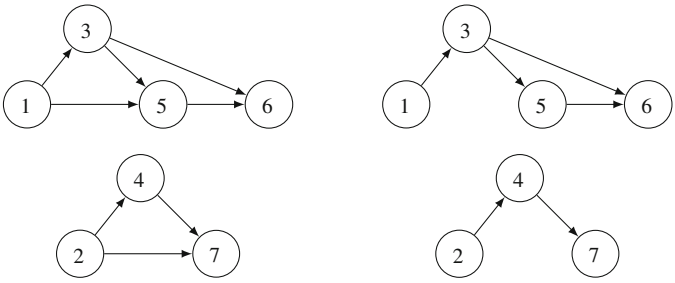


Figure 6.5 The DAG $\mathcal{G}(U)$ for the matrix from Figure 6.2 (left) and $\mathcal{G}'(U)$ which is equireachable with $\mathcal{G}(U)$ (right).

$$\mathcal{S}\{L_{j:n,j}\} = \mathcal{S}\{A_{j:n,j}\} \bigcup_{k<j, u_{kj}\neq 0} \mathcal{S}\{L_{j:n,k}\}, \quad 1 \leq j \leq n.$$

Theorem 6.7 implies that not all the terms in this union are needed to obtain $\mathcal{S}\{L_{j:n,j}\}$. This result is given in Theorem 6.9, which shows how $\mathcal{S}\{L\}$ can be computed by columns if $\mathcal{G}'(U)$ that is equireachable with $\mathcal{G}(U)$ is known.

Theorem 6.9 (Gilbert & Liu 1993) *If $\mathcal{G}'(U)$ is equireachable with $\mathcal{G}(U)$, then*

$$\mathcal{S}\{L_{j:n,j}\} = \mathcal{S}\{A_{j:n,j}\} \bigcup_{(k\rightarrow j)\in\mathcal{E}(\mathcal{G}'(U))} \mathcal{S}\{L_{j:n,k}\}, \quad 1 \leq j \leq n. \tag{6.2}$$

Proof Consider an edge $(k \rightarrow j)$ in $\mathcal{G}(U)$ but not in $\mathcal{G}'(U)$. Repeatedly applying (6.1) along the directed path $k \overset{\mathcal{G}'(U)}{\Longrightarrow} j$, we see that $L_{j:n,k}$ is contained in the right-hand side of (6.2) and therefore $\mathcal{S}\{L_{j:n,j}\}$ is contained in the right-hand side of (6.2). Because the right-hand side of (6.2) is trivially contained in the left-hand side, the result follows. □

An analogous result holds for the rows of U.

Theorem 6.10 (Gilbert & Liu 1993) *If $\mathcal{G}'(L)$ is equireachable with $\mathcal{G}(L)$, then*

$$\mathcal{S}\{U_{i,i:n}\} = \mathcal{S}\{A_{i,i:n}\} \bigcup_{(k\rightarrow i)\in\mathcal{E}(\mathcal{G}'(L^T))} \mathcal{S}\{U_{k,i:n}\}, \quad 1 \leq i \leq n.$$

As an example of Theorem 6.9, consider the matrix in Figure 6.2. Because $(3 \rightarrow 5)$ is the only edge of $\mathcal{G}'(U)$ in the union on the right-hand side of (6.2), $\mathcal{S}\{L_{5:7,5}\}$ is given by

$$\mathcal{S}\{L_{5:7,5}\} = \mathcal{S}\{A_{5:7,5}\} \cup \mathcal{S}\{L_{5:7,3}\}.$$

We can see this from the graph $\mathcal{G}'(U)$ in Figure 6.5 (top right).

6.1.3 Symbolic LU Factorizations Using DAGs

Factorization by bordering can be used to obtain $\mathcal{S}\{L\}$ by rows and $\mathcal{S}\{U\}$ by columns. Assume the sparsity patterns of the first $k-1$ rows of L and the first $k-1$ columns of U ($1 < k \leq n$) have been computed. At step k, the factors satisfy

$$A_{1:k,1:k} = \begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix}. \tag{6.3}$$

Equating terms for the (2, 1) block, row k of L satisfies

$$L_{k,1:k-1}U_{1:k-1,1:k-1} = A_{k,1:k-1},$$

or, equivalently, if y denotes the off-diagonal part of the column k of L^T, then it is the solution of the lower triangular system

$$U^T_{1:k-1,1:k-1}y = A^T_{k,1:k-1}.$$

From Theorem 5.2, the sparsity pattern of y is the set of all vertices reachable in the DAG $\mathcal{G}(U_{1:k-1,1:k-1})$ (or in a graph that is equireachable with it) from the nonzeros in $A_{k,1:k-1}$. Similarly, equating terms in (6.3) for the (1, 2) block, column k of U satisfies

$$L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}.$$

Again, its sparsity pattern can be determined using Theorem 5.2 and the DAG $\mathcal{G}(L^T_{1:k-1,1:k-1})$. The diagonal entry u_{kk} is then computed as $a_{kk} - L_{k,1:k-1}U_{1:k-1,k}$. This shows that determining the sparsity patterns of L and U and computing their numerical values is coupled: computation of the factors needs be mutually interleaved because computing part of one requires information from a part of the other.

6.1.4 Graph Pruning

Consider the matrices in Figure 6.6. The one in the centre is the same as the one on the left except that the entries in positions (4, 6) and (6, 4) have been removed (that is, pruned). Both matrices have the same sets of reachable vertices in $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$. This suggests how to find $\mathcal{G}'(L^T)$ and $\mathcal{G}'(U)$ that are equireachable with $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$, respectively.

Theorem 6.11 (Eisenstat & Liu 1992) *If for some $j < s$ both $l_{sj} \neq 0$ and $u_{js} \neq 0$, then there are no edges $(j \to k)$ with $k > s$ in the transitive reductions of $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$.*

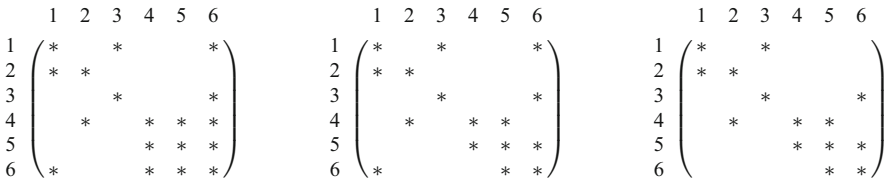


Figure 6.6 An example of symmetric pruning. On the left is $\mathcal{S}\{L+U\}$. In the centre is the reduced sparsity pattern obtained by symmetric pruning. On the right is the reduced sparsity pattern that results from symmetric path pruning.

Proof Let $(j \to k)$ be an edge of $\mathcal{G}(U)$, that is, $u_{jk} \neq 0$. Because $l_{sj} \neq 0$ and $u_{jk} \neq 0$ implies that $u_{sk} \neq 0$, there is a path $j \to s \to k$ in $\mathcal{G}(U)$ and the edge $(j \to k)$ does not belong to the transitive reduction of $\mathcal{G}(U)$. The result for $\mathcal{G}(L^T)$ can be seen analogously. □

This theorem implies that if for some $s > 1$ there are edges

$$j \xrightarrow{\mathcal{G}(L^T)} s \quad \text{and} \quad j \xrightarrow{\mathcal{G}(U)} s,$$

then all edges $(j \to k)$ in $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$ with $k > s$ can be pruned. The resulting DAGs $\mathcal{G}'(U)$ and $\mathcal{G}'(L^T)$ have fewer edges and are equireachable with $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$, respectively. The removal of redundant edges based on Theorem 6.11 is called **symmetric pruning**.

There are other ways to perform pruning. For example, if for some $s > 1$ there are paths

$$j \xRightarrow{\mathcal{G}(L^T)} s \quad \text{and} \quad j \xRightarrow{\mathcal{G}(U)} s,$$

then for all $k > s$ **symmetric path pruning** removes the edges $(j \to k)$ from $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$. Consider again Figure 6.6. In the centre is the sparsity pattern after symmetric pruning and on the right is the reduced sparsity pattern that results from symmetric path pruning. The edge $(1 \to 6)$ is not required in $\mathcal{G}'(L^T)$ or $\mathcal{G}'(U)$ because there are paths

$$1 \xrightarrow{\mathcal{G}(L^T)} 2 \xrightarrow{\mathcal{G}(L^T)} 4 \xrightarrow{\mathcal{G}(L^T)} 5 \xrightarrow{\mathcal{G}(L^T)} 6 \quad \text{and} \quad 1 \xrightarrow{\mathcal{G}(U)} 3 \xrightarrow{\mathcal{G}(U)} 6.$$

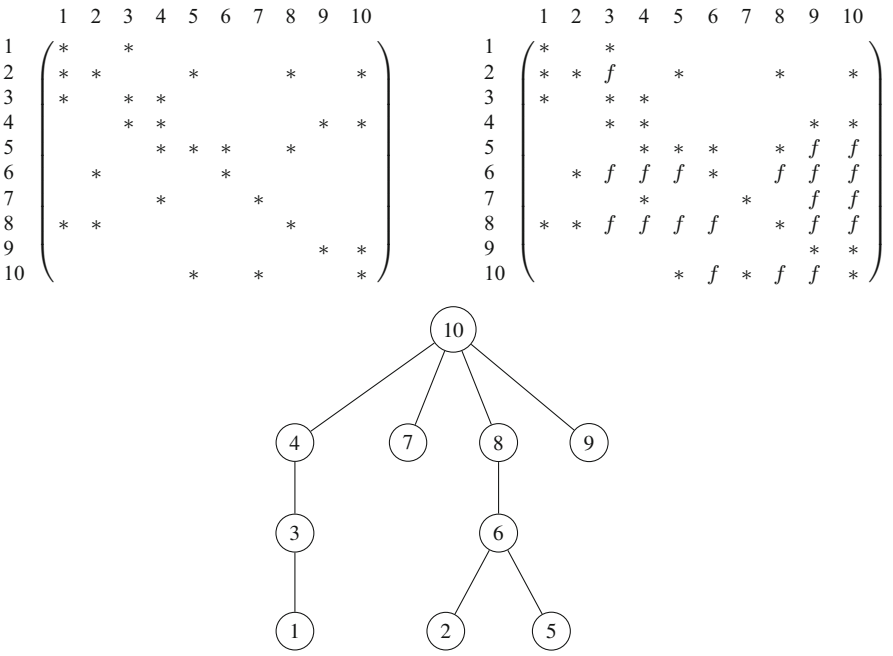


Figure 6.7 An example of the sparsity pattern of a nonsymmetric matrix A (left), $\mathcal{S}\{L+U\}$ with filled entries denoted by f (right) and its elimination tree.

6.1.5 *Elimination Trees for Nonsymmetric Matrices*

The elimination DAGs $\mathcal{G}(L)$ and $\mathcal{G}(U)$ can be combined into a single structure called the **nonsymmetric elimination tree** in which edges are replaced by paths. This can be advantageous because it is more compact. From (4.3), if $\mathcal{S}\{A\}$ is symmetric, then its elimination tree is defined in terms of the mapping

$$parent(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}.$$

The condition $l_{ij} \neq 0$ is equivalent to $i \xrightarrow{\mathcal{G}(L)} j \xrightarrow{\mathcal{G}(L^T)} i$. In the nonsymmetric case, the definition can be generalized using directed paths

$$parent(j) = \min\{i \mid i > j \text{ and } i \overset{\mathcal{G}(L)}{\Longrightarrow} j \overset{\mathcal{G}(U)}{\Longrightarrow} i\}. \tag{6.4}$$

This is illustrated in Figure 6.7. Vertices 6, 8, and 10 are the only ones with cycles of the form

$$i \overset{\mathcal{G}(L)}{\Longrightarrow} 2 \overset{\mathcal{G}(U)}{\Longrightarrow} i,$$

namely,

ALGORITHM 6.2 Basic computation of the elimination tree for nonsymmetric A
Input: Digraph $\mathcal{G}(A)$.
Output: The elimination tree given by the mapping $parent$.

```
1:  parent(1 : n) = 0
2:  for i = 1 : n do
3:      Find the vertex set V_C of the strong component of G(A_{1:i,1:i}) that contains i
4:      for j ∈ V_C \ {i} do
5:          if parent(j) = 0 then
6:              parent(j) = i
7:          end if
8:      end for
9:      parent(i) = 0
10: end for
```

$$6 \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} 5 \xrightarrow{\mathcal{G}(U)} 6, \quad 8 \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} 8 \quad \text{and} \quad 10 \xrightarrow{\mathcal{G}(L)} 6 \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} 10.$$

In this example, $parent(2) = 6$.

Theorem 6.12, which can be regarded as a generalization of Corollary 4.6, shows how the elimination tree for nonsymmetric A can be constructed.

Theorem 6.12 (Eisenstat & Liu 2005a) *Let A be a nonsymmetric matrix. $i = parent(j)$ if and only if $i > j$ and i is the smallest vertex that belongs to the same strong component of $\mathcal{G}(A_{1:i,1:i})$ as vertex j.*

This result is employed in Algorithm 6.2. The complexity of finding the strong components of a digraph with m edges and n vertices is $O(n + m)$. Hence, the complexity of Algorithm 6.2 is $O(nz(A)\, n)$. More sophisticated approaches with complexity $O(nz(A) \log n)$ exist.

To illustrate Algorithm 6.2, consider the matrix and its elimination tree depicted in Figure 6.7. The main loop sets the first nonzero value in the array $parent$ when $i = 3$ because this is the first i for which the set $\mathcal{V}_C \setminus \{i\}$ is non empty; it is equal to $\{1\}$ and thus $parent(1) = i = 3$. For $i = 4$, the vertex set $\{1, 3, 4\}$ forms a strong component of $\mathcal{G}(A_{1:4,1:4})$ and so $parent(3) = 4$. For $i = 5$, the single vertex $\{5\}$ is a strong component of $\mathcal{G}(A_{1:5,1:5})$ and, therefore, 5 is not a parent of any other vertex (it is a leaf vertex). $\mathcal{G}(A_{1:6,1:6})$ has two strong components with vertex sets $\{1, 3, 4\}$ and $\{2, 5, 6\}$. $i = 6$ belongs to the second of these and thus the algorithm sets $parent(j) = i = 6$ for $j = 2$ and 5.

An attractive idea for constructing $\mathcal{S}\{L + U\}$ and subsequently computing the LU factorization is based on using the **column elimination tree** $\mathcal{T}(A^T A)$.

Theorem 6.13 (George & Ng 1985; Grigori et al. 2009) *Assume all the diagonal entries of A are nonzero and let $\widetilde{L}\widetilde{L}^T$ be the Cholesky factorization of $A^T A$. Then for any row permutation matrix P such that $PA = LU$ the following holds:*

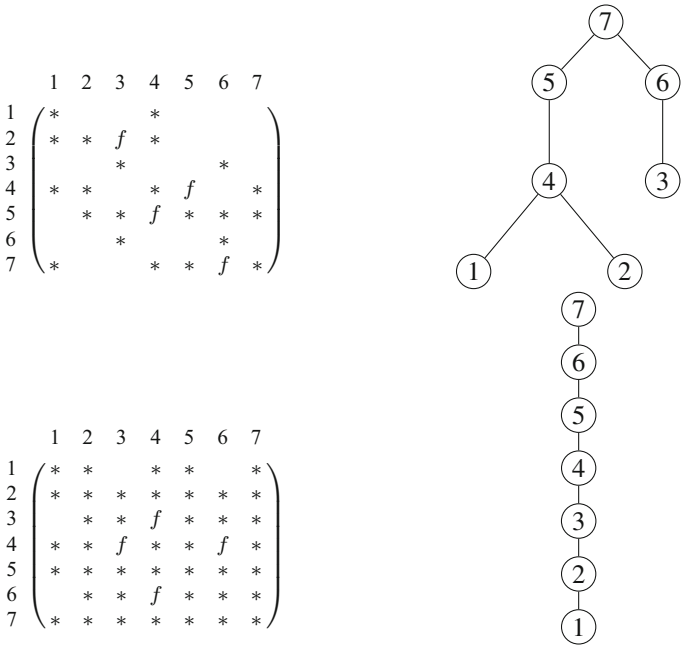


Figure 6.8 The sparsity patterns of A and $L + U$ (top) and of $A^T A$ and $\widehat{L} + \widehat{L}^T$, where $A^T A = \widehat{L}\widehat{L}^T$ (bottom). Filled entries are denoted by f. The corresponding elimination trees are also given.

$$\mathcal{S}\{L + U\} \subseteq \mathcal{S}\{\widehat{L} + \widehat{L}^T\}.$$

An important feature of Theorem 6.13 is that it holds for *any* row permutation matrix P applied to A. This allows partial pivoting (Section 3.1.2) to be used. The following result states that $\mathcal{T}(A^T A)$ represents the potential dependencies among the columns in an LU factorization and that for strong Hall matrices no tighter prediction is possible from the sparsity structure of A.

Theorem 6.14 (Gilbert & Ng 1993) *If $PA = LU$ is any factorization of A with partial pivoting, then the following hold.*

1. *If vertex i is an ancestor of vertex j in $\mathcal{T}(A^T A)$, then $i > j$.*
2. *If $l_{ij} \neq 0$, $i \neq j$, then vertex i is an ancestor of vertex j in $\mathcal{T}(A^T A)$.*
3. *If $u_{ij} \neq 0$, $i \neq j$, then vertex j is an ancestor of vertex i in $\mathcal{T}(A^T A)$.*
4. *Suppose in addition that A is a strong Hall matrix. If $l = parent(k)$ in $\mathcal{T}(A^T A)$, then there are values of the nonzero entries of A for which $u_{kl} \neq 0$.*

Figure 6.8 illustrates the differences in the sparsity patterns of A and $A^T A$ and of their factors; the corresponding elimination trees are also given. This reveals a potential problem with the column elimination tree: $\mathcal{S}\{A^T A\}$ can have significantly more entries than $\mathcal{S}\{L + U\}$. An extreme example is when A has one or more dense rows because $A^T A$ is then fully dense.

6.1.6 *Supernodes in LU Factorizations*

Supernodes group together columns of the factors with the same nonzero structure, allowing them to be treated as a dense submatrix for storage and computation. When solving SPD systems, supernodes can be determined during the symbolic phase. For nonsymmetric matrices, supernodes are harder to characterize. The need to incorporate pivoting means it may not be possible to predict the sparsity structures of the factors before the numerical factorization and they must be identified on-the-fly. While there are several possible ways to define supernodes, the simplest (which is widely used in practice) follows the symmetric case and defines a supernode to be a set of contiguously numbered columns of L with the triangular diagonal block treated as dense and the columns as having the same structure below the diagonal block.

In a Cholesky solver, fundamental supernodes (Section 4.6.1) are made contiguous by symmetrically permuting the matrix according to a postordering of its elimination tree; this does not change the sparsity of the Cholesky factor. For nonsymmetric A, before the numerical factorization, $\mathcal{T}(A^TA)$ can be constructed and the columns of A then permuted according to its postordering to bring together supernodes. The following result extends Theorem 4.9.

Theorem 6.15 (Li 1996) *Let A have column elimination tree $\mathcal{T}(A^TA)$. Let p be a permutation vector such that if p_i is an ancestor of p_j in $\mathcal{T}(A^TA)$, then $i > j$. Let P be the permutation matrix corresponding to p and let $\widehat{A} = PAP^T$. Then $\mathcal{T}(\widehat{A}^T\widehat{A})$ is isomorphic to $\mathcal{T}(A^TA)$; in particular, relabelling each vertex i of $\mathcal{T}(\widehat{A}^T\widehat{A})$ as p_i yields $\mathcal{T}(A^TA)$. If, in addition, $\widehat{A} = \widehat{L}\widehat{U}$ is an LU factorization without pivoting then $P^T\widehat{L}P$ and $P^T\widehat{U}P$ are lower triangular and upper triangular matrices, respectively, so that $A = (P^T\widehat{L}P)(P^T\widehat{U}P)$ is also an LU factorization.*

In practice, for many matrices the average size of a supernode is only 2 or 3 columns and many comprise a single column. Larger artificial supernodes may be created by merging vertex j with its parent vertex i in $\mathcal{T}(A^TA)$ if the subtree rooted at i has fewer than some chosen number of vertices.

6.2 LU Multifrontal Method

The multifrontal method (Section 5.4) can be generalized to nonsymmetric A by modifying the definitions of the frontal matrices and generated elements to conform to an LU factorization. But natural generalizations to rectangular frontal and generated element matrices do not simultaneously satisfy the statements from Observation 5.1. These statements can be rewritten for the LU factorization as follows.

(a) Each generated element V_j is used only once to contribute to a frontal matrix.

(b) The row and column index lists for the rectangular frontal matrix F_j correspond to the nonzeros in column $L_{j:n,j}$ and nonzeros in row $U_{j,j:n}$, respectively.

These conditions cannot both hold. An approach that satisfies (a) can be based on the sparsity pattern of $\mathcal{S}\{A + A^T\}$ and storing some explicit zeros if $\mathcal{S}\{A\}$ is not symmetric. It is then analogous to the symmetric multifrontal method. In this case, although the frontal and generated elements may be numerically nonsymmetric, they are square and structurally symmetric. This approach performs well if $\mathcal{S}\{A\}$ is close to symmetric, that is, the symmetry index of A is close to unity.

An approach that satisfies (b) and not necessarily (a) splits the generated elements into smaller ones that are embedded into further rectangular frontal matrices. We illustrate this using the example from Figure 6.7, that is,

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \left(\begin{array}{cccccccccc} * & & * & & & & & & & \\ * & * & f & & * & & & * & & * \\ * & & * & * & & & & & & \\ & & * & * & & & & & * & * \\ & & & * & * & * & & * & f & f \\ & * & f & f & f & * & & f & f & f \\ & & & * & & & * & & f & f \\ * & * & f & f & f & f & & * & f & f \\ & & & & & & & & * & * \\ & & & & * & f & * & f & f & * \end{array}\right) \end{array},$$

where $*$ are entries in A and filled entries in $L + U$ are denoted by f. Taking the entries in the first row and column, the sparsity patterns of the first frontal matrix and the corresponding generated element are

$$F_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{cc} 1 & 3 \end{array} \\ \left(\begin{array}{cc} * & * \\ * & \\ * & \\ * & \end{array}\right) \end{array}, \quad V_1 = \begin{array}{c} \\ 2 \\ 3 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{c} 3 \end{array} \\ \left(\begin{array}{c} f \\ * \\ f \end{array}\right) \end{array}.$$

To construct F_2 that satisfies (b) we can only use part of V_1. From the row and column replication principles, because $a_{13} \neq 0$, the sparsity pattern of column 1 is replicated in that of column 3 of the factors. While the entry in position $(2, 3)$ belongs to F_2, because of the row replication of the sparsity pattern of the first row in that of the second row, the remaining entries contribute to F_3 and so we split V_1 into two as follows

$$V_1^2 = \begin{array}{c} \\ 2 \end{array}\!\!\begin{array}{c} \begin{array}{c} 3 \end{array} \\ \left(\begin{array}{c} f \end{array}\right) \end{array}, \quad V_1^3 = \begin{array}{c} 3 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{c} 3 \end{array} \\ \left(\begin{array}{c} * \\ f \end{array}\right) \end{array}, \quad V_1 = V_1^2 \Updownarrow V_1^3,$$

where $\boxplus$ is the extend-add operator and V_1^2 and V_1^3 contribute to F_2 and F_3, respectively. Then F_2 and the corresponding generated element V_2 are

$$F_2 = \begin{matrix} \\ 2 \\ 6 \\ 8 \end{matrix} \begin{matrix} \begin{matrix} 2 & 5 & 8 & 10 \end{matrix} \\ \begin{pmatrix} * & * & * & * \\ * & & & \\ * & & & \end{pmatrix} \end{matrix} \boxplus V_1^2 = \begin{matrix} \\ 2 \\ 6 \\ 8 \end{matrix} \begin{matrix} \begin{matrix} 2 & 3 & 5 & 8 & 10 \end{matrix} \\ \begin{pmatrix} * & f & * & * & * \\ * & & & & \\ * & & & & \end{pmatrix} \end{matrix}, \quad V_2 = \begin{matrix} \\ 6 \\ 8 \end{matrix} \begin{matrix} \begin{matrix} 3 & 5 & 8 & 10 \end{matrix} \\ \begin{pmatrix} f & f & f & f \\ f & f & * & f \end{pmatrix} \end{matrix}.$$

Consider the following splitting of V_2

$$V_2 = \begin{matrix} \\ 6 \\ 8 \end{matrix} \begin{matrix} 3 \\ \begin{pmatrix} f \\ f \end{pmatrix} \end{matrix} \boxplus \begin{matrix} \\ 6 \\ 8 \end{matrix} \begin{matrix} 5 \\ \begin{pmatrix} f \\ f \end{pmatrix} \end{matrix} \boxplus \begin{matrix} \\ 6 \\ 8 \end{matrix} \begin{matrix} 8 & 10 \\ \begin{pmatrix} f & f \\ * & f \end{pmatrix} \end{matrix} \equiv V_2^3 \boxplus V_2^5 \boxplus V_2^6.$$

The next frontal matrix is

$$F_3 = \begin{matrix} \\ 3 \\ 4 \end{matrix} \begin{matrix} \begin{matrix} 3 & 4 \end{matrix} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{matrix} \boxplus V_1^3 \boxplus V_2^3 = \begin{matrix} \\ 3 \\ 4 \\ 6 \\ 8 \end{matrix} \begin{matrix} \begin{matrix} 3 & 4 \end{matrix} \\ \begin{pmatrix} * & * \\ * & * \\ f & \\ f & \end{pmatrix} \end{matrix}, \quad V_3 = \begin{matrix} \\ 4 \\ 6 \\ 8 \end{matrix} \begin{matrix} 4 \\ \begin{pmatrix} * \\ f \\ f \end{pmatrix} \end{matrix}.$$

The subsequent steps can be described in a similar way.

Theorem 6.16 expresses the nested relationship between the nonsymmetric multifrontal method and the nonsymmetric elimination tree.

Theorem 6.16 (Eisenstat & Liu 2005b) *Assume A is a general nonsymmetric matrix and $t = parent(k)$ in $\mathcal{T}(A)$. Then*

$$\mathcal{S}\{L_{t:n,k}\} \subseteq \mathcal{S}\{L_{t:n,t}\} \quad \textit{and} \quad \mathcal{S}\{U_{k,t:n}\} \subseteq \mathcal{S}\{U_{t,t:n}\}.$$

Proof Because t is the parent of k, by definition $t \overset{\mathcal{G}(L)}{\Longrightarrow} k \overset{\mathcal{G}(U)}{\Longrightarrow} t$. If $u_{ij} \neq 0$, then a multiple of column i is added to column j during the LU factorization. Thus, by a simple induction argument, for each j on the path $k \overset{\mathcal{G}(U)}{\Longrightarrow} t$, we must have $\mathcal{S}\{L_{j:n,k}\} \subseteq \mathcal{S}\{L_{j:n,j}\}$. In particular, this holds for column t. The second part follows by a similar argument using the path $t \overset{\mathcal{G}(L)}{\Longrightarrow} k$. □

This result shows that the parent relationship in the nonsymmetric elimination tree guarantees that both row and column replications can be applied at the same time. Hence all entries of the submatrices of the generated element V_k with indices greater than or equal to $parent(k)$ can be added to $V_{parent(k)}$ using the operation $\boxplus$. To illustrate this, consider again the 10×10 example above for which $parent(1) = 3$. Theorem 6.16 guarantees that V_1 can be embedded into F_3 because $\mathcal{S}\{L_{3:n,1}\} \subseteq \mathcal{S}\{L_{3:n,3}\}$ and $\mathcal{S}\{U_{1,3:n}\} \subseteq \mathcal{S}\{U_{3,3:n}\}$.

6.3 Preprocessing Sparse Matrices

We now turn our attention to preprocessing techniques that can help in computing an LU factorization. In particular, we consider when A does not have a full transversal (that is, it has one or more zeros on the diagonal). For numerical stability and to reduce the number of permutations required during the factorization, it can be useful to permute A before the factorization begins to put large nonzero entries on the diagonal. As an example, consider the matrix A in Figure 6.9. It has $a_{22} = 0$ and we want to know whether it can be permuted so that all the diagonal entries are nonzero. This question and its answer can be formulated in terms of matchings and bipartite graphs.

6.3.1 Bipartite Graphs and Matchings

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an edge subset $\mathcal{M} \subseteq \mathcal{E}$ is called a **matching** (or assignment) if no two edges in $\mathcal{M}$ are incident to the same vertex. In matrix terms, a matching corresponds to a set of nonzero entries with no two belonging to the same row or column. A vertex is matched if there is an edge in the matching incident on the vertex, and is unmatched (or free) otherwise. The **cardinality** of a matching is the number of edges in it. A **maximum cardinality matching** (or **maximum matching**) is a matching of maximum cardinality. A matching is **perfect** if all the vertices are matched.

A **bipartite graph** is an undirected graph whose vertices can be partitioned into two disjoint sets such that no two vertices within the same set are adjacent, that is, each set is an **independent set**. Let the $n \times n$ matrix A have entries $\{a_{ij'}\}$. Associated with A is a bipartite graph defined as a triple $\mathcal{G}_b = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$, where the row vertex set $\mathcal{V}_{row} = \{i \,|a_{ij'} \neq 0\}$ and the column vertex set $\mathcal{V}_{col} = \{j' \,|a_{ij'} \neq 0\}$ correspond to the rows and columns of A and there is an (undirected) edge $(i, j') \in \mathcal{E}$ if and only if $a_{ij'} \neq 0$. This is illustrated in Figure 6.9. We use prime to distinguish between the independent set of row vertices and the independent set of column vertices, that is, i denotes a row vertex and i' denotes a column vertex.

If A is structurally nonsingular, a matching $\mathcal{M}$ in $\mathcal{G}_b$ is perfect if it has cardinality n. A perfect matching defines an $n \times n$ permutation matrix Q with entries q_{ij} given by

$$q_{ij} = \begin{cases} 1, & \text{if } (j, i') \in \mathcal{M}, \\ 0, & \text{otherwise.} \end{cases}$$

Both QA and AQ have the matching entries on the (zero-free) diagonal. Q and the column permuted matrix AQ for the example in Figure 6.9 are given in Figure 6.10.

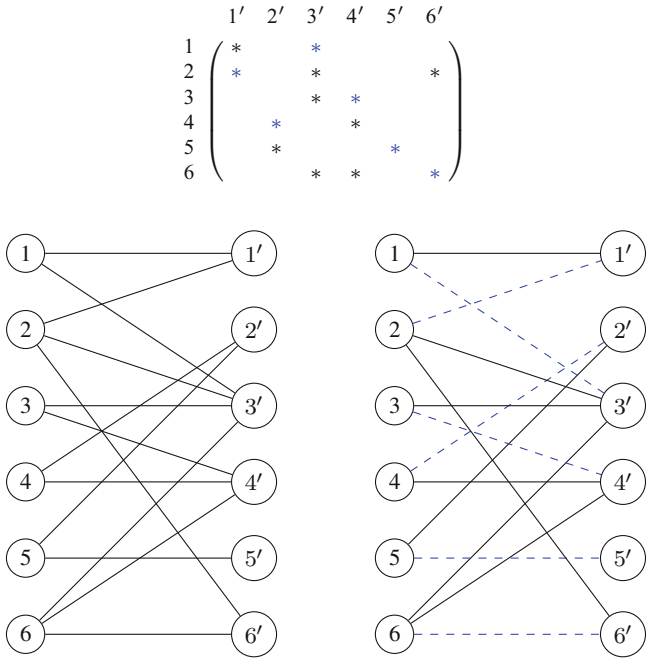


Figure 6.9 A sparse matrix and its bipartite graph $\mathcal{G}_b$ (left). The matched matrix entries are in blue and edges that belong to a perfect matching in $\mathcal{G}_b$ are given by the blue dashed lines (right). Note that the perfect matching is not unique (an alternative is in Figure 6.11).

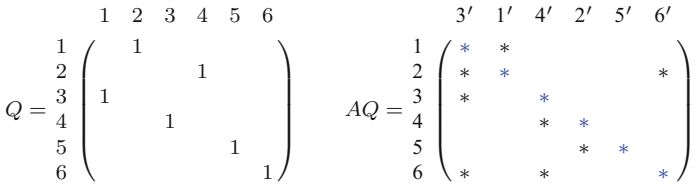


Figure 6.10 The permutation matrix Q and the column permuted matrix AQ corresponding to the matrix in Figure 6.9. The matched entries are on the diagonal of AQ.

6.3.2 Augmenting Paths

If a perfect matching exists, it can be found using augmenting paths. A path $\mathcal{P}$ in a graph is an ordered set of edges in which successive edges are incident to the same vertex. $\mathcal{P}$ is called an $\mathcal{M}$**-alternating path** if the edges of $\mathcal{P}$ are alternately in $\mathcal{M}$ and not in $\mathcal{M}$. An $\mathcal{M}$-alternating path is an $\mathcal{M}$**-augmenting path** in $\mathcal{G}_b$ if it connects an unmatched column vertex with an unmatched row vertex. Note that the length of an $\mathcal{M}$-augmenting path is an odd integer.

ALGORITHM 6.3 Maximum matching algorithm
Input: An undirected graph.
Output: Output maximum matching.

1: Find an initial matching $\mathcal{M}$ ▷ For example, $\mathcal{M} = \emptyset$
2: **while** there exists a $\mathcal{M}$-augmenting path $\mathcal{P}$ **do**
3: $\mathcal{M} = \mathcal{M} \oplus \mathcal{P}$ ▷ Augment the matching along $\mathcal{P}$
4: **end while**

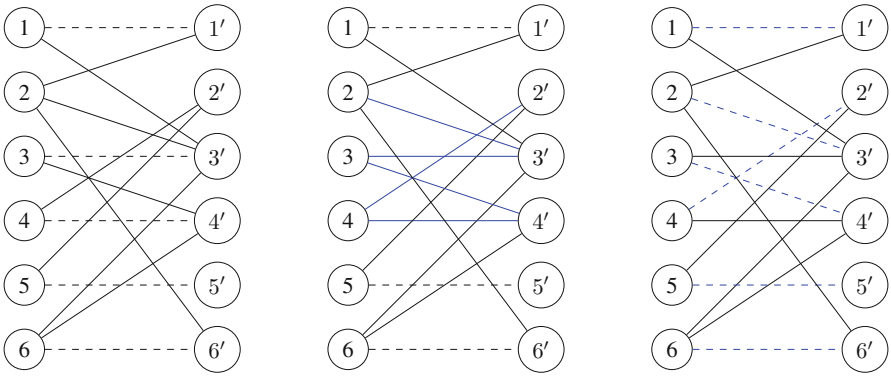


Figure 6.11 An illustration of the search for a perfect matching using augmenting paths. On the left, the dashed lines represent a matching with cardinality 5. In the centre, the blue line is an augmenting path with end vertices 2 and $2'$. On the right is the perfect matching with cardinality 6 that is obtained using the augmenting path.

Let $\mathcal{M}$ and $\mathcal{P}$ be subsets of $\mathcal{E}$ and define the symmetric difference

$$\mathcal{M} \oplus \mathcal{P} := (\mathcal{M} \setminus \mathcal{P}) \cup (\mathcal{P} \setminus \mathcal{M}),$$

that is, the set of edges that belongs to either $\mathcal{M}$ or $\mathcal{P}$ but not to both. If $\mathcal{M}$ is a matching and $\mathcal{P}$ is an $\mathcal{M}$-augmenting path, then $\mathcal{M} \oplus \mathcal{P}$ is a matching with cardinality $|\mathcal{M}|+1$. Growing the matching in this way is called augmenting along $\mathcal{P}$. The next result shows that augmenting paths can be used to find a maximum matching (Algorithm 6.3).

Theorem 6.17 (Berge 1957) *A matching $\mathcal{M}$ in an undirected graph is a maximum matching if and only if there is no $\mathcal{M}$-augmenting path*

Figure 6.11 demonstrates the procedure. On the left is a bipartite graph with a matching with cardinality 5. In the centre, an augmenting path $2 \Longrightarrow 3' \Longrightarrow 3 \Longrightarrow 4' \Longrightarrow 4 \Longrightarrow 2'$ is shown. Augmenting the matching along this path, the cardinality of the matching increases to 6 and $\mathcal{M} \oplus \mathcal{P}$ is a perfect matching.

6.3.3 Weighted Matchings

While the maximum matching algorithm finds a permutation of A such that the permuted matrix has nonzero diagonal entries, there are more sophisticated variations that aim to ensure the absolute values of the diagonal entries of the permuted matrix (or their product) are in some sense large. This can increase the likelihood that the permuted matrix is strongly regular and reduce the need for partial pivoting during the LU factorization. The core problem is as follows: given an $n \times n$ matrix A, find a matching of the rows to the columns such that the product of the matched entries is maximized. That is, find a permutation vector q that maximizes

$$\prod_{i=1}^{n} |a_{iq_i}|. \tag{6.5}$$

Define a matrix C corresponding to A with entries $c_{ij'} \geq 0$ as follows:

$$c_{ij'} = \begin{cases} \log(\max_i |a_{ij'}|) - \log |a_{ij'}|, & \text{if } a_{ij'} \neq 0 \\ \infty, & \text{otherwise.} \end{cases} \tag{6.6}$$

It is straightforward to see that finding a q that solves (6.5) is equivalent to finding a q that minimizes

$$\sum_{i=1}^{n} |c_{iq_i}|, \tag{6.7}$$

which is equivalent to finding a minimum weight perfect matching in an edge weighted bipartite graph. This is a well-studied problem and is known as the bipartite weighted matching or linear sum assignment problem.

If $\mathcal{G}_b = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ is the bipartite graph associated with A then let $\mathcal{G}_b(C) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ be the corresponding weighted bipartite graph in which each edge $(i, j') \in \mathcal{E}$ has a weight $c_{ij'} \geq 0$. The weight (or cost) of a matching $\mathcal{M}$ in $\mathcal{G}_b(C)$, denoted by $csum(\mathcal{M})$, is the sum of its edge weights; i.e.

$$csum(\mathcal{M}) = \sum_{(i,j') \in \mathcal{M}} c_{ij'}.$$

A perfect matching $\mathcal{M}$ in $\mathcal{G}_b(C)$ is said to be a **minimum weight perfect matching** if it has smallest possible weight, i.e. $csum(\mathcal{M}) \leq csum(\widehat{\mathcal{M}})$ for all possible perfect matchings $\widehat{\mathcal{M}}$.

The key concept for finding a minimum weight perfect matching is that of a **shortest augmenting path**. An $\mathcal{M}$-augmenting path $\mathcal{P}$ starting at an unmatched column vertex is called **shortest** if

$$csum(\mathcal{M} \oplus \mathcal{P}) \leq csum(\mathcal{M} \oplus \widehat{\mathcal{P}})$$

for all other possible $\mathcal{M}$-augmenting paths $\widehat{\mathcal{P}}$ starting at the same column vertex. A matching $\mathcal{M}_e$ is **extreme** if and only if there exist u_i and $v_{j'}$ (which are termed **dual variables**) satisfying

$$\begin{cases} c_{ij'} = u_i + v_{j'}, & \text{if } (i, j') \in \mathcal{M}_e, \\ c_{ij'} \geq u_i + v_{j'}, & \text{otherwise.} \end{cases} \tag{6.8}$$

This is employed by the `MC64` algorithm. The dual variables will be important when we discuss scaling sparse matrices in Section 7.4.2. The `MC64` algorithm is outlined here as Algorithm 6.4. It starts with a feasible solution and corresponding extreme matching and then proceeds to iteratively increase its cardinality by one by constructing a sequence of shortest augmenting paths until a perfect extreme matching is found. The algorithm can be made more efficient if a large initial extreme matching can be found. For example, Step 3 can be replaced by setting $u_i = \min\{c_{ij'} | \, j' \in \mathcal{S}\{A_{i,1:n}\}\}$ for $i \in \mathcal{V}_{row}$ and $v_{j'} = \min\{c_{ij'} - u_i | \, i \in \mathcal{S}\{A_{1:n,j'}\}\}$ for $j' \in \mathcal{V}_{col}$. In Step 4, an initial extreme matching can be determined from the edges for which $c_{ij'} - u_i - v_{j'} = 0$.

There are a number of potential problems with the `MC64` algorithm. First, the runtime is hard to predict and depends on the initial ordering of A. Second, it is a serial algorithm and as such it can represent a significant fraction of the total factorization time of a direct solver. Because the complexity of Step 6 of Algorithm 6.4 is $O((n + nz(A)) \log n)$ and the complexity of Step 7 is $O(n)$ and of Step 8 is $O(n+nz(A)$, `MC64` has a worst-case complexity of $O(n(n+nz(A)) \log n)$. In practice, this bound is not achieved and the algorithm is widely used.

ALGORITHM 6.4 Outline of the `MC64` algorithm
Input: Matrix A.
Output: A matching $\mathcal{M}$ and dual variables u_i, $v_{j'}$.

1: Define the weights $c_{ij'}$ using (6.6)
2: Construct the weighted bipartite graph $\mathcal{G}_b(C) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$
3: Set $u_i = 0$ for $i \in \mathcal{V}_{row}$ and $v_{j'} = \min\{c_{ij'} : (i, j') \in \mathcal{E}\}$ for $j' \in \mathcal{V}_{col}$ ▷ Initial solution
4: Set $\mathcal{M} = \{(i, j') | \, u_i + v_{j'}\}$ ▷ Initial extreme matching
5: **while** $\mathcal{M}$ is not perfect **do**
6: Find the shortest augmenting path $\mathcal{P}$ with respect to $\mathcal{M}$
7: Augment the matching $\mathcal{M} = \mathcal{M} \oplus \mathcal{P}$
8: Update u_i, $v_{j'}$ so that (6.8) is satisfied for new $\mathcal{M}$ ▷ Make $\mathcal{M}$ extreme
9: **end while**

6.3.4 Dulmage-Mendelsohn Decompositions

The importance of preordering A to block triangular form was discussed in Section 3.4. The **Dulmage-Mendelsohn decomposition** is based on matchings and is a generalization of the block triangular form. It provides a precise characterization of structurally rank deficient matrices and it can be used to reduce the work required for an LU factorization. It comprises row and column permutations P and Q such that

$$PAQ = \begin{array}{c} \\ \mathcal{R}_1 \\ \mathcal{R}_2 \\ \mathcal{R}_3 \end{array} \begin{array}{c} \begin{array}{ccc} \mathcal{C}_1 & \mathcal{C}_2 & \mathcal{C}_3 \end{array} \\ \begin{pmatrix} A_1 & A_4 & A_6 \\ 0 & A_2 & A_5 \\ 0 & 0 & A_3 \end{pmatrix} \end{array}. \tag{6.9}$$

Here A_1 is an $m_1 \times n_1$ underdetermined matrix ($m_1 < n_1$ or $m_1 = n_1 = 0$), A_2 is an $m_2 \times m_2$ square matrix and A_3 is an $m_3 \times n_3$ overdetermined matrix ($m_3 > n_3$ or $m_3 = n_3 = 0$). It can be shown that A_1^T and A_3 are strong Hall matrices but A_2 need not be a strong Hall matrix, in which case it can be permuted to block upper triangular form.

If row and column sets $\mathcal{R}$ and $\mathcal{C}$ form a maximum matching of A, then $\mathcal{R}_1$ and $\mathcal{R}_2$ are subsets of $\mathcal{R}$ and $|\mathcal{R}_3 \cap \mathcal{R}| = n_3$, and $\mathcal{C}_2$ and $\mathcal{C}_3$ are subsets of $\mathcal{C}$ and $|\mathcal{C}_1 \cap \mathcal{C}| = m_1$. An example decomposition for a 10×10 matrix is given in Figure 6.12. Here $\mathcal{R} = \{1, 2, \ldots, 9\}$ and $\mathcal{C} = \{2, 3, \ldots, 10\}$.

The **coarse Dulmage-Mendelsohn decomposition** orders the unmatched columns as the first columns in PAQ and orders the unmatched rows as the last rows in PAQ. If A is square and has a perfect matching then its coarse decomposition has only the matrix A_2; otherwise, both A_1 and A_3 are present. The coarse decomposition is computed by first finding a maximum matching. Assuming it is not a perfect matching, the rows in A_1 are determined by performing depth-first searches from the unmatched columns to find all of the row vertices that

$PAQ =$

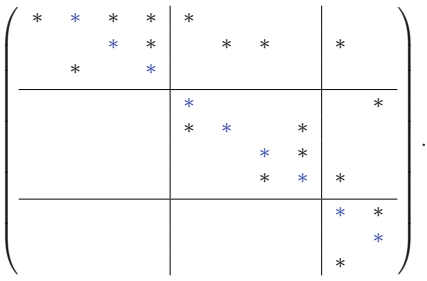


Figure 6.12 An example of a coarse Dulmage-Mendelsohn decomposition. The blue entries belong to the maximum matching. $m_1 = 3$, $m_2 = 4$, $m_3 = 3$, $n_1 = 4$, $n_2 = 4$, $n_3 = 2$. Column 1 and row 10 are unmatched.

are reachable from the unmatched columns via alternating augmenting paths. The columns in A_1 are defined to be the union of the set of unmatched columns and the set of columns matched with the rows in A_1. Similarly, the columns in A_3 are determined by performing depth-first searches from the unmatched rows to find all of the column vertices that are reachable from the unmatched rows via alternating augmenting paths. The rows in A_3 are defined to be the union of the set of rows matched to the columns in A_3 and the set of unmatched rows.

