NUMERICAL MATHEMATICS
AND SCIENTIFIC COMPUTATION

Direct Methods for Sparse Matrices

Second Edition

I. S. DUFF, A. M. ERISMAN,
AND J. K. REID

OXFORD SCIENCE PUBLICATIONS

GLOSSARY OF SYMBOLS

$\mathbf{A}$	User's original matrix.
$\mathbf{A}^T$	Transpose of $\mathbf{A}$.
$\mathbf{A}^{(k)}$	Reduced matrix (of order $n \times n$) before step k of Gaussian elimination.
$\mathbf{A}^{[k]}$	Matrix (order $n \times n$) associated with k^{th} finite element.
$\mathbf{A}_{i:}$	Row i of the matrix $\mathbf{A}$.
$\mathbf{A}_{:j}$	Column j of the matrix $\mathbf{A}$.
$\mathbf{B}, \mathbf{C}, \ldots$	Other matrices.
$\mathbf{b}$	Right-hand side vector.
$\mathbf{c}$	Intermediate solution vector.
c_k	Depending on the context, either component k of $\mathbf{c}$ or column count (number of entries in column k of matrix).
$\mathbf{D}$	Diagonal matrix.
$\mathbf{e}_i$	i^{th} column of $\mathbf{I}$.
$\mathbf{I}$	Identity matrix.
$\mathbf{L}$	Lower triangular matrix, usually formed by triangular factorization of $\mathbf{A}$.
$\mathbf{L}\backslash\mathbf{U}$	$\mathbf{LU}$ factorization of a matrix packed into a single array.
n	Order of the matrix $\mathbf{A}$.
$\mathcal{O}()$	If $f(n)/g(n) \to k$ as $n \to \infty$, where k is a constant, then $f(n) = \mathcal{O}(g(n))$.
$\mathbf{P}$	Permutation matrix (usually applied to matrix rows).
$\mathbf{Q}$	Orthogonal matrix (usually a permutation matrix). Note that $\mathbf{QQ}^T = \mathbf{I}$.
$\mathbf{R}$	Upper triangular matrix in a $\mathbf{QR}$ factorization.
$\mathbf{r}$	Residual vector $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$.
r_k	Row count (number of entries in row k of matrix).
$\mathbf{U}$	Upper triangular matrix, usually formed by triangular factorization of $\mathbf{A}$.
u	Threshold for numerical pivoting.
$\mathbf{v}_i$	i^{th} eigenvector of $\mathbf{A}$.
$\mathbf{x}$	Solution vector.
ϵ	Relative precision.
$\kappa(\mathbf{A})$	Condition number of $\mathbf{A}$.
λ_i	i^{th} eigenvalue of $\mathbf{A}$.
ρ	Largest element in any reduced matrix, that is $\max_{i,j,k} \lvert a_{ij}^{(k)} \rvert$.
τ	Number of entries in the matrix $\mathbf{A}$.
$\in$	Belongs to.

NUMERICAL MATHEMATICS AND SCIENTIFIC COMPUTATION

Series Editors

A.M. STUART E. SÜLI

NUMERICAL MATHEMATICS AND SCIENTIFIC COMPUTATION

Books in the series

Monographs marked with an asterisk (*) appeared in the series 'Monographs in Numerical Analysis' which is continued by the current series.

For a full list of titles please visit
https://global.oup.com/academic/content/series/n/numerical-mathematics-and-scientific-computation-nmsc/?lang=en&cc=gb

*J. H. Wilkinson: *The Algebraic Eigenvalue Problem*
*I. Duff, A. Erisman, and J. Reid: *Direct Methods for Sparse Matrices*
*M. J. Baines: *Moving Finite Elements*
*J. D. Pryce: *Numerical Solution of Sturm–Liouville Problems*