It may be possible to further permute the matrix to obtain the **fine Dulmage-Mendelsohn decomposition**. The fine Dulmage-Mendelsohn decomposition computes P and Q such that A_1 and A_3 are block diagonal matrices in which each diagonal block is irreducible, and A_2 is block upper triangular with strongly connected (square) diagonal blocks. Once the coarse decomposition has been computed, A_1 and A_3 are searched to find any irreducible blocks and the permutations required to place these on the diagonals of A_1 and A_3 are computed. Finally, following Section 3.4, strongly connected components in A_2 are found and a permutation is formed to reduce A_2 to block upper triangular form (with the strongly connected components lying on the diagonal). If A is reducible and nonsingular, the fine Dulmage-Mendelsohn decomposition can be used to solve the linear system $Ax = b$ using block back-substitution.

6.4 Notes and References

Early theoretical results related to sparse LU factorizations can be found in Rose & Tarjan (1978), which extends the systematic understanding of the symbolic elimination introduced in Rose et al. (1976). A key paper that influenced the discussion and development of both the theory and algorithms for predicting sparsity structures in LU factorizations is Gilbert (1994) (first available in 1986 as a Cornell technical report). As the primary and still very useful resource on transitive reduction, we refer to Aho et al. (1972); Gilbert & Liu (1993) extend the concept of an elimination tree to study sparse LU factorizations of nonsymmetric matrices and present theoretical concepts based on DAGs; see also the parallel counterpart in Grigori et al. (2007). Ways to simplify symbolic factorizations and prune DAGs are discussed in Eisenstat & Liu (1992, 1993a). An elegant treatment of both the theoretical and practical aspects of LU factorizations based on DAGs and the nonsymmetric elimination tree (including pruning and pivoting) is given in a series of three papers by Eisenstat & Liu (2005a,b, 2007).

Partial pivoting within the sparse column LU factorization is introduced in Gilbert & Peierls (1988). This paper influenced not only further developments in sparse LU factorizations but also the development of incomplete factorizations. Partial pivoting based on the column elimination tree is first discussed in George & Ng (1985); see also Gilbert & Ng (1993) and Li (1996) for further use of column elimination trees. Further research on exactness of structural predictions is presented by Grigori et al. (2009).

The proof of Theorem 6.17 is given by Berge (1957) but the result was observed earlier (for example, König (1931)). Preordering nonsymmetric matrices using matching algorithms is explained in Duff & Koster (1999, 2001). It is based on the Hungarian algorithm of Kuhn (1955) and a sparse variant of the shortest path algorithm of Dijkstra (1959). Duff and Koster implemented their algorithm in the widely used software package `MC64`. Because `MC64` can be expensive to run, there has been interest in developing efficient parallel algorithms for finding a perfect matching in a weighted bipartite graph (Azad et al., 2020) and also in relaxing the optimality requirement to allow the development of cheaper algorithms that can be parallelised; see, for example, Hogg & Scott (2015). A classical paper that describes the Dulmage-Mendelsohn decomposition is Pothen & Fan (1990).

The development of supernodal LU factorizations is closely connected with that of column LU factorizations. A key paper is by Demmel et al. (1999), in which different types of supernodes for nonsymmetric matrices are considered.

Duff & Reid (1984) describe a symmetric-pattern multifrontal algorithm for nonsymmetric matrices that generates an assembly tree based on the structure of $A+A^T$. This employs square frontal matrices and can incur a substantial overhead for highly nonsymmetric matrices because of unnecessary data dependencies in the assembly tree and extra explicit zeros in the artificially symmetrized frontal matrices. Davis & Duff (1997) introduce an nonsymmetric-pattern multifrontal algorithm that seeks to overcome these deficiencies by using rectangular frontal matrices. This work later developed into the package UMFPACK of Davis (2004), while Amestoy & Puglisi (2002) propose an nonsymmetric version of the multifrontal method that can be regarded as being intermediate between the nonsymmetric-pattern variant of UMFPACK and the symmetric-pattern multifrontal method. The Watson Sparse Matrix Package (WSMP, 2020) also uses a nonsymmetric multifrontal algorithm.

Notable early sparse LU solvers were the Yale Sparse Matrix Package (YSMP) of Eisenstat et al. (1977) and the Harwell Subroutine Library code `MA28` written by Duff (1980), followed later by `MA48` of Duff & Reid (1996). These codes address important practical considerations (for serial computations). Furthermore, the right-looking Markowitz packages `MA28` and `MA48`, which are designed particularly for highly nonsymmetric matrices, combine the symbolic and numerical factorization phases into a single analyse-factorize phase. Contemporary software packages such as PARDISO (2022), SuperLU (Li et al., 1999), UMFPACK and WSMP have been developed over many years. They provide one of the best ways of understanding the practical value of the ideas presented in research papers and technical reports. PARDISO combines left and right-looking updates in a parallel shared-memory code that assumes a symmetric nonzero sparsity pattern. SuperLU offers a left-looking supernodal variant for sequential machines, SuperLU_MT for shared-memory parallel machines, and the right-looking supernodal SuperLU_DIST (Li & Demmel, 2003) for highly parallel distributed memory hybrid systems. Demmel et al. (1999) and Li (2008) describe the algorithms and performance on various machines. The WSMP software is split into a serial and multithreaded single-process library for use on a single core or multiple cores on a shared-memory machine, and a separate library for distributed memory environments.

Chapter 7
Stability, Ill-Conditioning, and Symmetric Indefinite Factorizations

Solving sparse symmetric indefinite systems is more problematic. – Ashcraft et al. (1998).

The factorization of sparse symmetric indefinite systems is particularly challenging since pivoting is required to maintain stability of the factorization. Pivoting techniques generally offer limited parallelism and are associated with significant data movement hindering the scalability of these methods – Duff et al. (2018).

Practical computations are invariably based on finite precision arithmetic. Describing the accuracy of such computations often uses the concept of stability. Consider a computational algorithm $z = g(d)$ for computing z as a function g of given data d. The algorithm is said to be **backward stable** if the computed solution $\hat{z}$ is the exact solution of $\hat{z} = g(d + \Delta d)$, where the perturbation Δd is "small" for all possible inputs d. What is meant by small depends on the context. For example, if d is based on physical measurements that are necessarily inaccurate, Δd is small if it is of the same or smaller absolute value as the inaccuracies in determining d. The minimum absolute value $|\Delta d|$ among such perturbations is called the (absolute) **backward error** (or, if divided by $|d|$, the **relative backward error**). To distinguish them from backward errors, the absolute and relative errors of $\hat{z}$ are called **forward errors**. Backward stability is a property of the computational algorithm and to compute solutions with a small backward error we need to consider stable algorithms.

A related concept that influences the quality of the computed solution is **ill-conditioning**. The problem $z = g(d)$ is said to be ill-conditioned if small perturbations in the data d can lead to large changes in the computed $\hat{z}$. The **condition number** measures how sensitive the output of a function is to its input. Ill-conditioning, which is measured in terms of the condition number, is a property of the problem. Provided the backward error, forward error, and the condition number are defined in a consistent manner, the following approximate inequality holds:

$$\text{forward error} \lessapprox \text{condition number} \times \text{backward error}.$$

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_7

This says that the computed solution to an ill-conditioned problem can have a large forward error because even if the computed solution has a small backward error, this error can be amplified by a large condition number. By preprocessing the problem it may be possible to improve its conditioning. In this chapter, we discuss both the stability of numerical factorizations and preprocessing of the linear system to improve conditioning.

7.1 Backward Stability

We start with a simple backward error result. Here ϵ denotes the machine precision.

Theorem 7.1 (Demmel 1997; Watkins 2002) *Let the computed LU factorization of a matrix A be $A + \Delta A = \widehat{L}\,\widehat{U}$. The perturbation ΔA that results from using finite precision arithmetic satisfies*

$$||\Delta A||_\infty \leq n\, O(\epsilon)\, ||\widehat{L}||_\infty ||\widehat{U}||_\infty + O(\epsilon^2). \tag{7.1}$$

Moreover, the computed solution $\hat{x}$ of the linear system $Ax = b$ satisfies $(A + \Delta' A)\,\hat{x} = b$ with

$$||\Delta' A||_\infty \leq n\, O(\epsilon)\, ||\widehat{L}||_\infty ||\widehat{U}||_\infty + O(\epsilon^2). \tag{7.2}$$

At stage k of Gaussian elimination, the computed diagonal entry $a_{kk}^{(k)}$ is termed the **pivot** ($1 \leq k < n$). Gaussian elimination breaks down if a zero pivot is encountered. Provided A is nonsingular, row interchanges can be incorporated to prevent this happening (Theorem 1.1). The systematic use of row permutations is called **partial pivoting** and was introduced in Section 3.1.2. If $|a_{kk}^{(k)}|$ is very small (compared to other entries in the active submatrix), then it can cause difficulties in finite precision arithmetic because the absolute value of the corresponding computed multiplier $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$ can then be very large. Partial pivoting can be used to ensure $|l_{ik}| \leq 1$, that is, the rows of A that have not yet been pivoted on can be permuted so that the new pivot satisfies

$$\max_{i>k} |a_{ik}^{(k)}| \leq |a_{kk}^{(k)}|.$$

If P_k is the row permutation at stage k and $P = P_{n-1} P_{n-2} \ldots P_1$, then the computed factors of PA satisfy

$$||\widehat{L}||_\infty \leq n \quad \text{and} \quad ||\widehat{U}||_\infty \leq n\, \rho_{growth} ||A||_\infty,$$

where the **growth factor** ρ_{growth} is defined to be

$$\rho_{growth} = \max_{i,j,k} (\, |a_{ij}^{(k)}| \,/\, |a_{ij}| \,). \tag{7.3}$$

The bounds (7.1) and (7.2) can be rewritten as

$$||\Delta A||_\infty \leq n^3 \, \rho_{growth} \, O(\epsilon) \, ||A||_\infty, \quad ||\Delta' A||_\infty \leq n^3 \, \rho_{growth} \, O(\epsilon) \, ||A||_\infty.$$

In practice, these bounds are pessimistic and the actual errors are typically much smaller. Because backward stability of an LU factorization is influenced both by the initial ordering of A and the pivoting strategy, it is said to be **conditionally backward stable**.

For a symmetric positive definite (SPD) matrix A, pivoting for stability is not needed. The following states that the Cholesky factorization of A is **unconditionally backward stable**, allowing the stable computation of the solution of the corresponding linear system.

Theorem 7.2 (Demmel 1997; Watkins 2002) *Let the computed Cholesky factorization of an SPD matrix A be $A + \Delta A = \widehat{L}\widehat{L}^T$. The perturbation ΔA that results from using finite precision arithmetic satisfies*

$$||\Delta A||_\infty \leq n^2 \, O(\epsilon) \, ||A||_\infty.$$

Moreover, the computed solution $\hat{x}$ of the linear system $Ax = b$ satisfies $(A + \Delta' A)\hat{x} = b$ with

$$||\Delta' A||_\infty \leq n^2 \, O(\epsilon) \, ||A||_\infty.$$

Both the unconditional backward stability of a Cholesky factorization of an SPD matrix and the conditional backward stability of an LU factorization of a general A make algorithms for solving linear systems that are based on factorizing A preferable to computing and applying A^{-1}. The computed inverse is typically not the exact inverse of a nearby matrix $A + \Delta A$ for any small perturbation ΔA. Furthermore, the following pessimistic result shows it is impractical to compute and store A^{-1}, regardless of how sparse A is.

Theorem 7.3 (Duff et al. 1988) *If A is irreducible, then the sparsity pattern $\mathcal{S}\{A^{-1}\}$ of its inverse is fully dense.*

Proof Without loss of generality, assume A is factorizable. For if not, there is a permutation matrix P such that the LU factorization of the row permuted matrix PA is factorizable (Theorem 1.1). In this case, consider PA instead of A because for any permutation matrix P the inverse $(PA)^{-1}$ is fully dense if and only if A is fully dense. Let K be the matrix of order $2n$ given by

$$K = \begin{pmatrix} A & I_n \\ I_n & 0 \end{pmatrix}.$$

After applying n elimination steps to $K = K^{(1)}$, the order n active submatrix of $K^{(n+1)}$ is $-A^{-1}$. Consider entry $(A^{-1})_{ij}$ $(1 \leq i, j \leq n)$. Because A is irreducible

and the off-diagonal (1, 2) and (2, 1) blocks of K are equal to the identity matrix, there is a directed path $i \Longrightarrow j$ in $\mathcal{G}(K)$ such that the indices of all the intermediate vertices on the path are less than or equal to n. Theorem 3.1 and the non-cancellation assumption imply $(A^{-1})_{ij} \neq 0$. It follows that A^{-1} is fully dense. □

The above proof implies that entries of A^{-1} correspond to paths in $\mathcal{G}(A)$ when A is not irreducible. This result is given in the following corollary.

Corollary 7.4 (Rose & Tarjan 1978; Duff et al. 1988) *If A is factorizable, then $(A^{-1})_{ij} \neq 0$ $(1 \leq i, j \leq n)$ if and only if there exists a path $i \overset{\mathcal{G}(A)}{\Longrightarrow} j$.*

7.2 Pivoting Strategies for Dense Matrices

This section briefly describes the pivoting strategies that are used in LU factorizations of general dense matrices and, in the symmetric indefinite case, in LDLT factorizations. Here and in the following sections, all the quantities (such as $a_{ij}^{(k)}$) are the computed quantities.

7.2.1 Partial Pivoting

Partial pivoting interchanges rows at each stage of the factorization to select the entry of largest absolute value in its column as the next pivot (Section 3.1.2). If partial pivoting is used, it is straightforward to show that the growth factor (7.3) satisfies

$$\rho_{growth} \leq 2^{n-1}.$$

Although the bound can be achieved in nontrivial cases, it is generally extremely pessimistic, particularly when n is very large. In practice, Gaussian elimination with partial pivoting is often regarded as being a stable algorithm and is the pivoting strategy of choice for dense matrices.

7.2.2 Complete Pivoting

A much smaller bound can be obtained if complete (or full) pivoting is used. It chooses the pivot to be the largest entry (in absolute value) in the active submatrix, that is, at stage k the pivot $a_{kk}^{(k)}$ is chosen so that

$$\max_{i \geq k, j \geq k} |a_{ij}^{(k)}| \leq |a_{kk}^{(k)}|.$$

In this case,

$$\rho_{growth} \leq n^{1/2}(2.\, 3^{1/2}.\, 4^{1/3} \ldots n^{1/(n-1)})^{1/2}. \tag{7.4}$$

The disadvantages of complete pivoting are that it is expensive (the whole active submatrix must be searched for a pivot), and because the test is tougher than for partial pivoting, it is more likely that permutations (and hence more data movement) will be required.

7.2.3 Rook Pivoting

A pivoting strategy that is more restrictive than partial pivoting but cheaper than complete pivoting is **rook pivoting**. Here the pivot is chosen to be the largest entry in its row *and* its column, that is,

$$\max_{i>k} (|a_{ik}^{(k)}|, |a_{ki}^{(k)}|) \leq |a_{kk}^{(k)}|.$$

The strategy takes its name from the fact that the search for a pivot corresponds to the moves of a rook in the game of chess. Clearly, the search for a pivot in rook pivoting involves at least twice as many comparisons as for partial pivoting and if the whole active submatrix has to be searched, then the number of comparisons is the same as for complete pivoting. However, in practice, the cost is usually a small multiple of the cost of partial pivoting and significantly less than that of complete pivoting. The growth factor for rook pivoting satisfies

$$\rho_{growth} \leq 1.5\, n^{(3/4)\log n}\,.$$

7.2.4 2 × 2 Pivoting

When the matrix A is symmetric but indefinite, it may not be possible to select pivots from the diagonal (for example, if all the diagonal entries of A are zero). If rows of A are permuted (so that off-diagonal entries are selected as pivots), then symmetry is destroyed, which means an LU factorization must be performed and this essentially doubles the cost of the factorization in terms of both storage and operation counts. Symmetry can be preserved by extending the notion of a pivot to 2×2 blocks.

Consider the symmetric indefinite A given by

$$A = \begin{pmatrix} \delta & 1 \\ 1 & 0 \end{pmatrix}.$$

If $\delta = 0$, an LDLT factorization in which D is a diagonal matrix does not exist. Furthermore, if $\delta \ll 1$, then an LDLT factorization with D diagonal is not stable because $\rho_{growth} = 1/\delta$. However, if the LDLT factorization is generalized to allow D to be a block diagonal matrix with 1×1 and 2×2 blocks, then a factorization is obtained that preserves symmetry and is nearly as stable as an LU factorization. This is illustrated by the factorization of the following 3×3 symmetric indefinite matrix

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = LDL^T.$$

Here D has one 1×1 block and one 2×2 block.

Rook pivoting can be extended to include 2×2 pivots. An iterative procedure searches for an entry that is simultaneously the largest in absolute value in row i and column j of the active submatrix $A^{(k)}$. This entry is used to build a symmetric 2×2 pivot; the search terminates prematurely if a suitable 1×1 pivot is found, that is, a pivot that satisfies a threshold test. The standard choice for the threshold comes from requiring the same potential maximal growth in the absolute values of the entries of the partially eliminated matrix that results from either two consecutive 1×1 pivots or one 2×2 pivot. It can be shown that the appropriate choice is $(1 + \sqrt{17})/8$. In this case, the growth factor satisfies

$$\rho_{growth} < 3n\sqrt{2\, 3^{1/2} 4^{1/3} \ldots n^{1/(n-1)}},$$

which is only slightly worse than the bound (7.4) for an LU factorization with complete pivoting. Note that the number of partially eliminated matrices depends on the number of 2×2 pivots. If a 2×2 pivot is selected at stage k, then the next partially eliminated matrix is $A^{(k+2)}$.

7.3 Pivoting Strategies for Sparse Matrices

7.3.1 Threshold Partial Pivoting

While the growth factor is important, for sparse matrices the pivoting strategies discussed so far lack the scope to preserve sparsity. In the sparse case, it is necessary to balance pivoting for stability with limiting the amount of fill-in in the factors. The compromise strategy that seeks to achieve this is called **threshold partial pivoting**, which is a generalization of partial pivoting. At stage k of the numerical factorization phase of a sparse LU solver, the pivot is selected so that after permuting it to the first entry of the active submatrix $A^{(k)}$ it satisfies

$$\max_{i>k} |a_{ik}^{(k)}| \leq \gamma^{-1} |a_{kk}^{(k)}|, \tag{7.5}$$

where $\gamma \in (0, 1]$ is a chosen **threshold parameter**. It is straightforward to see that

$$\max_i |a_{ij}^{(k)}| \leq (1 + \gamma^{-1}) \max_i |a_{ij}^{(k-1)}| \leq (1 + \gamma^{-1})^{nz_j} \max_i |a_{ij}|,$$

where nz_j is the number of off-diagonal entries in the j-th column of the U factor. Furthermore,

$$\rho_{growth} \leq (1 + \gamma^{-1})^{nz_{cmax}},$$

where $nz_{cmax} = \max_j nz_j \leq n - 1$. Choosing $\gamma = 1$ reduces to partial pivoting; using a smaller value potentially leads to greater growth in the size of the entries in the factors but allows pivots to be chosen that are better able to preserve sparsity. The default choice for γ is typically between 0.1 and 0.01 but in some practical applications much smaller values are sometimes employed to speed up the factorization (at the possible cost of less accurate factors).

A threshold can also be incorporated into rook pivoting. The pivot must then be at least γ times the absolute value of any other entry in its row and column of the active submatrix. Threshold rook pivoting has the potential to limit growth more successfully than threshold partial pivoting. In the symmetric case, if pivots are selected from the diagonal (to preserve symmetry), threshold partial pivoting is the same as threshold rook pivoting.

7.3.2 *Threshold 2 × 2 Pivoting*

If A is a symmetric matrix, then standard fill-reducing ordering algorithms (which will be discussed in the next chapter) and the symbolic factorization phase employ only the sparsity pattern of A. In general, if A is indefinite, during the numerical factorization it is necessary to modify the chosen elimination order to maintain stability. As already observed, if symmetry is to be preserved, 1×1 and 2×2 pivots are needed, resulting in an LDLT factorization in which D is a block diagonal matrix with 1×1 and 2×2 blocks. Limiting the size of the entries of L so that

$$|l_{ij}| \leq \gamma^{-1} \tag{7.6}$$

for all i, j, together with a backward stable scheme for solving 2×2 linear systems, suffices to show backward stability for the entire solution process.

In the sparse symmetric indefinite case, the stability test for a 1×1 pivot in column t of the active submatrix at stage k is the standard threshold test

$$\max_{i \neq t,\, i \geq k} |a_{it}^{(k)}| \leq \gamma^{-1} |a_{tt}^{(k)}|. \tag{7.7}$$

For a 2×2 pivot in rows and columns s and t the corresponding test is

$$\left| \begin{pmatrix} a_{ss}^{(k)} & a_{st}^{(k)} \\ a_{st}^{(k)} & a_{tt}^{(k)} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i \neq s,t; i \geq k} |a_{is}^{(k)}| \\ \max_{i \neq s,t; i \geq k} |a_{it}^{(k)}| \end{pmatrix} \leq \gamma^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \tag{7.8}$$

where the absolute value of the matrix is interpreted element-wise. If $a_{tt}^{(k)}$ is accepted as a 1×1 pivot, it becomes the next diagonal entry of D and row and column t are permuted (if necessary) to the pivotal position k. The corresponding diagonal entry of L is 1 and from the inequality (7.7), the off-diagonal entries of column k of L are bounded in absolute value by γ^{-1}. If $\begin{pmatrix} a_{ss}^{(k)} & a_{st}^{(k)} \\ a_{st}^{(k)} & a_{tt}^{(k)} \end{pmatrix}$ is accepted as a 2×2 pivot, it becomes the next diagonal block of D and rows and columns s and t are permuted (if necessary) to the next two pivotal positions, k and $k+1$. The corresponding diagonal block of L is the identity matrix of order 2 and inequality (7.8) ensures that the off-diagonal entries of these columns of L are bounded in absolute value by γ^{-1}.

In addition to bounding the size of the entries in L, the ability to stably apply the inverse of D to a vector is required. This is trivially the case for 1×1 pivots, but for 2×2 pivots it is necessary to check that the determinant $|a_{ss}^{(k)} a_{tt}^{(k)} - a_{st}^{(k)} a_{st}^{(k)}|$ is sufficiently large and cancellation does not occur during the application of the inverse.

A major difficulty when stability tests are incorporated into sparse factorizations is that a pivot satisfying the stability criteria may not exist. We discuss this for symmetric indefinite A but the same problem occurs for general A. Consider the supernodal approach of Section 5.3 and the nodal matrix shown in Figure 7.1. Pivots can only be chosen from the block L_{diag} on the diagonal (the block is square and symmetric and only its lower triangular part is held) but the entries in the off-diagonal block L_{rect} are involved in the stability tests: large entries in L_{rect} can cause pivot candidates to fail the threshold tests (7.7), (7.8). If L_{diag} is of order p and only $q < p$ pivots can be found that satisfy the tests, then $p - q$ pivots must be **delayed**. That is, the variables that have not been pivoted on are passed up the assembly tree to the parent and the columns of the block column corresponding to these variables are appended to those of the nodal matrix at the parent. The delayed columns are retested at the parent and, if the stability test is still not satisfied, they are passed further up the assembly tree (at the root a full set of p pivots can be chosen provided the matrix is non-singular and $\gamma \leq 0.5$).

Observe that to be able to test for large entries, all the off-diagonal entries in a block column must be fully updated before the block on the diagonal is factorized. This means that the **factorize_block** task and all the **solve_block** tasks for a block column that are used in the SPD case (Section 5.3) are combined into a single **factorize_column** task. Thus there are fewer but larger tasks and this reduces the scope for parallelism.

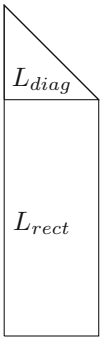


Figure 7.1 An illustration of a simple nodal matrix. Pivot candidates are restricted to the square block L_{diag} on the diagonal.

The problem of delayed pivots arises also in the multifrontal method. At each stage of the computation there is a dense symmetric indefinite frontal matrix F of order n_F of the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix}, \tag{7.9}$$

where F_{11} is a $p \times p$ matrix corresponding to the fully summed variables. Pivots can only be selected from F_{11} but the numerical values of the entries in F_{21} must be taken into account when testing for stability. If $q < p$ pivots are found, then the partial factorization of F is $P_F F P_F^T = L_F D_F L_F^T$, where $P_F = \begin{pmatrix} P_{11} & \\ & I \end{pmatrix}$ is a permutation matrix with P_{11} of order p, $L_F = \begin{pmatrix} L_{11} \\ L_{21}\ I \end{pmatrix}$ with L_{11} a unit lower triangular matrix of order q, and $D_F = \begin{pmatrix} D_1 & \\ & S \end{pmatrix}$, with D_1 a block diagonal matrix of order q and S a dense matrix of order $n_F - q$. A basic procedure for selecting pivots and partially factorizing F is summarized in Algorithms 7.1 and 7.2. Here updating means applying the elimination operations. Observe that candidate pivots are only permuted to the start of the frontal matrix once they have been accepted (passed the stability test). Algorithm 7.2 can be modified for a supernodal factorization, replacing the frontal matrix by a supernodal matrix.

So far, we have assumed that A is nonsingular, but consistent systems of linear equations with a (nearly) singular matrix can occur in practice and only minor modifications are needed to handle this. When a column is searched, if its largest entry is found to have absolute value less than a chosen threshold δ, the column (and, by symmetry, the row) is set to zero, the diagonal entry is accepted as a zero 1×1 pivot, and no update pivotal operations are applied to the remaining columns of F. This is equivalent to perturbing the entries of A in the pivotal column by at most

ALGORITHM 7.1 Simple partial sparse indefinite factorization
Input: Symmetric indefinite matrix F of order n_F of the form (7.9) with F_{11} of order p; threshold $\gamma \in (0, 0.5]$.
Output: Updated F; partial factors L_F and D_F and permutation P_F.

$q = 0, t = 0$ ▷ q holds the sum of the sizes (1 or 2) of the pivots chosen so far
while $q < p$ **do**
  find_pivot (piv_size) ▷ See Algorithm 7.2
  if $(piv\_size = 0)$ **exit while loop** ▷ Failed to find a pivot
  $q = q + piv\_size$
  Update columns $q + 1$ to p of F ▷ Right-looking
end while
Apply updates to columns $p + 1$ to n_F of F ▷ Left-looking

ALGORITHM 7.2 Find a pivot in F using threshold partial pivoting
Input: $F, L_F, D_F, P_F, p, q, t, \gamma$ are accessed from the environment of the call.
Output: Selected pivot of size piv_size; computed columns $q + 1 : q + piv\_size$ of L_F and D_F, updated P_F and t.

subroutine find_pivot (piv_size)
$piv\_size = 0$
for $test = 1 : p - q$ **do**
  $t = t + 1$; **if** $(t > p)$ set $t = q + 1$ ▷ Column t is searched for a pivot
  if (there is s such that $q + 1 \le s \le t - 1$ and $\begin{pmatrix} f_{ss} & f_{st} \\ f_{st} & f_{tt} \end{pmatrix}$ passes 2×2 pivot test) **then**
    $piv\_size = 2$
    Symmetrically permute rows/columns $q + 1$ and s of F ▷ Update P_F
    Symmetrically permute rows/columns $q + 2$ and t of F ▷ Update P_F
    Compute columns $q + 1$ and $q + 2$ of D_F and L_F
    return
  else if (f_{tt} passes 1×1 pivot test) **then**
    $piv\_size = 1$
    Symmetrically permute rows/columns $q + 1$ and t of F ▷ Update P_F
    Compute column $q + 1$ of D_F and L_F
    return
  end if
end for
end subroutine find_pivot

δ and the computed factorization is of a nearby singular matrix. It is convenient for the subsequent solve phase to store D_F^{-1} in place of D_F, with entries on the diagonal corresponding to zero pivots set to zero.

7.3.3 Relaxed and Static Pivoting

If pivots are delayed during the numerical factorization, then the data structures that were set up during the symbolic phase must be modified. This significantly complicates the development of general and symmetric indefinite sparse direct solvers compared to sparse Cholesky solvers. Furthermore, it increases the operation count and memory required to perform the factorization and, more importantly, it can severely limit the scope for parallelism. Maintaining stability and using static data structures are conflicting objectives.

If no candidate pivot satisfies the threshold test but the pivot that is nearest to satisfying it would satisfy it with a threshold $\gamma_1 < \gamma$, then provided γ_1 is at least some chosen minimum value, **relaxed pivoting** accepts this pivot and reduces γ to γ_1. The new value γ_1 is employed thereafter. This means that the factorization is potentially less stable but, with fewer delayed pivots, the factors may be sparser than if the original γ was used throughout.

With relaxed pivoting, delayed pivots can still occur and it may not be possible to use static data structures. Static pivoting allows static data structures because it permits no delayed pivots. When a candidate pivot is found to be too small (and no other eligible candidate passes the stability test), **static pivoting** replaces it by a user defined value. A small value may make the factorization more accurate but can lead to large growth in the size of the entries in the factors, while a large value controls this growth but reduces the accuracy of the factorization. As well as allowing the use of a static task graph and the structures predicted by the symbolic factorization, other benefits of static pivoting are improved use of BLAS 3 operations and parallelism and, because there is no additional fill-in, load imbalance in a parallel environment is less likely to be a problem. However, the factorization need not be stable and the factors are of a shifted matrix $A + D_\delta$ where D_δ is a diagonal matrix, and it may be necessary to seek to improve the accuracy of the solution using a refinement method (see Section 7.4.1). It is also possible that by the time a very small pivot is found it is too late to save the stability of the factorization and perturbing the pivot effectively just amplifies numerical noise. It is thus essential that static pivoting is used with care; it makes an LDLT or LU direct solver less of a "black box solver" because the guarantees are much weaker than when threshold partial pivoting is used. A more robust approach can be to incorporate the use of shifts into the algorithm that calls the linear system solver. For example, a standard technique in some optimization algorithms that involve symmetric linear systems is to employ regularization. This can avoid the need for an LDLT factorization in favour of a stable Cholesky factorization.

Observe that if an LDLT factorization of a symmetric indefinite matrix A is computed, then the **inertia** (that is, the number of positive eigenvalues, negative eigenvalues and eigenvalues equal to zero) of A can be found by computing the eigenvalues of the block diagonal factor D. In some applications, computing the inertia may be desired. For example, in interior point methods for minimizing a nonlinear objective function subject to constraints, each iteration involves solving a sparse symmetric indefinite linear system and it is important that the solution method for this system accurately reports the inertia to allow parameters within the interior point method to be chosen. One consequence of static pivoting or using a small threshold γ is that the computed inertia of A is less likely to be accurate.

7.3.4 *Special Indefinite Matrices that Avoid Pivoting*

Symmetric saddle point matrices are indefinite matrices of the form

$$A = \begin{pmatrix} G & R^T \\ R & -B \end{pmatrix}, \tag{7.10}$$

where $G \in \mathbb{R}^{n_1 \times n_1}$ is an SPD matrix, $B \in \mathbb{R}^{n_2 \times n_2}$ is a positive semidefinite matrix (including $B = 0$), and $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 + n_2 = n$. Such systems include the class of $\mathcal{F}$ matrices, where $B = 0$ and each column of R has at most two entries, and if there are two entries, they sum to zero. It is of interest to try and symmetrically permute A in such a way that the LDLT factorization of the permuted matrix PAP^T exists without the use of threshold pivoting. This is attractive because it then makes the factorization as efficient as for an SPD matrix.

Define the permutation matrix P to be

$$P = \left[e_1,\ e_{n_1+1},\ e_2,\ e_{n_1+2},\ \ldots\ e_{n_1},\ e_n,\ e_{n_2+1},\ \ldots,\ e_{n_1}\right]^T.$$

Then the permuted matrix PAP^T has a block form in which each entry $A_{i,j}$ is a 2×2 or 2×1 or 1×2 or 1×1 block. In particular, the diagonal blocks are

$$A_{i,i} = \begin{cases} \begin{pmatrix} g_{ii} & r_{ii} \\ r_{ii} & -b_{ii} \end{pmatrix}, & 1 \le i \le n_2 \\ b_{ii}, & n_2 + 1 \le i \le n_1. \end{cases}$$

The following theorem shows that a 2×2 pivot updated by the Schur complement of a 1×1 pivot is nonsingular and vice versa.

Theorem 7.5 (Lungten et al. 2018) *Let A be the symmetric saddle point matrix (7.10). Assume $R = (R_1\ R_2)$ is of full rank with $R_1 \in \mathbb{R}^{n_2 \times n_2}$ nonsingular. Let $G \in \mathbb{R}^{n_1 \times n_1}$ be SPD and partitioned conformally and let $B \in \mathbb{R}^{n_2 \times n_2}$ be*

positive semidefinite. If A is permuted to the form

$$\left(\begin{array}{cc|c} G_{11} & R_1^T & G_{12} \\ R_1 & -B & R_2 \\ \hline G_{12}^T & R_2^T & G_{22} \end{array}\right),$$

then the Schur complement of the symmetric indefinite matrix $\begin{pmatrix} G_{11} & R_1^T \\ R_1 & -B \end{pmatrix}$ *and the Schur complement of the SPD matrix* G_{22} *are nonsingular.*

A consequence of Theorem 7.5 is that provided R is of full rank and R_1 is nonsingular then the LDLT factorization of PAP^T exists, with 2×2 pivots and 1×1 pivots chosen from the diagonal blocks of PAP^T in any order. Assume all the 2×2 pivots are selected ahead of the 1×1 pivots. If $B = 0$ and $|r_{ii}| \geq \max_{i\leq j\leq n_1} |r_{ij}|$ $(1 \leq i \leq n_2)$, then the growth factor is bounded by 2^{2n_2}.

A potential difficulty is that permutation matrices P_r and P_c are needed such that $P_r R P_c = [R_1 \ R_2]$ with R_1 nonsingular. If P_r and P_c can be constructed so that

$$P_r R P_c = \begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix}, \tag{7.11}$$

where R_{11} is upper triangular with nonzero diagonal entries then the permuted R is said to have a **trapezoidal form**. A simple case where R can be permuted to this form is if it satisfies the following one-degree principle. Let R be of full rank and let $\mathcal{G}_b(R) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ be the bipartite graph of R (Section 6.3.1). R can be permuted to trapezoidal form if, for $k = 1, 2, \ldots, n_1 - 1$, the bipartite graph of $R^{(k)}$ has at least one vertex $j'_k \in \mathcal{V}_{col}$ of degree one, where $R^{(1)} = R$ and $R^{(k+1)}$ is obtained by removing from $R^{(k)}$ the column vertex j'_k and its matched row index i_k together with all edges involving j'_k or i_k.

To illustrate this, consider the 6×8 matrix R in Figure 7.2 and its associated bipartite graph $\mathcal{G}_b(R)$. The first column vertex with degree one is $2'$; it is matched with the row vertex 4. Deleting $2'$ and 4 removes edges $\{(4, 2'), (4, 3'), (4, 5'), (4, 6'), (4, 8')\}$. Column vertex $3'$ now has degree one; it is matched with row vertex 6. Repeating the process gives a perfect matching $\mathcal{M} = \{(4, 2'), (6, 3'), (1, 4'), (5, 5'), (2, 1'), (3, 6')\}$ together with row and column matched vertex sets $\{4, 6, 1, 5, 2, 3\}$ and $\{2', 3', 4', 5', 1', 6'\}$, respectively, and permutation matrices P_r and P_c of order 6 and 8 can be defined to obtain the trapezoidal form in Figure 7.2.

If after $k \geq 1$ steps all columns of the reduced matrix $R^{(k)}$ have degree greater than 1, the permuted matrix has the form (7.11) where R_{11} is $k \times k$ upper triangular, R_{12} is $k \times (n_1 - k)$ and the $(n_2 - k) \times (n_1 - k)$ block R_{22} has columns of degree greater than one. $n_1 - k$ steps of Gaussian elimination (with partial pivoting) can be applied to R_{22} to complete the transformation of R to trapezoidal form.

$$R = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1' & 2' & 3' & 4' & 5' & 6' & 7' & 8' \end{array} \\ \begin{pmatrix} & & & * & & & & \\ * & & & & & * & & \\ & & & & & * & & * \\ & * & * & & * & * & & * \\ * & & & & * & * & & \\ * & & * & * & & & * & \end{pmatrix} \end{array} \qquad P_r R P_c = \begin{array}{c} \\ 4 \\ 6 \\ 1 \\ 5 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccccc} 2' & 3' & 4' & 5' & 1' & 6' & 7' & 8' \end{array} \\ \begin{pmatrix} * & * & & * & & * & & * \\ & * & * & & * & & * & \\ & & * & & & & & \\ & & & * & * & * & & \\ & & & & * & * & & \\ & & & & & * & & * \end{pmatrix} \end{array}$$

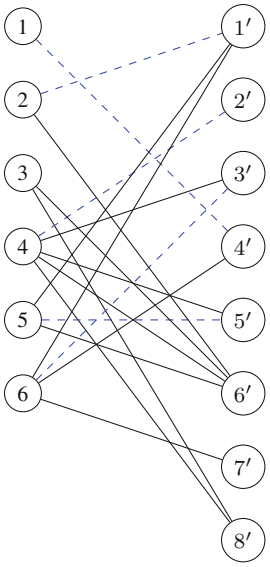


Figure 7.2 Illustration of permuting a full rank matrix to trapezoidal form using the one-degree principle. The matrix R and its bipartite graph $\mathcal{G}_B(R)$ are given. The edges that belong to the perfect matching in $\mathcal{G}_b(R)$ found using the one-degree principle are given by the dashed blue lines; the corresponding matrix entries are in blue. The trapezoidal form comprises a 6×6 upper triangular matrix R_1 and a 6×2 rectangular matrix R_2, where $P_r = [e_4,\ e_6,\ e_1,\ e_5,\ e_2,\ e_3]^T$ and $P_c = [e_2,\ e_3,\ e_4,\ e_5,\ e_1,\ e_6,\ e_7,\ e_8]$ are the row and column permutation matrices.

7.4 Solving Ill-Conditioned Problems

Ill-conditioning is connected to the input data: a problem is ill-conditioned if small changes in the data can lead to large changes in the solution. Assume for the general linear system $Ax = b$ that A and b are perturbed by ΔA and Δb, respectively, and the corresponding perturbation of the solution x is Δx, so that the perturbed problem

$$(A + \Delta A)(x + \Delta x) = b + \Delta b \tag{7.12}$$

has been solved. The perturbations in A and b may include both data uncertainty and algorithmic errors. Rearranging (7.12), we obtain

$$A\Delta x = \Delta b - \Delta A - \Delta A \Delta x.$$

Premultiplying by A^{-1} and considering *any* norm $\|.\|$ and the corresponding subordinate matrix norm yields

$$\|\Delta x\| \leq \|A^{-1}\| \, (\|\Delta b\| + \|\Delta A\| \, \|x\| + \|\Delta A\| \, \|\Delta x\|).$$

It follows that

$$(1 - \|A^{-1}\| \, \|\Delta A\|)\|\Delta x\| \leq \|A^{-1}\| \, (\|\Delta b\| + \|\Delta A\| \, \|x\|)$$

and, provided $\|A^{-1}\| \, \|\Delta A\| < 1$, this gives the following bound on the absolute error

$$\|\Delta x\| \leq \frac{\|A^{-1}\|}{\|A^{-1}\| \, \|\Delta A\|} (\|\Delta b\| + \|\Delta A\| \, \|x\|).$$

Dividing by $\|x\|$ and using $\|b\| \leq \|A\| \, \|x\|$, yields the relative error bound

$$\|\Delta x\|/\|x\| \leq \frac{\kappa(A)}{1 - \kappa(A)\|\Delta A\|/\|A\|} \left(\|\Delta A\|/\|A\| + \|\Delta b\|/\|b\|\right), \tag{7.13}$$

where

$$\kappa(A) = \|A\| \, \|A^{-1}\| \tag{7.14}$$

is the **condition number** of the matrix A. The inequality (7.13) shows that the condition number is a relative error magnification factor. If we have a stable algorithm, then a neighbouring problem has been solved, that is,

$$\|\Delta A\|/\|A\| + \|\Delta b\|/\|b\|$$

is small. This ensures an accurate solution if $\kappa(A)$ is small. A large condition number means that A is close to being singular ($\kappa(A)$ tends to infinity as A tends to singularity).

Observe that the condition number is very dependent on the scaling of A. Furthermore, $\kappa(A)$ takes no account of the right-hand side vector b or the fact that small entries of A (including zeros) may be known within much smaller tolerances than larger entries.

If the matrix norm is that induced by the Euclidean norm (that is, the 2-norm $\|.\|_2$) and A is symmetric, then (7.14) becomes

$$\kappa(A) = |\lambda_{\max}(A)|/|\lambda_{\min}(A)|, \tag{7.15}$$

ALGORITHM 7.3 Iterative refinement of the computed solution of $Ax = b$
Input: The vector b and matrix A.
Output: A sequence of approximate solutions $x^{(0)}, x^{(1)}, \ldots$.

1: Solve $Ax^{(0)} = b$ ▷ $x^{(0)}$ is the initial computed solution
2: **for** $k = 0, 1, \ldots$ **do**
3: Compute $r^{(k)} = b - Ax^{(k)}$ ▷ Residual on iteration k
4: Solve $A\,\delta x^{(k)} = r^{(k)}$ ▷ Solve correction equation
5: $x^{(k+1)} = x^{(k)} + \delta x^{(k)}$
6: **end for**

where $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ are eigenvalues of A of largest and smallest absolute values, respectively. This is called the **spectral condition number** of A. It is important when considering convergence of iterative solvers (Section 9.1.2).

7.4.1 Iterative Refinement

Iterative refinement can be used to overcome matrix ill-conditioning and improve the accuracy of the computed solution. It may also be used after relaxed or static pivoting. The basic method is outlined as Algorithm 7.3. Note that the solvers in Steps 1 and 4 do not have to be the same. The traditional and most common approach is to use the computed factors of A in both steps. Alternatively, the factors can be employed as a preconditioner for an iterative solver in Step 4 (preconditioning and iterative solvers are discussed in Chapter 9). Iterative refinement terminates when either the norm of the residual vector $r^{(k)}$ is sufficiently close to zero that the corresponding correction $\delta x^{(k)}$ is very small or the chosen maximum number of iterations is reached. If there were no roundoff errors in any of the refinement steps, the process would converge to the correct solution in a single iteration. In practice, the residual generally decreases significantly over the first few iterations before stagnating (i.e. reaching a point after which little further accuracy is achieved). If the required accuracy has not been achieved, then a possible approach is to switch to using the computed factors as a preconditioner for a Krylov subspace solver (see Chapter 9).

Observe that computing $r^{(k)}$ in Step 3 uses the original matrix A and if the residual is small, a nearby problem will have been solved. This is particularly useful when there is uncertainty in the accuracy of the computed factors as an approximation to A (for instance, if threshold pivoting or static pivoting has been employed).

There are a number of variants of iterative refinement that involve using different precisions for all or part of the process. In traditional iterative refinement, the

residuals are computed at twice the working precision (the precision at which the data A, b and the solution x are stored). In fixed precision refinement, all computations use the same precision. In mixed precision iterative refinement, the most expensive parts of the computation (the LU factorization of A and solving the correction equation) are performed in single precision and the residual computation in double precision. This is attractive because on modern computer architectures single precision arithmetic is usually significantly faster than double precision. Moreover, holding the factors in single precision substantially reduces the memory required and the amount of data movement. The use of half precision (16-bit) arithmetic is also a possibility, assuming it is considerably faster than single precision, with a proportional saving in energy consumption.

7.4.2 Scaling to Reduce Ill-Conditioning

We have discussed the importance of the condition number $\kappa(A)$. If it is large, then we would like to reduce it by transforming A. An important way of doing this is by scaling A before the numerical factorization begins.

Consider two nonsingular $n \times n$ diagonal matrices S_r and S_c. Diagonal scaling of the system $Ax = b$ transforms it to

$$S_r\, A\, S_c\, y = S_r\, b, \qquad y = S_c^{-1}\, x. \tag{7.16}$$

If A is symmetric, then selecting $S_r = S_c$ retains symmetry. For a general A, scaling and permuting to bring large entries onto the diagonal can reduce the need for numerical pivoting, resulting in fewer delayed pivots, less fill-in, faster factorization and solve times, and a reduction in the storage requirements. But finding a good scaling can represent a significant overhead (especially within a parallel solver) and there are limits on the reduction in $\kappa(A)$ that can be achieved by scaling, as illustrated by the following result.

Theorem 7.6 (van der Sluis 1969) *Let the matrix A be SPD and let D_A be the diagonal matrix with entries a_{ii} $(1 \le i \le n)$. Then for all diagonal matrices D with positive entries*

$$\kappa(D_A^{-1/2}\, A\, D_A^{-1/2}) \le nz_{rmax}\, \kappa(D^{-1/2}\, A\, D^{-1/2}),$$

where nz_{rmax} is the maximum number of entries in a row of A.

We remark that the original (unscaled) matrix A should be retained for iterative refinement of the computed solution. Using the scaled matrix generally results in a larger residual for the original system because, in effect, a perturbed system is solved.

Equilibration Scaling

How to find an appropriate scaling is an open question, but a number of heuristics have been proposed. An obvious choice is to seek to balance entries of the scaled matrix $S_r A S_c$ to have approximately equal absolute values. This is called (approximate) **equilibration** scaling. It is a natural scaling if the numerical values of the entries of A correspond to physical quantities that are measured using different scales.

One approach to equilibration scaling that is relatively cheap as well as easy to implement is to select the diagonal scaling matrices so that the infinity norm of each row and column of the scaled matrix is approximately equal to unity. Algorithm 7.4 presents an iterative procedure for computing such a scaling. Observe that this preserves symmetry. In the nonsymmetric case, Algorithm 7.4 yields the same results when applied to A and A^T in the sense that the scaled matrix obtained for A^T is the transpose of that for A.

The infinity norm in Algorithm 7.4 may be replaced by the 1-norm, resulting in a matrix whose row and column sums are exactly one (this is sometimes called a doubly stochastic matrix). It can be advantageous to combine the use of the infinity and one norms. For example, by performing one step of infinity norm scaling followed by one or more steps of one norm scaling.

ALGORITHM 7.4 Equilibration scaling in the infinity norm
Input: The matrix A and convergence tolerance $\delta > 0$.
Output: Diagonal scaling matrices S_r and S_c.

1: $B^{(1)} = A$, $D^{(1)} = I$, $E^{(1)} = I$
2: **for** $k = 1, 2, \ldots$ **do**
3: Compute $\|B^{(k)}_{i,1:n}\|_\infty$ and $\|B^{(k)}_{1:n,i}\|_\infty$, $1 \le i \le n$ ▷ i-th row and column of $B^{(k)}$
4: **if** $max_i \left\{ |1 - \|B^{(k)}_{i,1:n}\|_\infty| \right\} \le \delta$ and $max_i \left\{ |1 - \|B^{(k)}_{1:n,i}\|_\infty| \right\} \le \delta$ **then**
exit for loop
5: $R = diag\left(\sqrt{\|B^{(k)}_{i,1:n}\|_\infty}\right)$ and $C = diag\left(\sqrt{\|B^{(k)}_{1:n,i}\|_\infty}\right)$
6: $B^{(k+1)} = R^{-1} B^{(k)} C^{-1}$, $D^{(k+1)} = D^{(k)} R^{-1}$, $E^{(k+1)} = E^{(k)} C^{-1}$
7: **end for**
8: $S_r = D^{(k+1)}$ and $S_c = E^{(k+1)}$

Matching-Based Scalings

In Section 6.3.3, we discussed weighted matchings. In particular, the problem of finding a permutation vector q that maximizes the product

$$\prod_{i=1}^{n} |a_{iq_i}|.$$

The entries a_{iq_i} corresponding to the solution q are the matched entries. The dual variables u_i and v_j computed by the `MC64` algorithm (Algorithm 6.4) that seeks to compute q can be used to calculate a scaling as follows. Define the diagonal scaling matrices S_r and S_c to have entries

$$(S_r)_{ii} = \exp(u_i) \quad \text{and} \quad (S_c)_{jj} = \exp(v_j - \log(\max_i |a_{ij}|)), \quad 1 \le i, j \le n.$$

The entries of the scaled matrix $S_r A S_c$ satisfy

$$|(S_r A S_c)_{ij}| \begin{cases} = 1, & \text{if } (i, j) \in \mathcal{M}, \\ \le 1, & \text{otherwise,} \end{cases}$$

where $\mathcal{M}$ is the maximum weighted matching computed by the `MC64` algorithm. If A is symmetric, let S be the diagonal matrix with entries

$$(S)_{ii} = \sqrt{(S_r)_{ii}(S_c)_{ii}}.$$

Then the symmetric matrix SAS has the same property.

Combining Matching-Based Scalings and Orderings

The matching-based ordering and scaling can be used independently but they can also be combined. After scaling, if the matched entries are non-symmetrically permuted onto the diagonal, then because they are large, they provide good pivot candidates for an LU factorization. This approach is commonly used alongside static pivoting to obtain a factorization of a perturbed matrix, followed by iterative refinement to recover the solution to the original system.

In the symmetric indefinite case, symmetry needs to maintained and so the objective is to symmetrically permute a large off-diagonal entry a_{ij} onto the subdiagonal to give a 2×2 block $\begin{pmatrix} a_{ii} & a_{ij} \\ a_{ij} & a_{jj} \end{pmatrix}$ that is potentially a good 2×2 candidate pivot. Assume that a matching $\mathcal{M}$ has been computed using the `MC64` algorithm and let q be the corresponding permutation vector. Any diagonal entries that are in the matching are immediately considered as potential 1×1 pivots and are held in a set $\mathcal{M}_1$. A set $\mathcal{M}_2$ of potential 2×2 pivots is then built by expressing q in terms of its component cycles. A cycle of length 1 corresponds to an entry a_{ii} in the matching. A cycle of length 2 corresponds to two vertices i and j, where a_{ij} and a_{ji} are both in the matching. k potential 2×2 pivots can be extracted from even cycles of length $2k$ or from odd cycles of length $2k + 1$. A straightforward

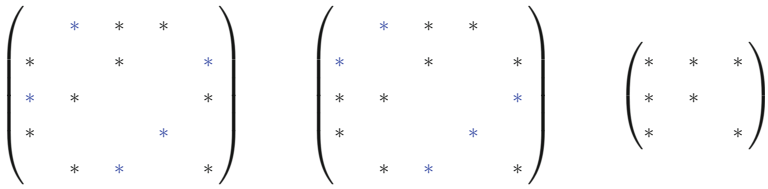


Figure 7.3 An illustration of a symmetric matching for a symmetric indefinite matrix. On the left is the matching $\mathcal{M}$ returned by the `MC64` algorithm and in the centre is a symmetric matching $\mathcal{M}_s$ obtained from $\mathcal{M}$. Entries in the matching are in blue. The pairs $(i, j) = (1, 2)$ and $(3, 5)$ are possible 2×2 pivot candidates. On the right is the compressed matrix that results from combining rows and columns 1 and 2 and rows and columns 3 and 5.

way to do this is to take the first two entries as the first 2×2 pivot, the next two as the next 2×2 pivot, and so on, until if the cycle is of odd length, a single entry remains, which is added to the set $\mathcal{M}_1$. In practice, most cycles in q are of length 1 or 2. A simple example is given in Figure 7.3. Here the matching from `MC64` is $\mathcal{M} = \{(1, 2), (2, 5), (3, 1), (4, 4), (5, 3)\}$, which is nonsymmetric. q has one cycle of length 4 ($1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 1$) and one of length 1, giving $\mathcal{M}_1 = \{(4, 4)\}$ and $\mathcal{M}_2 = \{(1, 2), (2, 1), (3, 5), (5, 3)\}$.

Let $\mathcal{M}_s = \mathcal{M}_1 \cup \mathcal{M}_2$ be the resulting symmetric matching obtained from $\mathcal{M}$ and let Q_s be the corresponding permutation matrix. To combine Q_s with a fill-reducing ordering (such as nested dissection or minimum degree), $Q_s A Q_s^T$ is compressed. The union of the sparsity structure of the two rows and columns belonging to a potential 2×2 pivot is built and used as the structure of a single row and column in the compressed matrix. A fill-reducing ordering algorithm is then applied to the (weighted) compressed graph, and the computed permutation is expanded to a permutation Q_f for $Q_s A Q_s^T$. The final permutation matrix is the product $Q_f Q_s$. The rows/columns of a potential 2×2 pivot are ordered consecutively.

This approach can reduce the overall computational cost when solving tough indefinite systems for which non-matching based orderings require substantial modifications to the pivot sequence during the numerical factorization to maintain stability. Unfortunately, although after applying the matching-based scaling and ordering there are pivot candidates with large entries, there is still no guarantee that the computed pivot sequence will not need modifying during the factorization. An important disadvantage of using matchings are that the numerical values of the entries of A are used so that, if a series of matrices with the same sparsity pattern but different numerical values need to be factorized (such as occurs when an iterative method is used to solve a nonlinear system), the whole symbolic factorization phase may have to be rerun for each matrix, potentially adding significantly to the total solution time.

7.5 Notes and References

There are many related but different results on the stability of matrix factorizations. While the seminal book of Higham (2002) includes component-wise accuracy and stability analysis (see also the classical text (Wilkinson, 1961), which introduced the terms partial pivoting and complete pivoting), the norm-wise results given in Section 7.1 are based on Demmel (1997); see also Watkins (2002).

Rook pivoting is introduced in Neal & Poole (1992) and analysed in Foster (1997). Early pivoting strategies for dense symmetric indefinite systems are presented in Bunch & Parlett (1971), Bunch (1971), and Bunch & Kaufman (1977). Static pivoting in sparse LU factorizations was first proposed by Li & Demmel (1998). A comprehensive overview of threshold-based pivoting strategies for dense and sparse symmetric indefinite problems is given in Ashcraft et al. (1998). This includes symmetric rook pivoting for dense problems and a discussion of the sparse 2×2 threshold partial pivoting strategy of Duff & Reid (1983), which was subsequently modified in Duff et al. (1991), and forms the basis of the approach of Section 7.3.2. Further implementation details (including incorporating working with blocks) are found in Reid & Scott (2011) (see also Hogg & Scott, 2013c). More recently, there has been work on new strategies that seek to offer greater potential for exploiting parallelism without sacrificing numerical robustness, including Hogg & Scott (2014), Hogg et al. (2016), and Duff et al. (2018).

Avoiding the need to pivot for special classes of indefinite matrices is from Lungten et al. (2018) (but see also Tůma, 2002 and de Niet & Wubs, 2009). Duff & Pralet (2005) and Schenk & Gärtner (2006) use weighted matchings for preprocessing, the latter implementing their strategy within the initial version of the solver PARDISO. The HSL mathematical software library (HSL, 2022) includes a number of packages that are designed for symmetric indefinite systems, most notably the multifrontal codes `MA57` (Duff, 2004) and `HSL_MA97`, and the supernodal DAG-based code `HSL_MA86` (Hogg & Scott, 2013b). In these solvers, the default setting for the threshold pivoting parameter γ is 0.01, although when used within the well-known interior point solver (IPOPT, 2022), a value of 10^{-8} is recommended (see also Saunders, 1996). Other well-known sparse direct solvers that handle symmetric indefinite systems include MUMPS (2022) and WSMP (2020).

The technique of iterative refinement was introduced by Doolittle (1878). It was probably first used in a computer program for improving the computed solution to a linear system by Wilkinson (1948), during the design and building of the ACE computer at the National Physical Laboratory; see also Wilkinson (1963) and Moler (1967). The book by Higham (2002) is an essential reference. For sparse systems, the paper by Arioli et al. (1989) is of interest. Hogg & Scott (2010) employ iterative refinement within a sparse mixed precision multifrontal solver. More recently, with a focus on dense systems, Carson & Higham (2017, 2018) and Carson et al. (2020) propose an alternative form of mixed precision iterative refinement that is able to handle highly ill-conditioned problems by solving for the correction using the

GMRES iterative method preconditioned by the computed LU factors. The survey by Abdelfattah et al. (2021) provides a comprehensive review of work on the use of mixed precision in numerical linear algebra.

For Theorem 7.6, we refer to van der Sluis (1969). The equilibration scaling in the infinite norm that is outlined in Algorithm 7.4 is given by Ruiz (2001) (see also Liu, 2015). Matching-based scalings are presented in Duff & Koster (1999, 2001), but see also Neumaier & Olschowka (1996) as well as the origins of the scaling factors in Edmonds (1965).

Chapter 8
Sparse Matrix Ordering Algorithms

The computational complexity of obtaining optimal reorderings for performing sparse Gaussian elimination justifies the heuristic nature of all practical reordering algorithms. – Erisman et al. (1987).

So far, our focus has been on the theoretical and algorithmic principles involved in sparse Gaussian elimination-based factorizations. To limit the storage and the work involved in the computation of the factors and in their use during the solve phase it is generally necessary to reorder (permute) the matrix before the factorization commences. The complexity of the most critical steps in the factorization is highly dependent on the amount of fill-in, as can be seen from the following observation.

Observation 8.1 *The operations to perform the sparse LU factorization $A = LU$ and the sparse Cholesky factorization $A = LL^T$ are $O(\sum_{j=1}^{n} |col_L\{j\}||row_U\{j\}|)$ and $O(\sum_{j=1}^{n} |col_L\{j\}|^2)$ respectively, where $|row_U\{j\}|$ and $|col_L\{j\}|$ are the number of off-diagonal entries in row j of U and column j of L, respectively.*

The problem of finding a permutation to minimize fill-in is NP complete and thus heuristics are used to determine orderings that limit the amount of fill-in; we refer to these as fill-reducing orderings. Frequently, this is done using the sparsity pattern $\mathcal{S}\{A\}$ alone, although sometimes for non-definite matrices, it is combined with the numerical factorization because additional permutations of A may be needed to make the matrix factorizable. Two main classes of methods that work with $\mathcal{S}\{A\}$ are commonly used.

Local orderings attempt to limit fill-in by repeated local decisions based on $\mathcal{G}(A)$ (or a relevant quotient graph).

Global orderings consider the whole sparsity pattern of A and seek to find a permutation using a divide-and-conquer approach. Such methods are normally used in conjunction with a local fill-reducing ordering, as the latter generally works well for problems that are not really large.

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_8

It is assumed throughout this chapter that A is irreducible. Otherwise, if $\mathcal{S}\{A\}$ is symmetric, the algorithms are applied to each component of $\mathcal{G}(A)$ independently and n is then the number of vertices in the component. If $\mathcal{S}\{A\}$ is nonsymmetric, we assume that A is in block triangular form and the algorithms are used on the graph of each block on the diagonal. We also assume that A has **no rows or columns that are (almost) dense**. If it does, a simple strategy is to remove them before applying the ordering algorithm to the remaining matrix; the variables corresponding to the dense rows and columns can be appended to the end of the computed ordering to give the final ordering.

Historically, ordering the matrix A before using a direct solver to factorize it was generally cheap compared to the numerical factorization cost. However, in the last couple of decades, the development of more sophisticated factorization algorithms and their implementations in parallel on modern architectures has affected this balance so that the ordering can be the most expensive step. If a sequence of matrices having the same sparsity pattern is to be factorized, then the ordering cost and the cost of the symbolic factorization can be amortized over the numerical factorizations. If not, it is important to have available a range of ordering algorithms because using a cheap but less effective algorithm may lead to faster complete solution times compared to using an expensive approach that gives some savings in the memory requirements and operation counts but not enough to offset the ordering cost.

8.1 Local Fill-Reducing Orderings for Symmetric $\mathcal{S}\{A\}$

In the symmetric case, the diagonal entries of A are required to be present in $\mathcal{S}\{A\}$ (thus zeros on the diagonal are included in the sparsity structure). The aim is to limit fill-in in the L factor of an LL^T (or LDL^T) factorization of A. Two greedy heuristics are the minimum degree (MD) criterion and the local minimum fill (MF) criterion.

8.1.1 Minimum Fill-in (MF) Criterion

One way to reduce fill-in is to use a local **minimum fill-in** (MF) criterion that, at each step, selects as the next variable in the ordering one that will introduce the least fill-in in the factor at that step. This is sometimes called the **minimum deficiency** approach. While MF can produce good orderings, its cost is often considered to be prohibitive because it requires the updated sparsity pattern and the fill-in associated with the possible candidates must be determined. The runtime can be reduced using an approximate variant (AMF) but it is not widely implemented in modern sparse direct solvers.

8.1.2 Basic Minimum Degree (MD) Algorithm

The minimum degree (MD) algorithm is the best-known and most widely used greedy heuristic for limiting fill-in. It seeks to find a permutation such that at each step of the factorization the number of entries in the corresponding column of L is minimized. This metric is easier and less expensive to compute compared to that used by the minimum fill-in criterion. If $\mathcal{G}(A)$ is a tree, then the MD algorithm results in no fill-in but, in most real applications, it does not minimize the amount of fill-in exactly.

The MD algorithm can be implemented using $\mathcal{G}(A)$ and it can predict the required factor storage without generating the structure of L. The basic approach is given in Algorithm 8.1. At step k, the number of off-diagonal nonzeros in a row or column of the active submatrix is the **current degree** of the corresponding vertex in the elimination graph $\mathcal{G}^k$. The algorithm selects a vertex of minimum current degree in $\mathcal{G}^k$ and labels it v_k, i.e. next for elimination. The set of vertices adjacent to v_k in $\mathcal{G}(A)$ is $\mathcal{R}each(v_k, \mathcal{V}_k)$, where $\mathcal{V}_k$ is the set of $k-1$ vertices that have already been eliminated. These are the only vertices whose degrees can change at step k. If $u \in \mathcal{R}each(v_k, \mathcal{V}_k)$, $u \neq v_k$, then its updated current degree is $|\mathcal{R}each(u, \mathcal{V}_{k+1})|$, where $\mathcal{V}_{k+1} = \mathcal{V}_k \cup v_k$.

At Step 4 of Algorithm 8.1, a tie-breaking strategy is needed when there is more than one vertex of current minimum degree. A straightforward strategy is to select the vertex that lies first in the original order. For the example in Figure 8.1, vertices 2, 3, and 6 are initially all of degree 2 and could be selected for elimination; as the lowest-numbered vertex, 2 is chosen. After it has been eliminated, vertices 3, 5, and 6 have current degree 2 and so vertex 3 is next. As all the remaining vertices have current degree 2, vertex 1 is eliminated, followed by 4, 5, and 6. It is possible to construct artificial matrices showing that some systematic tie-breaking choices can lead to a large amount of fill-in but such behaviour is not typical.

ALGORITHM 8.1 Basic minimum degree (MD) algorithm
Input: Graph $\mathcal{G}$ of a symmetrically structured matrix.
Output: A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.

1: Set $\mathcal{G}^1 = \mathcal{G}$ and compute the degree $deg_{\mathcal{G}^1}(u)$ of all $u \in \mathcal{V}(\mathcal{G}^1)$
2: **for** $k = 1 : n-1$ **do**
3: Compute $mdeg = \min\{deg_{\mathcal{G}^k}(u) \mid u \in \mathcal{V}(\mathcal{G}^k)\}$ ▷ $mdeg$ is the current minimum degree
4: Choose $v_k \in \mathcal{V}(\mathcal{G}^k)$ such that $deg_{\mathcal{G}^k}(v_k) = mdeg$
5: $p(k) = v_k$ ▷ v_k is the next vertex in the elimination order
6: Construct $\mathcal{G}^{k+1}$ and update the current degrees of its vertices
7: **end for**
8: $p(n) = v_n$ where v_n is the only vertex in $\mathcal{G}^n$

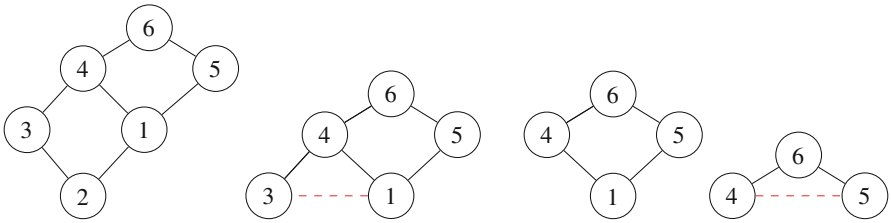


Figure 8.1 An illustration of three steps of the MD algorithm. The original graph $\mathcal{G}$ and the elimination graphs $\mathcal{G}^2$, $\mathcal{G}^3$ and $\mathcal{G}^4$ that result from eliminating vertex 2, then vertex 3 and then vertex 1 are shown red dashed lines denote fill edges.

The construction of each elimination graph $\mathcal{G}^{k+1}$ is central to the implementation of the MD algorithm. Because eliminating a vertex potentially creates fill-in, an efficient representation of the resulting elimination graph that accommodates this (either implicitly or explicitly) is needed. Moreover, recalculating the current degrees is time consuming. Consequently, various approaches have been developed to enhance performance; these are discussed in the following subsections.

8.1.3 Use of Indistinguishable Vertices

In Section 3.5.1, we introduced indistinguishable vertices and supervariables. The importance of exploiting these in MD algorithms is emphasized by the next two results. Here $\mathcal{G}_v$ denotes the elimination graph obtained from $\mathcal{G}$ when vertex $v \in \mathcal{V}(\mathcal{G})$ is eliminated.

Theorem 8.1 (George & Liu 1980b, 1989) *Let u and w be indistinguishable vertices in $\mathcal{G}$. If $v \in \mathcal{V}(\mathcal{G})$ with $v \neq u, w$, then u and w are indistinguishable in $\mathcal{G}_v$.*

Proof Two cases must be considered. First, let $u \notin adj_{\mathcal{G}}\{v\}$. Then $w \notin adj_{\mathcal{G}}\{v\}$ and if v is eliminated, the adjacency sets of u and w are unchanged and these vertices remain indistinguishable in the resulting elimination graph $\mathcal{G}_v$. Second, let $u, w \in adj_{\mathcal{G}}\{v\}$. When v is eliminated, because u and w are indistinguishable in $\mathcal{G}$, their adjacency sets in $\mathcal{G}_v$ will be modified in the same way, by adding the entries of $adj_{\mathcal{G}}\{v\}$ that are not already in $adj_{\mathcal{G}}\{u\}$ and $adj_{\mathcal{G}}\{w\}$. Consequently, u and w are indistinguishable in $\mathcal{G}_v$. □

Figure 8.2 demonstrates the two cases in the proof of Theorem 8.1. Here, u and w are indistinguishable vertices in $\mathcal{G}$. Setting $v \equiv v'$ illustrates $u \notin adj_{\mathcal{G}}\{v\}$. If v' is eliminated, then the adjacency sets of u and w are clearly unchanged. Setting $v \equiv v''$ illustrates $u, w \in adj_{\mathcal{G}}\{v\}$. In this case, if v'' is eliminated, then vertices s and t are added to both $adj_{\mathcal{G}}\{u\}$ and $adj_{\mathcal{G}}\{w\}$.

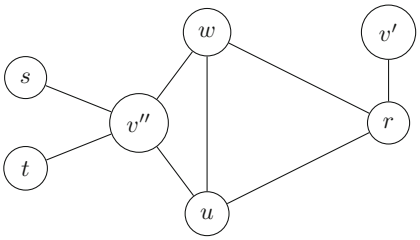


Figure 8.2 An example to illustrate Theorem 8.1. u and w are indistinguishable vertices in $\mathcal{G}$; $adj_{\mathcal{G}}\{u\} = \{r, w, v''\}$ and $adj_{\mathcal{G}}\{w\} = \{r, u, v''\}$.

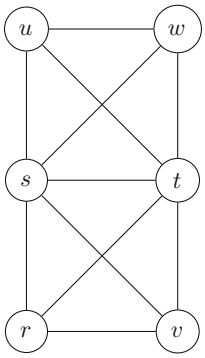


Figure 8.3 An illustration of Theorem 8.2. Vertices u and w are of minimum degree (with degree $mdeg = 3$) and are indistinguishable in $\mathcal{G}$. After elimination of w, the current degree of u is $mdeg - 1$ and the current degree of each of the other vertices is at most $mdeg - 1$. Therefore, u is of current minimum degree in $\mathcal{G}_w$. Note that vertices r and v are also of minimum degree and indistinguishable in $\mathcal{G}$; they are not neighbours of w and their degrees do not change when w is eliminated.

Theorem 8.2 (George & Liu 1980b, 1989) *Let u and w be indistinguishable vertices in $\mathcal{G}$. If w is of minimum degree in $\mathcal{G}$, then u is of minimum degree in $\mathcal{G}_w$.*

Proof Let $deg_{\mathcal{G}}(w) = mdeg$. Then $deg_{\mathcal{G}}(u) = mdeg$. Indistinguishable vertices are always neighbours. Eliminating w gives $deg_{\mathcal{G}_w}(u) = mdeg - 1$ because w is removed from the adjacency set of u and there is no neighbour of u in $\mathcal{G}_w$ that was not its neighbour in $\mathcal{G}$. If $x \neq w$ and $x \in adj_{\mathcal{G}}\{u\}$, then the number of neighbours of x in $\mathcal{G}_w$ is at least $mdeg - 1$. Otherwise, if $x \notin adj_{\mathcal{G}}\{u\}$, then its adjacency set in $\mathcal{G}_w$ is the same as in $\mathcal{G}$ and is of the size at least $mdeg$. The result follows. □

Theorem 8.2 is illustrated in Figure 8.3.

Theorems 8.1 and 8.2 can be extended to more than two indistinguishable vertices, which allows indistinguishable vertices to be selected one after another in the MD ordering. This is referred to as **mass elimination**. Treating indistinguishable vertices as a single supervariable cuts the number of vertices and edges in the elimination graphs, which reduces the work needed for degree updates.

In the basic MD algorithm, the current degree of a vertex is the number of adjacent vertices in the current elimination graph. The **external degree** of a vertex is the number of vertices adjacent to it that are not indistinguishable from it. The motivation comes from the underlying reason for the success of the minimum degree ordering in terms of fill reduction. Eliminating a vertex of minimum degree implies the formation of the smallest possible clique resulting from the elimination. If mass elimination is used, then the size of the resulting clique is equal to the external degree of the vertices eliminated by the mass elimination step. Using the external degree can speed up the time for computing the ordering and give worthwhile savings in the number of entries in the factors.

8.1.4 Degree Outmatching

A concept that is closely related to that of indistinguishable vertices is **degree outmatching**. This avoids computing the degrees of vertices that are known not to be of current minimum degree. Vertex w is said to be outmatched by vertex u if

$$adj_{\mathcal{G}}\{u\} \cup \{u\} \subseteq adj_{\mathcal{G}}\{w\} \cup \{w\}.$$

It follows that $deg_{\mathcal{G}}(u) \leq deg_{\mathcal{G}}(w)$. A simple example is given in Figure 8.4. Importantly, degree outmatching is preserved when vertex $v \in \mathcal{G}$ of minimum degree is eliminated, as stated in the following result.

Theorem 8.3 (George & Liu 1980b, 1989) *In the graph $\mathcal{G}$ let vertex w be outmatched by vertex u and vertex v ($v \neq u, w$) be of minimum degree. Then w is outmatched in $\mathcal{G}_v$ by u.*

Proof Three cases must be considered. First, if $u \notin adj_{\mathcal{G}}\{v\}$ and $w \notin adj_{\mathcal{G}}\{v\}$, then the adjacency sets of u and w in $\mathcal{G}_v$ are the same as in $\mathcal{G}$. Second, if v is a neighbour of both u and w in $\mathcal{G}$, then any neighbours of v that were not neighbours of u and

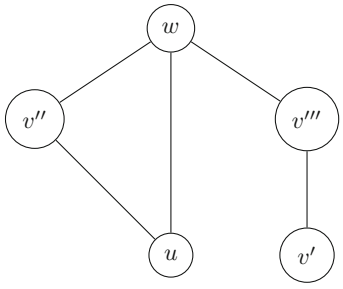


Figure 8.4 An example $\mathcal{G}$ in which vertex w is outmatched by vertex u. v' is not a neighbour of u or w; vertex v'' is a neighbour of both u and w; v''' is a neighbour of w but not of u.

w are added to their adjacency sets in $\mathcal{G}_v$. Third, if $u \notin adj_{\mathcal{G}}\{v\}$ and $w \in adj_{\mathcal{G}}\{v\}$, then the adjacency set of u in $\mathcal{G}_v$ is the same as in $\mathcal{G}$ but any neighbours of v that were not neighbours of w are added to the adjacency set of w in $\mathcal{G}_v$. In all three cases, w is still outmatched by u in $\mathcal{G}_v$. □

The three possible cases for v in the proof of Theorem 8.3 are illustrated in Figure 8.4 by setting $v \equiv v'$, v'' and v''', respectively. An important consequence of Theorem 8.3 is that if w is outmatched by u, then it is not necessary to consider w as a candidate for elimination and all updates to the data structures related to w can be postponed until u has been eliminated.

8.1.5 Cliques and Quotient Graphs

From Parter's rule, if vertex v is selected at step k, then the elimination matrix that corresponds to $\mathcal{G}^{k+1}$ contains a dense submatrix of size equal to the number of off-diagonal entries in row and column v in the matrix that corresponds to $\mathcal{G}^k$. For large matrices, creating and explicitly storing the edges in the sequence of elimination graphs is impractical and a more compact and efficient representation is needed. Each elimination graph can be interpreted as a collection of cliques, including the original graph $\mathcal{G}$, which can be regarded as having $|\mathcal{E}|$ cliques, each consisting of two vertices (or, equivalently, an edge). This gives a conceptually different view of the elimination process and provides a compact scheme to represent the elimination graphs. The advantage in terms of storage is based on the following.

Let $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_q\}$ be the set of cliques for the current graph and let v be a vertex of current minimum degree that is selected for elimination. Let $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ be the subset of cliques to which v belongs. Two steps are then required.

1. Remove the cliques $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ from $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_q\}$.
2. Add the new clique $\mathcal{V}_v = \{\mathcal{V}_{s_1} \cup \ldots \cup \mathcal{V}_{s_t}\} \setminus \{v\}$ into the set of cliques.

Hence

$$deg_{\mathcal{G}}(v) = |\mathcal{V}_v| < \sum_{i=1}^{t} |\mathcal{V}_{s_i}|,$$

and because $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ can now be discarded, the storage required for the representation of the sequence of elimination graphs never exceeds that needed for $\mathcal{G}(A)$. The storage to compute an MD ordering is therefore known beforehand in spite of the rather dynamic nature of the elimination process. The index of the eliminated vertex can be used as the index of the new clique. This is called an **element** or **enode** (the terminology comes from finite-element methods), to distinguish it from an uneliminated vertex, which is termed an **snode**.

A sequence of special quotient graphs $\mathcal{G}^{[1]} = \mathcal{G}(A), \mathcal{G}^{[2]}, \ldots, \mathcal{G}^{[n]}$ with the two types of vertices is generated in place of the elimination graphs. Each $\mathcal{G}^{[k]}$ has n vertices that satisfy

$$\mathcal{V}(\mathcal{G}) = \mathcal{V}_{snodes} \cup \mathcal{V}_{enodes}, \qquad \mathcal{V}_{snodes} \cap \mathcal{V}_{enodes} = \emptyset,$$

where $\mathcal{V}_{snodes}$ and $\mathcal{V}_{enodes}$ are the sets of snodes and enodes, respectively. When v is eliminated, any enodes adjacent to it are no longer required to represent the sparsity pattern of the corresponding active submatrix and so they can be removed. This is called **element absorption**.

Working with these graphs can be demonstrated by considering the computation of the vertex degrees. To compute the degree of an uneliminated vertex, the set of neighbouring snodes is counted. Then, if a neighbour of one of these snodes is an enode, its neighbours are also counted (avoiding double counting). More formally, if $v \in \mathcal{V}_{snodes}$, then the adjacency set of v is the union of its neighbours in $\mathcal{V}_{snodes}$ and the vertices reachable from v via its neighbours in $\mathcal{V}_{enodes}$. In this way, vertex degrees are computed by considering fill-paths, avoiding storing the fill-in entries explicitly. This reduces memory requirements and contributes to the computational efficiency, which can be further improved by amalgamating sets of indistinguishable enodes and snodes.

The sequences of elimination graphs and quotient graphs are illustrated in Figure 8.5. The top line shows $\mathcal{G}$ together with $\mathcal{G}^2$ and $\mathcal{G}^3$ after the elimination of vertices 1 and 2, respectively. When vertex 1 is eliminated, a new edge is added to make its neighbours into a clique. The elimination of vertex 2 creates no additional fill and the graph $\mathcal{G}^3$ with three nodes represents the sparsity structure of the corresponding active submatrix $A^{(3)}$. The bottom line shows the corresponding quotient graphs. After the first elimination, vertex 1 is an enode and the fill edge is represented implicitly. After the second elimination, the enodes 1 and 2 can be amalgamated and so too can the snodes 3 and 4 because they are indistinguishable.

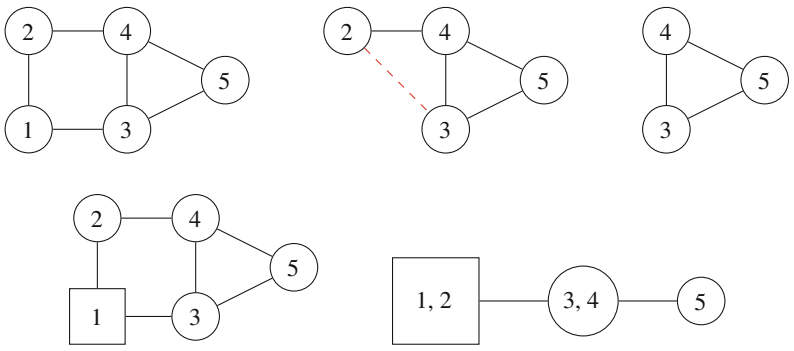


Figure 8.5 The top line shows $\mathcal{G} = \mathcal{G}^1$, $\mathcal{G}^2$ and $\mathcal{G}^3$. The red dashed line denotes a fill edge. The bottom line shows the quotient graphs $\mathcal{G}^{[2]}$ and $\mathcal{G}^{[3]}$ after the first and second elimination steps. A circle represents a vertex in $\mathcal{G}$ (an snode), while a square represents an enode.

ALGORITHM 8.2 Basic multiple minimum degree (MMD) algorithm
Input: Graph $\mathcal{G}$ of a symmetrically structured matrix.
Output: A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.

Set $k = 1$, $\mathcal{G}^1 = \mathcal{G}$ and compute the degree $deg_{\mathcal{G}^1}(u)$ of all $u \in \mathcal{V}(\mathcal{G}^1)$
while $k \leq n$ **do**
    Compute $mdeg = \min\{deg_{\mathcal{G}^k}(u) \mid u \in \mathcal{V}(\mathcal{G}^k)\}$
    Find all mutually non-adjacent $v_j \in \mathcal{V}(\mathcal{G}^k)$, $j = 1, \ldots, t$ with $deg_{\mathcal{G}^k}(v_j) = mdeg$
    for $j = 1 : t$ **do**
        $p(k) = v_j$ ▷ Vertex v_j is the next vertex in the elimination order
        $k = k + 1$
    end for
    if $k < n$ **then**
        Construct $\mathcal{G}^{k+1}$ and update the current degrees of its vertices
    end if
end while

8.1.6 *Multiple Minimum Degree (MMD) Algorithm*

The multiple minimum degree (MMD) algorithm aims to improve efficiency by processing several independent vertices that are each of minimum current degree together in the same step, before the degree updates are performed. The basic approach is outlined as Algorithm 8.2. At each outer loop, $t \geq 1$ denotes the number of vertices of minimum current degree that are mutually non-adjacent and so can be put into the elimination order one after another. An example in which the four corner vertices have the same minimum degree is depicted in Figure 8.6. Here, on the first step, $mdeg = 2$ and $t = 4$. Note that the MMD strategy is complementary to the mass elimination approach in which the set S of indistinguishable vertices that can be eliminated one after another is fully interconnected and all vertices of S have the same set of neighbours outside S.

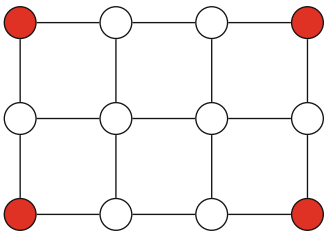


Figure 8.6 The red (corner) vertices of $\mathcal{G}$ are each of degree 2 and are ordered consecutively during the first step of Algorithm 8.2.

The complexity of the MD and MMD algorithms is $O(nz(A)n^2)$ but because for MMD the outer loop of the algorithm update is performed fewer times, it can be significantly faster than MD. MMD orderings can also lead to less fill-in, possibly a consequence of introducing some kind of regularity into the ordering sequence.

8.1.7 Approximate Minimum Degree (AMD) Algorithm

The idea behind the widely used **approximate minimum degree** (AMD) algorithm is to inexpensively compute an upper bound on a vertex degree in place of the degree, and to use this bound as an approximation to the external degree when selecting vertices within the MD algorithm. Even though vertex degrees are not determined exactly, the quality of the orderings obtained using the AMD algorithm are competitive with those computed using the MD algorithm and can surpass them. The complexity of AMD is $O(nz(A)n)$ and, in practice, its runtime is typically significantly less than that of the MD and MMD approaches.

8.2 Minimizing the Bandwidth and Profile

An alternative way of reducing the fill-in locally is to add another criterion to the relabelling of the vertices, such as restricting the nonzeros of the permuted matrix to specific positions. The most popular approach is to force them to lie close to the main diagonal. If Gaussian elimination is applied without further permutations, then all fill-in takes place between the first entry of a row and the diagonal or between the first entry of a column and the diagonal. It is therefore sufficient to store all the entries in the lower triangular part from the first entry in each row to the diagonal and all the entries in the upper triangular part from the first entry in each column to the diagonal. This allows straightforward implementations of Gaussian elimination that employ static data structures. Here we again consider symmetric and, for simplicity, we assume that $\mathcal{G}(A)$ is connected; generalizations of the terminology and ideas to nonsymmetric matrices are possible.

8.2.1 The Band and Envelope of a Matrix

To characterize the positions within $\mathcal{S}\{A\}$ that are close to the main diagonal, we denote the leftmost entries in the lower triangular part of A using the mapping η_i as follows:

$$\eta_i(A) = \min\{j \mid 1 \le j \le i \text{ with } a_{ij} \neq 0\}, \quad 1 \le i \le n, \tag{8.1}$$

that is, $\eta_i(A)$ is the column index of the first entry in the i-th row of A.

Define

$$\beta_i(A) = i - \eta_i(A), \quad 1 \le i \le n.$$

The **semibandwidth** of A is

$$\max\{\beta_i(A) |\ 1 \le i \le n\},$$

and the **bandwidth** is

$$\beta(A) = 2 * \max\{\beta_i(A) \mid\ 1 \le i \le n\} + 1.$$

The **band** of A is the following set of index pairs in A

$$band(A) = \{(i, j) \mid\ 0 < i - j \le \beta(A)\}.$$

The **envelope** is the set of index pairs that lie between the first entry in each row and the diagonal

$$env(A) = \{(i, j) \mid\ 0 < i - j \le \beta_i(A)\}.$$

Note that the band and envelope of a sparse symmetrically structured matrix A include only entries of the strict lower triangular part of A. The envelope is easily visualized: picture the lower triangular part of A, and remove the diagonal and the leading zero entries in each row. The remaining entries (whether nonzero or zero) comprise the envelope of A. The **profile** of A is defined to be the number of entries in the envelope (the envelope size) plus n.[1] An illustrative example is given in Figure 8.7. Here $\eta_1(A) = 1$, $\beta_1(A) = 0$, $\eta_2(A) = 1$, $\beta_2(A) = 1$, $\eta_3(A) = 2$, $\beta_3(A) = 1$, and so on.

Figure 8.7 Illustration of the band and envelope of a matrix A whose sparsity pattern is on the left. In the centre, the positions of $band(A)$ are circled and on the right, the positions of $env(A)$ are circled. The bandwidth is 5 and the envelope size and the profile are 7 and 14, respectively.

[1] Sometimes in the literature the profile is defined to be the envelope size.

The next result shows that the static data structures determined for A are sufficient for its Cholesky factors and by permuting A to minimize its band or profile, the fill-in is also approximately minimized.

Theorem 8.4 (Liu & Sherman 1976; George & Liu 1981) *If L is the Cholesky factor of A, then*

$$env(A) = env(L).$$

Proof The proof uses mathematical induction on the principal leading submatrices of A of order k. The result is clearly true for $k = 1$ and $k = 2$. Assume it holds for $2 \leq k < n$ and consider the block factorization

$$\begin{pmatrix} A_{1:k,1:k} & u_{1:k} \\ u_{1:k}^T & \alpha \end{pmatrix} = \begin{pmatrix} L_{1:k,1:k} & 0 \\ v_{1:k}^T & \beta \end{pmatrix} \begin{pmatrix} L_{1:k,1:k}^T & v_{1:k} \\ 0 & \beta \end{pmatrix},$$

where α and β are scalars. Equating the left and right sides, $L_{1:k,1:k} v_{1:k} = u_{1:k}$. Because $u_j = 0$ for $j < \eta_{k+1}(A)$ and $u_{\eta_{k+1}} \neq 0$, it follows that $v_j = 0$ for $j < \eta_{k+1}(A)$ and $v_{\eta_{k+1}} \neq 0$. This proves the induction step. □

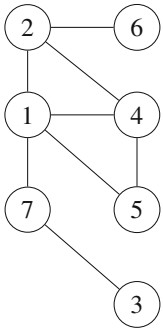
A straightforward corollary of Theorem 8.4 is that $band(A) = band(L)$.

8.2.2 Level-Based Orderings

Finding a permutation P to minimize the band or profile of PAP^T is combinatorially hard and again heuristics are used to efficiently find an acceptable P. The popular Cuthill McKee (CM) approach chooses a suitable starting vertex s and labels it 1. Then, for $i = 1, 2, \ldots, n-1$, all vertices adjacent to vertex i that are still unlabelled are labelled successively in order of increasing degree, as described in Algorithm 8.3. A very important variation is the Reverse Cuthill McKee (RCM) algorithm, which incorporates a final step in which the CM ordering is reversed. The CM- and RCM-permuted matrices have the same bandwidth but the latter can decrease the envelope, as demonstrated in Figure 8.8.

The importance of the CM and RCM orderings is expressed in the following theorem. The full envelope of the Cholesky factor of the permuted matrix implies cache efficiency when performing the triangular solves once the factorization is complete.

Theorem 8.5 (Liu & Sherman 1976; George & Liu 1981) *Let A be symmetrically structured and irreducible. If P corresponds to the CM labelling obtained from Algorithm 8.3 and L is the Cholesky factor of $P^T AP$, then $env(L)$ is full, that is, all entries of the envelope are nonzero.*



$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{pmatrix}
* & * & & * & * & & * \\
* & * & & * & & * & \\
 & & * & & & & * \\
* & * & & * & * & & \\
* & & & * & * & & \\
 & * & & & & * & \\
* & & * & & & & *
\end{pmatrix}
\end{array}
,
\begin{array}{c} \\ 3 \\ 7 \\ 1 \\ 5 \\ 2 \\ 4 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 3 & 7 & 1 & 5 & 2 & 4 & 6 \end{array} \\
\begin{pmatrix}
* & * & & & & & \\
* & * & * & & & & \\
 & * & * & * & * & * & \\
 & & * & * & & * & \\
 & & * & & * & * & * \\
 & & * & * & * & * & \\
 & & & & * & & *
\end{pmatrix}
\end{array}
,
\begin{array}{c} \\ 6 \\ 4 \\ 2 \\ 5 \\ 1 \\ 7 \\ 3 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 6 & 4 & 2 & 5 & 1 & 7 & 3 \end{array} \\
\begin{pmatrix}
* & & * & & & & \\
 & * & * & * & * & & \\
* & * & * & & * & & \\
 & * & & * & * & & \\
 & * & * & * & * & * & \\
 & & & & * & * & * \\
 & & & & & * & *
\end{pmatrix}
\end{array}
,
$$

Figure 8.8 An example to illustrate Algorithm 8.3. The starting vertex is $s = 3$; it has degree 1. The graph $\mathcal{G}(A)$ is given and the sparsity patterns of A (left), A symmetrically permuted by the CM algorithm (centre) and A symmetrically permuted by the RCM algorithm (right). The profiles of these matrices are 25, 17, and 16, respectively.

A crucial difference between profile reduction ordering algorithms and minimum degree strategies is that the former is based solely on $\mathcal{G}$: the costly construction of quotient graphs is not needed. However, unless the profile after reordering is very small, there can be significantly more fill-in in the factor.