C. Schwab: *p- and hp- Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*
J. W. Jerome: *Modelling and Computation for Applications in Mathematics, Science, and Engineering*
A. Quarteroni and A. Valli: *Domain Decomposition Methods for Partial Differential Equations*
G. Em Karniadakis and S. J. Sherwin: *Spectral/hp Element Methods for Computational Fluid Dynamics*
I. Babuška and T. Strouboulis: *The Finite Element Method and its Reliability*
B. Mohammadi and O. Pironneau: *Applied Shape Optimization for Fluids*
S. Succi: *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*
P. Monk: *Finite Element Methods for Maxwell's Equations*
A. Bellen and M. Zennaro: *Numerical Methods for Delay Differential Equations*
J. Modersitzki: *Numerical Methods for Image Registration*
M. Feistauer, J. Felcman, and I. Straškraba: *Mathematical and Computational Methods for Compressible Flow*
W. Gautschi: *Orthogonal Polynomials: Computation and Approximation*
M. K. Ng: *Iterative Methods for Toeplitz Systems*
M. Metcalf, J. Reid, and M. Cohen: *Fortran 95/2003 Explained*
G. Em Karniadakis and S. Sherwin: *Spectral/hp Element Methods for Computational Fluid Dynamics, Second Edition*
D. A. Bini, G. Latouche, and B. Meini: *Numerical Methods for Structured Markov Chains*
H. Elman, D. Silvester, and A. Wathen: *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*
M. Chu and G. Golub: *Inverse Eigenvalue Problems: Theory, Algorithms, and Applications*
J.-F. Gerbeau, C. Le Bris, and T. Lelièvre: *Mathematical Methods for the Magnetohydrodynamics of Liquid Metals*
G. Allaire and A. Craig: *Numerical Analysis and Optimization: An Introduction to Mathematical Modelling and Numerical Simulation*
K. Urban: *Wavelet Methods for Elliptic Partial Differential Equations*
B. Mohammadi and O. Pironneau: *Applied Shape Optimization for Fluids, Second Edition*
K. Boehmer: *Numerical Methods for Nonlinear Elliptic Differential Equations: A Synopsis*
M. Metcalf, J. Reid, and M. Cohen: *Modern Fortran Explained*
J. Liesen and Z. Strakoš: *Krylov Subspace Methods: Principles and Analysis*
R. Verfürth: *A Posteriori Error Estimation Techniques for Finite Element Methods*
H. Elman, D. Silvester, and A. Wathen: *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics, Second Edition*
I. Duff, A. Erisman, and J. Reid: *Direct Methods for Sparse Matrices, Second Edition*

Direct Methods for Sparse Matrices

SECOND EDITION

I. S. DUFF

Rutherford Appleton Laboratory, CERFACS, Toulouse, France, and Strathclyde University

A. M. ERISMAN

The Boeing Company, Seattle (retired) and Seattle Pacific University

J. K. REID

Rutherford Appleton Laboratory and Cranfield University

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

First Edition published in 1986
Second Edition published in 2017

Impression: 1

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data
Data available

Library of Congress Control Number: 2016946839

ISBN 978–0–19–850838–0

Printed and bound by
CPI Group (UK) Ltd, Croydon, CR0 4YY

PREFACE

The subject of sparse matrices has its roots in such diverse fields as management science, power systems analysis, surveying, circuit theory, and structural analysis. Mathematical models in all of these areas give rise to very large systems of linear equations that could not be solved were it not for the fact that the matrices contain relatively few nonzeros. It has become apparent that the equations can be solved even when the pattern is irregular, and it is primarily the solution of such problems that we consider.

A great deal has changed since the first edition of this book was published 30 years ago. There has been considerable research progress. Simply updating the book to account for more recent results is important.

In addition, our world has become much more dependent on mathematical models for the complex designs we produce. To take just one example, The Boeing Company used wind tunnels as the primary design tool and mathematical models as a tool for independent insight in the design of large aircraft at the time of the first edition. Since that time, the mathematical model has become the primary design tool, and the wind tunnel is used for validation. In 1994, Boeing had the first test flight of the 777, its first all-digitally-designed aeroplane. This major transition raised the stakes on the importance of large-scale simulations. There has been a similar shift in many other fields.

Furthermore, models have been extended to other areas since 1986. At that time, it was common in aerodynamics, structural analysis, chemical processing design, electronics design, power systems design and analysis, weather forecasting, and linear programming to include a sparse matrix model at the core. Today, these same models continue to be developed, but are solved for significantly larger problems. In addition, sparse matrices appear in models for speech recognition, language processing, computer gaming, signal processing, big data, bioscience, social networks, and process simulations of all kinds. In many cases, these models started in off-line design applications, but have moved to real-time decision support applications. Thus, the rapid and reliable solution of large sparse systems is more important than ever.

On the other hand, computing has also advanced at a rapid rate. In over 25 years, it is safe to say that computing price performance has improved approximately 100,000 times based on Moore's Law. Chris Anderson, editor of Wired Magazine and author of Free, argues that computing has advanced to the point where waste is good. All of the work we did to reduce memory and to get the most out of computing cycles is (sometimes) no longer needed. Computing is so cheap that it is better not to take expensive people time to save what does not cost very much.

Since the essence of this book is to describe efficient algorithms for solving sparse matrix problems arising at the heart of most large-scale simulations, Anderson's argument may suggest this work is no longer needed. However, in this case he is wrong, as any scientist working in the field would already know.

Here is why. As the problem size and hence the matrix size grows, ignoring sparsity would cause the computing times to grow as the cube of the size of the problem. Thus, a problem that is 10 times as large will require 1000 times the computation. Computation for sparse problems is much more problem dependent, but we might expect a problem 10 times as large could require 30–50 times the computation. At the time of the first edition of the book in 1986, a problem with 10 000 variables was regarded as very large. Today, problems with 10 000 000 variables are solved and efficient algorithms for making this possible are discussed in this book.

The subject is intensely practical and we have written this book with practicalities ever in mind. Whenever two methods are applicable we have considered their relative merits, basing our conclusions on practical experience in cases where a theoretical comparison is not possible. We hope that the reader with a specific problem may get some real help in solving it. Non-numeric computing techniques have been included, as well as frequent illustrations, in an attempt to bridge the usually wide gap between the printed page and the working computer code. Despite this practical bias, we believe that many aspects of the subject are of interest in their own right and we have aimed for the book to be suitable also as a basis for a student course, probably at MSc level. Exercises have been included to illustrate and strengthen understanding of the material, as well as to extend it. In this second edition, we have introduced a few research exercises that will extend the reader's understanding even further and could be used as the basis for research.

We have aimed to make modest demands on the mathematical expertise of the reader, and familiarity with elementary linear algebra is almost certainly needed. Similarly, only modest computing background is expected, although familiarity with modern Fortran is helpful.

After the introductory Chapter 1, the computational tools required to handle sparsity are explained in Chapter 2. We considered making the assumption of familiarity with the numerical analysis of full matrices, but felt that this might provide a barrier for an unreasonably large number of potential readers and, in any case, have found that we could summarize the results required in this area without lengthening the book unreasonably. This summary constitutes Chapters 3 and 4. The reader with a background in computer science will find Chapter 2 straightforward, while the reader with a background in numerical analysis will find the material in Chapters 3 and 4 familiar. Note that the rest of the book is built around the basic material in these chapters.

Sparsity is considered in earnest in the remainder of the book. In Chapter 5, Gaussian elimination for sparse matrices is introduced, and the utility of standard software packages to realize the potential saving is emphasized. Chapters 6–9 are

focused on ordering methods to preserve sparsity. These include transforming a matrix to block triangular form when possible (Chapter 6), variations of local pivoting strategies to preserve sparsity (Chapter 7), orderings to achieve band and variable band form (Chapter 8), and dissection strategies (Chapter 9). The implementation of the solution of sparse systems using these orderings is the subject of Chapters 10–14. In these chapters, we consider the cases where the factorizations and solutions can be developed without concern for numerical values, as well as the cases where account of numerical values needs to be taken during factorization. We examine such implementations for modest to very large problems across a variety of computer architectures. Finally, in Chapter 15, we consider other exploitations of sparsity.

Since the previous edition, much work in sparse matrix research has involved the development of algorithms that exploit parallel architectures. We decided that, rather than have a single chapter on this, we would discuss parallel aspects relevant to the work in each chapter.

We use bold face in the body of the text when we are defining terms.

For the purposes of illustration, we include fragments of Fortran code conforming to the ISO Fortran 95 standard, and some exercises ask the reader to write code fragments in Fortran or another language of his or her choice. Where a more informal approach is needed, we use syntax loosely based on Algol 60, with ':=' meaning assignment. We use equal-width font for Fortran and normal font for informal code, sometimes using both in the same figure.

Throughout the book we present timing data drawn from a variety of computers to illustrate various points. This diversity of computers is due to the different papers, environments, and time periods from which our results are drawn. It does not cause a difficulty of interpretation because it is the relative performance within a particular area that is compared; absolute numbers are less meaningful.

We refer to many sparse matrix packages in the book and aim to provide somewhere (usually at its first occurrence) a short description and a reference. This will be indexed in bold so that whenever we refer to the package the reader will be able to find the description and reference easily.

Three appendices cover matrix norms, some pictures of sparse matrices from various applications, and solutions to selected exercises. These appendices precede the collected set of references and indices to authors and subjects.

We find it convenient to estimate operation counts and storage requirements by the term that dominates for large problems. We will use the symbol $\mathcal{O}$ for this. For instance, if a certain computation needs $\frac{1}{3}n^3 - \frac{1}{6}n(n+1)$ multiplications we might write this as $\frac{1}{3}n^3 + \mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$.

Efficient use of sparsity is a key to solving large problems in many fields. We hope that this book will help people doing research in sparse matrices and supply both insight and answers for those attempting to solve these problems.