Key to the success of Algorithm 8.3 is the choice of the starting vertex s: the quality of the ordering is highly dependent on s. A good candidate is a vertex for which the maximum distance between it and some other vertex in $\mathcal{G}$ is large. Formally, the **eccentricity** $\epsilon(u)$ of the vertex u in the connected undirected graph $\mathcal{G}$ is defined to be

$$\epsilon(u) = \max\{d(u, v) \mid v \in \mathcal{V}\},$$

where $d(u, v)$ is the distance between the vertices u and v (the length of the shortest path between these vertices). The maximum eccentricity taken over all the vertices is the **diameter** of $\mathcal{G}$ (that is, the maximum distance between any pair of vertices). The endpoints of a diameter (also termed **peripheral vertices**) provide good starting vertices. The complexity of finding a diameter is $O(n^3)$ because the shortest paths amongst all the vertices have to be checked. Thus, a pseudo-diameter defined by any pair of vertices for which $d(u, v)$ is close to the diameter is used instead. The vertices defining a pseudo-diameter are **pseudo-peripheral** vertices.

ALGORITHM 8.3 CM and RCM algorithms for band and profile reduction
Input: Graph $\mathcal{G}$ of a symmetrically structured irreducible matrix and a starting vertex s.
Output: Permutation vectors p_{cm} and p_{rcm} that define new labellings of the vertices of $\mathcal{G}(A)$.

1: $label(1:n) = false$
2: Compute $adj_{\mathcal{G}}\{u\}$ and $deg_{\mathcal{G}}(u)$ for all $u \in \mathcal{V}(\mathcal{G})$
3: $k = 1,\ v_1 = s,\ p_{cm}(1) = v_1,\ label(v_1) = true$
4: **for** $i = 1 : n-1$ **do**
5: **for** $w \in adj_{\mathcal{G}}\{v_i\}$ with $label(w) = false$ in order of increasing degree **do**
6: $k = k+1,\ v_k = w,\ p_{cm}(k) = v_k,\ label(v_k) = true$
7: **end for**
8: **end for**
9: For the RCM ordering, $p_{rcm}(i) = p_{cm}(n-i+1), i = 1, 2, \ldots, n$.

A heuristic algorithm is used to find pseudo-peripheral vertices. A commonly used approach is based on level sets. A level structure rooted at a vertex r is defined as the partitioning of $\mathcal{V}$ into disjoint **levels** $\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_h(r)$ such that

(i) $\mathcal{L}_1(r) = \{r\}$ and
(ii) for $1 < i \leq h$, $\mathcal{L}_i(r)$ is the set of all vertices that are adjacent to vertices in $\mathcal{L}_{i-1}(r)$ but are not in $\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_{i-1}(r)$.

The level structure rooted at r may be expressed as the set $\mathcal{L}(r) = \{\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_h(r)\}$, where h is the total number of levels and is termed the **depth**. The level sets can be found using a breadth-first search that starts at the root r. The Gibbs-Poole-Stockmeyer (GPS) algorithm presented as Algorithm 8.4 can be used to finding pseudo-peripheral vertices, one of which may then be used as a starting vertex for the CM and RCM algorithms. Here the root vertex r is normally taken to be an arbitrary vertex of minimum degree. $\mathcal{L}(r)$ is constructed and then the level structures rooted at each of the vertices in the last level set $\mathcal{L}_h(r)$. If, for some $w \in \mathcal{L}_h(r)$, the depth of $\mathcal{L}_w$ exceeds that of $\mathcal{L}(r)$, w replaces r as the root vertex, and the procedure is repeated. If no such vertex is found, r is chosen as a pseudo-peripheral vertex.

A simple example is given in Figure 8.9. Starting with $r = 2$, after two passes through the while loop, the GPS algorithm returns $s = 8$ and $t = 1$ as pseudo-peripheral vertices.

To obtain an efficient implementation of the GPS algorithm, it is necessary to limit the number of level set structures that are fully constructed. For example, "short circuiting" can be incorporated in which wide level structures are rejected as soon as they are detected (wide levels will not lead to a deep level structure which is

ALGORITHM 8.4 Basic GPS algorithm to find a pair of pseudo-peripheral vertices

Input: Graph $\mathcal{G}$ of a symmetrically structured irreducible matrix and a root vertex r.

Output: Pseudo-peripheral vertices s, t.

1: Construct $\mathcal{L}(r)$ and set $flag = false$
2: **while** $flag = false$ **do**
3: $flag = true$
4: **for** $i = 1 : |\mathcal{L}(r)|$ **do**
5: $w_i \in \mathcal{L}(r)$ ▷ Select vertex w_i from last level set
6: **if** $flag = true$ **then**
7: Construct $\mathcal{L}(w_i)$
8: **if** $depth(\mathcal{L}(w_i)) > depth(\mathcal{L}(r))$ **then**
9: $flag = false$ ▷ Flag that w_i will be used as new initial vertex
10: **end if**
11: **end if**
12: **end for**
13: **if** $flag = true$ **then**
14: $s = r$ and $t = w_i$ ▷ s is chosen; while loop will terminate algorithm
15: **else**
16: $r = w_i$
17: **end if**
18: **end while**

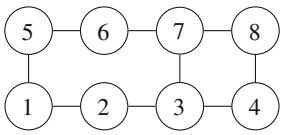


Figure 8.9 An example to illustrate Algorithm 8.4 for finding pseudo-peripheral vertices. With root vertex $r = 2$, the first level set structure is $\mathcal{L}(2) = \{\{2\}, \{1, 3\}, \{4, 5, 7\}, \{6, 8\}\}$. Setting $r = 8$ at Step 16, the second level set structure is $\mathcal{L}(8) = \{\{8\}, \{4, 7\}, \{3, 6\}, \{2, 5\}, \{1\}\}$ and the algorithm terminates with $s = 8$ and $t = 1$.

needed for a narrow band). Furthermore, to reduce the number of vertices in the last level set $\mathcal{L}_h(r)$ for which it is necessary to generate the rooted level structures, a "shrinking" strategy can be used. This typically involves considering the degrees of the vertices in $\mathcal{L}_h(r)$ (for example, only those of smallest degree will be tried). Such modifications can lead to significant time savings while still returning a good starting vertex for the CM and RCM algorithms. As with the MD algorithm, tie-breaking rules must be built into any implementation.

8.2.3 Spectral Orderings

Spectral methods offer an alternative approach that does not use level structures. The spectral algorithm associates a positive semidefinite Laplacian matrix L_p with the symmetric matrix A as follows:

$$(L_p)_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } a_{ij} \neq 0, \\ deg_{\mathcal{G}}(i) & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

An eigenvector corresponding to the smallest positive eigenvalue of the Laplacian matrix is called a **Fiedler vector.** If $\mathcal{G}$ is connected, L_p is irreducible and the second smallest eigenvalue is positive. The vertices of $\mathcal{G}$ are ordered by sorting the entries of the Fiedler vector into monotonic order. Applying the permutation symmetrically to A yields the spectral ordering.

The use of the Fiedler vector for reordering A comes from considering the matrix envelope. The size of the envelope can be written as

$$|env(A)| = \sum_{i=1}^{n} \beta_i = \sum_{i=1}^{n} \max_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k).$$

Observation 8.1 implies that the asymptotic upper bound on the operation count for the factorization based on $env(A)$ is

$$work_{env} = \sum_{i=1}^{n} \beta_i^2 = \sum_{i=1}^{n} \max_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k)^2.$$

Ordering the vertices using the Fiedler vector is closely related to minimizing $weight_{env}$ over all possible vertex reorderings, where

$$weight_{env} = \sum_{i=1}^{n} \sum_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k)^2.$$

Thus, while minimizing the profile and envelope is related to the infinity norm, minimizing $weight_{env}$ is related to the Euclidean norm of the distance between graph vertices.

Although computing the Fiedler vector can be computationally expensive it does have the advantage of easy vectorization and parallelization and the resulting ordering can give small profiles and low operation counts.

8.3 Local fill-reducing orderings for nonsymmetric $\mathcal{S}\{A\}$

If $\mathcal{S}\{A\}$ is nonsymmetric, then an often-used strategy is to apply the minimum degree algorithm (or one of its variants) or a band or profile-reducing ordering to the undirected graph $\mathcal{G}(A+A^T)$. This can work well if the symmetry index $s(A)$ is close to 1. But if A is highly nonsymmetric (typically, for values of $s(A)$ less than 0.5, A is considered to be highly nonsymmetric), then a different approach is required. **Markowitz pivoting** generalizes the MD algorithm by choosing the pivot entry based on vertex degrees computed directly from the nonsymmetric $\mathcal{S}\{A\}$; the result is a nonsymmetric permutation. It can be described using a sequence of bipartite graphs of the active submatrices but here we use a matrix-based description that permutes A on-the-fly. Note that Markowitz pivoting is generally incorporated into the numerical factorization phase of an LU solver, rather than being used to derive an initial reordering of A.

At step k of the LU factorization, consider the $(n-k+1) \times (n-k+1)$ active submatrix, that is, the Schur complement $S^{(k)}$ given by (3.2). Let $nz(row_i)$ and $nz(col_j)$ denote the number of entries in row i and column j of $S^{(k)}$ $(1 \le i, j \le n-k+1)$. Markowitz pivoting selects as the k-th pivot the entry of $S^{(k)}$ that minimizes the **Markowitz count** given by the product

$$(nz(row_i)-1)(nz(col_j)-1).$$

This strategy is summarized in Algorithm 8.5 and illustrated in Figure 8.10. Here the first pivot is a_{24} with Markowitz count 1; it does not cause fill-in. The second pivot has Markowitz count 2 in $S^{(2)}$; it results in one filled entry. Note that the interchanges of rows and columns that are potentially performed at each of the first $n-1$ steps of the factorization give the row and column permutation matrices on the output of Algorithm 8.5. Implementation of the algorithm requires access to the rows and the columns of the matrix.

ALGORITHM 8.5 Markowitz pivoting

Input: Matrix A with a nonsymmetric sparsity pattern.

Output: $A' = PAQ$, where P and Q are permutation matrices chosen to limit fill in.

1: Set $S^{(1)} = A$ and $A' = A$
2: **for** $k = 1 : n-1$ **do**
3: Compute $nz(row_i)$ and $nz(col_j)$ $(1 \le i, j \le n-k+1)$
4: Find an entry $s_{ij}^{(k)}$ of $S^{(k)}$ that minimizes $(nz(row_i)-1)(nz(col_j)-1)$
5: Permute the rows and columns so that $s_{ij}^{(k)}$ is the (1, 1) entry of the permuted $S^{(k)}$
6: Compute Schur complement $S^{(k+1)}$ of the permuted $S^{(k)}$ with respect to its (1, 1) entry
7: **end for**

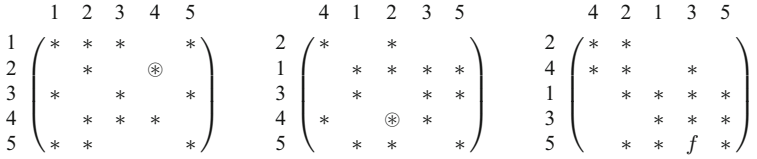


Figure 8.10 Illustration of Markowitz pivoting. The first and second pivots are circled. The sparsity pattern of $A = S^{(1)}$ is on the left. In the centre is the sparsity pattern after permuting the pivot in position $(2, 4)$ to the $(1, 1)$ position of $S^{(1)}$. There is no fill-in after the first factorization step. On the right is the sparsity pattern after selecting the second pivot that has the original position $(4, 2)$ and permuting it to the $(1, 1)$ position of $S^{(2)}$. The resulting filled entry is denoted by f. Note that the nonsymmetric permutations transform the originally irreducible matrix into a reducible one.

Markowitz pivoting as described here only considers the sparsity of A and the subsequent Schur complements. In practice, the pivoting strategy also needs to avoid small pivots because, as discussed in the last chapter, they can lead to numerical instability. A simple improvement is to break ties in Step 4 by choosing from the entries with the minimum Markowitz count the one of largest absolute value.

Because computing row and column counts is expensive, practical implementations may restrict computing them to a limited number of rows and columns. Alternatively, the search may be restricted to a predetermined number of rows of lowest row count (typically two or three rows), choosing entries with best Markowitz count and breaking ties on numerical grounds. Another option is to restrict the pivot choice to diagonal entries, in which case A is permuted symmetrically.

Algorithm 8.5 needs storage formats that can accommodate dynamic changes to the Schur complements. For example, the DS format described in Section 1.3.2, which allows access to both the rows and the columns. However, this format is only feasible if the amount of fill-in during the factorization is not large.

8.4 Global Nested Dissection Orderings

Nested dissection is the most important and widely used global ordering strategy for direct methods when $\mathcal{S}\{A\}$ is symmetric; it is particularly effective for ordering very large matrices. It proceeds by identifying a small set of vertices $\mathcal{V}_\mathcal{S}$ (known as a **vertex separator**) that if removed separates the graph into two disjoint subgraphs described by the vertex subsets $\mathcal{B}$ and $\mathcal{W}$ (commonly called "black" and "white", respectively). The rows and columns belonging to $\mathcal{B}$ are labelled first, then those belonging to $\mathcal{W}$ and finally those in $\mathcal{V}_\mathcal{S}$. The reordered matrix has the form

$$\begin{pmatrix} A_{\mathcal{B},\mathcal{B}} & 0 & A_{\mathcal{B},\mathcal{V}_\mathcal{S}} \\ 0 & A_{\mathcal{W},\mathcal{W}} & A_{\mathcal{W},\mathcal{V}_\mathcal{S}} \\ A^T_{\mathcal{B},\mathcal{V}_\mathcal{S}} & A^T_{\mathcal{W},\mathcal{V}_\mathcal{S}} & A_{\mathcal{V}_\mathcal{S},\mathcal{V}_\mathcal{S}} \end{pmatrix}. \tag{8.2}$$

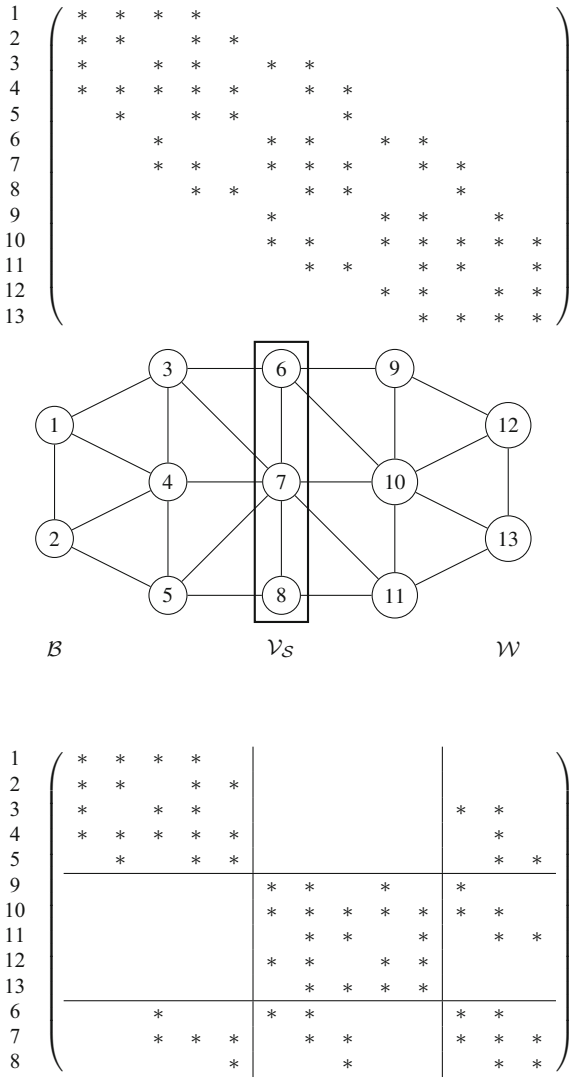


Figure 8.11 A simple example to illustrate nested dissection. The pattern of the original matrix (top), the partitioned graph (centre), and the corresponding symmetrically permuted matrix (bottom) are given.

This is shown for a 13 × 13 example in Figure 8.11. Provided the variables are eliminated in the permuted order, no fill occurs within the zero off-diagonal blocks. If $|\mathcal{V}_S|$ is small and $|\mathcal{B}|$ and $|\mathcal{W}|$ are similar, these zero blocks account for approximately half the possible entries in the matrix. The reordering can be applied recursively to the submatrices $A_{\mathcal{B},\mathcal{B}}$ and $A_{\mathcal{W},\mathcal{W}}$ until the vertex subsets

ALGORITHM 8.6 Nested dissection algorithm
Input: Graph $\mathcal{G}$ of a symmetrically structured matrix A and a partitioning algorithm **PartitionAlg**.
Output: A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.

recursive function (p = **nested_dissection**(A, **PartitionAlg**))
  if dissection has terminated **then** ▷ Vertex subsets are smaller than some threshold
    $p = \text{AMD}(\mathcal{V}, \mathcal{E})$ ▷ Compute an AMD ordering
  else
    Use **PartitionAlg**($\mathcal{V}, \mathcal{E}$) to obtain the vertex partitioning ($\mathcal{B}, \mathcal{W}, \mathcal{V}_\mathcal{S}$)
    $p_\mathcal{B}$ = **nested_dissection**($A_{\mathcal{B},\mathcal{B}}$, **PartitionAlg**)
    $p_\mathcal{W}$ = **nested_dissection**($A_{\mathcal{W},\mathcal{W}}$, **PartitionAlg**)
    $p_{\mathcal{V}_\mathcal{S}}$ is an ordering of $\mathcal{V}_\mathcal{S}$
    Set $p = \begin{pmatrix} p_\mathcal{B} \\ p_\mathcal{W} \\ p_{\mathcal{V}_\mathcal{S}} \end{pmatrix}$
  end if
end recursive function

are of size less than some prescribed threshold. At this stage, a local ordering technique (such as AMD) is normally more effective than nested dissection, and so a switch is made. The general form of the nested dissection algorithm is summarized in Algorithm 8.6. The parameter **PartitionAlg** specifies the algorithm used in determining the partitioning of the vertices. The performance and efficacy is highly dependent on the choice of **PartitionAlg**. Originally, level set based methods were used but most current approaches use multilevel techniques that create a hierarchy of graphs, each representing the original graph, but with a smaller dimension. The smallest (that is, the coarsest) graph in the sequence is partitioned. This partition is propagated back through the sequence of graphs, while being periodically refined.

8.5 Bordered Forms

Another possibility to exploit the global matrix structure is to use bordered block forms. These forms can arise naturally in some practical applications.

8.5.1 Doubly Bordered Form

The matrix (8.2) is an example of a **doubly bordered block diagonal (DBBD)** form. More generally, a matrix is said in DBBD form if it has the block structure

$$A_{DB} = \begin{pmatrix} A_{1,1} & & & & C_1 \\ & A_{2,2} & & & C_2 \\ & & \ldots & & . \\ & & & A_{Nb,Nb} & C_{Nb} \\ R_1 & R_2 & \ldots & R_{Nb} & B \end{pmatrix}, \tag{8.3}$$

where $Nb > 1$, the blocks $A_{lb,lb}$ on the diagonal are **square** $n_{lb} \times n_{lb}$ matrices and the border blocks C_{lb} and R_{lb} are $n_{lb} \times n_S$ and $n_S \times n_{lb}$ matrices, respectively, with $n_S \ll n_{lb}$ ($1 \le lb \le Nb$). B is an $n_S \times n_S$ matrix. The blocks can have very different sizes. A nested dissection ordering can be used to permute a symmetrically structured matrix A to a symmetrically structured DBBD form ($\mathcal{S}\{R_i\} = \mathcal{S}\{C_i^T\}$). If $\mathcal{S}\{A\}$ is close to symmetric, then nested dissection can be applied to $\mathcal{S}\{A + A^T\}$. In finite-element applications, the DBBD form corresponds to partitioning the underlying finite-element domain into non-overlapping subdomains; each $A_{lb,lb}$ represents the interior of a subdomain and the variables in the borders are those that lie on an interface between two or more subdomains.

Coarse-grained parallel approaches aim to factorize the $A_{lb,lb}$ blocks in parallel before solving the interface problem that connects the blocks. The block factorization of A_{DB} is

$$A_{DB} = \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ldots & & \\ & & & L_{Nb} & \\ \widehat{R}_1 & \widehat{R}_2 & \ldots & \widehat{R}_{Nb} & L_S \end{pmatrix} \begin{pmatrix} U_1 & & & & \widehat{C}_1 \\ & U_2 & & & \widehat{C}_2 \\ & & \ldots & & . \\ & & & U_{Nb} & \widehat{C}_{Nb} \\ & & & & U_S \end{pmatrix},$$

where

$$\widehat{R}_{lb} = R_{lb} U_{lb}^{-1}, \quad \widehat{C}_{lb} = L_{lb}^{-1} C_{lb} \ (1 \le lb \le Nb), \quad L_S U_S = B - \sum_{lb=1}^{Nb} \widehat{R}_{lb} \widehat{C}_{lb}.$$

The process is summarized in Algorithm 8.7. Here, for simplicity of notation, the permutation matrices for the block factorizations are set to the identity; in practice, $A_{lb,lb} = P_{lb} L_{lb} U_{lb} Q_{lb}$ for some permutation matrices P_{lb} and Q_{lb} ($1 \le lb \le Nb$) and $S = P_S L_S U_S Q_S$ for some permutation matrices P_s and Q_S.

There are several opportunities to incorporate parallelism. First, the factorizations of the blocks $A_{lb,lb}$ on the diagonal are completely independent. In addition,

ALGORITHM 8.7 Coarse-grained parallel LU factorization using DBBD form
Input: Matrix A_{DB} in DBBD form (8.3).
Output: Block LU factorization.

1: Initialise $S = B$
2: **for** $lb = 1 : Nb$ **do**
3: $A_{lb,lb} = L_{lb}U_{lb}$ ▷ LU factorization of square block on diagonal
4: $\widehat{R}_{lb} = R_{lb}U_{lb}^{-1}$ ▷ Triangular solve for bottom-border blocks
5: $\widehat{C}_{lb} = L_{lb}^{-1}C_{lb}$ ▷ Triangular solve for right-border blocks
6: **end for**
7: $S = S - \sum_{lb=1}^{Nb} \widehat{R}_{lb}\widehat{C}_{lb}$ ▷ Assemble updates to interface block
8: $S = L_S U_S$ ▷ Factorize updated interface block (Schur complement)

the factorization of each individual $A_{lb,lb}$ can be parallelized. The same is true for the triangular solves that update the border blocks. Second, the assembly of the interface block S can be partially parallelized (it can be started as soon as the first updated border blocks are available). Third, the LU factorization of S can be parallelized.

Observe that S is generally significantly denser than the other blocks and can present a computational bottleneck. In fact, not only is factorizing S expensive in terms of the memory and operations required, assembly updates to it can be time consuming. This is because multiple submatrices may contribute to the same entry of S, and these cannot be performed at the same time. Furthermore, for an efficient parallel implementation, load balance must be considered. If the work required for factorizing each of the blocks on the diagonal is not similar, then the time will be dominated by the most expensive block. One possible solution is to choose Nb to be greater than the number of processors and use dynamic scheduling to achieve good load balance. Unfortunately, if the number of blocks increases, so too does the size of S.

If A is not SPD, then factorizing the $A_{lb,lb}$ blocks without considering the entries in the border can potentially lead to stability problems. Consider the first step in factorizing $A_{lb,lb}$ and the threshold pivoting test (7.5) for a sparse LU factorization. The pivot candidate $(A_{lb,lb})_{11}$ must satisfy

$$\max\{\max_{i>1} |(A_{lb,lb})_{i1}|, \max_{k} |(R_{lb})_{k1}|\} \leq \gamma^{-1}|(A_{lb,lb})_{11}|,$$

where $\gamma \in (0, 1]$ is the threshold parameter. Large entries in the row border matrix R_{lb} can prevent pivots being selected within $A_{lb,lb}$. Stability can be maintained by moving rows and columns that cannot be eliminated to the borders. This increases the border size and may adversely affect the a priori sparse data structures for holding the factors, increase the work required to perform the factorization, and reduce the potential for parallelism within the factorization of the block.

8.5.2 Singly Bordered Form

An alternative strategy is to permute A to **singly bordered block diagonal (SBBD)** form

$$A_{SB} = \begin{pmatrix} A_{1,1} & & & C_1 \\ & A_{2,2} & & C_2 \\ & & \cdots & \cdot \\ & & A_{Nb,Nb} & C_{Nb} \end{pmatrix},$$

where the blocks $A_{lb,lb}$ are **rectangular** $m_{lb} \times n_{lb}$ matrices with $m_{lb} \geq n_{lb}$ and $\sum_{lb=1}^{Nb} m_l = n$, and the border blocks C_{lb} are of order $m_{lb} \times n_I$ ($n_I \ll n_{lb}$), where $n_I = \sum_{bl=1}^{Nb} (m_{lb} - n_{lb})$. The linear system becomes

$$\begin{pmatrix} A_{1,1} & & & C_1 \\ & A_{2,2} & & C_2 \\ & & \cdots & \cdot \\ & & A_{Nb,Nb} & C_{Nb} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{Nb} \\ x_I \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{Nb} \end{pmatrix}, \tag{8.4}$$

where x_{lb} is of length n_{lb}, x_I is a vector of length n_I of interface variables, and the right-hand side vectors b_{lb} are of length m_{lb}, such that

$$\begin{pmatrix} A_{lb,lb} & C_{lb} \end{pmatrix} \begin{pmatrix} x_{lb} \\ x_I \end{pmatrix} = b_{lb}, \quad 1 \leq lb \leq Nb.$$

A partial factorization of each block matrix is performed, that is,

$$\begin{pmatrix} A_{lb,lb} & C_{lb} \end{pmatrix} = P_{lb} \begin{pmatrix} L_{lb} & \\ \bar{L}_{lb} & I \end{pmatrix} \begin{pmatrix} U_{lb} & \bar{U}_{lb} \\ & S_{lb} \end{pmatrix} Q_{lb}, \tag{8.5}$$

where P_{lb} and Q_{lb} are permutation matrices, L_{lb} and U_{lb} are $n_{lb} \times n_{lb}$ lower and upper triangular matrices, respectively, and if q_{lb} is the number of columns in C_{lb} with at least one entry, S_{lb} is a $(m_{lb} - n_{lb}) \times q_{lb}$ local Schur complement matrix. Pivots can only be chosen from the columns of $A_{lb,lb}$ because the columns of C_{lb} have entries in at least one other border block C_{jb} ($jb \neq lb$). The pivot candidate $(A_{lb,lb})_{11}$ at the first elimination step must satisfy

$$\max_{i>1} |(A_{lb,lb})_{i1}| \leq \gamma^{-1} |(A_{lb,lb})_{11}|,$$

and provided A is nonsingular, there will always be a numerically satisfactory pivot in column 1 of $A_{lb,lb}$. The same is true at each elimination step so that n_{lb} pivots can be chosen. An $n_I \times n_I$ matrix S is obtained by assembling the Nb local

ALGORITHM 8.8 Coarse-grained parallel LU factorization and solve using SBBD form
Input: Linear system in SBBD form (8.4).
Output: Block LU factorization and computed solution x.

1: $S = 0$ and $z_I = 0$
2: **for** $lb = 1 : Nb$ **do**
3: Perform a partial LU factorization (8.5) of $(A_{lb,lb},\ C_{lb})$.
4: Solve $P_{lb} \begin{pmatrix} L_{lb} \\ \bar{L}_{lb} & I \end{pmatrix} \begin{pmatrix} y_{lb} \\ \bar{y}_{lb} \end{pmatrix} = b_{lb}$
5: $S = S + S_{lb}$ and $z_I = z_I + \bar{y}_{lb}$ ▷ Assemble S and z_I
6: **end for**
7: $S = P_s L_s U_s Q_s$ ▷ P_s and Q_s are permutation matrices
8: Solve $P_s L_s\, y_I = z_I$ and then $U_s Q_s\, x_I = y_I$ ▷ Forward then back substitution
9: **for** $lb = 1 : Nb$ **do**
10: Solve $U_{lb}\, Q_{lb}\, x_{lb} = y_{lb} - \bar{U}_{lb}\, Q_{lb}\, x_I$
11: **end for**

Schur complement matrices S_{lb}. The approach is summarized as Algorithm 8.8. The operations on the submatrices can be performed in parallel.

8.5.3 *Ordering to Singly Bordered Form*

The objective is to permute A to an SBBD form with a narrow column border. One way to do this is to choose the number $Nb > 1$ of required blocks and use nested dissection to compute a vertex separator $\mathcal{V}_S$ of $\mathcal{G}(A + A^T)$ such that removing $\mathcal{V}_S$ and its incident edges splits $\mathcal{G}(A + A^T)$ into Nb components. Then initialize the set $\mathcal{S}_C$ of border columns to $\mathcal{V}_S$ and let $\mathcal{V}_{1b}, \mathcal{V}_{2b}, \ldots, \mathcal{V}_{Nb}$ be the subsets of column indices of A that correspond to the Nb components and let $n_{i,kb}$ be the number of column indices in row i that belong to $\mathcal{V}_{kb}$. If $lb = \arg\max_{1 \le kb \le Nb} |n_{i,kb}|$, then row i is assigned to partition lb. All column indices in row i that do not belong to $\mathcal{V}_{lb}$ are moved into $\mathcal{S}_C$. Once all the rows have been considered, the only rows that remain unassigned are those that have all their nonzero entries in $\mathcal{V}_S$. Such rows can be assigned equally to the Nb partitions. If $j \in \mathcal{S}_C$ is such that column j of A has nonzero entries only in rows belonging to partition kb, then j can be removed from $\mathcal{S}_C$ and added to $\mathcal{V}_{kb}$. The procedure is outlined as Algorithm 8.9. The computed vector $block$ and set $\mathcal{S}_C$ can be used to define permutation matrices P and Q such that $PAQ = A_{SB}$. In practice, it may be necessary to modify the algorithm to ensure a good row balance between the number of rows in the blocks; this may lead

ALGORITHM 8.9 SBBD ordering of a general matrix

Input: Matrix A, the number $Nb > 1$ of blocks and corresponding vertex separator $\mathcal{V}_S$ of $\mathcal{G}(A + A^T)$.

Output: Vector $block$ such that $block(i)$ denotes the partition in the SBBD form to which row i is assigned ($1 \leq i \leq n$) and $\mathcal{S}_C$ is the set of border columns.

Initialise $\mathcal{S}_C = \mathcal{V}_S$ and $block(1:n) = 0$
Initialise $\mathcal{V}_{kb}$ to hold the column indices of A that correspond to component kb of $\mathcal{G}(A + A^T)$ after the removal of $\mathcal{V}_S$, $1 \leq kb \leq Nb$
for each row i **do**
    Add up the number $n_{i,kb}$ of column indices belonging to $\mathcal{V}_{kb}$, $1 \leq kb \leq Nb$
    Find $lb = \arg\max_{1 \leq kb \leq Nb} n_{i,kb}$
    $block(i) = lb$
    for each column index j in row i **do**
        if $j \in \mathcal{V}_{kb}$ and $kb \neq lb$ **then**
            Remove j from $\mathcal{V}_{kb}$ and add to $\mathcal{S}_C$
        end if
    end for
end for
Assign the rows i for which $block(i) = 0$ equally between the Nb partitions.
If some column $j \in \mathcal{S}_C$ has nonzero entries only in rows belonging to partition kb then remove j from $\mathcal{S}_C$ and add to $\mathcal{V}_{kb}$

to a larger $\mathcal{S}_C$. It is also necessary to avoid adding in duplicate column indices into $\mathcal{S}_C$ (alternatively, a final step can be added that removes duplicates).

The matching-based orderings discussed in Section 6.3 that permute off-diagonal entries onto the diagonal can increase the symmetry index of the resulting reordered matrix, particularly in cases where A is very sparse with a large number of zeros on the diagonal. Frequently, applying a matching ordering before ordering to SBBD form reduces the number of columns in $\mathcal{S}_C$.

8.6 Notes and References

The most influential early paper on orderings for sparse symmetric matrices is that of Tinney & Walker (1967). It first proposed the minimum degree algorithm (referred to as scheme 2) and the minimum fill-in algorithm (referred to as scheme 3). The fast implementation of the minimum degree algorithm using quotient graphs is summarized by George & Liu (1980a). Further developments were made throughout the 1980s, including the multiple minimum degree variant, mass

elimination and external degree; key references are Liu (1985) and George & Liu (1989). An important development in the 1990s was the approximate minimum degree algorithm of Amestoy et al. (1996). Modifying the AMD algorithm for matrices with some dense rows is discussed in Dollar & Scott (2010). For a careful description of different variants of the minimum degree strategy and their complexity we recommend Heggernes et al. (2001). Rothberg & Eisenstat (1998) consider both minimum degree and minimum fill strategies and (Erisman et al., 1987) provide an early evaluation of different strategies for nonsymmetric matrices.

Jennings (1966) presents the first envelope method for sparse Cholesky factorizations. The Cuthill-McKee algorithm comes from the paper by Cuthill & McKee (1969). The GPS algorithm was originally introduced in Gibbs et al. (1976). The book by George & Liu (1981) gives a detailed description of the algorithm while Meurant (1999) includes an enlightening discussion of the relation between the CM and RCM algorithms. A quick search of the literature shows that a large number of bandwidth and profile reduction algorithms have been (and continue to be) reported. Many have their origins in the Cuthill-McKee and GPS algorithms. A widely used two-stage variant that employs level sets is the so-called Sloan algorithm (Sloan, 1986); see also Reid & Scott (1999) for details of an efficient implementation. The use of the Fiedler vector to obtain spectral orderings is introduced in Barnard et al. (1995), with analysis given in George & Pothen (1997). A hybrid algorithm that combines the spectral method with the second stage of Sloan's algorithm to further reduce the profile is proposed in Kumfert & Pothen (1997) and a multilevel variant is given by Hu & Scott (2001). de Oliveira et al. (2018) provide a recent comparison of many bandwidth and profile reduction algorithms.

Reducing the bandwidth when A is nonsymmetric is discussed by Reid & Scott (2006). For highly nonsymmetric A, Scott (1999) applies a modified Sloan algorithm applied to the row graph (that is, $\mathcal{G}(AA^T)$) to derive an effective ordering of the rows of A for use with a frontal solver. The approach originally proposed by Markowitz (1957) for finding pivots during an LU factorization is incorporated (in modified form) in a number of serial LU factorization codes, including the early solvers `MA28` and Y12M (Duff, 1980 and Zlatev, 1991, respectively) as well as `MA48` (Duff & Reid, 1996). The book of Duff et al. (2017) includes detailed discussions. To limit permutations to being symmetric, Amestoy et al. (2007) propose minimizing the Markowitz count among the diagonal entries.

A seminal paper on global orderings is George (1973), but a real revolution in the field followed the theoretical analysis of the application of nested dissection for general symmetrically structured sparse matrices given in Lipton et al. (1979). For subsequent extensions discussing separator sizes we suggest Agrawal et al. (1993), Teng (1997), and Spielman & Teng (2007).

From the early 1990s onwards, there have been numerous contributions to graph partitioning algorithms. Significant developments, including multilevel algorithms, have been driven in part by the design and development of mathematical software, notably the well-established packages METIS (2022) and Scotch (2022); both offer versions for sequential and parallel graph partitioning (see also the papers by Karypis & Kumar, 1998a,b and Chevalier & Pellegrini, 2008). The book by

Bichot & Siarry (2013) discusses a number of contributions, including hypergraph partitioning, which is well suited to parallel computational models (see, for example, Uçar & Aykanat, 2007 and references to the use of hypergraphs given in the survey article of Davis et al., 2016; they can also be used for profile reduction Acer et al., 2019).

Hu et al. (2000) present a serial algorithm for ordering nonsymmetric A to SBBD form; an implementation is available as `HSL_MC66` within the HSL mathematical software library. Algorithm 8.9 is from Hu & Scott (2005) (see also Duff & Scott, 2005). Alternatively, hypergraphs can be used for SBBD orderings. The best-known packages are the serial code PaToH of Aykanat et al. (2004) and the parallel code PHG from Zoltan (2022).

Chapter 9
Algebraic Preconditioners and Approximate Factorizations

In conjunction with iterative methods, preconditioning is often the vital component in enabling the solution of such (linear) systems when the dimension is large. – Wathen (2015)

Preconditioning involves exploiting ideas from sparse direct solvers. Gradually, iterative methods started to approach the quality of direct solvers. In earlier times, iterative methods were often special purpose in nature... Now iterative methods are almost mandatory. – Saad (1996b).

When a matrix factorization is performed using finite precision arithmetic, the computed factors are not the exact factors. Despite this, the objective of sparse direct methods is normally to compute solutions that are accurate within the precision used. As discussed in Chapter 7, theoretical results can be used to assess both stability and accuracy.

The effort to obtain results that are as accurate as possible can lead to complex coding and unavoidable inefficiencies that can be magnified by modern computer architectures. Furthermore, in some situations, more accuracy than is needed (or is justified by the input data) is sought by a direct method. These issues can potentially be addressed by intentionally relaxing the required accuracy of the computed factors. In Section 7.3.3, we discussed static pivoting that allows pivots to be explicitly perturbed during a matrix factorization to enable them to be selected, thereby reducing the computational costs of the factorization (in terms of time and memory). The penalty is that the factorization may be less stable and a refinement process (such as described in Algorithm 7.3) may be needed to improve the accuracy of the computed solution. However, even with sophisticated theoretical and algorithmic tools, factorizations that use such strategies can still be prohibitively expensive and may not be fully robust. An alternative approach is to compute a simpler and cheaper and sparser approximate factorization of A (or of A^{-1}) and to use this as a preconditioner in combination with an iterative solver to derive a suitable solution of the linear system. The main obstacle is that the choice of an efficient preconditioner is highly problem dependent: what works well for problems

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_9

from one application may not help for those of a different origin. Our focus is on algebraic preconditioners that are often successfully used in the solution of linear systems arising from a range of diverse applications.

Algebraic preconditioners do not require knowledge of the provenance of the linear system, and their construction relies solely on the matrix A (which may only be available implicitly, that is, the action of A on vectors is known, but A itself is not supplied). They are general methods that are particularly important when little is known about the underlying problem and they are widely applicable because they are designed with few restrictions. However, if more information is known, it can be more effective to use a specialized preconditioner that is designed for the specific application. This division between approaches to preconditioning essentially amounts to whether we are "given a problem" or "given a matrix": algebraic preconditioning is primarily concerned with the latter.

In the following, we refer to an approximate factorization as an **incomplete factorization** to distinguish it from a **complete factorization** of a direct method.

9.1 Introduction to Iterative Solvers

The two main classes of iterative methods for solving $Ax = b$ are **stationary** iterative methods (also sometimes called **relaxation** or **simple** methods) and **Krylov subspace** methods. We briefly introduce each class.

9.1.1 Stationary Iterative Methods

Stationary iterative methods work by splitting A as follows:

$$A = M - N,$$

where the matrix M is chosen to be nonsingular and easy to invert. Starting with an initial guess $x^{(0)}$, the iterations are then given by

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b, \quad k = 0, 1, \ldots \tag{9.1}$$

This can be rewritten as

$$x^{(k+1)} = x^{(k)} + M^{-1}(b - Ax^{(k)}) = x^{(k)} + M^{-1}\,r^{(k)}, \quad k = 0, 1, \ldots, \tag{9.2}$$

where the vector $r^{(k)} = b - Ax^{(k)}$ is the **residual** on the k-th iteration. Observe that by substituting $b = r^{(k)} + Ax^{(k)}$ into $x = A^{-1}\,b$, we obtain

$$x = A^{-1}(r^{(k)} + Ax^{(k)}) = x^{(k)} + A^{-1}\,r^{(k)},$$

and if M is used to approximate A, we again get the iteration (9.2). From (9.2),

$$r^{(k+1)} = b - A(x^{(k)} + M^{-1} r^{(k)}) = (I - AM^{-1}) r^{(k)} = \ldots = (I - AM^{-1})^{k+1} r^{(0)}, \tag{9.3}$$

and if $e^{(k)} = x - x^{(k)}$ is the error vector on iteration k, then

$$e^{(k+1)} = M^{-1} N e^{(k)} = \ldots = (M^{-1}N)^{k+1} e^{(0)} = (I - M^{-1}A)^{k+1} e^{(0)}. \tag{9.4}$$

The matrix $I - M^{-1}A$ or $I - AM^{-1}$ is called the **iteration matrix**. In general, (9.3) is evaluated rather than (9.4) because $e^{(0)}$ is unknown and (9.3) computes the residuals that are often used to monitor convergence.

Theorem 9.1 (Saad 2003b; Greenbaum 1997) *For any initial $x^{(0)}$ and vector b, the iteration (9.1) converges if and only if the spectral radius of the iteration matrix $(I - M^{-1}A)$ is less than unity.*

Proof The **spectral radius** of an $n \times n$ matrix C with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ is defined to be

$$\rho(C) = \max\{|\lambda_i| \,|\, 1 \leq i \leq n\}. \tag{9.5}$$

Furthermore, the sequence of matrix powers C^k, $k = 0, 1, \ldots$, converges to zero if and only if $\rho(C) < 1$. It follows from (9.4) that if the spectral radius of $(I - M^{-1}A)$ is less than unity, then the iteration (9.1) converges for any $x^{(0)}$ and b. Conversely, the relation

$$x^{(k+1)} - x^{(k)} = (I - M^{-1}N)(x^{(k)} - x^{(k-1)}) = \ldots = (I - M^{-1}N)^k M^{-1}(b - Ax^{(0)})$$

shows that if the iteration converges for any $x^{(0)}$ and b, then $(I - M^{-1}N)^k v$ converges to zero for any v. Consequently, $\rho(I - M^{-1}A)$ must be less than unity, and the result follows. □

It is generally impractical to compute the spectral radius and sufficient conditions that guarantee convergence are used. Because $\rho(C) \leq \|C\|$ for any matrix norm, a sufficient condition is $\|I - M^{-1}A\| < 1$. A small spectral radius leads to rapid convergence, and the closer the eigenvalues of $M^{-1}A$ are to unity, the faster the convergence. However, the eigenvalue distribution (not just the spectral radius) is important in evaluating the rate of convergence.

Several standard stationary methods are obtained from the splitting

$$A = D_A + L_A + U_A, \tag{9.6}$$

where D_A is a diagonal matrix that represents the diagonal part of A, and L_A and U_A are the strictly lower and upper triangular parts of A, respectively. If $\omega > 0$ is a scalar parameter, classical methods include:

- Richardson method: $M = \omega^{-1} I$
- Jacobi and damped Jacobi methods: $M = D_A$ and $M = \omega^{-1} D_A$
- Gauss–Seidel and SOR methods: $M = D_A + L_A$ and $M = \omega^{-1} D_A + L_A$

9.1.2 Krylov Subspace Methods

Non-stationary iterative methods are of the form

$$x^{(k+1)} = x^{(k)} + \omega^{(k)} M^{-1} r^{(k)}, \quad k = 0, 1, \ldots,$$

where the $\omega^{(k)}$ are scalars. In this class of methods, Krylov subspace methods are the most effective. Given a vector y, the k-th Krylov subspace $\mathcal{K}^{(k)}(A, y)$ generated by A from the vector y is defined to be

$$\mathcal{K}^{(k)}(A, y) = span(y, Ay, \ldots, A^{k-1} y).$$

The idea behind Krylov subspace methods is to generate a sequence of approximate solutions $x^{(k)} \in x^{(0)} + \mathcal{K}^{(k)}(A, r^{(0)})$ such that the norm of the corresponding residuals $r^{(k)} \in \mathcal{K}^{(k+1)}(A, r^{(0)})$ converges to zero. For symmetric positive definite (SPD) systems, the Krylov subspace method of choice is the conjugate gradient (CG) method. For nonsymmetric systems, there are a number of popular methods, including the generalized minimal residual (GMRES) method and the biconjugate gradient (BiCG) method, but there is no single method of choice. The key feature they have in common is that at each iteration only matrix-vector products with A (and possibly with A^T in the nonsymmetric case) are required.

Krylov subspace methods are powerful and nowadays, when combined with a preconditioner, comprise the most widely used class of preconditioned iterative methods. Because they build a basis, in exact arithmetic, convergence is achieved in at most n iterations (but in the presence of rounding errors, this is not guaranteed). If n is large, it is impractical to perform $O(n)$ iterations; the hope is that the process returns a sufficiently accurate solution far earlier. Unfortunately, for a given A, right-hand side vector b, and initial guess $x^{(0)}$, it is usually not possible to predict the rate of convergence. If A is an SPD matrix, then it can be shown that the approximate solution $x^{(k)}$ at iteration k computed using the CG method satisfies

$$\|x - x^{(k)}\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x - x^{(0)}\|_A,$$

where $\| \cdot \|_A$ is the A-norm, and $\kappa(A)$ is the spectral condition number given by (7.15). Clearly, there is good (fast) convergence when $\kappa(A)$ is small, but poor (slow) convergence usually occurs if $\kappa(A) \gg 1$. But this error bound can be highly pessimistic. It does not show the potential for CG to converge superlinearly or

that the rate of convergence depends on the distribution of all the eigenvalues of A. In practice, it is not normally possible to obtain detailed spectral information. Thus, even for CG, preconditioning is often based on experimentation. For non-SPD matrices, less is known and methods that guarantee the monotonic reduction of a relevant quantity at each iteration are sometimes favoured. For example, if the minimal residual (MINRES) method is used for solving symmetric indefinite systems, then in exact arithmetic, the norm of the residual is monotonically decreasing. However, no general descriptive convergence theory is available for Krylov subspace methods for nonsymmetric systems (including GMRES). This is a significant problem because, without theory to guide us, preconditioning must be heuristic.