Acknowledgements for the first edition

The authors wish to acknowledge the support of many individuals and institutions in the development of this book. The international co-authorship has been logistically challenging. Both AERE Harwell and Boeing Computer Services (our employers) have supported the work through several exchange visits. Other institutions have been the sites of extended visits by the authors, including the Australian National University and Argonne National Laboratory (ISD), Carnegie-Mellon University (AME), and the Technical University of Denmark (JKR).

The book was typeset at Harwell on a Linotron 202 typesetter using the TSSD (Typesetting System for Scientific Documents) package written by Mike Hopper. We wish to thank Harwell for supporting us in this way, the staff of the Harwell Reprographic Section for their rapid service, Mike Hopper for answering so many of our queries, and Rosemary Rosier for copying successive drafts for us to check.

Oxford University Press has been very supportive (and patient) over the years. We would like to thank the staff involved for the encouragement and help that they have given us, and for their rapid response to queries.

Many friends and colleagues have read and commented on chapters. We are particularly grateful to the editors, Leslie Fox and Joan Walsh, for going far beyond their expected duties in reading and commenting on the book. Others who have commented include Pat Gaffney, Ian Gladwell, Nick Gould, Nick Higham, John Lewis, and Jorge Moré.

Finally, we wish to thank our supportive families who accepted our time away from them, even when at home, during this lengthy project. Thanks to Diana, Catriona, and Hamish Duff; Nancy, Mike, Andy, and Amy Erisman; and Alison, Martin, Tom, and Pippa Reid.

Harwell, Oxon, England and I. S. D.
Seattle, USA, A. M. E.
May, 1986. J. K. R.

Acknowledgements for the second edition

The challenge of writing a second edition in a very active research area has been greatly assisted by the help of others. We wish particularly to thank the following people for reading parts of the draft and making comments that led to major improvements: Mario Arioli, Cleve Ashcraft, Tim Davis, Pat Gaffney, Abdou Guermouche, Anshul Gupta, Jonathan Hogg, Sherry Li, Bora Uçar, and Clément Weisbecker. We would also like to thank our colleagues Mario Arioli, Nick Gould, Jonathan Hogg, and Jennifer Scott for their encouragement and for the many discussions over the years about the algorithms that we describe.

Oxford University Press has been very supportive (and patient). We would like to thank the staff involved for the encouragement and help that they have given us, and for their rapid response to queries.

Modern technology has made this book far easier to revise than it was to write the first edition. Now we can exchange chapters by email instead of airmail taking 3 or 4 days at best. We typeset the first edition, but each page involved an expensive photographic master. Now the whole book can be typeset in seconds, viewed on the screen, and printed inexpensively.

Iain (ISD) wishes to acknowledge the support of the Technical University of Denmark and the Australian National University through exchange visits that facilitated the writing of this book. He would like to acknowledge the close collaboration with his colleagues and friends in France, particularly in Toulouse, at CERFACS and IRIT. Stimulating interactions and invitations to the juries of theses have helped him to keep abreast of developments in the field.

Al (AME) would like to thank his supportive wife Nancy, both in enabling the work and hosting visits during the project. He also thanks his two co-authors; he has worked with Iain and John since the early 1970s, values their different perspectives, and appreciates them as friends. In the course of writing this second edition, John has routed his trips to the US through Seattle for blocks of working time, and has hosted Al at his home in Benson.

John (JKR) would like to thank Al (AME) and his wife Nancy for the many occasions when he has stayed in their house in order to make progress with writing the book. He would like to thank his grand-daughter Poppy Reid for the careful work she did during 2 weeks of work experience at RAL. She selected test problems from the Florida collection, ran the HSL code `MA48` on them and collected statistics. Her results are the basis of eight tables, starting with Table 5.5.1.

All of us would like to thank our wives, Di Duff, Nancy Erisman, and Alison Reid for their support over so many years.

RAL-STFC, Oxon, England and — I. S. D.
Seattle, USA, — A. M. E.
March, 2016. — J. K. R.

CONTENTS

1

INTRODUCTION

The use of graph theory to 'visualize' the relationship between sparsity patterns and Gaussian elimination is introduced. The potential of significant savings from the exploitation of sparsity is illustrated by one example. The effect of computer hardware on efficient computation is discussed. Realization of sparsity means more than faster solutions; it affects the formulation of mathematical models and the feasibility of solving them.