9.2 Introduction to Algebraic Preconditioners

Preconditioning corresponds to the application of a matrix (or a linear operator) to the original linear system to yield a different linear system that has more favourable properties. Consider the preconditioned linear system

$$M^{-1}Ax = M^{-1}b. \tag{9.7}$$

Here M^{-1} is applied to A from the left. We say that A is preconditioned from the left and M is a left preconditioner. Analogously, the linear system can be preconditioned from the right

$$AM^{-1}y = b, \qquad x = M^{-1}y. \tag{9.8}$$

The following result states that it is not possible to determine a priori which variant is the best.

Theorem 9.2 (Mendelsohn 1956) *Let δ and Δ be positive numbers. Then, for any $n \geq 3$, there exist nonsingular $n \times n$ matrices A and M such that all the entries of $M^{-1}A - I$ have absolute value less than δ and all the entries of $AM^{-1} - I$ have absolute values greater than Δ.*

Nevertheless, the choice between left and right preconditioning is still important and may be based on the properties of the coupling of the preconditioner with the iterative method or on the distribution of the eigenvalues of A. The computed quantities that are readily available during a preconditioned iterative method depend on how the preconditioner is applied and this may influence the choice. These quantities may be used, for example, to decide when to terminate the iterations. An obvious advantage of right preconditioning is that in exact arithmetic, the residuals for the right preconditioned system are identical to the true residuals, enabling convergence to be monitored accurately. In some cases, the numerical properties of an implementation and/or the computer architecture may also play a part.

For M in factorized form $M = M_1 M_2$, **two-sided** (or **split**) preconditioning is an option. The iterative method then solves the transformed system

$$M_1^{-1} A M_2^{-1}\, y = M_1^{-1}\, b, \qquad x = M_2^{-1}\, y. \tag{9.9}$$

If A and M are SPD matrices, then $M_2 = M_1^T$ and we would like the preconditioned matrix $M_1^{-1} A M_1^{-T}$ to be SPD. However, it is not necessary to use a two-sided transformation with the preconditioned conjugate gradient (PCG) method because it can be formulated using the M-inner product in which the matrix $M^{-1}A$ is self-adjoint.

Theorem 9.3 (Saad 2003b; van der Vorst 2003) *Let A and M be SPD matrices. Then $M^{-1}A$ is self-adjoint in the M-inner product.*

Proof Self-adjointness is implied by the following chain of equivalences.

$$\begin{aligned}\langle M^{-1}Ax, y\rangle_M &= \langle Ax, y\rangle = \langle x, Ay\rangle = \langle x, MM^{-1}Ay\rangle \\ &= \langle Mx, M^{-1}Ay\rangle = \langle x, M^{-1}Ay\rangle_M.\end{aligned}$$

□

Left preconditioned CG with the M-inner product is mathematically equivalent to right preconditioned CG with the M^{-1}-inner product. If A is symmetric but not positive definite, the PCG method can formally be written down, but the necessary conditions for convergence may not be satisfied and the method may break down (division by a zero quantity).

9.2.1 Desirable Preconditioner Properties

An obvious objective is for the preconditioner to lead to rapid convergence. As already noted, if the matrix A is SPD, then the convergence rate of the CG method depends on the distribution of its eigenvalues. The preconditioner should aim to reduce the condition number, but this is not necessarily sufficient to give fast convergence. For general matrices, despite the lack of theoretical guarantees regarding convergence, many useful preconditioners have nevertheless been motivated by bounding the condition number of the preconditioned matrix.

Choosing a preconditioner is often based on how costly it is to compute and on some indicators that potentially reflect its quality. In particular, the **accuracy** of a preconditioner M can be assessed using the norm of the error matrix

$$\|E\| = \|M - A\|,$$

and its **stability** can be measured using

$$\|M^{-1}E\| = \|I - M^{-1}A\| \quad \text{or} \quad \|EM^{-1}\| = \|I - AM^{-1}\|.$$

If a preconditioner is used to solve a large number of systems over which the cost of constructing it can be amortized, then the expense of constructing M in terms of time may not be the driving factor. However, as the preconditioner must be applied at each iteration of the solver, unless very few iterations are performed, it is essential that each application is inexpensive. Each application $M^{-1}w$ involves solving a linear system $Mv = w$. If M is in factorized form and the factors are (block) triangular, this is straightforward but because they are inherently serial and hard to parallelize, repeated substitutions can be a critical computational bottleneck. In some cases, rather than M, the inverse M^{-1} is computed directly. In this case, we have an **approximate inverse preconditioner**. Applying such a preconditioner involves only matrix-vector multiplications, which are normally easier to parallelize. However, because the inverse of an irreducible matrix is dense (Theorem 7.3), it is important that M^{-1} is constructed to be sparse. Such preconditioners are discussed in Chapter 11.

9.2.2 Simple Algebraic Preconditioners

The simplest preconditioner consists of the diagonal of the matrix $M = D_A$. This is known as the (point) Jacobi preconditioner. Block versions can be derived by partitioning $\mathcal{V} = \{1, 2, \ldots, n\}$ into mutually disjoint subsets $\mathcal{V}_1, \ldots, \mathcal{V}_l$ and then setting

$$m_{ij} = \begin{cases} a_{ij} & \text{if } i \text{ and } j \text{ belong to the same subset } \mathcal{V}_k \text{ for some } k, 1 \leq k \leq l, \\ 0 & \text{otherwise.} \end{cases}$$

Often, natural choices for the partitioning suggest themselves. For example, supervariables can be used or the partitioning may be chosen to coincide with the division of variables over the processors in a parallel environment. Jacobi preconditioners need very little storage and are easy to implement.

The SSOR preconditioner, like the Jacobi preconditioner, can be derived from A without any work. If A is symmetric, then using the notation (9.6), the SSOR preconditioner is defined to be

$$M = (D_A + L_A)D_A^{-1}(D_A + L_A)^T, \tag{9.10}$$

or, using a parameter $0 < \omega < 2$, as

$$M = \frac{1}{2-\omega}(\frac{1}{\omega}D_A + L_A)(\frac{1}{\omega}D_A)^{-1}(\frac{1}{\omega}D_A + L_A)^T.$$

The optimal value of ω will reduce the number of iterations needed for convergence of the iterative solver, but it is usually prohibitively expensive to compute the spectral information needed to calculate it. Again, block variants are possible.

9.2.3 The Eisenstat Trick

Within a preconditioned iterative solver, it is generally cheaper to apply M^{-1} and A separately, rather than explicitly forming and storing the preconditioned matrix. However, in special cases, it is possible to improve efficiency by combining the action of the preconditioner with the matrix-vector multiplication. One such approach is called the **Eisenstat trick**. Consider the matrix splitting (9.6), and let M be given by

$$M = (D + L_A)\,[D^{-1}(D + U_A)] = M_1\,M_2, \tag{9.11}$$

where D is a nonsingular diagonal matrix. The SSOR matrix (9.10) is one example in the symmetric case but more generally $D \neq D_A$. Using two-sided preconditioning, (9.9) becomes

$$A'y = M_1^{-1}AM_2^{-1}\,y = (D + L_A)^{-1}A[D^{-1}(D + U_A)]^{-1}\,y = (D + L_A)^{-1}b. \tag{9.12}$$

Setting

$$\bar{L} = D^{-1}L_A, \quad \bar{U} = D^{-1}U_A, \quad \bar{A} = D^{-1}A, \quad \text{and} \quad \bar{b} = (I + \bar{L})^{-1}D^{-1}\,b,$$

and using (9.6), we obtain

$$A' = (D + L_A)^{-1}A[D^{-1}(D + U_A)]^{-1} = [(D + L_A)^{-1}D]D^{-1}A[D^{-1}(D + U_A)]^{-1}$$

$$= [D^{-1}(D + L_A)]^{-1}D^{-1}A(I + D^{-1}U_A)^{-1} = (I + \bar{L})^{-1}\bar{A}(I + \bar{U})^{-1}.$$

That is, the system in (9.12) becomes

$$A'y = (I + \bar{L})^{-1}\bar{A}(I + \bar{U})^{-1}\,y = (I + \bar{L})^{-1}D^{-1}\,b = \bar{b}. \tag{9.13}$$

If y solves (9.13), then the solution x of $(I + \bar{U})\,x = y$ solves $Ax = b$. But the expression for A' can be further transformed as

$$A' = (I + \bar{L})^{-1}\,(I + \bar{L} + D^{-1}D_A - 2I + I + \bar{U})(I + \bar{U})^{-1}$$

$$= (I + \bar{L})^{-1}\,[(I + \bar{L})(I + \bar{U})^{-1} + (D^{-1}D_A - 2I)(I + \bar{U})^{-1} + I]$$

$$= (I + \bar{U})^{-1} + (I + \bar{L})^{-1}\,[(D^{-1}D_A - 2I)(I + \bar{U})^{-1} + I].$$

Thus, to compute $z = A'w = (I + \bar{L})^{-1}\bar{A}(I + \bar{U})^{-1}w$ for a given w, it is necessary only to solve two triangular systems

$$(I + \bar{U})\, z_1 = w \quad \text{followed by} \quad (I + \bar{L})\, z_2 = (D^{-1}D_A - 2I)\, z_1 + w$$

and then set $z = z_1 + z_2$. Note that this trick is not a preconditioner: it is simply a way of applying the preconditioner (9.11).

9.3 Some Special Classes of Matrices

The development of algebraic preconditioners has historically been closely connected to their earliest application, which was solving linear systems arising from the discretization of partial differential equations. Consider a two-dimensional Poisson problem discretized on a given domain by a uniform regular grid using finite differences, with zero Dirichlet conditions on the boundary. The resulting matrix for a 3×3 rectangular grid using the natural ordering of the vertices is given by

$$A = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}. \tag{9.14}$$

If the spatial discretization on the domain is characterized by the mesh parameter h, then the size of A is inversely proportional to h. Expressing some matrix-related quantities asymptotically as functions of h can be useful if the discretized domain is bounded. For example, the condition number of the matrix (9.14) depends asymptotically on h^{-2}. Matrices with similar banded sparsity patterns with nonzeros on only a small number of subdiagonals arise from simple finite difference or finite element discretizations of other partial differential equations. They can be considered as particular cases of more general special classes of matrices whose properties can be derived using the theoretical background behind the discretizations.

M-matrices is one such class. Let the off-diagonal entries of the nonsingular matrix A be nonpositive (that is, $a_{ij} \leq 0$ for all $i \neq j$). Then A is a (nonsingular) **M-matrix** if one of the following holds:

- $A + D$ is nonsingular for any diagonal matrix D with nonnegative entries.
- All the entries of A^{-1} are nonnegative.
- All principal minors of A are positive.

The matrix (9.14) is an example of an M-matrix. A symmetric M-matrix is known as a Stieltjes matrix, and such a matrix is positive definite.

The class of **nonsingular H-matrices** includes matrices coming from simple discretizations of convection–diffusion problems. The **comparison** matrix $C(A)$ of A is defined to have entries

$$C(A)_{ij} = \begin{cases} -|a_{ij}|, & i \neq j, \\ |a_{ij}|, & i = j. \end{cases}$$

If $C(A)$ is a nonsingular M-matrix, then A is a nonsingular H-matrix.

We also recall diagonally dominant matrices. A is **diagonally dominant by rows** if

$$\sum_{j=1,\ j\neq i}^{n} |a_{ij}| \leq |a_{ii}|, \quad 1 \leq i \leq n. \tag{9.15}$$

A is **strictly diagonally dominant by rows** if strict inequality holds in (9.15) for all i. A is (strictly) diagonally dominant by columns if A^T is (strictly) diagonally dominant by rows. A is said to be **irreducibly diagonally dominant** if it is irreducible and (9.15) is satisfied with strict inequality for at least one row i. If A is strictly diagonally dominant by rows or columns or is irreducibly diagonally dominant, then it is nonsingular and factorizable. The class of diagonally dominant matrices is closely connected to that of nonsingular H-matrices. For example, the property that there exists a diagonal matrix D with positive entries such that AD is strictly diagonally dominant is equivalent to A being a nonsingular H-matrix.

9.4 Introduction to Incomplete Factorizations

Preconditioners based on an incomplete factorization of A in which entries are dropped during the factorization are widely used in computational science and engineering, especially when the underlying physics of a problem is difficult to exploit. Besides being used as standalone preconditioners, incomplete factorizations are important within more sophisticated methods. For example, they can be used to precondition subdomain solves in domain decomposition schemes or as a smoother in multigrid methods. Incomplete factorizations fall into three main classes:

(i) Threshold-based methods in which the locations of permissible fill-in are determined in conjunction with the numerical factorization of A; entries of the computed factors of absolute value less than a prescribed threshold $\tau > 0$ are dropped. Success relies on determining a suitable τ. This is highly problem dependent and is influenced by the scaling of A.

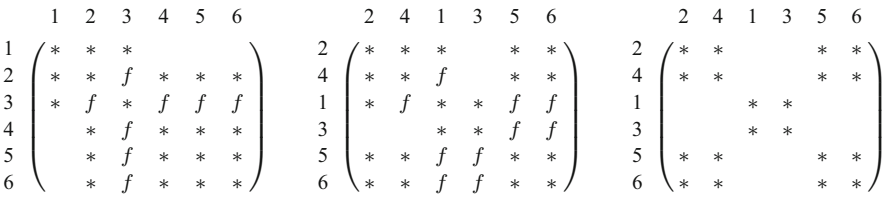


Figure 9.1 Illustration of matrix sparsification. f denotes filled entries in the factors. On the left is the original matrix A with its filled entries, in the centre is the permuted matrix with its filled entries, and on the right is the sparsified permuted matrix after dropping the entries of A in positions $(1, 3)$ and $(3, 1)$ (it has no filled entries).

(ii) Memory-based methods in which the amount of memory available for the incomplete factorization is prescribed and only the largest entries in each row (or column) are retained.

(iii) Structure-based methods in which an initial symbolic factorization phase determines the location of permissible entries using $\mathcal{S}\{A\}$. This allows the memory requirements to be determined before an incomplete numerical factorization is performed. The specified set of positions is called the **target sparsity pattern**. A widely used example allows the incomplete factors to have entries only in the positions corresponding to $\mathcal{S}\{A\}$.

The basic dropping approaches can be combined and they can be employed in conjunction with discarding entries in A before the factorization commences. This initial sparsification is appealing because it may be possible to obtain an incomplete factorization by computing a complete factorization of the sparsified matrix. Sparsification can be performed by value or by position. Figure 9.1 illustrates sparsification of A after permuting it reveals a block structure (the permutation can be found using, for example, Algorithm 3.7 or 3.8).

9.4.1 Incomplete Factorization Breakdown

Dropping entries can lead to **breakdown** of the incomplete factorization, that is, a zero pivot may be encountered during the factorization (or a non-positive pivot in the Cholesky case). It is only possible to predict when this will happen in special cases, as stated in the following theorem, which is a consequence of the fact that being an M-matrix or an H-matrix is preserved in the sequence of the Schur complements during the factorization. This result does not hold for general SPD matrices.

Theorem 9.4 (Meijerink & van der Vorst 1977; Manteuffel 1980; Varga et al. 1980) *Let A be a nonsingular M-matrix or H-matrix. If the target sparsity pattern of the incomplete factors contains the positions of the diagonal entries, then the incomplete factorization of A does not break down.*

To illustrate the error accumulation in the incomplete factorization of an M-matrix using dropping, consider the example given in (9.14). Let E be the error matrix. E is initialized to zero, and at each stage of the factorization, the dropped entries are added into it. After one step of the complete factorization of A, the partially eliminated matrix $A^{(2)}$ is

$$A^{(2)} = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ & 3.75 & -1 & -0.25 & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ & -0.25 & & 3.75 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}.$$

Suppose the filled entries -0.25 in positions $(2, 4)$ and $(4, 2)$ are dropped. Then the values of the corresponding diagonal entries in the subsequent elimination matrices are larger than they would have been without any dropping. Furthermore, as all the off-diagonal nonzero entries are negative, for any target sparsity pattern the dropped entries are negative. The M-matrix property applies to all subsequent Schur complements, which implies that all the entries added into E are negative and so the absolute values of the entries in E grow as the factorization proceeds (the contributions can never cancel each other out). Thus, although the factorization does not break down, the growth in the error is potentially a problem for the accuracy of an incomplete factorization of an M-matrix.

9.4.2 Perturbing Entries to Prevent Breakdown

Modifying the diagonal entries of A is a common approach to avoid breakdown in an incomplete factorization. Breakdown is illustrated in Figure 9.2. A simple a posteriori remedy is to perturb the diagonal value that has caused breakdown. In this example, increasing a_{44} so that $\tilde{d}_{44}$ has a (small) positive value. Unfortunately, practical experience of making simple ad hoc modifications is generally not very positive. This is because making a local perturbation when breakdown occurs (or is close to occurring) may be too late for the resulting factorization to be good enough to be useful as a preconditioner (growth may already have happened in some of the factor entries). This applies to standard incomplete factorizations and to approximate inverses.

An alternative and more effective strategy to avoid breakdown is to modify all the diagonal entries of A a priori and then compute an incomplete factorization of $A + \alpha I$, where the shift $\alpha > 0$ is a scalar parameter. It is always possible to find α such that $A + \alpha I$ is nonsingular and diagonally dominant and is thus an H-matrix. However, being an H-matrix is not a necessary condition for a

$$A = \begin{pmatrix} 3 & -2 & & 2 \\ -2 & 3 & -2 & \\ & -2 & 3 & -2 \\ 2 & & -2 & 8 \end{pmatrix}, \; L = \begin{pmatrix} 1 & & & \\ -2/3 & 1 & & \\ & -6/5 & 1 & \\ 2/3 & 4/5 & -2/3 & 1 \end{pmatrix}, \; D = \begin{pmatrix} 3 & & & \\ & 5/3 & & \\ & & 3/5 & \\ & & & 16/3 \end{pmatrix}.$$

$$\widetilde{L} = \begin{pmatrix} 1 & & & \\ -2/3 & 1 & & \\ & -6/5 & 1 & \\ 2/3 & & -10/3 & 1 \end{pmatrix}, \; \widetilde{D} = \begin{pmatrix} 3 & & & \\ & 5/3 & & \\ & & 3/5 & \\ & & & 0 \end{pmatrix}.$$

Figure 9.2 An example to illustrate breakdown. The matrix A and its square root-free factors are given together with the incomplete factors $\widetilde{L}$ and $\widetilde{D}$ that result from dropping the entry l_{24} during the factorization. $\tilde{d}_{44} = 0$ means the incomplete factorization has broken down.

ALGORITHM 9.1 Trial-and-error global shifted incomplete factorization

Input: Matrix A, incomplete factorization algorithm, initial shift $\alpha^{(0)}$
Output: Shift α and incomplete factors $\widetilde{L}$ and $\widetilde{U}$ such that $A + \alpha \approx \widetilde{L}\widetilde{U}$

1: **for** $k = 0, 1, 2, \ldots$ **do**
2: $\quad A + \alpha^{(k)} I \approx \widetilde{L}\widetilde{U}$ ▷ Perform incomplete factorization
3: $\quad$ If successful, $\alpha = \alpha^{(k)}$ and **return**
4: $\quad \alpha^{(k+1)} = 2\alpha^{(k)}$
5: **end for**

matrix to be factorizable and, in practice, much smaller values of α can provide incomplete factorizations for which $\|E\|$ is small. A simple trial-and-error procedure for choosing a shift is given in Algorithm 9.1. The initial shift $\alpha^{(0)} = 0$ is reasonable if A is an SPD matrix or, more generally, has positive diagonal entries. If $\alpha^{(0)} > 0$ and the incomplete factorization of $A + \alpha^{(0)} I$ is successful, then the algorithm can be modified to reduce $\alpha^{(0)}$ (for example, it could be replaced by $\alpha^{(0)}/2$) and then restarted. The potential benefit is a smaller $\|E\|$ (and hopefully a higher quality preconditioner) but at the cost of performing further incomplete factorizations. Observe that A should be prescaled to try and limit the size of α.

9.4.3 Pivoting to Prevent Breakdown

An alternative approach to avoid small pivots is to follow what is done in sparse direct solvers and incorporate partial or threshold pivoting within the incomplete factorization algorithm. This potentially makes the factorization significantly more expensive and much more complicated to implement efficiently. As with sparse direct solvers, preprocessing can limit the need for pivoting. If A is nonsymmetric, then row and column permutations can be used to bring large entries onto the diagonal before the factorization commences. In particular, the weighted matching

ordering and scaling discussed in Section 7.4.2 can be used. In the symmetric case, symmetry is preserved by choosing pivots from the diagonal. Again, the matrix should be prescaled, and then at each stage, a straightforward choice is to select as the next pivot the diagonal entry of the largest absolute value in the remaining active submatrix. If there is no suitable diagonal entry (for example, if the absolute values of all the remaining diagonal entries are less than some threshold), then either the diagonal can be modified or 2×2 pivots that preserve symmetry can be used.

One way to attempt to minimize the norm of the error matrix E is to select the pivot candidate to minimize the sum of the absolute values of the dropped (discarded) entries. However, this **minimum discarded fill** ordering is typically too expensive to be useful in practice.

9.5 Factorizations as Preconditioner Components

Sometimes (incomplete) factorizations are employed as components in the construction of more complex preconditioners. Here some possible approaches are briefly discussed.

9.5.1 Polynomial Preconditioning

Polynomial preconditioning selects a polynomial ϕ and applies a Krylov subspace method to solve either

$$\phi(A)Ax = \phi(A)\,b$$

(left preconditioning) or

$$A\,\phi(A)\,y = b, \quad x = \phi(A)\,y$$

(right preconditioning). ϕ should be of small degree and chosen to enhance convergence. Consider the characteristic polynomial $\phi_n(\mu) = \det(A - \mu I)$ of A (det denotes the determinant). The Cayley–Hamilton theorem states that A satisfies its own characteristic equation so that

$$\phi_n(A) = \sum_{j=0}^{n} \beta_j\, A^j = 0,$$

where β_j $(0 \le j \le n)$ are the coefficients of the characteristic polynomial ($\beta_n = 1$, $\beta_0 = (-1)^n \det(A)$). Provided A is nonsingular,

$$A^{-1} = (-1)^{n+1} \frac{1}{\det(A)} \sum_{j=1}^{n} \beta_j \, A^{j-1}.$$

A preconditioner can be constructed by taking the first k terms, possibly weighted by some suitable scalar coefficients, that is,

$$M^{-1} = \sum_{j=0}^{k} \gamma_j \, A^k.$$

An important question is why such a preconditioner can help in the presence of the optimality properties of Krylov subspace methods. For example, at iteration $k+1$ of the CG method, $x^{(k+1)}$ satisfies

$$x^{(k+1)} = x^{(0)} + \phi_k(A)\, r^{(0)},\ k = 0, 1, \ldots,$$

where ϕ_k is a monic polynomial of degree k. This polynomial is optimal in the sense that $x^{(k+1)}$ minimizes

$$\|x - x^{(k+1)}\|_A^2. \tag{9.16}$$

A preconditioner that is a polynomial in A cannot speed the convergence because the resulting iteration again forms the new $x^{(k+1)}$ as $x^{(0)}$ plus a polynomial in A times $r^{(0)}$, and thus the same or a higher degree polynomial is needed to achieve the same value of (9.16). Consequently, the number of matrix-vector multiplications cannot decrease. Nevertheless, polynomial preconditioning can be useful for a number of reasons.

- The polynomial can improve the eigenvalue distribution of the preconditioned matrix and result in a reduction in the number of iterations required for convergence (even though the overall complexity may increase).
- It requires very little memory and its implementation can be straightforward.
- It can decrease the number of synchronization points in iterative methods as represented by inner products. This is potentially important for message-passing parallel architectures.

Even if only a small number of terms are used in the polynomial approximating A^{-1}, a crucial issue is determining the coefficients $\gamma_0, \ldots, \gamma_k$. A straightforward way of doing this is based on the Neumann series of a matrix C given by $\sum_{j=0}^{+\infty} C^j$, which is convergent if and only if $\rho(C) < 1$. In this case,

$$(I - C)^{-1} = \sum_{j=0}^{+\infty} C^j. \tag{9.17}$$

Now let $\bar{M}$ be a nonsingular matrix and $\omega > 0$ a scalar such that the matrix $C = I - \omega \bar{M}^{-1} A$ satisfies $\rho(C) < 1$. Using (9.17),

$$A^{-1} = \omega(\omega \bar{M}^{-1} A)^{-1} \bar{M}^{-1} = \omega\,(I - C)^{-1} \bar{M}^{-1} = \omega \left(\sum_{j=0}^{+\infty} C^j \right) \bar{M}^{-1}.$$

Truncating the summation gives as a possible preconditioner

$$M^{-1} = \omega \left(\sum_{j=0}^{k} C^j \right) \bar{M}^{-1}.$$

Observe that

$$I - M^{-1} A = I - \omega \left(\sum_{j=0}^{k} C^j \right) \bar{M}^{-1} A = I - \left(\sum_{j=0}^{k} C^j \right) (I - C) = C^{k+1},$$

which shows the positive effect of increasing k. If A and $\bar{M}$ are SPD matrices, then M can be used with the CG method preconditioned from the left because $M^{-1} A$ is self-adjoint in the $\bar{M}$-inner product. Generalizations of the approach weight the powers of C in M^{-1} using additional scalars. The choice of $\bar{M}$ is crucial for the effectiveness of the approach.

9.5.2 Schur Complement Approach and Deflation

Many contemporary preconditioners are constructed hierarchically. A straightforward example is represented by the approximate solution of saddle point problems using the Schur complement approach. Consider the following general saddle point system:

$$Ax = \begin{pmatrix} G & C \\ R & B \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}. \tag{9.18}$$

Assuming G is nonsingular, eliminating x_1 from the second block row yields the reduced system

$$Sx_2 = b_2 - RG^{-1} b_1, \tag{9.19}$$

where $S = B - RG^{-1}C$ is the Schur complement of G in A. Solving (9.19) involves solving a linear system with G and with S. One option is to compute an LU factorization of G and then employ a preconditioned iterative method; this is

ALGORITHM 9.2 Simple Schur complement approach for saddle point systems
Input: Nonsingular saddle point system (9.18) with G nonsingular.
Output: Computed solution x.

1: Compute LU factorization of G
2: Solve $Gz = b_1$ ▷ Use LU factors
3: Compute $\widetilde{S}^{-1} \approx (B - RG^{-1}C)^{-1}$ ▷ $\widetilde{S}^{-1}$ chosen to approximate S^{-1}
4: Solve $Sx_2 = b_2 - Rz$ ▷ Use iterative method with $M^{-1} = \widetilde{S}^{-1}$
5: Solve $Gx_1 = b_1 - Cx_2$ ▷ Use LU factors

outlined in Algorithm 9.2. Combining direct and iterative techniques is sometimes referred to as a **hybrid** approach.

The Schur complement (or substructuring) approach can be extended to matrices that are split into more blocks. Blocks may arise naturally from the underlying application, but they can also be defined using purely algebraic rules. For example, consider an SPD matrix A. Applying graph partitioning techniques (such as the nested dissection approach of Section 8.4) to the adjacency graph $\mathcal{G}(A)$, A can be symmetrically permuted to the doubly bordered block diagonal (DBBD) form

$$P^T A P = A_{DB} = \begin{pmatrix} G_D & R^T \\ R & B \end{pmatrix},$$

where G_D is an SPD block diagonal matrix (Section 8.5.1). A_{DB} is a special case of a symmetric saddle point matrix. A block LDLT factorization of A_{DB} is given by

$$A_{DB} = \begin{pmatrix} I & \\ RG_D^{-1} & I \end{pmatrix} \begin{pmatrix} G_D & \\ & S \end{pmatrix} \begin{pmatrix} I & G_D^{-1}R^T \\ & I \end{pmatrix},$$

where the matrix S is the SPD Schur complement. The blocks within G_D can be factorized in parallel using a sparse Cholesky solver. However, S is typically large and significantly denser than B and, in large-scale practical applications, it may not be possible to explicitly assemble and factorize it; in this case, a preconditioned iterative method is needed.

If $\widetilde{S}^{-1} \approx S^{-1}$, then an approximate block factorization of A_{DB}^{-1} is

$$M^{-1} = \begin{pmatrix} I & -G_D^{-1}R^T \\ & I \end{pmatrix} \begin{pmatrix} G_D^{-1} & \\ & \widetilde{S}^{-1} \end{pmatrix} \begin{pmatrix} I & \\ -RG_D^{-1} & I \end{pmatrix}.$$

Employing M^{-1} as a preconditioner for A_{DB} gives the preconditioned matrix

$$M^{-1}A_{DB} = \begin{pmatrix} I & G_D^{-1}R^T(I - \widetilde{S}^{-1}S) \\ & \widetilde{S}^{-1}S \end{pmatrix}.$$

Applying M^{-1} requires the efficient solution of linear systems with $\widetilde{S}^{-1}S$ and G_D. As in other preconditioning approaches, bounding the condition number of the preconditioned matrix may be a useful indicator of the expected convergence of CG. The eigenvalues of $M^{-1}A_{DB}$ are those of $\widetilde{S}^{-1}S$ and unity. Note that the spectrum of $M^{-1}A_{DB}$ is the same as the spectrum of $M^{-1/2}A_{DB}M^{-1/2}$. Thus, $\kappa(M^{-1/2}A_{DB}M^{-1/2})$ depends on the extremal eigenvalues of $\widetilde{S}^{-1}S$. A one-level preconditioner for S is obtained by setting

$$\widetilde{S}_1^{-1} = B^{-1}.$$

Let the matrix B be $m \times m$ and let $\lambda_1 \geq \cdots \geq \lambda_m > 0$ be the eigenvalues of the generalized eigenvalue problem

$$Sz = \lambda \widetilde{S}_1 z.$$

Because $\widetilde{S}^{-1}S = I - B^{-1}RG_D^{-1}R^T$, it follows that $\lambda_1 \leq 1$ and so

$$\kappa(\widetilde{S}_1^{-1}S) = \kappa(\widetilde{S}_1^{-1/2}\, S\, \widetilde{S}_1^{-1/2}) = \frac{\lambda_1}{\lambda_m} \leq \frac{1}{\lambda_m},$$

which is unbounded as λ_m approaches zero. In general, one-level algebraic preconditioners successfully bound the largest eigenvalues of the preconditioned matrix but encounter difficulties in controlling the smallest ones, which can lie close to the origin, hindering convergence. Strategies that involve a second-level component aim to overcome this and include **deflation** preconditioners and **domain decomposition** preconditioners.

The basic idea behind deflation is to "hide" parts of the spectrum from the CG method such that the CG iteration "sees" a system that has a much smaller condition number and hopefully a more favourable eigenvalue distribution than the original matrix. The part of the spectrum that is hidden is determined by the deflation subspace and the improvement in the convergence rate of the deflated CG method depends on the choice of this subspace. The ideal deflation subspace is the invariant subspace spanned by the eigenvectors corresponding to the smallest eigenvalues. There are practical cases showing convergence of the preconditioned iterative method may profit from this restriction of the spectrum to its "effective" part. To illustrate the approach, let Λ be the $k \times k$ diagonal matrix with entries equal to the k smallest eigenvalues and let Z be the $m \times k$ matrix whose columns are the corresponding eigenvectors. A two-level deflation preconditioner is defined to be

$$\widetilde{S}_2^{-1} = B^{-1} + Z(\Lambda^{-1} - I)Z^T = \widetilde{S}_1^{-1} + Z(\Lambda^{-1} - I)Z^T.$$

In practice, challenges remain because Λ and Z are typically not readily available.

9.5.3 Domain Decomposition

In the last section, the vertices $\mathcal{V} = \{1, 2, \ldots, n\}$ of $\mathcal{G}(A)$ were partitioned into non-overlapping subsets. Alternatively, overlapping subsets (which are generally termed subdomains because the approach was originally proposed for problems that had an underlying grid) may be used. Domain decomposition methods based on overlapping subdomains are often referred to as **Schwarz methods**. Given $N > 1$, let $\Omega_{\Gamma i}$ be the subset of size $n_{\Gamma i}$ of vertices that are distance one in $\mathcal{G}(A)$ from the vertices in Ω_{Ii} $(1 \leq i \leq N)$. The overlapping subdomain Ω_i is defined to be $\Omega_i = [\Omega_{Ii}, \Omega_{\Gamma i}]$, with size $n_i = n_{\Gamma i} + n_{Ii}$.

Associate with Ω_i an $n_i \times n$ restriction (or projection) matrix given by $R_i = I_n(\Omega_i, :)$. R_i maps from the global domain to subdomain Ω_i; its transpose R_i^T is a prolongation matrix that maps from subdomain Ω_i to the global domain. The **one-level additive Schwarz preconditioner** is defined to be

$$M_{AS}^{-1} = \sum_{i=1}^{N} R_i^T A_i^{-1} R_i, \qquad A_i = R_i A R_i^T. \tag{9.20}$$

Applying this preconditioner to a vector involves solving concurrent local problems in the overlapping subdomains. Increasing N reduces the sizes n_i of the overlapping subdomains, leading to smaller local problems and faster computations. However, the preconditioned system using M_{AS}^{-1} may not be well conditioned and the convergence of the iterative solver may be inhibited. In fact, the local nature of this preconditioner can lead to a deterioration in its effectiveness as the number of subdomains increases because of the lack of global information from the matrix A. To maintain robustness with respect to N, an artificial subdomain is added to the preconditioner (also known as second-level or coarse space correction) that includes global information. Let $0 < n_0 \ll n$. If the $n_0 \times n$ matrix R_0 is of full row rank, the **two-level additive Schwarz preconditioner** is defined to be

$$M_{AS2}^{-1} = M_{AS}^{-1} + R_0^T A_0^{-1} R_0, \qquad A_0 = R_0 A R_0^T.$$

The coarse space correction can also be applied in a multiplicative way, which can lead to more robust variants. A sparse direct method can be used for the solves with each A_i, which has the advantage of being robust and is another example of a hybrid approach. Alternatively, for very large systems, incomplete IC factorization preconditioners or approximate inverse preconditioners and an iterative method can be used. While this may result in a slower convergence rate, it can lead to a faster method overall because each iteration is less expensive (and may be the only option if the direct solver requires too much memory). Generalizing the approach to a hierarchy of additions of artificial domains leads to the class of **multilevel** methods. Again, employing them as preconditioners requires solves with the domain matrices, which can be based on sparse direct methods or preconditioned iterative methods.

An attractive feature of domain decomposition methods is that they are naturally parallel because all subdomain computations can be performed simultaneously. The restricted additive Schwarz preconditioner is obtained by a simple and efficient change that removes the overlap in the prolongation, replacing (9.20) by

$$M_{RAS}^{-1} = \sum_{i=1}^{N} \widehat{R}_i^T A_i^{-1} R_i,$$

where $\widehat{R}_i = I_n(\Omega_{Ii}, :)$. The main motivation here is to reduce the communication cost by half because computing products such as $\widehat{R}_i w$ does not involve any data exchange with neighbouring processors.

9.6 Notes and References

A useful textbook on iterative methods is Saad (2003b). It includes the result stated in Theorem 9.3, while the proof of Theorem 9.2 is given in Mendelsohn (1956). Other key books include Meurant (1999), van der Vorst (2003), and the recent monograph of Bertaccini & Durastante (2018), as well as Liesen & Strakoš (2013) and Meurant & Duintjer Tebbens (2020), which targets theoretical and practical properties of iterative methods. The excellent surveys of Benzi (2002) and Wathen (2015), Pearson & Pestana (2020) present overviews of preconditioning techniques and the monograph Chen (2005) describes several approaches and includes many example applications, while Bollhöfer (2015) gives a practically oriented survey that mainly targets multilevel and parallel aspects of algebraic preconditioners. A discussion of the desirable properties of preconditioners can be found in Chow & Saad (1997). More sophisticated dropping strategies and the relation between ILU factorizations and factorized approximate inverses are considered by Bollhöfer & Saad (2002, 2006); while Kopal et al. (2016) discuss adaptive dropping.

For a basic introduction to the stability problems of LU-based preconditioners, see Elman (1986, 1989). The Eisenstat trick of Section 9.2.3 is presented by Eisenstat (1981). An interesting discussion putting this into the context of other similar ideas is given in Ortega (1988a).

The issue of potential breakdown during incomplete factorizations was pointed out by Kershaw (1978). This strengthened interest in classes of matrices for which breakdown cannot occur. Theorem 9.4 for M-matrices is from Meijerink & van der Vorst (1977); the extension to H-matrices is given independently by Manteuffel (1980) and Varga et al. (1980). Favourable asymptotic bounds for the condition number of M-matrices preconditioned by modified incomplete factorizations were an important impetus behind the development of algebraic preconditioners. These are described in Axelsson (1972) and Gustafsson (1978, 1979), but see also the early sophisticated analysis of relaxation methods presented in Dupont et al. (1968). Some of the assumptions that were used to obtain early asymptotic bounds were later shown to be unnecessary (Bern et al., 2006). Practical choices of polynomial

preconditioners, particularly for SPD systems, are discussed in the book by Saad (2003b) (and the earlier introductory paper of Saad, 1985). Note the recent interest of Loe & Morgan (2021) and Ye et al. (2021), the former motivated by the potential to reduce communication in parallel computing.

For preconditioning saddle point problems using algebraic approaches, the highly cited survey of Benzi et al. (2005) and monograph of Rozložník (2018) are good starting points. We also refer to the papers by Maryška et al. (1996, 2000a,b) and Arioli et al. (2006) on the iterative solution of algebraically preconditioned saddle point problems from PDE applications.

There are a number of monographs on domain decomposition methods. An important algorithmically oriented introduction is Smith et al. (1996), but see also Quarteroni & Valli (1999) and Toselli & Widlund (2005) as well as the books by Olshanskii & Tyrtyshnikov (2014) and Dolean et al. (2015), which emphasize connections to PDEs and solution techniques motivated by them. We recommend the paper of Tang et al. (2009) for an algebraic comparison of different classes of domain decomposition and deflation preconditioners. A further line of research resulting in general algebraic preconditioners has been developed using hierarchical matrices; the papers include Bebendorf & Fischer (2008) and Bebendorf et al. (2013) and the monograph on hierarchical matrices of Bebendorf (2008). The ShyLU software package developed by Rajamanickam et al. (2012) is a fully algebraic hybrid package for solving sparse linear systems using domain decomposition methods. It offers distributed memory domain decomposition solvers and node level solvers and kernels that support the distributed memory solvers. The node level solvers include sparse LU and Cholesky factorizations, a multithreaded triangular solver, and a fast iterative ILU algorithm. ShyLU is available as part of Trilinos (ShyLU Project Team, 2022).

Algebraic multigrid (AMG) methods are another important class of frequently used methods. AMG methods can be used to precondition a wide spectrum of problems, but their development has been mainly motivated by systems arising from the discretization of PDEs, often exploiting specific properties of discretized models. A recommended overview is by Xu & Zikatanov (2017); see also Stüben et al. (2017).

Chapter 10
Incomplete Factorizations

They [incomplete factorizations] can be thought of as approximating the exact LU factorization of a given matrix A (e.g. computed via Gaussian elimination) by disallowing certain fill-ins. As opposed to other PDE-based preconditioners such as multigrid and domain decomposition, this class of preconditioners are primarily algebraic in nature and can in principle be applied to any sparse matrices. When applied to PDE problems, they are usually not optimal ... On the other hand, they are often quite robust. – Chan & van der Vorst (1997).

Having introduced incomplete factorization preconditioners in the previous chapter, the focus in this chapter is on different ways to compute such factorizations and their relationship to the complete factorizations used in sparse direct methods. We denote the incomplete factors by $\widetilde{L}$ and $\widetilde{U}$; in the SPD case, $\widetilde{U} = \widetilde{L}^T$. We assume that the sparsity patterns of A and its incomplete factors always include the positions of the diagonal entries.

10.1 ILU(0) Factorization

The simplest sparsity pattern for an incomplete factorization is $\mathcal{S}\{\widetilde{L} + \widetilde{U}\} = \mathcal{S}\{A\}$, that is, no entries in $\widetilde{L}$ or $\widetilde{U}$ are allowed outside the sparsity pattern of A and only entries in positions $(i, j) \in \mathcal{S}\{A\}$ are retained in the (incomplete) elimination matrices. The resulting incomplete factorization is called an ILU(0) factorization (or an IC(0) factorization if A is SPD).

Motivation for considering a sparsity pattern that is a superset of $\mathcal{S}\{A\}$ is given by the following straightforward but important result.

Theorem 10.1 (Chan & van der Vorst 1997; van der Vorst 2003) *Consider the incomplete LU factorization $A + E = \widetilde{L}\widetilde{U}$ with sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. The entries of the error matrix E are zero at positions $(i, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$.*

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_10

Proof The result clearly holds for $j = 1$. Let $(i, j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ and assume without loss of generality that $i > j > 1$. The (i, j) entry of $\widetilde{L}$ is computed as

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\, \tilde{u}_{kj} \right) / \tilde{u}_{jj}$$

with the sums over k implying $(i, k) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ and $(k, j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$. This gives

$$a_{ij} = \widetilde{L}_{i,1:j-1}\widetilde{U}_{1:j-1,j} + \tilde{l}_{ij}\tilde{u}_{jj} = \widetilde{L}_{i,1:j}\widetilde{U}_{1:j,j} = L_{i,1:j}U_{1:j,j},$$

and the corresponding entry of E is zero. □

A consequence of Theorem 10.1 is that extending $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ gives a larger set of entries of A for which the error is zero. This is attractive provided the incomplete factorization can still be computed and employed cheaply and does not require prohibitive amounts of memory. In some situations, there are straightforward ways to extend $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$. For example, consider a simple discretization of a PDE on a rectangular grid. The sparsity pattern of the corresponding SPD matrix A and its graph $\mathcal{G}(A)$ together with the first three steps of the Cholesky factorization of A (in which variables 1, 2, and 3 are eliminated in turn) are given in Figure 10.1. A has entries on the diagonal and four of its subdiagonals and the fill-in lies within $band(A)$. A natural choice is to allow $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ to include fill-in along a few additional diagonals within the band.

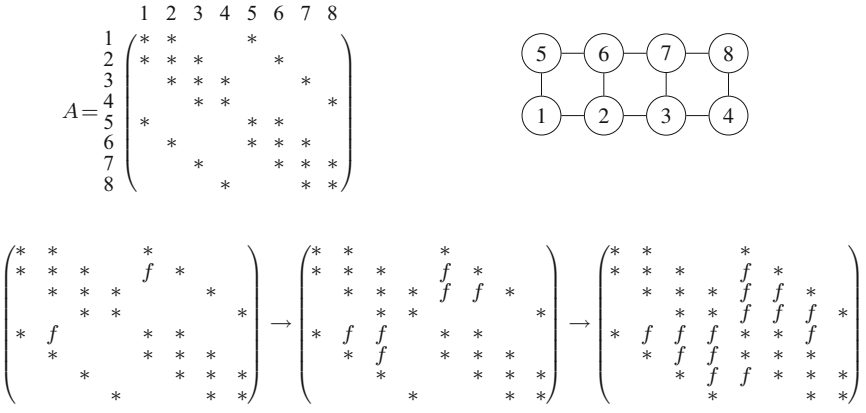


Figure 10.1 An 8×8 banded sparse SPD matrix A and its graph $\mathcal{G}(A)$. The first three steps of a Cholesky factorization are shown. Filled entries are denoted by f.

10.2 Basic Incomplete Factorizations

We start with the two basic incomplete factorizations. Here and elsewhere, section notation is used but operations are performed only on nonzero entries. The Crout variant given in Algorithm 10.1 computes $\widetilde{U}$ row-by-row and $\widetilde{L}$ column-by-column and sparsifies each row and column as soon as they are computed using a target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. The widely used variant outlined in Algorithm 10.2 constructs both $\widetilde{L}$ and $\widetilde{U}$ by rows. Prescribing an appropriate sparsity pattern in advance can be difficult. If it is not supplied, sparsification can be applied inside the k loops (for instance, entries with absolute value less than a chosen tolerance may be dropped) and the sparsity patterns of the factors updated as the factorization proceeds.

Algorithms 10.1 and 10.2 are straightforward to implement using sparse data structures. At major step i, Algorithm 10.2 computes $\widetilde{L}_{i,1:i-1}$ and $\widetilde{U}_{i,i+1:n}$; both rows can be held using a single auxiliary vector. Note that, in Algorithm 10.1, sparsification of the partially computed vectors is performed outside the k loops, whereas in Algorithm 10.2 it is inside the k loop. In practice, either approach can be used, leading to slightly different variants.

ALGORITHM 10.1 Crout incomplete LU factorization

Input: Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

1: **for** $j = 1 : n$ **do**
2: $\quad \tilde{l}_{jj} = 1,\ \widetilde{L}_{j+1:n,j} = A_{j+1:n,j}$
3: $\quad \widetilde{U}_{j,j:n} = A_{j,j:n}$
4: $\quad$ **for** $k = 1 : j-1$ such that $(j,k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
5: $\quad\quad \widetilde{U}_{j,j:n} = \widetilde{U}_{j,j:n} - \tilde{l}_{jk}\,\widetilde{U}_{k,j:n}$ ▷ Sparse linear combination
6: $\quad$ **end for**
7: $\quad$ Sparsify $\widetilde{U}_{j,j+1:n}$ ▷ Drop entries from row j of $\widetilde{U}$
8: $\quad$ **for** $k = 1 : j-1$ such that $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
9: $\quad\quad \widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j} - \tilde{u}_{kj}\,\widetilde{L}_{j+1:n,k}$ ▷ Sparse linear combination
10: $\quad$ **end for**
11: $\quad$ Sparsify $\widetilde{L}_{j+1:n,j}$ ▷ Drop entries from column j of $\widetilde{L}$
12: $\quad \widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j}/\tilde{u}_{jj}$
13: **end for**

ALGORITHM 10.2 Row incomplete LU factorization

Input: Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

```
for i = 1 : n do
    l̃_ii = 1, L̃_{i,1:i−1} = A_{i,1:i−1}
    Ũ_{i,i:n} = A_{i,i:n}
    Sparsify L̃_{1,1:i−1} and Ũ_{i,i+1:n}
    for k = 1 : i − 1 such that (i, k) ∈ S{L̃} do
        l̃_ik = l̃_ik / ũ_kk
        L̃_{i,k+1:i−1} = L̃_{i,k+1:i−1} − l̃_ik Ũ_{k,k+1:i−1}
        Sparsify L̃_{i,k+1:i−1}
        Ũ_{i,i:n} = Ũ_{i,i:n} − l̃_ik Ũ_{k,i:n}
        Sparsify Ũ_{i,i+1:n}
    end for
end for
```

10.3 Incomplete Factorizations Based on the Shortest Fill-Paths

We next consider an incomplete LU factorization that uses a structure-based dropping strategy. Entries of the factors that correspond to nonzero entries of A are assigned the level 0, while each potential filled entry in position (i, j) is assigned a level as follows:

$$level(i, j) = \min_{1 \le k < \min\{i,j\}} (level(i, k) + level(k, j) + 1). \tag{10.1}$$

Given $\ell \ge 0$, during the factorization, a filled entry is permitted at position (i, j) provided $level(i, j) \le \ell$. The resulting **level-based** incomplete factorization is denoted by ILU(ℓ) (or IC(ℓ)); the basic row variant is given in Algorithm 10.3.

Figure 10.2 depicts $\mathcal{S}\{\widetilde{L} + \widetilde{L}^T\}$ for the IC(ℓ) factorization of A from the discretized Laplace equation on a square grid (see the smaller problem in (9.14)) and for a matrix with a more general symmetric sparsity structure. The fill-in is typically generated irregularly throughout the factorization: initially few updates are needed, but later steps involve many updates, leading to large amounts of dropping. Furthermore, the amount of fill-in can grow quickly with increasing ℓ and, as a result, ℓ is typically small and level-based dropping is often combined with threshold-based dropping or with sparsifying A before the factorization commences (for example, by discarding entries of A with small absolute values).

ALGORITHM 10.3 Level-based incomplete LU factorization
Input: Matrix A and the level parameter $\ell \geq 0$.
Output: ILU(ℓ) factorization $A \approx \widetilde{L}\widetilde{U}$.

1: Initialise $level$ to 0 for nonzeros and diagonal entries of A and to $n+1$ otherwise
2: **for** $i = 1 : n$ **do** ▷ Loop over rows
3: $\tilde{l}_{ii} = 1$, $\widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$ and $\widetilde{U}_{i,i:n} = A_{i,i:n}$ ▷ Initialise row i of $\widetilde{L}$ and $\widetilde{U}$
4: **for** $k = 1 : i-1$ such that $level(i,k) \leq \ell$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
6: **for** $j = k+1 : i-1$ **do**
7: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ and update $level(i,j)$
8: **end for**
9: **for** $j = i : n$ **do**
10: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ and update $level(i,j)$
11: **end for**
12: **end for**
13: **for** $k = 1 : i-1$ **do** ▷ Drop entries in row i for which $level$ is too high
14: **if** $level(i,k) > \ell$ **then** $\tilde{l}_{ik} = 0$
15: **end for**
16: **for** $k = i : n$ **do**
17: **if** $level(i,k) > \ell$ **then** $\tilde{u}_{ik} = 0$
18: **end for**
19: **end for**

The level-based strategy comes from observing that in practical examples the absolute values of the entries in the factors in positions for which $level$ is large are often small. This is the case for model problems arising from discretized PDEs. A closer look shows a surprising connection between the level-based ILU factorization and the complete factorization: entries with large values of $level$ correspond to long fill-paths. This is expressed in Theorem 10.2, which allows the sparsity patterns of the incomplete factors to be determined a priori.

Theorem 10.2 (Hysom & Pothen 2002) *Consider the ILU(ℓ) factorization of A. $level(i,j) = k$ for some $k \leq \ell$ if and only if there is a shortest fill-path $i \Longrightarrow j$ of length $k+1$ in the adjacency graph $\mathcal{G}(A)$.*

Algorithm 10.4 outlines finding the pattern of row i of $\widetilde{U}$; finding the pattern of columns of $\widetilde{L}$ is analogous. Only $\mathcal{G}(A)$ is required, and hence the sparsity pattern of each row in the factor can be computed independently, in parallel. The algorithm operates via a simple breadth-first search that finds a shortest path between vertex

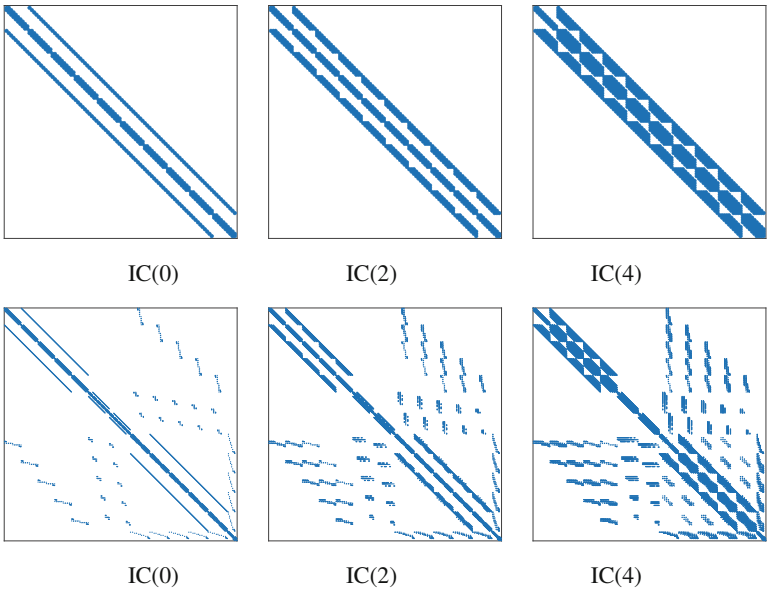


Figure 10.2 The sparsity patterns of the IC(ℓ) factors of A from the discretized Laplace equation on a square grid (top) and a more general symmetric sparse matrix (bottom).

i and vertices reachable from i via a graph traversal of $l + 1$ or fewer edges. The correctness of the algorithm follows from Theorem 10.2.

10.4 Modifications Based on Maintaining Row Sums

We assume in this section that the target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ contains $\mathcal{S}\{A\}$. **Modified incomplete factorizations** (MILU or MIC in the SPD case) seek to maintain equality between the row sums of A and $\widetilde{L}\widetilde{U}$, that is, $\widetilde{L}\widetilde{U}e = Ae$ (e is the vector of all ones). Rather than discarding potential fill-in outside the target sparsity pattern, the approach subtracts it from the diagonal entries of $\widetilde{U}$; this is outlined in Algorithm 10.5. Note that an MILU factorization may break down. If the target sparsity pattern corresponds to that of an ILU(ℓ) factorization, then an MILU(ℓ) factorization is computed.

Equality of the row sums of A and $\widetilde{L}\widetilde{U}$ can be seen as follows. If all the filled entries are retained (that is, $\mathcal{S}\{\widetilde{L} + \widetilde{U}\} = \mathcal{S}\{L + U\}$), then the claim holds trivially. Now assume some filled entries are not kept. If an entry in column j of row i of A belongs to the target sparsity pattern, then its value is modified in Step 8 if $i \leq j$ or in Step 15 if $i > j$. Otherwise, the i-th diagonal entry of $\widetilde{U}$ is modified (Step 10 or Step 17). In each case, $\tilde{l}_{ik}\,\tilde{u}_{kj}$ is subtracted from entries of the i-th row of the incomplete factors. Consider row i of $\widetilde{L}\widetilde{U}$. This product is given by

ALGORITHM 10.4 Find the sparsity pattern of row i of the ILU(ℓ) factor $\widetilde{U}$ of A
Input: Graph $\mathcal{G}(A)$, the level parameter $\ell \geq 0$ and row index i.
Output: Sparsity pattern $\mathcal{S}\{\widetilde{U}_{i,i:n}\}$ of row i of the ILU(ℓ) factorization $A \approx \widetilde{L}\widetilde{U}$.

```
S{Ũ_{i,i:n}} = {i}, Q = {i}                          ▷ Queue holds i initially
length(i) = 0
visited(i) = i
while Q is not empty do
    pop(Q, k)                                        ▷ Take k from the queue
    for j ∈ adj_{G(A)}(k) with visited(j) ≠ i do
        visited(j) = i
        if j < i and length(k) < ℓ then
            append(Q, j)                             ▷ Add j to the queue
            length(j) = length(k) + 1
        else if j > i then
            S{Ũ_{i,i:n}} = S{Ũ_{i,i:n}} ∪ {j}        ▷ Add j to the sparsity pattern of row i
        end if
    end for
end while
```

$$\sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j}^{n} \tilde{u}_{jk} = \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jj} + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} + \sum_{k=i}^{n} \tilde{u}_{ik} =$$

$$= \sum_{j=1}^{i-1} \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\tilde{u}_{kj} \right) + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} + \sum_{k=i}^{n} \left(a_{ik} - \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jk} \right)$$

$$= \sum_{j=1}^{n} a_{ij} + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} - \left(\sum_{j=1}^{i-1} \sum_{k=1}^{j-1} \tilde{l}_{ik}\tilde{u}_{kj} + \sum_{k=i}^{n} \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jk} \right).$$

Rearranging the indices in the double summations, the last three sums cancel out. Moreover, the added double summation is the sum of all the modification terms $\tilde{l}_{ik}\,\tilde{u}_{kj}$ in Algorithm 10.5, and the sum of the two subtracted double summations also comprises all the modification terms. Consequently, the row sums of A are preserved in the product of the incomplete factors.

Theorem 10.3 provides motivation for maintaining constant row sums in the case of a model PDE problem. The result is also valid for Neumann or mixed boundary conditions, and there are extensions to three-dimensional problems and MIC(ℓ)

ALGORITHM 10.5 Modified incomplete factorization (MILU)

Input: Matrix $A = L_A + D_A + U_A$ (see (9.6)) and a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ containing $\mathcal{S}\{A\}$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

1: $\tilde{l}_{ij} = (I + L_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{L})$ ▷ $\mathcal{S}(L_A) \subseteq \mathcal{S}(\widetilde{L})$
2: $\tilde{u}_{ij} = (D_A + U_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{U})$ ▷ $\mathcal{S}(U_A) \subseteq \mathcal{S}(\widetilde{U})$
3: **for** $k = 1 : n - 1$ **do**
4: **for** $i = k + 1 : n$ such that $(i, k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$ ▷ Check that $\tilde{u}_{kk}$ is nonzero
6: **for** $j = i : n$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
7: **if** $(i, j) \in \mathcal{S}\{\widetilde{U}\}$ **then**
8: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
9: **else**
10: $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ ▷ Modify diagonal instead of creating fill-in
11: **end if**
12: **end for**
13: **for** $j = k + 1 : i - 1$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
14: **if** $(i, j) \in \mathcal{S}\{\widetilde{L}\}$ **then**
15: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
16: **else**
17: $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ ▷ Modify diagonal instead of creating fill-in
18: **end if**
19: **end for**
20: **end for**
21: **end for**

with $\ell > 0$. However, although Theorem 10.1 holds for MILU factorizations, the approach may not be useful for general A.

Theorem 10.3 (Gustafsson 1978; Bern et al. 2006) *Let A come from a discretized Poisson problem on a uniform two-dimensional rectangular grid with Dirichlet boundary conditions and discretization parameter h. Then the condition number $\kappa((\widetilde{L}\widetilde{U})^{-1}A)$ for the level-based MIC(0) preconditioner is $O(h^{-1})$.*

Optionally, in Steps 10 and 17 of Algorithm 10.5, the update term $\tilde{l}_{ik}\,\tilde{u}_{kj}$ may be multiplied by a parameter θ $(0 < \theta < 1)$ before it is subtracted from the diagonal entry $\tilde{u}_{ii}$. The introduction of θ was proposed as a practical way to extend MILU to linear systems not coming from discretized PDEs. Clearly, using $\theta < 1$ reduces the amount by which the diagonal entries are modified.

10.5 Dynamic Compensation

As discussed in Section 9.4.1, dropping entries can lead to breakdown. One way to avoid this (in exact arithmetic) is to dynamically modify the computed entries; this is outlined as Algorithm 10.6. Instead of accepting a filled entry in position (i, j), the idea is to add a (weighted) multiple of its absolute value to the corresponding diagonal entries $\tilde{u}_{ii}$ and $\tilde{u}_{jj}$. Provided the number of modifications is small, this can be useful if A is a diagonally dominant matrix and scaled so that its diagonal entries are nonnegative. The parameter ω controls the amount by which the diagonal entries of $\widetilde{U}$ are modified, but if $\omega < 1$, then breakdown can still occur. Dynamic compensation can be successful when incorporated into an IC factorization of

ALGORITHM 10.6 ILU factorization with dynamic compensation

Input: Matrix $A = L_A + D_A + U_A$ (see (9.6)), a target sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and parameter ω $(0 \le \omega \le 1)$.
Output: Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.

$\tilde{l}_{ij} = (I + L_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{L})$
$\tilde{u}_{ij} = (D_A + U_A)_{ij}$ for all $(i, j) \in \mathcal{S}(\widetilde{U})$
for $k = 1 : n - 1$ **do**
  for $i = k + 1 : n$ such that $(i, k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
    $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
    for $j = i : n$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
      if $(i, j) \in \mathcal{S}\{\widetilde{U}\}$ **then**
        $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
      else
        $\rho = (\tilde{u}_{ii}/\tilde{u}_{jj})^{1/2}$
        $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega\rho\,|\tilde{l}_{ik}\,\tilde{u}_{kj}|, \quad \tilde{u}_{jj} = \tilde{u}_{jj} + \omega|\tilde{l}_{ik}\,\tilde{u}_{kj}|\,/\rho, \quad \tilde{u}_{ij} = 0.$
      end if
    end for
    for $j = k + 1 : i - 1$ such that $(k, j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
      if $(i, j) \in \mathcal{S}\{\widetilde{L}\}$ **then**
        $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
      else
        $\rho = (\tilde{u}_{ii}/\tilde{u}_{jj})^{1/2}$
        $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega\rho\,|\tilde{l}_{ik}\,\tilde{u}_{kj}|, \quad \tilde{u}_{jj} = \tilde{u}_{jj} + \omega|\tilde{l}_{ik}\,\tilde{u}_{kj}|\,/\rho, \quad \tilde{l}_{ij} = 0.$
      end if
    end for
  end for
end for

an SPD matrix A because the resulting local modifications correspond to adding positive semidefinite matrices to A. In practice, the behaviour of the resulting preconditioner can be very different from that computed using the MIC approach of the previous section.

A related scheme, called **diagonally compensated reduction**, modifies A before the factorization begins by adding the values of all of its positive off-diagonal entries to the corresponding diagonal entries and then setting these off-diagonal entries to zero. If A is SPD, then the resulting matrix is a symmetric M-matrix and the incomplete factorization will not break down (Theorem 9.4). However, the modified matrix may be too far from A for its incomplete factors to be useful.

10.6 Memory-Limited Incomplete Factorizations

We next consider a more sophisticated modification scheme that introduces the use of intermediate memory that is employed during the construction of the incomplete factors but is then discarded. The aim is to obtain a high quality preconditioner while maintaining sparsity and allowing the user to control how much memory is used (both in the construction of the preconditioner and in the incomplete factor $\widetilde{L}$). Let the matrix A be SPD and consider the decomposition

$$A = (\widetilde{L} + \widetilde{R})\,(\widetilde{L} + \widetilde{R})^T - E.$$

Here the incomplete factor $\widetilde{L}$ is a lower triangular matrix with positive diagonal entries, $\widetilde{R}$ is a strictly lower triangular matrix with "small" entries, and the error matrix is $E = \widetilde{R}\widetilde{R}^T$. At each step, the next column of $\widetilde{L}$ is computed, and then the remaining Schur complement is modified. On step j of the incomplete factorization, the first column of the Schur complement $S^{(j)}$ is split into the sum

$$\widetilde{L}_{j:n,j} + \widetilde{R}_{j:n,j},$$

where $\widetilde{L}_{j:n,j}$ contains the entries that are retained in column j of the final incomplete factorization, $(\widetilde{R})_{jj} = 0$ and $\widetilde{R}_{j+1:n,j}$ contains the entries that are discarded. If a complete factorization was being computed, then the Schur complement would be updated by subtracting

$$(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})\,(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})^T.$$

However, the incomplete factorization discards the term

$$E^{(j)} = \widetilde{R}_{j+1:n,j}\,\widetilde{R}^T_{j+1:n,j}.$$

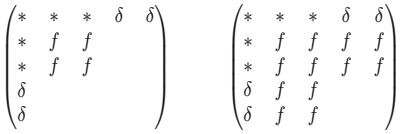


Figure 10.3 An illustration of the fill-in in a standard sparsification-based IC factorization (left) and in the approach that uses intermediate memory (right) after one step of the factorization. Entries with a small absolute value in row and column 1 are denoted by δ. The filled entries are denoted by f.

Figure 10.4 On the left is an SPD matrix with an entry of small absolute in positions $(1, 3)$ and $(3, 1)$. In the centre is $\mathcal{S}\{\widetilde{L}\}$ computed using a standard IC factorization that drops the small entry δ at position $(3, 1)$ (there are no filled entries in this case). On the right is the lower triangular part of the elimination matrix after the first step of the incomplete factorization using intermediate memory. The filled entry is denoted by f.

Thus, the matrix $E^{(j)}$ is implicitly added to A, and because $E^{(j)}$ is positive semidefinite, the approach is naturally breakdown-free.

The obvious choice for $\widetilde{R}_{j+1:n,j}$ is the smallest off-diagonal entries in the column (those that are smaller in absolute value than a chosen tolerance). Then implicitly adding $E^{(j)}$ is combined with the standard steps of an IC factorization, with entries dropped from $\widetilde{L}$ after the updates have been applied to the Schur complement.

Figure 10.3 depicts the first step of this approach. In the first row and column, $*$ and δ denote the entries of $\widetilde{L}_{1:n,1}$ and $\widetilde{R}_{1:n,1}$, respectively. Because a standard sparsification scheme does not store the smallest entries, using such a scheme gives no fill-in in the rows and columns corresponding to the discarded entries; this is shown on the left. The fill-in in the factorization that uses intermediate memory is depicted on the right. Clearly, more filled entries are used in constructing $\widetilde{L}$.

This strategy enables the structure of the complete factorization to be followed more closely than is possible using a standard approach. This is illustrated in Figure 10.4. If the small entries at positions $(1, 3)$ and $(3, 1)$ are not discarded, then there is a filled entry in position $(3, 2)$ and this allows the incomplete factorization using intermediate memory to involve the (large) off-diagonal entries in positions $(5, 2)$ and $(6, 2)$ in the second step of the IC factorization.

Unfortunately, because the column $\widetilde{R}_{j+1:n,j}$ must be retained to perform the updates on the next step, the total memory requirements are essentially as for a

ALGORITHM 10.7 Crout memory-limited IC factorization
Input: SPD matrix A, memory control parameters $lsize > 0$ and $rsize \geq 0$.
Output: Incomplete Cholesky factorization $A \approx \widetilde{L}\widetilde{L}^T$.

$w_i = 0, \quad 1 \leq i \leq n$
for $j = 1 : n$ **do**
  for $i = j : n$ such that $a_{ij} \neq 0$ **do**
    $w_i = a_{ij}$
  end for
  for $k < j$ such that $\tilde{l}_{jk} \neq 0$ **do**
    for $i = j : n$ such that $\tilde{l}_{ik} \neq 0$ **do**
      $w_i = w_i - \tilde{l}_{ik}\tilde{l}_{jk}$
    end for
    for $i = j : n$ such that $\tilde{r}_{ik} \neq 0$ **do**
      $w_i = w_i - \tilde{r}_{ik}\tilde{l}_{jk}$
    end for
  end for
  for $k < j$ such that $\tilde{r}_{jk} \neq 0$ **do**
    for $i = j : n$ such that $\tilde{l}_{ik} \neq 0$ **do**
      $w_i = w_i - \tilde{l}_{ik}\tilde{r}_{jk}$
    end for
  end for
  Copy into $\widetilde{L}_{j:n,j}$ the $lsize + nz(A_{j:n,j})$ entries of w of largest absolute value
  Copy into $\widetilde{R}_{j+1:n,j}$ the $rsize$ entries of w that are the next largest in absolute value
  Scale $\tilde{l}_{jj} = (w_j)^{1/2}$, $\widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j}/\tilde{l}_{jj}$, $\widetilde{R}_{j+1:n,j} = \widetilde{R}_{j+1:n,j}/\tilde{l}_{jj}$
  Reset entries of w to zero.
end for
Optionally discard $\widetilde{R}$ ▷ $\widetilde{R}$ is often only used in the construction of $\widetilde{L}$

complete factorization. However, the memory can be reduced by introducing two drop tolerances so that only entries of absolute value at least τ_1 are kept in $\widetilde{L}$ and entries smaller than τ_2 are dropped from $\widetilde{R}$. The factorization is no longer guaranteed to be breakdown-free. Furthermore, the numbers of entries in $\widetilde{L}$ and $\widetilde{R}$ are not known a priori.

An alternative idea that limits both the number of entries in the incomplete factor and the intermediate memory is to fix the maximum number of entries in each column of $\widetilde{L}$ and $\widetilde{R}$. This is outlined in Algorithm 10.7. Here $lsize \geq 0$ and $rsize \geq 0$ are the maximum number of filled entries in each column of $\widetilde{L}$ and the maximum number of entries in each column of $\widetilde{R}$, respectively, and $nz(A_{j:n,j})$

denotes the number of entries in the lower triangular part of column j of A. The number of entries in $\widetilde{L}$ is less than $nz(A)+(n-1)lsize$ (where $nz(A)$ is the number of entries in the lower triangular part of A) and $\widetilde{R}$ has at most $(n-1)rsize$ entries. If the parameter $rsize$ is set to 0, then no intermediate memory is used but in general choosing $rsize > 0$ leads to the computed $\widetilde{L}$ being a higher quality preconditioner. In case of breakdown, the algorithm can incorporate the use of a global shift; see Algorithm 9.1.

10.7 Fixed-Point Iterations for Computing ILU Factorizations

The fixed-point ILU algorithm is fundamentally different from Gaussian elimination-based approaches. Given the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, the goal is to iteratively generate incomplete factors fulfilling the ILU property

$$(\widetilde{L}\widetilde{U})_{ij} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$$

(see Theorem 10.1). The idea is appealing because the entries of $\widetilde{L}$ and $\widetilde{U}$ can be computed iteratively in parallel using the constraints

$$\sum_{\substack{k=1 \\ (i,k),(k,j)\in\mathcal{S}\{\widetilde{L}+\widetilde{U}\}}}^{\min(i,j)} \tilde{l}_{ik}\,\tilde{u}_{kj} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\},$$

and the normalization $\tilde{l}_{ii} = 1$. Using the relations

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\,\tilde{u}_{kj}\right) / \tilde{u}_{jj}, \quad i > j, \tag{10.2}$$

$$\tilde{u}_{ij} = a_{ij} - \sum_{k=1}^{i-1} \tilde{l}_{ik}\,\tilde{u}_{kj}, \quad i \le j, \tag{10.3}$$

the approach can be formulated as a fixed-point iteration method of the form $w^{k+1} = f(w^k)$, $k = 0, 1, \ldots$, where w is a vector containing the unknowns $\tilde{l}_{ij}$ and $\tilde{u}_{ij}$. Each fixed-point iteration is called a **sweep**. Algorithm 10.8 outlines the method.

An important question is how to choose initial values for the factor entries. In some applications, a natural initial guess is available. For example, in time-dependent problems, the $\widetilde{L}$ and $\widetilde{U}$ computed in the previous time step may provide appropriate initial guesses for the current time step. In other cases, a possible strategy is to symmetrically scale A to have a unit diagonal and then take the initial $\widetilde{L}$

ALGORITHM 10.8 Fixed-point ILU factorization

Input: Matrix A, the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, and initial incomplete factors $\widetilde{L}$ and $\widetilde{U}$.
Output: Updated incomplete factors.

```
for (i, j) ∈ S{L̃ + Ũ} do
    Set l̃_ij and ũ_ij to the given initial values
end for
for sweep = 1, 2, ... do
    for (i, j) ∈ S{L̃ + Ũ} do
        if i > j then
            Compute l̃_ij using (10.2)
        else
            Compute ũ_ij using (10.3)
        end if
    end for
end for
```

and $\widetilde{U}$ to be the lower and upper parts of the scaled matrix, respectively. In practice, a few sweeps may be sufficient to generate preconditioners that are competitive in terms of quality to those generated via classical incomplete Gaussian elimination algorithms.

The following features differentiate the fixed-point ILU algorithm from classical methods and make it attractive for parallel computations on modern architectures.

- The algorithm is fine-grained, allowing for scaling to a very large number of processors, limited only by the number of nonzero entries in the target sparsity patterns.
- Preordering A is not needed to enhance parallelism, and thus orderings that improve the accuracy of the incomplete factorization can be used.
- The algorithm can utilize an initial guess for the ILU factorization.

To enhance the preconditioner quality, it is possible to interleave employing Algorithm 10.8 with a strategy that dynamically adapts $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ to the problem characteristics. In an iterative process based on highly parallel building blocks, this allows threshold-based ILU factorizations to be computed on parallel shared-memory architectures and enables the efficient use of streaming-based architectures such as GPUs.

10.8 Ordering in Incomplete Factorizations

Ordering algorithms designed for sparse direct solvers (see Chapter 8) can have a positive effect on the robustness and performance of preconditioned Krylov subspace methods. However, the best choice of ordering for an incomplete factorization preconditioner may not be the same as for a complete factorization, and although the effects of orderings and how much fill-in is allowed have been widely demonstrated, they are not yet fully understood.

When the natural (lexicographic) ordering is used, the incomplete triangular factors resulting from a no-fill ILU factorization can be highly ill-conditioned, even if the matrix A is well-conditioned. Allowing more fill-in in the factors, for example, using ILU(1) instead of ILU(0), may solve the problem, but it is not guaranteed. In some cases, preordering A can lead to more stable factors, and hence more effective preconditioners, but, again, this is not understood.

Minimum degree orderings (Section 8.1.2) are popular for direct methods, but for incomplete factorizations care is needed to ensure the dropping strategy is compatible with the ordering. This is because the rows (and columns) of the permuted matrix can have significantly different counts. In this situation, using memory-based dropping in which the maximum allowable number of filled entries in a row of $\widetilde{L}$ is the same for all rows may not be a good approach. An alternative strategy is to specify that the permitted fill-in is proportional to that of the complete factorization (which can be computed using Algorithm 4.3).

A level set ordering that reduces the bandwidth or profile of a matrix can be employed (Section 8.2). For complete factorizations, the fill-in in the factors can be much greater than for nested dissection or minimum degree, but for incomplete factorizations they can be highly effective. In particular, using an RCM ordering (Algorithm 8.3) is often found to lead to a higher quality preconditioner than using the natural ordering. RCM-based orderings are generally inexpensive to compute and can provide good reuse of computer caches.

Global orderings based on a divide-and-conquer approach and, in particular, nested dissection (Section 8.4) are important for complete factorizations. But such orderings cut local connections within the graph of A and, when used with incomplete factorizations, can lead to poor quality preconditioners. A global ordering that specifically targets incomplete factorizations is a **red–black** (or checker board) ordering. Consider the graph $\mathcal{G}(A)$ of an SPD matrix A that arises from a simple 5-point discretization of a PDE on a regular two-dimensional grid and colour its vertices using two colours so that no vertices of the same colour are incident to the same edge (see Figure 10.5). Because no red vertex is adjacent to any other red vertex, the red vertices are an independent set; similarly, the black vertices are an independent set. The red vertices can be processed in any order, provided they are all processed before any of the black vertices. This can make red–black orderings convenient for parallel implementations and is the main reason that they are often employed with stationary iterative methods.

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\
\begin{pmatrix}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
-1 & & & 4 & & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 3 \\ 5 \\ 7 \\ 9 \\ 2 \\ 4 \\ 6 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} 1 & 3 & 5 & 7 & 9 & 2 & 4 & 6 & 8 \end{array} \\
\begin{pmatrix}
4 & & & & & -1 & -1 & & \\
 & 4 & & & & -1 & -1 & & \\
 & & 4 & & & -1 & -1 & -1 & -1 \\
 & & & 4 & & & -1 & & -1 \\
 & & & & 4 & & & -1 & -1 \\
-1 & -1 & -1 & & & 4 & & & \\
-1 & & -1 & -1 & & & 4 & & \\
 & -1 & -1 & & -1 & & & 4 & \\
 & & -1 & -1 & -1 & & & & 4
\end{pmatrix}
\end{array}
$$

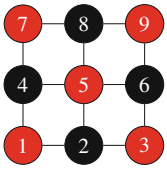


Figure 10.5 A model problem to illustrate a red–black ordering. The grid-based graph $\mathcal{G}(A)$ with coloured vertices is given together with the matrix A (left) and the symmetrically permuted matrix using the red–black ordering (right).

A bipartite graph is an undirected graph whose vertices can be partitioned into two disjoint sets such that each set is an independent set (Section 6.3.1). It follows that the red–black ordering exists if and only if $\mathcal{G}(A)$ is bipartite. The ordering is often generalized as follows. Start by finding a set of mutually non-adjacent vertices (that is, an independent set) and flag them as red vertices. After the elimination of the variables corresponding to the red vertices and employing a sparsification strategy, a Schur complement matrix is obtained. Proceed by finding a set of mutually non-adjacent vertices in this matrix, flag them as red vertices and continue recursively. This approach can lead to a significant decrease in the condition number of the preconditioned matrix. Another generalization for arbitrary graphs is to employ more colours (multicolouring). Again, the colouring can be exploited in parallel computations. For efficiency, load balancing of the coloured vertices needs to be considered. Because reordering the vertices can affect the convergence rate of an iterative solver, the potential gain in parallel performance at each iteration may be offset by a slower convergence rate.

10.9 Exploiting Block Structure

Blocking methods for complete factorizations can be adapted to incomplete factorizations. The aim is to speed up the computation of the factors and to obtain more effective preconditioners. In a block factorization, scalar operations of the form

$$\tilde{l}_{ik} = a_{ik}/\tilde{u}_{kk}$$

are replaced by matrix operations

$$\widetilde{L}_{ib,kb} = A_{ib,kb}\widetilde{U}_{kb,kb}^{-1},$$

and scalar multiplications of entries of the factors are replaced by matrix–matrix products. When dropping entries, instead of considering the absolute values, simple norms of the block entries (such as the one norm, max norm, or Frobenius norm) are used.

An incomplete factorization can start with the supernodal structure of the complete factors. If dropping is applied to individual columns, this structure is generally lost. To try and retain it, the dropping strategy can be modified either to drop the set of nonzeros of a row in the current supernode or to keep it. To obtain sufficiently sparse incomplete factors, it may be necessary to subdivide each supernode, allowing greater flexibility on how many rows are dropped. It is also possible to relax blocking operations in such a way that the supernodes are not exact but are allowed to incur some fill-in.

10.10 Notes and References

Sparsity structure was the main ingredient of the first algebraic preconditioners that were developed in the late 1950s. The nonzero structure represented the stencils resulting from the discretization of PDEs on structured grids. The earliest contribution is Buleev (1959), and this was later generalized to three-dimensional problems. An independent derivation and its interpretation as an incomplete factorization for a sparse matrix coming from a simple 5-point stencil is given in Varga (1960); other early work is by Baker & Oliphant (1960). For an overview of early contributions and the motivations behind incomplete factorizations, see Il′in (1992); we also refer to the survey of Chan & van der Vorst (1997).

Important breakthroughs in the use of preconditioning using incomplete factorizations for practical problems came in two key papers. The first by Meijerink & van der Vorst (1977) recognized the importance of preconditioning for the conjugate gradient method. In the second, Kershaw (1978) proposed locally replacing pivots by a small positive number to prevent breakdown of the factorization. This paved the way for incomplete factorizations in which dropping is based solely on the size of the computed entries and which were introduced even earlier by Tuff & Jennings (1973).

The Crout incomplete LU factorization outlined in Algorithm 10.1 was implemented in a successful code for symmetric problems by Lin & Moré (1999), building on earlier ideas of Jones & Plassmann (1995) and Eisenstat et al. (1982) (see also Li et al., 2003 for later contributions to this approach). Algorithm 10.2

with a sparsification strategy that uses both a drop tolerance and a limit on the number of entries in each column of the incomplete factors was published in Saad (1994a) as the dual threshold ILUT method. For general nonsymmetric matrices, ILUT has proved very popular and has been developed further (see, for example, MacLachlan et al., 2012). But because it is based on the row factorization, it ignores symmetry in A and, if A is symmetric, the computed sparsity patterns of L and U^T are normally different. In this case, a Crout incomplete factorization may be preferable. The hierarchy of sparsity structures based on the concept of levels is introduced in Watts-III (1981). The initial work has since been significantly improved, notably for parallel implementations by Hysom & Pothen (2002). The Euclid library is a scalable implementation of a parallel level-based ILU algorithm that is available as part of the *hypre* library of linear solvers (see Falgout et al., 2006, 2021). Scalable means that the incomplete factorization and triangular solve timings remain nearly constant when the problem size n is scaled in proportion to the number of processors. Another parallel level-based ILU preconditioner that uses an adaptive block implementation is proposed in Hénon et al. (2008).

The modified incomplete factorizations of Section 10.4 are described in Saad (2003b). A proof of Theorem 10.3 can be found in Bern et al. (2006), but it is also of interest to follow earlier work on asymptotic bounds for the condition number of matrices preconditioned by modified incomplete factorizations given in Dupont et al. (1968), Axelsson (1972), and Gustafsson (1978), while an elegant description is in Meurant (1999).

Incomplete factorizations with dynamic compensation originally introduced by Ajiz & Jennings (1984) have been routinely employed in practice. However, memory-limited approaches based on relaxing the strategy of Tismenetsky (1991) often lead to more efficient preconditioners; see Kaporin (1998) for a row-based construction that has recently been used by Konshin et al. (2017, 2019) to solve challenging practical problems. Scott & Tůma (2014b) present a Crout construction of a sophisticated memory-limited incomplete factorization and provide a robust implementation for SPD systems as the package `HSL_MI28` within the HSL mathematical software library (Scott & Tůma, 2014a); a variant for symmetric saddle point systems is also included in HSL.

Using fixed-point iterations for the parallel computation of incomplete factorizations is a relatively new idea that was proposed and analysed by Chow & Patel (2015). Interleaving a fixed-point iteration with a procedure that adjusts the sparsity pattern is proposed by Anzt et al. (2018). Other attempts to compute and use ILU preconditioners in parallel that build on the software package ILUPACK (Bollhöfer et al., 2012) are presented in Aliaga et al. (2016, 2019). A different approach to parallelize incomplete factorizations by relaxing supernodes is given by Gupta & George (2010).

Significant attention has been devoted to using orderings of A to try and improve the quality of incomplete factorization preconditioners. An early and often quoted comparison of reorderings for SPD problems is by Duff & Meurant (1989). For more general matrices, see Benzi et al. (1999), Oliker et al. (2002), or Osei-Kuffuor

et al. (2015). Saad (1996a) and Saad & Zhang (1999) generalize red–black orderings and consider blocks and/or more colours; also of interest are the papers of Saad & Suchomel (2002), Li et al. (2003), and Carpentieri et al. (2014)).

Chapter 11
Sparse Approximate Inverse Preconditioners

While it is recognized that preconditioning the system often improves the convergence of a particular method, this is not always so. In particular, a successful preconditioner for one class of problems may prove ineffective on another class. – Gould & Scott (1998).

There is, of course, no such concept as a best preconditioner ... However, every practitioner knows when they have a good preconditioner which enables feasible computation and solution of problems. In this sense, preconditioning will always be an art rather than a science. – Wathen (2015).

Consider a preconditioner M based on an incomplete LU (or Cholesky) factorization of a matrix A. M^{-1}, which represents an approximation of A^{-1}, is applied by performing forward and back substitution steps; this can present a computational bottleneck. An alternative strategy is to directly approximate A^{-1} by explicitly computing M^{-1}. Preconditioners of this kind are called **sparse approximate inverse** preconditioners. They constitute an important class of algebraic preconditioners that are complementary to the approaches discussed in the previous chapter. They can be attractive because when used with an iterative solver, they can require fewer iterations than standard incomplete factorization preconditioners that contain a similar number of entries while offering significantly greater potential for parallel computations.

From Theorem 7.3, the sparsity pattern of the inverse of an irreducible matrix A is dense, even when A is sparse. Therefore, if A is large, the exact computation of its inverse is not an option, and aggressive dropping is needed to obtain a sufficiently sparse approximation to A^{-1} that can be used as a preconditioner. Fortunately, for a wide class of problems of practical interest, many of the entries of A^{-1} are small in absolute value, so that approximating the inverse with a sparse M^{-1} may be feasible, although capturing the large (important) values of A^{-1} is a nontrivial task. Importantly, the computed M^{-1} can have nonzeros at positions that cannot be obtained by either a complete or an incomplete factorization, and this can be

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_11

beneficial. Furthermore, although A^{-1} is fully dense, the following result shows this is not the case for the factors of factorized inverses.

Theorem 11.1 (Bridson & Tang 1999; Benzi & Tůma 2000) *Assume the matrix A is SPD, and let L be its Cholesky factor. Then $\mathcal{S}\{L^{-1}\}$ is the union of all entries (i, j) such that i is an ancestor of j in the elimination tree $\mathcal{T}(A)$.*

A consequence of this result is that L^{-1} need not be fully dense. Considering this implication algorithmically, if A is SPD, it may be advantageous to preorder A to limit the number of ancestors that the vertices in $\mathcal{T}(A)$ have. For example, nested dissection may be applied to $\mathcal{S}\{A\}$ (Section 8.4). If $\mathcal{S}\{A\}$ is nonsymmetric, then it may be possible to reduce fill-in in the factors of A^{-1} by applying nested dissection to $\mathcal{S}\{A + A^T\}$.

11.1 Basic Approaches

An obvious way to obtain an approximate inverse of A in factorized form is to compute an incomplete LU factorization of A and then perform an approximate inversion of the incomplete factors. For example, if incomplete factors $\widetilde{L}$ and $\widetilde{U}$ are available, approximate inverse factors can be found by solving the $2n$ triangular linear systems

$$\widetilde{L}x_i = e_i, \quad \widetilde{U}y_i = e_i, \quad 1 \le i \le n,$$

where e_i is the i-th column of the identity matrix. These systems can all be solved independently, and hence, there is the potential for significant parallelism. To reduce costs and to preserve sparsity in the approximate inverse factors, they may not need to be solved accurately. A disadvantage is that the computation of the preconditioner involves two levels of incompleteness, and because information from the incomplete factorization of A is passed into the second step, the loss of information can be excessive.

Another straightforward approach is based on bordering. Let A_j denote the principal leading submatrix of A of order j ($A_j = A_{1:j,1:j}$), and assume that its inverse factorization

$$A_j^{-1} = W_j D_j^{-1} Z_j^T$$

is known. Here W_j and Z_j are unit upper triangular matrices, and D_j is a diagonal matrix. Consider the following scheme:

$$\begin{pmatrix} Z_j^T & 0 \\ z_{j+1}^T & 1 \end{pmatrix} \begin{pmatrix} A_j & A_{1:j,j+1} \\ A_{j+1,1:j} & a_{j+1,j+1} \end{pmatrix} \begin{pmatrix} W_j & w_{j+1} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} D_j & 0 \\ 0 & d_{j+1,j+1} \end{pmatrix},$$

where for $1 \leq j < n$

$$w_{j+1} = -W_j D_j^{-1} Z_j^T A_{1:j,j+1},$$

$$z_{j+1} = -Z_j D_j^{-1} W_j^T A_{j+1,1:j}^T,$$

$$d_{j+1,j+1} = a_{j+1,j+1} + z_{j+1}^T A_j w_{j+1} + A_{j+1,1:j} w_{j+1} + z_{j+1}^T A_{1:j,j+1}.$$

Starting from $j = 1$, this suggests a procedure for computing the inverse factors of A. Sparsity can be preserved by dropping some entries from the vectors w_{j+1} and z_{j+1} once they have been computed. Sparsity and the quality of the preconditioner can be influenced by preordering A.

If A is symmetric, $W = Z$ and the required work is halved. Furthermore, if A is SPD, then it can be shown that, in exact arithmetic, $d_{jj} > 0$ for all j and the process does not break down. In the general case, diagonal modifications may be required, which can limit the effectiveness of the resulting preconditioner.

Observe that the computations of Z and W are tightly coupled, restricting the potential to exploit parallelism. At each step j, besides a matrix–vector product with A_j, four sparse matrix–vector products involving W_j, Z_j and their transposes are needed; these account for most of the work. The implementation is simplified if access to the triangular factors is available by columns as well as by rows.

11.2 Approximate Inverses Based on Frobenius Norm Minimization

It is clear from the above discussion that alternative techniques for constructing sparse approximate inverse preconditioners are needed. We start by looking at schemes based on Frobenius norm minimization. Historically, these were the first to be proposed and offer the greatest potential for parallelism because both the construction of the preconditioner and its subsequent application can be performed in parallel.

11.2.1 SPAI Preconditioner

To describe the sparse approximate inverse (SPAI) preconditioner, it is convenient to use the notation $K = M^{-1}$. The basic idea is to compute $K \approx A^{-1}$ with its columns denoted by k_j as the solution of the problem of minimizing

$$\|I - AM^{-1}\|_F^2 = \|I - AK\|_F^2 = \sum_{j=1}^{n} \|e_j - Ak_j\|_2^2, \tag{11.1}$$

over all K with pattern $\mathcal{S}$. This produces a right approximate inverse. A left approximate inverse can be computed by solving a minimization problem for $\|I - KA\|_F = \|I - A^T K^T\|_F$. This amounts to computing a right approximate inverse for A^T and taking the transpose of the resulting matrix. For nonsymmetric matrices, the distinction between left and right approximate inverses can be important. Indeed, there are situations where it is difficult to compute a good right approximate inverse but easy to find a good left approximate inverse (or vice versa). In the following discussion, we assume that a right approximate inverse is being computed.

The Frobenius norm is generally used because the minimization problem then reduces to least squares problems for the columns of K that can be computed independently and, if required, in parallel. Further, these least squares problems are all of small dimension when $\mathcal{S}$ is chosen to ensure K is sparse. Let $\mathcal{J} = \{i \mid k_j(i) \neq 0\}$ be the set of indices of the nonzero entries in column k_j. The set of indices of rows of A that can affect a product with column k_j is $\mathcal{I} = \{m \mid A_{m,\mathcal{J}} \neq 0\}$. Let $|\mathcal{I}|$ and $|\mathcal{J}|$ denote the number of entries in $\mathcal{I}$ and $\mathcal{J}$, respectively, and let $\widehat{e}_j = e_j(\mathcal{I})$ be the vector of length $|\mathcal{I}|$ that is obtained by taking the entries of e_j with row indices belonging to $\mathcal{I}$. To solve (11.1) for k_j, construct the $|\mathcal{I}| \times |\mathcal{J}|$ matrix $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and solve the small unconstrained least squares problem

$$\min_{\widehat{k}_j} \|\widehat{e}_j - \widehat{A}\,\widehat{k}_j\|_2^2. \tag{11.2}$$

This can be done using a dense QR factorization of $\widehat{A}$. Extending $\widehat{k}_j$ to have length n by setting entries that are not in $\mathcal{J}$ to zero gives k_j.

A straightforward way to construct $\mathcal{S}$ that does not depend on a sophisticated initial choice (but could, for example, be the identity or be equal to $\mathcal{S}\{A\}$) proceeds as follows. Starting with a chosen column sparsity pattern $\mathcal{J}$ for k_j, construct $\widehat{A}$, solve (11.2) for $\widehat{k}_j$, set $k_j(\mathcal{J}) = \widehat{k}_j$, and define the residual vector

$$r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j.$$

If $\|r_j\|_2 \neq 0$, then k_j is not equal to the j-th column of A^{-1}, and a better approximation can be derived by augmenting $\mathcal{J}$. To do this, let $\mathcal{L} = \{l \mid r_j(l) \neq 0\}$ and define

$$\widetilde{\mathcal{J}} = \{i \mid A_{\mathcal{L},i} \neq 0\} \setminus \mathcal{J}. \tag{11.3}$$

These are candidate indices that can be added to $\mathcal{J}$, but as there may be many of them, they need to be chosen to most effectively reduce $\|r_j\|_2$. A possible heuristic is to solve for each $i \in \widetilde{\mathcal{J}}$ the minimization problem

$$\min_{\mu_i} ||r_j - \mu_i A e_i\|_2^2.$$

This has the solution $\mu_i = r_j^T Ae_i / \|Ae_i\|_2^2$ with residual $\|r_j\|^2 - (r_j^T Ae_i)^2 / \|Ae_i\|_2^2$. Indices $i \in \widetilde{\mathcal{J}}$ for which this is small are appended to $\mathcal{J}$. The process can be repeated until either the required accuracy is attained or the maximum number of allowed entries in $\mathcal{J}$ is reached.

Solving the unconstrained least squares problem (11.2) after extending $\widehat{A}$ to $A_{\mathcal{I}\cup\mathcal{I}',\,\mathcal{J}\cup\mathcal{J}'}$ is typically performed using updating. Assume the QR factorization of $\widehat{A}$ is

$$\widehat{A} = A_{\mathcal{I},\mathcal{J}} = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix},$$

where Q_1 is $|\mathcal{I}| \times |\mathcal{J}|$. Here Q is an orthogonal matrix and R is an upper triangular matrix. The QR factorization of the extended matrix is

$$A_{\mathcal{I}\cup\mathcal{I}',\,\mathcal{J}\cup\mathcal{J}'} = \begin{pmatrix} \widehat{A} & A_{\mathcal{I},\mathcal{J}'} \\ & A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix} = \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} R & Q_1^T A_{\mathcal{I},\mathcal{J}'} \\ & Q_2^T A_{\mathcal{I},\mathcal{J}'} \\ & A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix}$$

$$= \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} I & \\ & Q' \end{pmatrix} \begin{pmatrix} R & Q_1^T A_{\mathcal{I},\mathcal{J}'} \\ & R' \\ & 0 \end{pmatrix},$$

where Q' and R' are from the QR factorization of the $(|\mathcal{I}'| + |\mathcal{I}| - |\mathcal{J}|) \times |\mathcal{J}'|$ matrix

$$\begin{pmatrix} Q_2^T A_{\mathcal{I},\mathcal{J}'} \\ A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix}.$$

Factorizing this matrix and updating the trailing QR factorization to get the new $\widehat{k}_j$ is much more efficient than computing the QR factorization of the extended matrix from scratch.

Construction of the SPAI preconditioner is summarized in Algorithm 11.1. The maximum number of entries nz_j that is permitted in k_j must be at least as large as the number of entries in the initial sparsity pattern $\mathcal{J}_j$. Updating can be used to compute a new $\widehat{k}_j$ for each pass through the while loop; the number of passes is typically small (for example, if a good initial sparsity pattern is available, a single pass may be sufficient).

The example in Figure 11.1 illustrates Algorithm 11.1. Starting with a tridiagonal matrix, it considers the computation of the first column k_1 of the inverse matrix K. The algorithm starts with $\mathcal{J}_1 = \{1, 2\}$.

When A is symmetric, there is no guarantee that the computed K will be symmetric. One possibility is to use $(K + K^T)/2$ to approximate A^{-1}. The SPAI preconditioner is not sensitive to the ordering of A. This has the advantage that A can be partitioned and preordered in whatever way is convenient, for instance,

ALGORITHM 11.1 SPAI preconditioner (right-looking approach)
Input: Nonsymmetric matrix A, a convergence tolerance $\eta > 0$, an initial sparsity pattern $\mathcal{J}_j$ and the maximum number nz_j of permitted entries for column j of K $(1 \le j \le n)$.
Output: $K \approx A^{-1}$ with columns k_j $(1 \le j \le n)$.

1: **for** $j = 1 : n$ **do** ▷ The columns may be computed in parallel
2: Set $\mathcal{J} = \mathcal{J}_j$ and $\mathcal{I} = \{m \mid A(m, \mathcal{J}) \neq 0\}$, $\|r_j\|_2 = \infty$
3: Construct $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and solve (11.2) for $\widehat{k}_j$
4: $r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j$
5: **while** $|\mathcal{J}| < nz_j$ and $\|r_j\|_2 > \eta$ **do**
6: Construct $\widetilde{\mathcal{J}}$ given by (11.3) ▷ $\widetilde{\mathcal{J}}$ is the candidate set
7: Determine new indices $\mathcal{J}' \subset \widetilde{\mathcal{J}}$ to add to $\mathcal{J}$
8: $\mathcal{I}' = \{m \mid A_{m,\mathcal{J}'} \neq 0\} \setminus \mathcal{I}$
9: $\mathcal{I} = \mathcal{I} \cup \mathcal{I}'$ and $\mathcal{J} = \mathcal{J} \cup \mathcal{J}'$ ▷ Augment the sparsity pattern
10: Construct new $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and new $\widehat{k}_j$ ▷ Update the QR factorization
11: $r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j$
12: **end while**
13: $k_j(\mathcal{J}) = \widehat{k}_j$ ▷ Extend $\widehat{k}_j$ to k_j by setting entries not in $\mathcal{J}$ to zero.
14: **end for**

$$A = \begin{pmatrix} 10 & -2 & & & \\ -1 & 10 & -2 & & \\ & -1 & 10 & -2 & \\ & & -1 & 10 & -2 \\ & & & -1 & 10 \end{pmatrix}, \widehat{A} = \begin{pmatrix} 10 & -2 \\ -1 & 10 \\ & -1 \end{pmatrix}, \widehat{k}_1 = \begin{pmatrix} 0.1020 \\ 0.0101 \end{pmatrix}, r_1 = \begin{pmatrix} 1.00 \times 10^{-4} \\ 1.00 \times 10^{-3} \\ 1.01 \times 10^{-2} \\ 0 \\ 0 \end{pmatrix}.$$

$$\widehat{A} = \begin{pmatrix} 10 & -2 & \\ -1 & 10 & -2 \\ & -1 & 10 \\ & & -1 \end{pmatrix}, \widehat{k}_1 = \begin{pmatrix} 0.1021 \\ 0.0104 \\ 0.0010 \end{pmatrix}, r_1 = \begin{pmatrix} 1.0 \times 10^{-5} \\ 1.1 \times 10^{-4} \\ 1.1 \times 10^{-3} \\ 1.0 \times 10^{-2} \\ 0 \end{pmatrix}, k_1 = \begin{pmatrix} 0.1021 \\ 0.0104 \\ 0.0010 \\ 0 \\ 0 \end{pmatrix}.$$

Figure 11.1 An illustration of computing the first column of a sparse approximate inverse using the SPAI algorithm with $nz_1 = 3$. On the top line is the initial tridiagonal matrix A followed by the matrix $\widehat{A}$ and the vectors $\widehat{k}_1$ and r_1 on the first loop of Algorithm 11.1. The bottom line presents the updated matrix $\hat{A}$ that is obtained on the second loop by adding the third row and column of A and the corresponding vectors $\widehat{k}_1$ and r_1 and, finally, k_1. Here the numerical values have been appropriately rounded.

to better suit the needs of a distributed implementation, without worrying about the impact on the subsequent convergence rate of the solver. The disadvantage is that orderings cannot be used to reduce fill-in and/or improve the quality of this

approximate inverse. For instance, if A^{-1} has no small entries, SPAI will not find a sparse K, and because the inverse of a permutation of A is just a permutation of A^{-1}, no permutation of A will change this.

11.2.2 FSAI Preconditioner: SPD Case

We next consider a class of preconditioners based on an incomplete inverse factorization of A^{-1}. The factorized sparse approximate inverse (FSAI) preconditioner for an SPD matrix A is defined as the product

$$M^{-1} = G^T G,$$

where the sparse lower triangular matrix G is an approximation of the inverse of the (complete) Cholesky factor L of A. Theoretically, a FSAI preconditioner is computed by choosing a lower triangular sparsity pattern $\mathcal{S}_L$ and minimizing

$$\|I - GL\|_F^2 = tr\left[(I - GL)^T (I - GL)\right], \tag{11.4}$$

over all G with sparsity pattern $\mathcal{S}_L$. Here tr denotes the matrix trace operator (that is, the sum of the entries on the diagonal). The computation of G can be performed without knowing L explicitly. Differentiating (11.4) with respect to the entries of G and setting to zero yields

$$(GLL^T)_{ij} = (GA)_{ij} = (L^T)_{ij} \quad \text{for all} \quad (i, j) \in \mathcal{S}_L. \tag{11.5}$$

Because L^T is an upper triangular matrix while $\mathcal{S}_L$ is a lower triangular pattern, the matrix equation (11.5) can be rewritten as

$$(GA)_{ij} = \begin{cases} 0 & i \neq j, \quad (i, j) \in \mathcal{S}_L, \\ l_{ii} & i = j. \end{cases} \tag{11.6}$$

G is not available from (11.6) because L is unknown. Instead, $\overline{G}$ is computed such that

$$(\overline{G}A)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i, j) \in \mathcal{S}_L, \tag{11.7}$$

where $\delta_{i,j}$ is the Kronecker delta function ($\delta_{i,j} = 1$ if $i = j$ and is equal to 0, otherwise). The FSAI factor G is then obtained by setting

$$G = D\overline{G},$$

where D is a diagonal scaling matrix. An appropriate choice for D is

$$D = [diag(\overline{G})]^{-1/2}, \tag{11.8}$$

so that

$$(GAG^T)_{ii} = 1, \quad 1 \le i \le n.$$

The following result shows that the FSAI preconditioner exists for any nonzero pattern $\mathcal{S}_L$ that includes the main diagonal of A.

Theorem 11.2 (Kolotilina & Yeremin 1993) *Assume A is SPD. If the lower triangular sparsity pattern $\mathcal{S}_L$ includes all diagonal positions, then G exists and is unique.*

Proof Set $\mathcal{I}_i = \{j \mid (i, j) \in \mathcal{S}_L\}$, and let $A_{\mathcal{I}_i, \mathcal{I}_i}$ denote the submatrix of order $nz_i = |\mathcal{I}_i|$ of entries a_{kl} such that $k, l \in \mathcal{I}_i$. Let $\bar{g}_i$ and g_i be dense vectors containing the nonzero coefficients in row i of $\overline{G}$ and G, respectively. Using this notation, solving (11.7) decouples into solving n independent SPD linear systems

$$A_{\mathcal{I}_i, \mathcal{I}_i}\, \bar{g}_i = e_{nz_i}, \quad 1 \le i \le n,$$

where the unit vectors are of length nz_i. Moreover,

$$(\overline{G}A\overline{G}^T)_{ii} = \sum_{j \in \mathcal{I}_i} \delta_{i,j} \overline{G}_{ij} = \overline{G}_{ii} = (A^{-1}_{\mathcal{I}_i, \mathcal{I}_i})_{ii}.$$

This implies that the diagonal entries of D given by (11.8) are nonzero. Consequently, the computed rows of G exist and provide a unique solution. □

The procedure for computing a FSAI preconditioner is summarized in Algorithm 11.2. The computation of each row of G can be performed independently; thus, the algorithm is inherently parallel, but each application of the preconditioner requires the solution of triangular systems.

The following theorem states that G computed using Algorithm 11.2 is in some sense optimal.

Theorem 11.3 (Kolotilina et al. 2000) *Let L be the Cholesky factor of the SPD matrix A. Given a lower triangular sparsity pattern $\mathcal{S}_L$ that includes all diagonal positions, let G be the FSAI preconditioner computed using Algorithm 11.2. Then any lower triangular matrix G_1 with its sparsity pattern contained in $\mathcal{S}_L$ and $(G_1 A G_1^T)_{ii} = 1$ $(1 \le i \le n)$ satisfies*

$$||I - GL||_F \le ||I - G_1 L||_F.$$

ALGORITHM 11.2 FSAI preconditioner
Input: SPD matrix A and a lower triangular sparsity pattern $\mathcal{S}_L$ that includes all diagonal positions.
Output: Lower triangular matrix G such that $A^{-1} \approx GG^T$.

1: **for** $i = 1 : n$ **do**
2: Construct $\mathcal{I}_i = \{j \mid (i, j) \in \mathcal{S}_L\}$, $A_{\mathcal{I}_i,\mathcal{I}_i}$ and set $nz_i = |\mathcal{I}_i|$
3: Solve $A_{\mathcal{I}_i,\mathcal{I}_i}\, \bar{g}_i = e_{nz_i}$
4: Scale $g_i = d_{ii}\bar{g}_i$ with $d_{ii} = (\bar{g}_{i,nz_i})^{-1/2}$ ▷ $\bar{g}_{i,nz_i}$ is the last component of $\bar{g}_i$
5: Extend g_i to the row $G_{i,1:i}$ by setting entries that are not in $\mathcal{I}_i$ to zero
6: **end for**

The performance of the FSAI preconditioner is highly dependent on the choice of $\mathcal{S}_L$. If entries are added to the pattern, then, as the following result shows, the preconditioner is more accurate, but it is also more expensive.

Theorem 11.4 (Kolotilina et al. 2000) *Let L be the Cholesky factor of the SPD matrix A. Given the lower triangular sparsity patterns $\mathcal{S}_{L1}$ and $\mathcal{S}_{L2}$ that include all diagonal positions, let the corresponding FSAI preconditioners computed using Algorithm 11.2 be G_1 and G_2, respectively. If $\mathcal{S}_{L1} \subseteq \mathcal{S}_{L2}$, then*

$$||I - G_2 L||_F \leq ||I - G_1 L||_F.$$

11.2.3 FSAI Preconditioner: General Case

The FSAI algorithm can be extended to a general matrix A. Two input sparsity patterns are required: a lower triangular sparsity pattern $\mathcal{S}_L$ and an upper triangular sparsity pattern $\mathcal{S}_U$, both containing the diagonal positions. First, lower and upper triangular matrices $\overline{G}_L$ and $\overline{G}_U$ are computed such that

$$(\overline{G}_L A)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i, j) \in \mathcal{S}_L,$$

$$(A\overline{G}_U)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i, j) \in \mathcal{S}_U.$$

Then D is obtained as the inverse of the diagonal of the matrix $\overline{G}_L A \overline{G}_U$, and the final nonsymmetric FSAI factors are given by $G_L = \overline{G}_L$ and $G_U = \overline{G}_U D$. The computation of the two approximate factors can be performed independently.

This generalization is well defined if, for example, A is nonsymmetric positive definite. There is also theory that extends existence to special classes of matrices, including M- and H-matrices. In more general cases, solutions to the reduced

systems may not exist, and modifications (such as perturbing the diagonal entries) are needed to circumvent breakdown.

11.2.4 Determining a Good Sparsity Pattern

The role of the input pattern is to preserve sparsity by filtering out entries of A^{-1} that contribute little to the quality of the preconditioner. For instance, it might be appropriate to ignore entries with a small absolute value, while retaining the largest ones. Unfortunately, the locations of large entries in A^{-1} are generally unknown, and this makes the a priori sparsity choice difficult. A possible exception is when A is a banded SPD matrix. In this case, the entries of A^{-1} are bounded in an exponentially decaying manner along each row or column. Specifically, there exist $0 < \rho < 1$ and a constant c such that for all i, j

$$|(A^{-1})_{ij}| \leq c\rho^{|i-j|}.$$

The scalars ρ and c depend on the bandwidth and the condition number of A. For matrices with a large bandwidth and/or a high condition number, c can be very large and ρ close to one, indicating extremely slow decay. However, if the entries of A^{-1} can be shown to decay rapidly, then a banded M^{-1} should be a good approximation to A^{-1}. In this case, $\mathcal{S}_L$ can be chosen to correspond to a matrix with a prescribed bandwidth.

A common choice for a general A is $\mathcal{S}_L + \mathcal{S}_U = \mathcal{S}\{A\}$, motivated by the empirical observation that entries in A^{-1} that correspond to nonzero positions in A tend to be relatively large. However, this simple choice is not robust because entries of A^{-1} that lie outside $\mathcal{S}\{A\}$ can also be large. An alternative strategy based on the Neumann series expansion of A^{-1} is to use the pattern of a small power of A, i.e., $\mathcal{S}\{A^2\}$ or $\mathcal{S}\{A^3\}$. By starting from the lower and upper triangular parts of A, this approach can be used to determine candidates $\mathcal{S}_L$ and $\mathcal{S}_U$. While approximate inverses based on higher powers of A are often better than those corresponding to A, there is no guarantee they will result in good preconditioners. Furthermore, even small powers of A can be very dense, thus slowing down the construction and application of the preconditioner. A possible remedy is to use the power of a sparsified A. Alternatively, the pattern can be chosen dynamically by retaining the largest terms in each row of the preconditioner as it is computed, which is what the SPAI algorithm does. Another possibility is to implicitly determine $\mathcal{S}_L + \mathcal{S}_U$ as follows. Starting with a simple sparsity pattern, compute the corresponding FSAI preconditioner G_1. Then choose a pattern based on $G_1 A G_1^T$ and apply the FSAI algorithm to $G_1 A G_1^T$ to obtain G_2. Finally, set the preconditioner to $G_2 G_1$. Despite running the FSAI algorithm twice, this approach can be worthwhile. Unfortunately, the choice of the best technique for generating a FSAI preconditioner and its sparsity pattern is highly problem dependent.

11.3 Factorized Approximate Inverses Based on Incomplete Conjugation

An alternative way to obtain a factorized approximate inverse is based on incomplete conjugation (A-orthogonalization) in the SPD case and on incomplete A-biconjugation in the general case. For SPD matrices, the approach represents an approximate Gram–Schmidt orthogonalization that uses the A-inner product $\langle ., . \rangle_A$. An important attraction is that the sparsity patterns of the approximate inverse factors need not be specified in advance; instead, they are determined dynamically as the preconditioner is computed.

11.3.1 AINV Preconditioner: SPD Case

When A is an SPD matrix, the AINV preconditioner is defined by an approximate inverse factorization of the form

$$A^{-1} \approx M^{-1} = ZD^{-1}Z^T,$$

where the matrix Z is unit upper triangular and D is a diagonal matrix with positive entries. The factor Z is a sparse approximation of the inverse of the L^T factor in the square root-free factorization of A. Z and D are computed directly from A using an incomplete A-orthogonalization process applied to the columns of the identity matrix. If entries are not dropped, then a complete factorization of A^{-1} is computed and Z is significantly denser than L^T. To preserve sparsity, at each step of the computation, entries are discarded (for example, using a prescribed threshold, or according to the positions of the entries, or by retaining a chosen number of the largest entries in each column), resulting in an approximate factorization of A^{-1}.

There are several variants. Algorithms 11.3 and 11.4 outline left-looking and right-looking approaches, respectively. Practical implementations need to employ sparse matrix techniques. The left-looking scheme computes the j-th column z_j of Z as a sparse linear combination of the previous columns $z_1, \ldots, z_{j-1}$. The key is determining which multipliers (the α's in Steps 4 and 5 of the two algorithms, respectively) are nonzero and need to be computed. This can be achieved very efficiently by having access to both the rows and columns of A (although the algorithm does not require that A is explicitly stored—only the capability of forming inner products involving the rows of A is required). For the right-looking approach, the crucial part for each j is the update of the sparse submatrix of Z composed of the columns $j + 1$ to n that are not yet fully computed. Here, only one row of A is used in the outer loop of the algorithm. Therefore, A can be generated on-the-fly by rows. The DS format can be used to store the partially computed Z (Section 1.3.2). As with complete factorizations, the efficiency of the computation and application of AINV preconditioners can benefit from incorporating blocking.

ALGORITHM 11.3 AINV preconditioner (left-looking approach)
Input: SPD matrix A and sparsifying rule.
Output: $A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.

1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ ▷ Initialise Z to hold the columns of the identity matrix
2: **for** $j = 1 : n$ **do**
3: **for** $k = 1 : j - 1$ **do**
4: $\alpha = A_{k,1:n}\, z_j^{(k-1)}/d_{kk}$
5: $z_j^{(k)} = z_j^{(k-1)} - \alpha z_k^{(k-1)}$
6: Sparsify $z_j^{(k)}$ ▷ Drop entries from $z_j^{(k)}$
7: **end for**
8: $d_{jj} = A_{j,1:n}\, z_j^{(j-1)}$
9: **end for**
10: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$

11.3.2 AINV Preconditioner: General Case

In the general case, the AINV preconditioner is given by an approximate inverse factorization of the form

$$A^{-1} \approx M^{-1} = WD^{-1}Z^T,$$

where Z and W are unit upper triangular matrices and D is a diagonal matrix. Z and W are sparse approximations of the inverses of the L^T and U factors in the LDU factorization of A, respectively. Starting from the columns of the identity matrix, A-biconjugation is used to compute the factors. Algorithm 11.5 outlines the right-looking approach. Note it offers two possibilities for computing the entries d_{jj} of D that are equivalent in exact arithmetic if the factorization is breakdown-free. The left-looking variant given in Algorithm 11.3 can be generalized in a similar way.

Figure 11.2 illustrates the sparsity patterns of the AINV factors for a matrix arising in circuit simulation. $\mathcal{S}\{A\}$ is symmetric, but the values of the entries of A are nonsymmetric. The sparsity pattern $\mathcal{S}\{W + Z^T\}$ is given, where W and Z are computed using Algorithm 11.5 with sparsification based on a dropping tolerance of 0.5. Also given are the patterns $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and $\mathcal{S}\{\widetilde{L}^{-1} + \widetilde{U}^{-1}\}$ for the incomplete factors $\widetilde{L}$ and $\widetilde{U}$ computed using Algorithm 10.2 (see Section 10.2) with a dropping tolerance of 0.1 and at most 10 entries in each row of $\widetilde{L} + \widetilde{U}$. Note that this dual dropping strategy is one of the most popular ways of employing

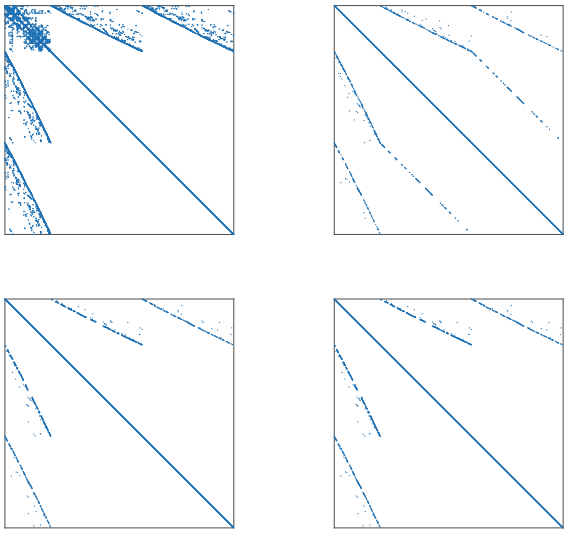


Figure 11.2 An example to illustrate the difference between the sparsity patterns of the AINV factors and those of the inverse of the ILU factors. The sparsity pattern $\mathcal{S}\{A\}$ of the matrix A is given (top left) together with the patterns of the factorized approximate inverse factors $\mathcal{S}\{W + Z^T\}$ (top right), the ILU factors $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ (bottom left), and their inverses $\mathcal{S}\{\widetilde{L}^{-1} + \widetilde{U}^{-1}\}$ (bottom right).

Algorithm 10.2; it is often denoted as ILUT(p, τ), where p is the maximum number of entries allowed in each row and τ is the dropping tolerance. In this example, the parameters were chosen so that the number of entries in both $W + Z^T$ and $\widetilde{L} + \widetilde{U}$ is approximately equal, but the resulting sparsity patterns are clearly different. In particular, potentially important information is lost from $\mathcal{S}\{\widetilde{L}^{-1} + \widetilde{U}^{-1}\}$.

11.3.3 SAINV: Stabilization of the AINV Method

The following result is analogous to Theorem 9.4.

Theorem 11.5 (Benzi et al. 1996) *If A is a nonsingular M- or H-matrix, then the AINV factorization of A does not break down.*

For more general matrices, breakdown can happen because of the occurrence of a zero d_{jj} or, in the SPD case, negative d_{jj}. In practice, exact zeros are unlikely but very small d_{jj} can occur (near breakdown), which may lead to uncontrolled growth in the size of entries in the incomplete factors and, because such entries are not dropped when using a threshold parameter, a large amount of fill-in. The next theorem indicates how breakdown can be prevented when A is SPD through reformulating the A-orthogonalization.

ALGORITHM 11.4 AINV preconditioner (right-looking approach)
Input: SPD matrix A and sparsifying rule.
Output: $A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.

1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ ▷ Initialise Z to hold the columns of the identity matrix
2: **for** $j = 1 : n$ **do**
3: $\quad d_{jj} = A_{j,1:n}\, z_j^{(j-1)}$
4: $\quad$ **for** $k = j + 1 : n$ **do**
5: $\quad\quad \alpha = A_{j,1:n}\, z_k^{(j-1)} / d_{jj}$
6: $\quad\quad z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
7: $\quad\quad$ Sparsify $z_k^{(j)}$ ▷ Drop entries from $z_k^{(j)}$
8: $\quad$ **end for**
9: **end for**
10: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$

Theorem 11.6 (Benzi et al. 2000; Kopal et al. 2012) *Consider Algorithm 11.4 with no sparsification (Step 7 is removed). The following identity holds*

$$A_{j,1:n}\, z_k^{(j-1)} \equiv e_j^T A z_k^{(j-1)} = \langle z_j^{(j-1)}, z_k^{(j-1)} \rangle_A, \quad 1 \le j \le k \le n.$$

Proof Because $AZ = Z^{-T}D$ and $Z^{-T}D$ is lower triangular, entries 1 to $j-1$ of the vector $Az_k^{(j-1)}$ are equal to zero. Z is unit upper triangular so entries $j+1$ to n of its j-th column $z_j^{(j-1)}$ are also equal to zero. Thus, $z_j^{(j-1)}$ can be written as the sum $z + e_j$, where entries j to n of the vector z are zero. The result follows. □

This suggests using alternative computations within the AINV approach based on the whole of A instead of on its rows. The reformulation, which is called the stabilized AINV algorithm (SAINV), is outlined in Algorithm 11.6. It is breakdown-free for any SPD matrix A because the diagonal entries are $d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A > 0$. Practical experience shows that, while slightly more costly to compute, the SAINV algorithm gives higher quality preconditioners than the AINV algorithm. However, the computed diagonal entries can still be very small and may need to be modified.

The factors Z and D obtained with no sparsification can be used to compute the square root-free Cholesky factorization of A. The L factor of A and the inverse factor Z computed using Algorithm 11.6 without sparsification satisfy

ALGORITHM 11.5 Nonsymmetric AINV preconditioner (right-looking approach)

Input: Nonsymmetric matrix A and sparsifying rule.

Output: $A^{-1} \approx WD^{-1}Z^T$ with W and Z unit upper triangular matrices and D a diagonal matrix.

1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ and $[w_1^{(0)}, \ldots, w_n^{(0)}] = [e_1, \ldots, e_n]$
2: **for** $j = 1 : n$ **do**
3: $\quad d_{jj} = (A_{1:n,j})^T z_j^{(j-1)}$ or $d_{jj} = A_{j,1:n}\, w_j^{(j-1)}$
4: $\quad$ **for** $k = j + 1 : n$ **do**
5: $\quad\quad \alpha = (A_{1:n,j})^T z_k^{(j-1)}/d_{jj}$
6: $\quad\quad z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
7: $\quad\quad$ Sparsify $z_k^{(j)}$ ▷ Drop entries from $z_k^{(j)}$
8: $\quad\quad \beta = A_{j,1:n}\, w_k^{(j-1)}/d_{jj}$
9: $\quad\quad w_k^{(j)} = w_k^{(j-1)} - \beta w_j^{(j-1)}$
10: $\quad\quad$ Sparsify $w_k^{(j)}$ ▷ Drop entries from $w_k^{(j)}$
11: $\quad$ **end for**
12: **end for**
13: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$ and $W = [w_1^{(0)}, \ldots, w_n^{(n-1)}]$

$$AZ = LD \quad \text{or} \quad L = AZD^{-1}.$$

Using $d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A$, and equating corresponding entries of AZD^{-1} and L, gives

$$l_{ij} = \frac{\langle z_j^{(j-1)}, z_i^{(j-1)} \rangle_A}{\langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A}, \quad 1 \le j \le i \le n.$$

Thus, the SAINV algorithm generates the L factor of the square root-free Cholesky factorization of A as a by-product of orthogonalization in the inner product $\langle ., . \rangle_A$ at no extra cost and without breakdown.

The stabilization strategy can be extended to the nonsymmetric AINV algorithm using the following result.

Theorem 11.7 (Benzi & Tůma 1998; Bollhöfer & Saad 2002) *Consider Algorithm 11.5 with no sparsification (Steps 7 and 10 removed). The following identities hold:*

ALGORITHM 11.6 SAINV preconditioner (right-looking approach)
Input: SPD matrix A and sparsifying rule.
Output: $A^{-1} \approx Z D^{-1} Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.

$[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$
for $j = 1 : n$ **do**
  $d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A$
  for $k = j + 1 : n$ **do**
    $\alpha = \langle z_k^{(j-1)}, z_j^{(j-1)} \rangle_A / d_{jj}$
    $z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
    Sparsify $z_k^{(j)}$ ▷ Drop entries from $z_k^{(j)}$
  end for
end for
$Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$

$$A_{j,1:n}\, z_k^{(j-1)} = e_j^T A z_k^{(j-1)} = \langle w_j^{(j-1)}, z_k^{(j-1)} \rangle_A,$$

$$(A_{1:n,j})^T w_k^{(j-1)} = e_j^T A^T w_k^{(j-1)} = \langle z_j^{(j-1)}, w_k^{(j-1)} \rangle_A, \quad 1 \le j \le k \le n.$$

The nonsymmetric SAINV algorithm obtained using this reformulation can improve the preconditioner quality, but it is not guaranteed to be breakdown-free.

11.4 Notes and References

Benzi & Tůma (1999) present an early comparative study that puts preconditioning by approximate inverses into the context of alternative preconditioning techniques; see also Bollhöfer & Saad (2002, 2006), Benzi & Tůma (2003), and Bru et al. (2008, 2010). The inverse by bordering method mentioned in Section 11.1 is from Saad (2003b).

The first use of approximate inverses based on Frobenius norm minimization is given by Benson (1973). A SPAI approach that can exploit a dynamically changing sparsity pattern $\mathcal{S}$ is introduced in Cosgrove et al. (1992); an independent and enhanced description is given in the influential paper by Grote & Huckle (1997). Later developments are presented in Holland et al. (2005), Jia & Zhang (2013), and Jia & Kang (2019). A comprehensive discussion on the choice of the sparsity pattern $\mathcal{S}$ can be found in Huckle (1999). Huckle & Kallischko (2007) consider modifying the SPAI method by probing or symmetrizing the approximate inverse

and Bröker et al. (2001) look at using approximate inverses based on Frobenius norm minimization as smoothers for multigrid methods. Choosing sparsity patterns for a related approximate inverse with a particular emphasis on parallel computing is described by Chow (2000).

For nonsymmetric matrices, `MI12` within the HSL mathematical software library computes SPAI preconditioners (see Gould & Scott, 1998 for details and a discussion of the merits and limitations of the approach). An early parallel implementation is given by Barnard et al. (1999). Dehnavi et al. (2013) present an efficient parallel implementation that uses GPUs and include comparisons with ParaSails (Chow, 2001). The latter handles SPD problems using a factored sparse approximate inverse and general problems with an unfactored sparse approximate inverse. A priori techniques determine $\mathcal{S}$ as a power of a sparsified matrix.

Original work on the FSAI preconditioner is by Kolotilina & Yeremin (1986, 1993). Its use in solving systems on massively parallel computers is presented in Kolotilina et al. (1992), while an interesting iterative construction can be found in Kolotilina et al. (2000). A parallel variant called ISAI preconditioning that combines a Frobenius norm-based approach with traditional ILU preconditioning is proposed by Anzt et al. (2018). FSAI preconditioning has attracted significant theoretical and practical attention. Recent contributions discuss not only its efficacy but also parallel computation, the use of blocks, supernodes, and multilevel implementations (Ferronato et al., 2012, 2014; Janna & Ferronato, 2011; Janna et al., 2010, 2013, 2015; Ferronato & Pini, 2018; Magri et al., 2018). Many of these enhancements are exploited in the FSAIPACK software of Janna et al. (2015).

The AINV preconditioner for SPD and nonsymmetric systems is introduced in Benzi et al. (1996) and Benzi & Tůma (1998), respectively; see also Benzi et al. (1999) for a parallel implementation. However, the development of this type of preconditioner follows much earlier interest in factorized matrix inverses (for example, Morris, 1946 and Fox et al., 1948). For the SAINV algorithm, see Benzi et al. (2000) and Kharchenko et al. (2001). Theoretical and practical properties of the AINV and SAINV factorizations are studied in a series of papers by Kopal et al. (2012, 2016, 2020).

References

Abdelfattah, A., Anzt, H., Boman, E., Carson, E., Cojean, T., Dongarra, J., Fox, A., Gates, M., N. Higham, X. S. L., Liu, Y., Loe, J., Luszczek, P., Pranesh, S., Rajamanickam, S., Ribizel, T., Smith, B., Swirydowicz, K., Thomas, S., Tomov, S., Tzai, M., Yamazaki, I., & Yang, U. M. (2021). A survey of numerical linear algebra methods utilizing mixed precision. *International Journal of High Performance Computing Applications, 35*(4), 344–369.

Acer, S., Kayaaslan, E., & Aykanat, C. (2019). A hypergraph partitioning model for profile minimization. *SIAM Journal on Scientific Computing, 41*(1), A83–A108.

Agrawal, A., Klein, P., & Ravi, R. (1993). Cutting down on fill using nested dissection: Provably good elimination orderings. In A. George, J. R. Gilbert, & J. W. H. Liu (Eds.), *Graph Theory and Sparse Matrix Computation* (pp. 31–55). New York: Springer.

Aho, A. V., Garey, M. R., & Ullman, J. D. (1972). The transitive reduction of a directed graph. *SIAM Journal on Computing, 1*(2), 131–137.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data Structures and Algorithms*. Computer Science and Information Processing. Reading, Mass: Addison-Wesley Publishing Co.

Ajiz, M. A. & Jennings, A. (1984). A robust incomplete Choleski-conjugate gradient algorithm. *International Journal for Numerical Methods in Engineering, 20*(5), 949–966.

Aliaga, J. I., Badia, R. M., Barreda, M., Bollhöfer, M., Dufrechou, E., Ezzatti, P., & Quintana-Ortí, E. S. (2016). Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. *Parallel Computing, 54*, 97–107.

Aliaga, J. I., Dufrechou, E., Ezzatti, P., & Quintana-Ortí, E. S. (2019). Accelerating the task/data-parallel version of ILUPACK's BiCG in multi-CPU/GPU configurations. *Parallel Computing, 85*, 79–87.

Amestoy, P. R., Davis, T. A., & Duff, I. S. (1996). An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications, 17*(4), 886–905.

Amestoy, P. R., Li, X. S., & Ng, E. G. (2007). Diagonal Markowitz scheme with local symmetrization. *SIAM Journal on Matrix Analysis and Applications, 29*(1), 228–244.

Amestoy, P. R. & Puglisi, C. (2002). An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications, 24*(2), 553–569.

Anzt, H., Chow, E., & Dongarra, J. (2018). ParILUT - a new parallel threshold ILU factorization. *SIAM Journal on Scientific Computing, 40*, C503–C519.

Anzt, H., Huckle, T. K., Bräckle, J., & Dongarra, J. (2018). Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Computing, 71*, 1–22.

Arioli, M., Demmel, J. W., & Duff, I. S. (1989). Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications, 10*(2), 165–190.

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6

Arioli, M., Maryška, J., Rozložník, M., & Tůma, M. (2006). Dual variable methods for mixed-hybrid finite element approximation of the potential fluid flow problem in porous media. *Electronic Transactions on Numerical Analysis, 22*, 17–40.

Ashcraft, C. (1995). Compressed graphs and the minimum degree algorithm. *SIAM Journal on Scientific Computing, 16*(6), 1404–1411.

Ashcraft, C., Grimes, R. G., & Lewis, J. G. (1998). Accurate symmetric indefinite linear equation solvers. *SIAM Journal on Matrix Analysis and Applications, 20*(2), 513–561.

Axelsson, O. (1972). A generalized SSOR method. *BIT, 12*, 443–467.

Axelsson, O. (1994). *Iterative Solution Methods*. Cambridge: Cambridge University Press.

Aykanat, C., Pinar, A., & Çatalyürek, U. V. (2004). Permuting sparse rectangular matrices into block-diagonal form. *SIAM Journal on Scientific Computing, 25*(6), 1860–1879.

Azad, A., Buluç, A., Li, X. S., Wang, X., & Langguth, J. (2020). A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs. *SIAM Journal on Scientific Computing, 42*(4), C143–C168.

Bai, Z.-Z. & Pan, J.-Y. (2021). *Matrix Analysis and Computations*. Philadelphia, PA: SIAM.

Baker, Jr., G. A. & Oliphant, T. A. (1960). An implicit, numerical method for solving the two-dimensional heat equation. *Quarterly of Applied Mathematics, 17*(4), 361–373.

Barnard, S. T., Clay, R. L., & Simon, H. D. (1999). An MPI implementation of the SPAI preconditioner on the T3E. *International Journal of High Performance Computing Applications, 13*(2), 107–123.

Barnard, S. T., Pothen, A., & Simon, H. D. (1995). A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications, 2*(4), 317–334.

Bebendorf, M. (2008). *Hierarchical Matrices*. *Lecture Notes in Computational Science and Engineering*, vol. 63. Berlin, Heidelberg: Springer.

Bebendorf, M., Bollhöfer, M., & Bratsch, M. (2013). Hierarchical matrix approximation with blockwise constraints. *BIT, 53*(2), 311–339.

Bebendorf, M. & Fischer, T. (2008). A purely algebraic approach to preconditioning based on hierarchical *LU* factorizations. In K. Kunisch, G. Of, & O. Steinbach (Eds.), *Numerical Mathematics and Advanced Applications, Proceedings of ENUMATH 2007*, (pp. 135–142). Berlin, Heidelberg: Springer.

Benson, M. W. (1973). Iterative solution of large scale linear systems. Master's thesis, Thunder Bay, Canada: Lakehead University.

Benzi, M. (2002). Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics, 182*(2), 418–477.

Benzi, M. (2017). Review of *André-Louis Cholesky: Mathematician, Topographer and Army Officer* by Claude Brezinski and Dominique Tournès. *Math Intelligencer, 39*(2), 99–191.

Benzi, M., Cullum, J. K., & Tůma, M. (2000). Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM Journal on Scientific Computing, 22*(4), 1318–1332.

Benzi, M., Golub, G., & Liesen, J. (2005). Numerical solution of saddle point problems. *Acta Numerica, 14*, 1–137.

Benzi, M., Marín, J., & Tůma, M. (1999). Parallel preconditioning with factorized sparse approximate inverses. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. Philadelphia, PA: SIAM.

Benzi, M., Meyer, C. D., & Tůma, M. (1996). A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing, 17*(5), 1135–1149.

Benzi, M., Szyld, D. B., & van Duin, A. (1999). Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM Journal on Scientific Computing, 20*(5), 1652–1670.

Benzi, M. & Tůma, M. (1998). A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM Journal on Scientific Computing, 19*(3), 968–994.

Benzi, M. & Tůma, M. (1999). A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics, 30*(2-3), 305–340.

Benzi, M. & Tůma, M. (2000). Orderings for factorized sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing, 21*(5), 1851–1868.

Benzi, M. & Tůma, M. (2003). A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications, 10*(5-6), 385–400.

Berge, C. (1957). Two theorems in graph theory. *Proceedings National Academy of Sciences of USA, 43*(9), 842–844.

Bern, M., Gilbert, J. R., Hendrickson, B., Nguyen, N. T., & Toledo, S. (2006). Support-graph preconditioners. *SIAM Journal on Matrix Analysis and Applications, 27*(4), 930–951.

Bertaccini, D. & Durastante, F. (2018). *Iterative Methods and Preconditioning for Large and Sparse Linear Systems with Applications*. Monographs and Research Notes in Mathematics. Boca Raton, FL: CRC Press.

Bichot, C.-E. & Siarry, P. (Eds.). (2013). *Graph Partitioning*. New York: John Wiley.

Björck, Å. (1996). *Numerical Methods for Least Squares Problems*. Philadelphia, PA: SIAM.

Björck, Å. (2015). *Numerical Methods in Matrix Computations*. Texts in Applied Mathematics, vol. 59. Cham: Springer.

Bollhöfer, M. (2015). Algebraic preconditioning approaches and their applications. In P. Benner, M. Bollhöfer, D. Kressner, C. Mehl, & T. Stykel (Eds.), *Numerical Algebra, Matrix Theory, Differential-Algebraic Equations and Control Theory* (pp. 257–295). Berlin: Springer International Publishing.

Bollhöfer, M., Aliaga, J., Martín, A., & Quintana-Ortí, E. (2012). ILUPACK. In Padua, D. (Ed.), *Encyclopedia of Parallel Computing*, (pp. 917–926). Berlin: Springer.

Bollhöfer, M. & Saad, Y. (2002). On the relations between ILUs and factored approximate inverses. *SIAM Journal on Matrix Analysis and Applications, 24*(1), 219–237.

Bollhöfer, M. & Saad, Y. (2006). Multilevel preconditioners constructed from inverse-based ILUs. *SIAM Journal on Scientific Computing, 27*(5), 1627–1650.

Bollhöfer, M. & Schenk, O. (2006). Combinatorial aspects in sparse elimination methods. *GAMM-Mitteilungen, 29*(2), 342–367.

Bollhöfer, M., Schenk, O., Janalik, R., Hamm, S., & Gullapalli, K. (2020). State-of-the-art sparse direct solvers. In A. Grama & A. Sameh (Eds.), *Parallel Algorithms in Computational Science and Engineering, Modeling and Simulation in Science*. Modelling and Simulation in Science, Engineering and Technology (pp. 3–33). Birkhäuser: Springer Nature Switzerland AG.

Bondy, J. A. & Murty, U. S. R. (2008). *Graph Theory*. Graduate Texts in Mathematics, vol. 244. New York: Springer.

Brayton, R. K., Gustavson, F. G., & Willoughby, R. A. (1970). Some results on sparse matrices. *Mathematics of Computation, 24*, 937–954.

Bridson, R. & Tang, W.-P. (1999). Ordering, anisotropy, and factored sparse approximate inverses. *SIAM Journal on Scientific Computing, 21*(3), 867–882.

Bröker, O., Grote, M. J., Mayer, C., & Reusken, A. (2001). Robust parallel smoothing for multigrid via sparse approximate inverses. *SIAM Journal on Scientific Computing, 23*(4), 1396–1417.

Bru, R., Marín, J., Mas, J., & Tůma, M. (2008). Balanced incomplete factorization. *SIAM Journal on Scientific Computing, 30*(5), 2302–2318.

Bru, R., Marín, J., Mas, J., & Tůma, M. (2010). Improved balanced incomplete factorization. *SIAM Journal on Matrix Analysis and Applications, 31*(5), 2431–2452.

Brualdi, R. A. & Ryser, H. J. (1991). *Combinatorial Matrix Theory*. Cambridge: Cambridge University Press.

Buleev, N. I. (1959). A numerical method for solving two-dimensional diffusion equations (in Russian). *Atomnaja Energija, 6*, 338–340.

Buluç, A., Gilbert, J., & Shah, V. B. (2011). Implementing sparse matrices for graph algorithms. In J. Kepner & J. Gilbert (Eds.), *Graph Algorithms in the Language of Linear Algebra* (pp. 287–313). Philadelphia, PA: SIAM.

Bunch, J. R. (1971). Analysis of the diagonal pivoting method. *SIAM Journal on Numerical Analysis, 8*(4), 656–680.

Bunch, J. R. & Kaufman, L. (1977). Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation, 31*, 162–179.

Bunch, J. R. & Parlett, B. (1971). Direct methods for solving symmetric indefinite systems of linear systems. *SIAM Journal on Numerical Analysis, 8*(4), 639–655.

Carpentieri, B., Liao, J., & Sosonkina, M. (2014). VBARMS: a variable block algebraic recursive multilevel solver for sparse linear systems. *Journal of Computational and Applied Mathematics, 259*(A), 164–173.

Carson, E. & Higham, N. J. (2017). A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM Journal on Scientific Computing, 39*(6), A2834–A2856.

Carson, E. & Higham, N. J. (2018). Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing, 40*(2), A817–A847.

Carson, E., Higham, N. J., & Pranesh, S. (2020). Three-precision GMRES-based iterative refinement for least squares problems. *SIAM Journal on Scientific Computing, 42*(6), A4063–A4083.

Chan, T. F. & van der Vorst, H. A. (1997). Approximate and incomplete factorizations. In A. S. D.E. Keyes & V. Venkatakrishnan (Eds.), *Parallel Numerical Algorithms* (pp. 167–202). Dordrecht: Kluwer Academic Publishers.

Chen, K. (2005). *Matrix preconditioning and applications*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge: Cambridge University Press.

Chen, Y., Davis, T. A., Hager, W. W., & Rajamanickam, S. (2008). Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software, 35*(3), Art. 22, 1–14.

Chevalier, C. & Pellegrini, F. (2008). PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing, 34*(6–8), 318–331.

Chow, E. (2000). A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing, 21*(5), 1804–1822.

Chow, E. (2001). Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *International Journal of High Performance Computing Applications, 15*(1), 56–74.

Chow, E. & Patel, A. (2015). Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing, 37*(2), C169–C193.

Chow, E. & Saad, Y. (1997). Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics, 86*(2), 387–414.

Ciaramella, G. & Gander, M. J. (2022). *Iterative Methods and Preconditioners for Systems of Linear Equations*. Fundamentals of Algorithms. Philadelphia, PA: SIAM.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). Cambridge, MA: MIT Press.

Cosgrove, J. D. F., Díaz, & Griewank, A. (1992). Approximate inverse preconditioning for sparse linear systems. *International Journal of Computer Mathematics, 44*, 91–110.

Cuthill, E. H. & McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings* 24^{th} *National Conference of the ACM* (pp. 157–172). New York: ACM Press.

Davis, T. A. (2004). Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software, 30*(2), 196–199.

Davis, T. A. (2006). *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. Philadelphia, PA: SIAM.

Davis, T. A. & Duff, I. S. (1997). An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications, 18*(1), 140–158.

Davis, T. A., Rajamanickam, S., & Sid-Lakhdar, W. M. (2016). A survey of direct methods for sparse linear systems. *Acta Numerica, 25*, 383–566.

de Niet, A. C. & Wubs, F. W. (2009). Numerically stable LDL^T -factorization of F-type saddle point matrices. *IMA Journal of Numerical Analysis, 29*(1), 208–234.

de Oliveira, S. L. G., Bernardes, J. A. B., & Chagas, G. O. (2018). An evaluation of low-cost heuristics for matrix bandwidth and profile reductions. *Computational and Applied Mathematics, 37*(2), 1412–1471.

Dehnavi, M. M., Becerra, F., Moises, D., Gaudiot, J.-L., & Giannacopoulos, D. D. (2013). Parallel sparse approximate inverse preconditioning on graphic processing units. *IEEE Transactions on Parallel and Distributed Systems, 24*(9), 1852–1862.

Demmel, J. W. (1997). *Applied Numerical Linear Algebra*. Philadelphia, PA: SIAM.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., & Liu, J. W. H. (1999). A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications, 20*(3), 720–755.

Demmel, J. W., Heath, Michael, T., & van der Vorst, H. A. (1993). Parallel numerical linear algebra. *Acta Numerica, 2*, 111–197.

Dijkstra, E. W. (1959). A note on two problems connected with graphs. *Numerische Mathematik, 1*, 269–271.

Dolean, V., Jolivet, P., & Nataf, F. (2015). *An Introduction to Domain Decomposition Methods*. Philadelphia, PA: SIAM.

Dollar, H. S. & Scott, J. A. (2010). A note on fast approximate minimum degree orderings for matrices with some dense rows. *Numerical Linear Algebra with Applications, 17*(1), 43–55.

Dongarra, J. J., Duff, I. S., Sorensen, D. C., & van der Vorst, H. A. (1998). *Numerical Linear Algebra for High-Performance Computers*. Software, Environments, and Tools. Philadelphia, PA: SIAM.

Dongarra, J. J., Gustavson, F. G., & Karp, A. (1984). Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review, 26*, 91–112.

Doolittle, M. H. (1878). Method employed in this office in the solution of normal equations used in the adjustment of a triangulation. In *Report of the Superintendent, US Coast and Geodetic Survey (1878). Appendix 8, Paper 3*, (pp. 115–120).

Duff, I. (1977). A survey of sparse matrix research. *Proceedings of the IEEE, 65*(4), 500–535.

Duff, I. (1984). Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core. *SIAM Journal on Scientific Computing, 5*(2), 270–280.

Duff, I. S. (1980). MA28–a set of Fortran subroutines for sparse unsymmetric linear equations. Technical Report AERE-R.8730, UK: Harwell Laboratory.

Duff, I. S. (1981). A sparse future. In Duff, I. S. (Ed.), *Sparse Matrices and their Uses*, Institute of Mathematics and its Applications Conference Series, (pp. 1–29). New York: Academic Press.

Duff, I. S. (2004). MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software, 30*(2), 118–154.

Duff, I. S., Erisman, A. M., Gear, C. W., & Reid, J. K. (1988). Sparsity structure and Gaussian elimination. *ACM SIGNUM Newsletter, 23*(2), 2–8.

Duff, I. S., Erisman, A. M., & Reid, J. K. (2017). *Direct Methods for Sparse Matrices* (2nd ed.). Oxford: Oxford University Press.

Duff, I. S., Gould, N. I. M., Reid, J. K., Scott, J. A., & Turner, K. (1991). The factorization of sparse symmetric indefinite matrices. *IMA Journal of Numerical Analysis, 11*(2), 181–204.

Duff, I. S., Hogg, J. D., & Lopez, L. (2018). A new sparse symmetric indefinite solver using a posteriori threshold pivoting. Technical Report RAL-TR-2018-008, Chilton, Oxfordshire, England: Rutherford Appleton Laboratory.

Duff, I. S. & Koster, J. (1999). The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications, 20*(4), 889–901.

Duff, I. S. & Koster, J. (2001). On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications, 22*(4), 973–996.

Duff, I. S. & Meurant, G. A. (1989). The effect of ordering on preconditioned conjugate gradients. *BIT, 29*(4), 635–657.

Duff, I. S. & Pralet, S. (2005). Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications, 27*(2), 313–340.

Duff, I. S. & Reid, J. K. (1978). An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software, 4*(2), 137–147.

Duff, I. S. & Reid, J. K. (1983). The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software, 9*(3), 302–325.

Duff, I. S. & Reid, J. K. (1984). The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific Computing, 5*(3), 633–641.

Duff, I. S. & Reid, J. K. (1996). The design of MA48: A code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Transactions on Mathematical Software, 22*(2), 187–226.

Duff, I. S. & Scott, J. A. (1994). The use of multiple fronts in Gaussian elimination. In Lewis, J. (Ed.), *Proceedings of the 5th SIAM Conference Applied Linear Algebra*. Philadelphia, PA: SIAM.

Duff, I. S. & Scott, J. A. (1996). The design of a new frontal code for solving sparse, unsymmetric systems. *ACM Transactions on Mathematical Software, 22*(1), 30–45.

Duff, I. S. & Scott, J. A. (1999). A frontal code for the solution of sparse positive-definite symmetric systems arising from finite-element applications. *ACM Transactions on Mathematical Software, 25*(1), 404–424.

Duff, I. S. & Scott, J. A. (2005). Stabilized bordered block diagonal forms for parallel sparse solvers. *Parallel Computing, 31*, 275–289.

Dupont, T., Kendall, R. P., & Rachford, H. H. J. (1968). An approximate factorization procedure for the solving self-adjoint elliptic difference equations. *SIAM Journal on Numerical Analysis, 5*, 559–573.

Edmonds, J. (1965). Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards Section B, 69B*, 125–130.

Eisenstat, S. (1981). Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM Journal on Scientific and Statistical Computing, 2*, 1–4.

Eisenstat, S. C., Gursky, M. C., Schultz, M. H., & Sherman, A. H. (1977). The Yale Sparse Matrix Package (YSMP) – II : The non-symmetric codes. Technical Report No. 114, Dept. of Computer Science, Yale University.

Eisenstat, S. C., Gursky, M. C., Schultz, M. H., & Sherman, A. H. (1982). The Yale Sparse Matrix Package (YSMP) – I : The symmetric codes. *International Journal for Numerical Methods in Engineering, 18*, 1145–1151.

Eisenstat, S. C. & Liu, J. W. H. (1992). Exploiting structural symmetry in unsymmetric sparse symbolic factorization. *SIAM Journal on Matrix Analysis and Applications, 13*(1), 202–211.

Eisenstat, S. C. & Liu, J. W. H. (1993a). Exploiting structural symmetry in a sparse partial pivoting code. *SIAM Journal on Scientific Computing, 14*(1), 253–257.

Eisenstat, S. C. & Liu, J. W. H. (1993b). Structural representations of Schur complements in sparse matrices. In A. George, J. Gilbert, & J. W. H. (Eds.), *Graph Theory and Sparse Matrix Computation*. *The IMA Volumes in Mathematics and its Applications*, vol. 56 (pp. 85–100). New York: Springer.

Eisenstat, S. C. & Liu, J. W. H. (2005a). The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications, 26*(3), 686–705.

Eisenstat, S. C. & Liu, J. W. H. (2005b). A tree-based dataflow model for the unsymmetric multifrontal method. *Electronic Transactions on Numerical Analysis, 21*, 1–19.

Eisenstat, S. C. & Liu, J. W. H. (2007). Algorithmic aspects of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications, 29*(4), 1363–1381.

Elman, H. C. (1986). A stability analysis of incomplete LU factorizations. *Mathematics of Computation, 47*, 191–217.

Elman, H. C. (1989). Relaxed and stabilized incomplete factorizations for non-self-adjoint linear systems. *BIT, 29*, 890–915.

Erisman, A. M., Grimes, R. G., Lewis, J. G., Poole, W. G., & Simon, H. D. (1987). Evaluation of orderings for unsymmetric sparse matrices. *SIAM Journal on Scientific Computing, 8*(4), 600–624.

Evans, D. J. (Ed.). (1985). *Sparsity and Its Applications*. Cambridge: Cambridge University Press.

Falgout, R. D., Jones, J. E., & Yang, U. M. (2006). The design and implementation of *hypre*, a library of parallel high performance preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Lecture Notes in Computational Science and Engineering, vol. 51 (pp. 267–294). Berlin: Springer.

Falgout, R. D., Li, R., Sjögreen, B., Wang, L., & Yang, U. M. (2021). Porting *hypre* to heterogeneous computer architectures: strategies and experiences. *Parallel Computing, 108*, Paper No. 102840, 1–12.

Ferronato, M., Janna, C., & Pini, G. (2012). Shifted FSAI preconditioners for the efficient parallel solution of non-linear groundwater flow models. *International Journal for Numerical Methods in Engineering, 89*(13), 1707–1719.

Ferronato, M., Janna, C., & Pini, G. (2014). A generalized Block FSAI preconditioner for nonsymmetric linear systems. *Journal of Computational and Applied Mathematics, 256*, 230–241.

Ferronato, M. & Pini, G. (2018). A supernodal block factorized sparse approximate inverse for non-symmetric linear systems. *Numerical Algorithms, 78*(1), 333–354.

Foster, L. V. (1997). The growth factor and efficiency of Gaussian elimination with rook pivoting. *Journal of Computational and Applied Mathematics, 86*(1), 177–194.

Fox, L., Huskey, H. D., & Wilkinson, J. H. (1948). Notes on the solution of algebraic linear simultaneous equations. *The Quarterly Journal of Mechanics and Applied Mathematics, 1*, 149–173.

Fritzsche, D., Frommer, A., Shank, S. D., & Szyld, D. B. (2013). Overlapping blocks by growing a partition with applications to preconditioning. *SIAM Journal on Scientific Computing, 35*(1), A453–A473.

George, A. (1973). Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis, 10*, 345–363.

George, A. (1998). On finding and analyzing the structure of the Cholesky factor. In G. W. Althaus, & E. Spedicato (Eds.), *Algorithms for Large Scale Linear Algebraic Systems: Applications in Science and Engineering*. Nato Science Series C (pp. 73–105). Dordrecht: Springer.

George, A. & Liu, J. W. H. (1980a). A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Transactions on Mathematical Software, 6*(3), 337–358.

George, A. & Liu, J. W. H. (1980b). A minimal storage implementation of the minimum degree algorithm. *SIAM Journal on Numerical Analysis, 17*(2), 282–299.

George, A. & Liu, J. W. H. (1980c). An optimal algorithm for symbolic factorization of symmetric matrices. *SIAM Journal on Computing, 9*(3), 583–593.

George, A. & Liu, J. W. H. (1981). *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice Hall.

George, A. & Liu, J. W. H. (1989). The evolution of the minimum degree ordering algorithm. *SIAM Review, 31*(1), 1–19.

George, A. & Ng, E. (1984). A new release of SPARSPAK: the Waterloo sparse matrix package. *ACM SIGNUM Newsletter, 19*(4), 9–13.

George, A. & Ng, E. (1985). An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM Journal on Scientific Computing, 6*(2), 390–409.

George, A. & Pothen, A. (1997). An analysis of spectral envelope-reduction via quadratic assignment problems. *SIAM Journal on Matrix Analysis and Applications, 18*(3), 706–732.

Gibbs, N. E., Poole, W. G. J., & Stockmeyer, P. K. (1976). An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis, 13*(2), 236–250.

Gilbert, J. R. (1994). Predicting structure in sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications, 15*(1), 62–79.

Gilbert, J. R. & Liu, J. W. H. (1993). Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications, 14*(2), 334–352.

Gilbert, J. R., Ng, E., & Peyton, B. W. (1994). An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications, 15*(4), 1075–1091.

Gilbert, J. R. & Ng, E. G. (1993). Predicting structure in nonsymmetric sparse matrix factorizations. In G. George, J. R. Gilbert, & J. W. H. Liu (Eds.), *Graph Theory and Sparse Matrix Computation*. *The IMA Volumes in Mathematics and its Applications* (vol. 56, pp. 107–139). New York: Springer.

Gilbert, J. R. & Peierls, T. (1988). Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing, 9*(5), 862–874.

Gilbert, J. R., Reinhardt, S., & Shah, V. B. (2006). High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*, (pp. 260–269). Berlin: Springer.

Golub, G. H. & Van Loan, C. F. (1996). *Matrix Computations* (4th ed.). Baltimore and London: The Johns Hopkins University Press.

Gould, N. I. M. & Scott, J. A. (1998). On approximate inverse preconditioners. *SIAM Journal on Scientific Computing, 19*(2), 605–625.

Gould, N. I. M., Scott, J. A., & Hu, Y. (2007). A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software, 33*(2), Art. 10, 1–32.

Grcar, J. F. (2011). Mathematicians of Gaussian elimination. *Notices of the AMS, 58*(6), 782–792.

Greenbaum, A. (1997). *Iterative Methods For Solving Linear Systems*. Philadelphia, PA: SIAM.

Grigori, L., Demmel, J. W., & Li, X. S. (2007). Parallel symbolic factorization for sparse LU with static pivoting. *SIAM Journal on Scientific Computing, 29*(3), 1289–1314.

Grigori, L., Gilbert, J. R., & Cosnard, M. (2009). Symbolic and exact structure prediction for sparse Gaussian elimination with partial pivoting. *SIAM Journal on Matrix Analysis and Applications, 30*(4), 1520–1545.

Grote, M. J. & Huckle, T. (1997). Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing, 18*(3), 838–853.

Gupta, A. & George, T. (2010). Adaptive techniques for improving the performance of incomplete factorization preconditioning. *SIAM Journal on Scientific Computing, 32*(1), 84–110.

Gustafsson, I. (1978). A class of first order factorization methods. *BIT, 18*(2), 142–156.

Gustafsson, I. (1979). On modified incomplete Cholesky factorization methods for the solution of problems with mixed boundary conditions and problems with discontinuous material coefficients. *International Journal for Numerical Methods in Engineering, 14*(8), 1127–1140.

Haskins, L. & Rose, D. J. (1973). Toward characterization of perfect elimination digraphs. *SIAM Journal on Computing, 2*(4), 217–224.

Heggernes, P., Eisenstat, S., Kumfert, G., & Pothen, A. (2001). The computational complexity of the minimum degree algorithm. ICASE Report No. 2001-42, Hampton, Virginia: Institute for Computer Applications in Science and Engineering.

Hénon, P., Ramet, P., & Roman, J. (2002). PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing, 28*(2), 301–321.

Hénon, P., Ramet, P., & Roman, J. (2008). On finding approximate supernodes for an efficient block-$ILU(k)$ factorization. *Parallel Computing, 34*(6–8), 345–362.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Philadelphia, PA: SIAM.

Higham, N. J. (2011). Gaussian elimination. *Wiley Interdisciplinary Reviews: Computational Statistics, 3*(3), 230–238.

Hogg, J. D., Ovtchinnikov, E., & Scott, J. A. (2016). A sparse symmetric indefinite direct solver for GPU architectures. *ACM Transactions on Mathematical Software, 42*(1), Art. 1, 1–25.

Hogg, J. D., Reid, J. K., & Scott, J. A. (2010). Design of a multicore sparse Cholesky factorization using DAGs. *SIAM Journal on Scientific Computing, 32*(6), 3627–3649.

Hogg, J. D. & Scott, J. A. (2010). A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Transactions on Mathematical Software, 37*(2), Art. 17, 1–19.

Hogg, J. D. & Scott, J. A. (2013a). An efficient analyse phase for element problems. *Numerical Linear Algebra with Applications, 20*(3), 397–412.

Hogg, J. D. & Scott, J. A. (2013b). New parallel sparse direct solvers for multicore architectures. *Algorithms, 6*(4), 702–725.

Hogg, J. D. & Scott, J. A. (2013c). Pivoting strategies for tough sparse indefinite systems. *ACM Transactions on Mathematical Software, 40*(1), Art. 4, 2–19.

Hogg, J. D. & Scott, J. A. (2014). Compressed threshold pivoting for sparse symmetric indefinite systems. *SIAM Journal on Matrix Analysis and Applications, 35*(2), 783–817.

Hogg, J. D. & Scott, J. A. (2015). On the use of suboptimal matchings for scaling and ordering sparse symmetric matrices. *Numerical Linear Algebra with Applications, 22*(4), 648–663.

Holland, R. M., Wathen, A. J., & Shaw, G. J. (2005). Sparse approximate inverses and target matrices. *SIAM Journal on Scientific Computing, 26*(3), 1000–1011.

Hood, P. (1976). Frontal solution program for unsymmetric matrices. *International Journal for Numerical Methods in Engineering, 10*, 379–400.

HSL (accessed 2022). HSL. A collection of Fortran codes for large-scale scientific computation. http://www.hsl.rl.ac.uk.

Hu, Y. & Scott, J. (2005). Ordering techniques for singly bordered block diagonal forms for unsymmetric parallel sparse direct solvers. *Numerical Linear Algebra with Applications, 12*(9), 877–894.

Hu, Y. F., Maguire, K. C. F., & Blake, R. J. (2000). A multilevel unsymmetric matrix ordering for parallel process simulation. *Computers and Chemical Engineering, 23*, 1631–1647.

Hu, Y. F. & Scott, J. A. (2001). A multilevel algorithm for wavefront reduction. *SIAM Journal on Scientific Computing, 23*(4), 1352–1375.

Huckle, T. (1999). Approximate sparsity patterns for the inverse of a matrix and preconditioning. *Applied Numerical Mathematics, 30*(2-3), 291–303.

Huckle, T. & Kallischko, A. (2007). Frobenius norm minimization and probing for preconditioning. *International Journal of Computer Mathematics, 84*(8), 1225–1248.

Hysom, D. & Pothen, A. (2002). Level-based incomplete LU factorization: Graph model and algorithms. Preprint UCRL-JC-150789, US Department of Energy.

Il′in, V. P. (1992). *Iterative Incomplete Factorization Methods*. Singapore: World Scientific.

IPOPT (accessed 2022). COIN-OR interior point optimizer IPOPT. http://www.hsl.rl.ac.uk.

Irons, B. M. (1970). A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering, 2*, 5–32.

Janna, C. & Ferronato, M. (2011). Adaptive pattern research for block FSAI preconditioning. *SIAM Journal on Scientific Computing, 33*(6), 3357–3380.

Janna, C., Ferronato, M., & Gambolati, G. (2010). A block FSAI-ILU parallel preconditioner for symmetric positive definite linear systems. *SIAM Journal on Scientific Computing, 32*(5), 2468–2484.

Janna, C., Ferronato, M., & Gambolati, G. (2013). Enhanced block FSAI preconditioning using domain decomposition techniques. *SIAM Journal on Scientific Computing, 35*(5), S229–S249.

Janna, C., Ferronato, M., & Gambolati, G. (2015). The use of supernodes in factored sparse approximate inverse preconditioning. *SIAM Journal on Scientific Computing, 37*(1), C72–C94.

Janna, C., Ferronato, M., Sartoretto, F., & Gambolati, G. (2015). FSAIPACK: a software package for high-performance factored sparse approximate inverse preconditioning. *ACM Transactions on Mathematical Software, 41*(2), Art. 10, 1–26.

Jennings, A. (1966). A compact storage scheme for the solution of symmetric linear simultaneous equations. *The Computer Journal, 9*, 281–285.

Jia, Z. & Kang, W. (2019). A transformation approach that makes SPAI, PSAI and RSAI procedures efficient for large double irregular nonsymmetric sparse linear systems. *Journal of Computational and Applied Mathematics, 348*, 200–213.

Jia, Z. & Zhang, Q. (2013). An approach to making SPAI and PSAI preconditioning effective for large irregular sparse linear systems. *SIAM Journal on Scientific Computing, 35*(4), A1903–A1927.

Jones, M. T. & Plassmann, P. E. (1995). An improved incomplete Cholesky factorization. *ACM Transactions on Mathematical Software, 21*(1), 5–17.

Kaporin, I. E. (1998). High quality preconditioning of a general symmetric positive definite matrix based on its $U^T U + U^T R + R^T U$ decomposition. *Numerical Linear Algebra with Applications, 5*, 483–509.

Karypis, G. & Kumar, V. (1998a). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing, 20*(1), 359–392.

Karypis, G. & Kumar, V. (1998b). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing, 48*(1), 71–95.

Kepner, J. & Gilbert, J. (Eds.). (2011). *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA: SIAM.

Kershaw, D. S. (1978). The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics, 26*, 43–65.

Kharchenko, S. A., Kolotilina, L. Y., Nikishin, A. A., & Yeremin, A. Y. (2001). A robust AINV-type method for constructing sparse approximate inverse preconditioners in factored form. *Numerical Linear Algebra with Applications, 8*(3), 165–179.

Kolotilina, L. Y., Nikishin, A. A., & Yeremin, A. Y. (1992). Factorized sparse approximate inverse (FSAI) preconditionings for solving 3D FE systems on massively parallel computers. II. In *Iterative Methods in Linear Algebra* (pp. 311–312). Amsterdam: North-Holland.

Kolotilina, L. Y. & Yeremin, A. Y. (1986). On a family of two-level preconditionings of the incomplete block factorization type. *Soviet Journal of Numerical Analysis and Mathematical Modelling, 1*(4), 293–320.

Kolotilina, L. Y. & Yeremin, A. Y. (1993). Factorized sparse approximate inverse preconditionings. I. Theory. *SIAM Journal on Matrix Analysis and Applications, 14*(1), 45–58.

Kolotilina, L. Y., Yeremin, A. Y., & Nikishin, A. A. (2000). Factorized sparse approximate inverse preconditionings. III: Iterative construction of preconditioners. *Journal of Mathematical Sciences, 101*, 3237–3254.

König, D. (1931). Gráfok és mátrixok. (2020) *Matematikai és Fizikai Lapok, 38*, 116–119. See also G. Szárnyas: Graphs and matrices: A translation of "Graphok és matrixok" by Dénes König (1931), arXiv:2009.03780.

Konshin, I., Olshanskii, M., & Vassilevski, Y. (2017). LU factorizations and ILU preconditioning for stabilized discretizations of incompressible Navier-Stokes equations. *Numerical Linear Algebra with Applications, 24*(3), e2085, 15.

Konshin, I. N., Olshanskii, M. A., & Vassilevski, Y. V. (2019). An algebraic solver for the Oseen problem with application to hemodynamics. In B. Chetverushkin, W. Fitzgibbon, P. Kuznetsov, Y. A.and Neittaanmäki, J. Periaux, & P. O. (Eds.), *Contributions to Partial Differential Equations and Applications*. *Computational Methods in Applied Sciences* (vol. 47, pp. 339–357). Cham: Springer.

Kopal, J., Rozložník, M., & Tůma, M. (2016). Factorized approximate inverses with adaptive dropping. *SIAM Journal on Scientific Computing, 38*(3), A1807–A1820.

Kopal, J., Rozložník, M., & Tůma, M. (2020). A note on adaptivity in factorized approximate inverse preconditioning. *Analele Universitatii "Ovidius" Constanta-Seria Matematica, 28*(2), 149–159.

Kopal, J., Rozložník, M., Smoktunowicz, A., & Tůma., M. (2012). Rounding error analysis of orthogonalization with a non-standard inner product. *BIT, 52*, 1035–1058.

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly, 2*, 83–97.

Kumfert, G. & Pothen, A. (1997). Two improved algorithms for reducing the envelope and wavefront. *BIT, 37*(3), 559–590.

Lacoste, X., Ramet, P., Faverge, M., Ichitaro, Y., & Dongarra, J. (2012). Sparse direct solvers with accelerators over DAG runtimes. Technical Report RR-7972, France: Inria.

Li, N., Saad, Y., & Chow, E. (2003). Crout versions of ILU for general sparse matrices. *SISC, 25*(2), 716–728.

Li, X. S. (1996). *Sparse Gaussian elimination on high performance computers*. PhD thesis, California, Berkeley: University of California.

Li, X. S. (2008). Evaluation of SuperLU on multicore architectures. *Journal of Physics Conference Series, 125*, 012079.

Li, X. S. & Demmel, J. W. (1998). Making sparse Gaussian elimination scalable by static pivoting. In *ACM/IEEE Conference on Supercomputing. IEEE Computer Society, Washington, DC, USA*, (pp. 1–17).

Li, X. S. & Demmel, J. W. (2003). SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software, 29*(2), 110–140.

Li, X. S., Demmel, J. W., Gilbert, J. R., Grigori, L., Shao, M., & Yamazaki, I. (1999). SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Lab. https://portal.nersc.gov/project/sparse/superlu/ug.pdf Last update: June 2018.

Li, Z., Saad, Y., & Sosonkina, M. (2003). pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications, 10*(5–6), 485–509.

Liesen, J. & Strakoš, Z. (2013). *Krylov Subspace Methods*. Numerical Mathematics and Scientific Computation. Oxford: Oxford University Press.

Lin, C.-J. & Moré, J. J. (1999). Incomplete Cholesky factorizations with limited memory. *SIAM Journal on Scientific Computing, 21*(1), 24–45.

Lipton, R. J., Rose, D. J., & Tarjan, R. E. (1979). Generalized nested dissection. *SIAM Journal on Numerical Analysis, 16*(2), 346–358.

Liu, J. W. H. (1985). Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software, 11*(2), 141–153.

Liu, J. W. H. (1986). A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software, 12*(2), 127–148.

Liu, J. W. H. (1990). The role of elimination trees in sparse factorizations. *SIAM Journal on Matrix Analysis and Applications,11*(1), 134–172.

Liu, J. W. H. (1992). The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review, 34*(1), 82–109.

Liu, J. W. H., Ng, E. G., & Peyton, B. W. (1993). On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications, 14*(1), 242–252.

Liu, J. W. H. & Sherman, A. H. (1976). Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis, 13*(2), 198–213.

Liu, P. (2015). An exploration of matrix equilibration. Technical Report CPSC 517, University of British Columbia.

Loe, J. A. & Morgan, R. B. (2021). Toward efficient polynomial preconditioning for GMRES. *Numerical Linear Algebra with Applications, 29*(4), e2427.

Lungten, S., Schilders, W. H. A., & Scott, J. A. (2018). Preordering saddle-point systems for sparse LDL^T factorization without pivoting. *Numerical Linear Algebra with Applications, 25*(5), e2173, 13.

MacLachlan, S., Osei-Kuffuor, D., & Saad, Y. (2012). Modification and compensation strategies for threshold-based incomplete factorizations. *SIAM Journal on Scientific Computing, 34*(1), A48–A75.

Magri, V. A. P., Franceschini, A., Ferronato, M., & Janna, C. (2018). Multilevel approaches for FSAI preconditioning. *Numerical Linear Algebra with Applications, 25*(5), e2183, 18.

Manteuffel, T. A. (1980). An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation, 34*, 473–497.

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

Maryška, J., Rozložník, M., & Tůma, M. (1996). The potential fluid flow problem and the convergence rate of the minimal residual method. *Numerical Linear Algebra with Applications, 3*(6), 525–542.

Maryška, J., Rozložník, M., & Tůma, M. (2000a). Schur complement reduction in the mixed-hybrid approximation of Darcy's law: rounding error analysis. *Journal of Computational and Applied Mathematics, 117*(2), 159–173.

Maryška, J., Rozložník, M., & Tůma, M. (2000b). Schur complement systems in the mixed-hybrid finite element approximation of the potential fluid flow problem. *SIAM Journal on Scientific Computing, 22*(2), 704–723.

Meijerink, J. A. & van der Vorst, H. A. (1977). An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation, 31*(137), 148–162.

Mendelsohn, N. S. (1956). Some properties of approximate inverses of matrices. *Transactions of the Royal Society Canada Section III, 50*, 53–59.

METIS (accessed 2022). METIS: A family of multilevel partitioning algorithms. https://github.com/KarypisLab.

Meurant, G. (1999). *Computer Solution of Large Linear Systems*. North Holland: Elsevier.

Meurant, G. & Duintjer Tebbens, J. (2020). *Krylov Methods for Nonsymmetric Linear Systems - From Theory to Computations*. *Springer Series in Computational Mathematics*, vol. 57. Cham: Springer.

Moler, C. B. (1967). Iterative refinement in floating point. *Journal of the Association for Computing Machinery, 14*(2), 316–321.

Morris, J. (1946). An escalator process for the solution of linear simultaneous equations. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 37*(265), 106–120.

MUMPS (accessed 2022). MUMPS: A parallel sparse direct solver. Version 5.5.0. http://mumps.enseeiht.fr/.

Neal, L. & Poole, G. (1992). A geometric analysis of Gaussian elimination. II. *Linear Algebra and its Applications, 173*, 239–264.

Neumaier, A. & Olschowka, M. (1996). A new pivoting strategy for Gaussian elimination. *Linear Algebra and its Applications, 240*, 131–151.

Ng, E. G. & Peyton, B. W. (1993a). Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing, 14*(5), 1034–1056.

Ng, E. G. & Peyton, B. W. (1993b). A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM Journal on Scientific Computing, 14*(4), 761–769.

Oliker, L., Li, X., Husbands, P., & Biswas, R. (2002). Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review, 44*(3), 373–393.

Olshanskii, M. A. & Tyrtyshnikov, E. E. (2014). *Iterative Methods for Linear Systems*. Philadelphia, PA: SIAM.

Ortega, J. M. (1988a). Efficient implementations of certain iterative methods. *SIAM Journal on Scientific and Statistical Computing, 9*(5), 882–891.

Ortega, J. M. (1988b). *Introduction to Parallel and Vector Computing*. New York: Plenum Press.

Osei-Kuffuor, D., Li, R., & Saad, Y. (2015). Matrix reordering using multilevel graph coarsening for ILU preconditioning. *SIAM Journal on Scientific Computing, 37*(1), A391–A419.

Østerby, O. & Zlatev, Z. (1983). *Direct Methods for Sparse Matrices*. *Lecture Notes in Computer Science*, vol. 157. Berlin: Springer.

PARDISO (accessed 2022). PARDISO 7.2 Solver project. https://www.pardiso-project.org/.

Parlett, B. N. (1974). Review of *Large Sparse Sets of Linear Equations* by J. K. Reid and *Sparse Matrices and their Applications* by Donald J. Rose and Ralph A. Willoughby. *Mathematics of Computation, 28*(126), 669–671.

Parter, S. (1961). The use of linear graphs in Gaussian elimination. *SIAM Review, 3*, 119–130, 364–369.

Pissanetzky, S. (1984). *Sparse Matrix Technology*. London: Academic Press, Inc., Harcourt Brace Jovanovich, Publishers.

Pothen, A. & Fan, C. J. (1990). Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software, 16*(4), 303–324.

Pothen, A. & Toledo, S. (2004). Elimination structures in scientific computing. In D. Mehta & S. Sahni, (Eds.). *Handbook on Data Structures and Applications* . New York: Chapman and Hall.

Pearson, J. W & Pestana, J. (2020). Preconditioners for Krylov subspace methods: An overview. *GAMM-Mitteilungen, 43*(4), 1–35.

Quarteroni, A. & Valli, A. (1999). *Domain Decomposition Methods for Partial Differential Equations*. Numerical Mathematics and Scientific Computation. Oxford: Oxford University Press.

Rajamanickam, S., Boman, E. G., & Heroux, M. A. (2012). ShyLU: A hybrid-hybrid solver for multicore platforms. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, (pp. 631–643).

Reid, J. K. (Ed.). (1971). *Large Sparse Sets of Linear Equations*. New York: Academic Press. Proceedings of the Oxford Conference Organized by the Institute of Mathematics and its Applications.

Reid, J. K. (1974). Direct methods for sparse matrices. In *Software for numerical mathematics (Proceedings of the Conference Institute of Mathematics Application of Loughborough University Technology, Loughborough)*, (pp. 29–47).

Reid, J. K. (1981). Frontal methods for solving finite-element systems of linear equations. In *Sparse matrices and their uses*, Institute of Mathematics and its Applications Conference Series, (pp. 265–281). London: Academic Press.

Reid, J. K. & Scott, J. A. (1999). Ordering symmetric sparse matrices for small profile and wavefront. *International Journal for Numerical Methods in Engineering, 45*(12), 1737–1755.

Reid, J. K. & Scott, J. A. (2006). Reducing the total bandwidth of a sparse unsymmetric matrix. *SIAM Journal on Matrix Analysis and Applications, 28*(3), 805–821.

Reid, J. K. & Scott, J. A. (2009). An out-of-core sparse Cholesky solver. *ACM Transactions on Mathematical Software, 36*(2), Art.9, 1–33.

Reid, J. K. & Scott, J. A. (2011). Partial factorization of a dense symmetric indefinite matrix. *ACM Transactions on Mathematical Software, 38*(2), Art. 10, 1–19.

Rennich, S. C., Stosic, D., & Davis, T. A. (2016). Accelerating sparse Cholesky factorization on GPUs. *Parallel Computing, 59*, 140–150.

Rose, D. J. (1973). A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Read, R. (Ed.), *Graph Theory and Computing* (pp. 183–217). New York: Academic Press.

Rose, D. J. & Tarjan, R. E. (1978). Algorithm aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics, 34*(1), 176–197.

Rose, D. J., Tarjan, R. E., & Lueker, G. S. (1976). Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing, 5*(2), 266–283.

Rose, D. J. & Willoughby, R. A. (Eds.). (1972). *Sparse Matrices and Their Applications*. New York: Plenum Press.

Rothberg, E. & Eisenstat, S. (1998). Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications, 19*(3), 682–695.

Rothberg, E. E. & Gupta, A. (1993). An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. *International Journal of High Speed Computing, 5*(04), 537–593.

Rotkin, V. & Toledo, S. (2004). The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software, 30*(1), 19–46.

Rozin, E. & Toledo, S. (2005). Locality of reference in sparse Cholesky factorization methods. *Electronic Transactions on Numerical Analysis, 21*, 81–106.

Rozložník, M. (2018). *Saddle-Point Problems and Their Iterative Solution*. Nečas Center Series. Cham: Birkhäuser/Springer.

Ruiz, D. (2001). A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England.

Saad, Y. (1985). Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing, 6*(4), 865–881.

Saad, Y. (1994a). ILUT: a dual threshold incomplete *LU* factorization. *Numerical Linear Algebra with Applications, 1*(4), 387–402.

Saad, Y. (1994b). SPARSKIT: a basic tool kit for sparse matrix computations - Version 2. https://www-users.cse.umn.edu/~saad/software/SPARSKIT/.

Saad, Y. (1996a). ILUM: a multi-elimination ILU preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing, 17*(4), 830–847.

Saad, Y. (1996b). *Preface to the First Edition of Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM.

Saad, Y. (2003a). Finding exact and approximate block structures for ILU preconditioning. *SIAM Journal on Scientific Computing, 24*(4), 1107–1123.

Saad, Y. (2003b). *Iterative Methods for Sparse Linear Systems* (2nd ed.). Philadelphia, PA: SIAM.

Saad, Y. & Suchomel, B. (2002). ARMS: an algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications, 9*(5), 359–378.

Saad, Y. & Zhang, J. (1999). BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing, 20*(6), 2103–2121.

Saunders, M. A. (1996). Cholesky-based methods for sparse least squares: the benefits of regularization. In L. Adams & J. L. Nazareth (Eds.), *Linear and Nonlinear Conjugate Gradient-Related Methods* (pp. 92–100). Philadelphia, PA: SIAM.

Schenk, O. & Gärtner, K. (2006). On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Transactions on Numerical Analysis, 23*, 158–179.

Schreiber, R. (1982). A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software, 8*(3), 256–276.

Scotch (accessed 2022). Scotch Version 7.0: a software package for graph and mesh/hypergraph partitioning, graph clustering, and sparse matrix ordering. https://www.labri.fr/perso/pelegrin/scotch/ and https://gitlab.inria.fr/scotch/scotch.

Scott, J. A. (1999). A new row ordering strategy for frontal solvers. *Numerical Linear Algebra with Applications, 6*(3), 189–211.

Scott, J. A. (2001). A parallel frontal solver for finite element applications. *International Journal for Numerical Methods in Engineering, 50*(5), 1131–1144.

Scott, J. A. & Tůma, M. (2014a). `HSL_MI28`: an efficient and robust limited-memory incomplete Cholesky factorization code. *ACM Transactions on Mathematical Software, 40*(4), Art. 24, 1–19.

Scott, J. A. & Tůma, M. (2014b). On positive semidefinite modification schemes for incomplete Cholesky factorization. *SIAM Journal on Scientific Computing, 36*(2), A609–A633.

Sharir, M. (1981). A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications, 7*(1), 67–72.

ShyLU Project Team (accessed 2022). ShyLU: A collection of node-scalable sparse linear solvers. https://trilinos.github.io/shylu.html.

Skiena, S. S. (2020). *The Algorithm Design Manual* (3rd ed.). Texts in Computer Science. Berlin: Springer.

Sloan, S. W. (1986). An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering, 23*(2), 239–251.

Smith, B. F., Björstad, P. E., & Gropp, W. D. (1996). *Domain decomposition*. Cambridge: Cambridge University Press.

Speelpenning, B. (1978). The generalized element method. Report UIUCDCS-R-78-946, Department of Computer Science, Urbana, Illinois: University of Illinois.

Spielman, D. A. & Teng, S.-H. (2007). Spectral partitioning works: Planar graphs and finite element meshes. *Linear Algebra and its Applications, 421*(2–3), 284–305.

Strang, G. (2007). *Computational Science and Engineering*. Wellesley, MA: Wellesley-Cambridge Press.

Stüben, K., Ruge, J. W., Clees, T., & Gries, S. (2017). Algebraic multigrid: from academia to industry. In M. Griebel, A. Schüller, & M. A. Schweitzer (Eds.), *Scientific Computing and Algorithms in Industrial Simulations* (pp. 83–119). Cham, Switzerland: Springer.

Tang, J. M., Nabben, R., Vuik, C., & Erlangga, Y. A. (2009). Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *Journal of Scientific Computing, 39*(3), 340–370.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing, 1*(2), 146–160.

Tarjan, R. E. (1983). *Data Structures and Network Algorithms*. *CBMS-NSF Regional Conference Series in Applied Mathematics*, vol. 44. Philadelphia, PA: SIAM.

Teng, S.-H. (1997). Fast nested dissection for finite element meshes. *SIAM Journal on Matrix Analysis and Applications, 18*(3), 552–565.

Tewarson, R. P. (1970). Computations with sparse matrices. *SIAM Review, 12*(4), 527–543.

Tewarson, R. P. (1973). *Sparse Matrices*. Mathematics in Science and Engineering, vol. 99. New York: Academic Press.

Tinney, W. F. & Walker, J. W. (1967). Direct solutions of sparse network equations by optimally ordered triangular factorization. In *Proceedings of the IEEE*, vol. 55, (pp. 1801–1809).

Tismenetsky, M. (1991). A new preconditioning technique for solving large sparse linear systems. *Linear Algebra and its Applications, 154*, 331–353.

Toselli, A. & Widlund, O. (2005). *Domain Decomposition Methods—Algorithms and Theory*. *Computational Mathematics*, vol. 34. Berlin: Springer.

Trefethen, L. N. (1985). Three mysteries of Gaussian elimination. *ACM SIGNUM Newsletter, 20*(4), 2–5.

Trefethen, L. N. & Bau, III, D. (1997). *Numerical Linear Algebra*. Philadelphia, PA: SIAM.

Tůma, M. (2002). A note on the LDL^T decomposition of matrices from saddle-point problems. *SIAM Journal on Matrix Analysis and Applications, 23*(4), 903–925.

Tuff, A. D. & Jennings, A. (1973). An iterative method for large systems of linear structural equations. *International Journal for Numerical Methods in Engineering, 7*(2), 175–183.

Uçar, B. & Aykanat, C. (2007). Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review, 49*(4), 595–603.

van der Sluis, A. (1969). Condition numbers and equilibration of matrices. *Numerische Mathematik, 14*(1), 14–23.

van der Vorst, H. & Van Dooren, P. (Eds.). (2015). *Parallel Algorithms for Numerical Linear Algebra, Reprint edition*. Advances in Parallel Computing. Amsterdam: Elsevier Science.

van der Vorst, H. A. (2003). *Iterative Krylov methods for Large Linear Systems*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge: Cambridge University Press.

Varga, R. S. (1960). Factorizations and normalized iterative methods. In *Boundary problems in differential equations*, (pp. 121–142). Madison, WI: University of Wisconsin Press.

Varga, R. S., Saff, E. B., & Mehrmann, V. (1980). Incomplete factorizations of matrices and connections with H-matrices. *SIAM Journal on Numerical Analysis, 17*(6), 787–793.

Von Neumann, J. & Goldstine, H. H. (1947). Numerical inverting of matrices of high order. *Bulletin of the American Mathematical Society, 53*(11), 1021–1099.

Wathen, A. J. (2015). Preconditioning. *Acta Numerica, 24*, 329–376.

Watkins, D. S. (2002). *Fundamentals of Matrix Computations* (2nd ed.). Pure and Applied Mathematics. New York: Wiley-Interscience.

Watts-III, J. W. (1981). A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Society of Petroleum Engineers Journal, 21*, 345–353.

Wendland, H. (2017). *Numerical Linear Algebra: An Introduction*. Cambridge Texts in Applied Mathematics. Cambridge: Cambridge University Press.

Wilkinson, J. H. (1948). Progress report on the Automatic Computing Engine. Report MA/17/1024, Mathematics Division, Department of Scientific and Industrial Research, National Physical Laboratory, Teddington, UK.

Wilkinson, J. H. (1961). Error analysis of direct methods of matrix inversion. *Journal of the Association for Computing Machinery, 8*, 281–330.

Wilkinson, J. H. (1963). *Rounding Errors in Algebraic Processes*. Englewood Cliffs, NJ: Prentice Hall. Reprinted by Dover, New York, 1994.

Wilson, R. J. (1996). *Introduction to Graph Theory* (4th ed.). Advances in Parallel Computing. Prentice Hall, Addison Wesley Longman Limited.

WSMP (2020). Watson Sparse Matrix Package (WSMP Version 20.12). http://researcher.watson.ibm.com/researcher/view_group.php?id=1426.

Xu, J. & Zikatanov, L. (2017). Algebraic multigrid methods. *Acta Numerica, 26*, 591–721.

Ye, X., Xi, Y., & Saad, Y. (2021). Proxy-GMRES: preconditioning via GMRES in polynomial space. *SIAM Journal on Matrix Analysis and Applications, 42*(3), 1248–1267.

Zlatev, Z. (1991). *Computational Methods for General Sparse Matrices*. Dordrecht: Kluwer Academic Publishers.

Zoltan (2022). Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services (v3.901). https://sandialabs.github.io/Zoltan/.

Index

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6