1.1 Introduction

A matrix is sparse if many of its coefficients are zero. The interest in sparsity arises because its exploitation can lead to enormous computational savings and because many large matrix problems that occur in practice are sparse. How much of the matrix must be zero for it to be considered sparse depends on the computation to be performed, the pattern of the nonzeros, and even the architecture of the computer. Generally, we say that a matrix is **sparse** if there is an advantage in exploiting its zeros. This advantage is gained from both not having to store the zeros and not having to perform computation with them. We say that a matrix is **dense** if none of its zeros is treated specially.

This book is primarily concerned with direct methods for solving sparse systems of linear equations, although other operations with sparse matrices are also discussed. The significant benefits from sparsity come from solution time reductions, but more importantly from the fact that previously intractable problems can now be solved. These matrices may have hundreds of thousands or even millions of rows and columns.

Often, it is possible to gain insight into sparse matrix techniques by working with the graph associated with the matrix, and this is considered in Section 1.2. There is a well-defined relationship between the pattern of the nonzero coefficients of a square sparse matrix and its associated graph. Furthermore, results from graph theory sometimes provide answers to questions associated with algorithms for sparse matrices. We introduce this topic here in order to be able to use it later in the book.

To illustrate the potential saving from exploiting sparsity, we consider a small example in Section 1.3. Without going into detail, which is the subject of the rest of the book, we use this example to motivate the study.

When serial computers with a single level of memory were our only concern, there was a near-linear relationship between the number of **floating-point operations** (additions, subtractions, multiplications, and divisions) on real

Direct Methods for Sparse Matrices, second edition. I. S. Duff, A. M. Erisman, and J. K. Reid.

(or complex) values and the run time of the program. Thus, a program requiring 600 000 operations was about 20% more expensive than one requiring 500 000 operations unless there were unusual overheads. In such an environment, exploitation of sparsity meant reducing floating-point computation while keeping the overheads in proportion. Cache memories have introduced additional complications. It is necessary to consider the effect of data movement, since waiting for data to be loaded into cache takes much more time than a floating-point operation. Vector and parallel architectures, possibly using multicore chips, present further levels of difficulty for comparisons between algorithms. A brief discussion of modern architectures is contained in Section 1.4 and of computational performance in Section 1.5.

In Section 1.6, we discuss the formulation of mathematical models from the viewpoint of the exploitation of sparsity. If sparsity tools are available, it is useful to apply these tools in a straightforward manner within an existing formulation. However, going back to the model may make it possible to reformulate the problem so that more is achieved. This section is intended only to stimulate thinking along these lines, since a full discussion is outside the scope of this book. Finally, in Section 1.7, we explain the need for a collection of sparse matrix test problems.

1.2 Graph theory

Matrix sparsity and graph theory are subjects that can be closely linked. The pattern of a square sparse matrix can be represented by a graph, for example, and then results from graph theory can be used to obtain sparse matrix results. George and Liu (1981), among others, do this in their book. **Graph theory** is also a subject in its own right, and early treatments are given by König (1950) and Harary (1969). More recent references include Bondy and Murty (2008).

In this book, we use graph theory mainly as a tool to visualize what is happening in sparse matrix computation. As a result, we use only limited results from graph theory and make no assumption of knowledge of the subject by the reader. In this section, we introduce the basic concepts. Other concepts are introduced as they are used, for example, in Chapter 9.

For practical reasons, we do not necessarily exploit all the zeros. Particularly in the intermediate calculations, it may be better to store some zeros explicitly. We therefore use the term **entry** to refer to those coefficients that are handled explicitly. All nonzeros are entries and some zero coefficients may also be entries. In this discussion connecting a sparse matrix and its associated graph, we relate the graph and the entries.

A **directed graph** or **digraph** consists of a set of **nodes** (also called **vertices**) and directed **edges** between nodes. Any square sparse matrix pattern has an associated digraph, and any digraph has an associated square sparse matrix pattern. For a given square sparse matrix $\mathbf{A}$, a node is associated with each row/column. If a_{ij} is an entry, there is an edge from node i to node j in the directed graph. This is usually written diagrammatically as a line with an

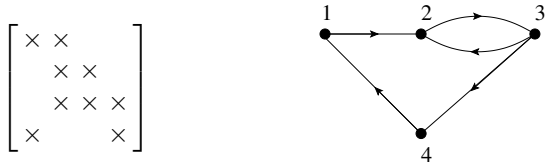

FIG. 1.2.1. An unsymmetric matrix and its digraph.

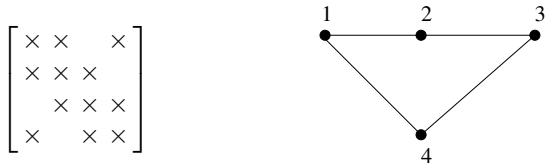

FIG. 1.2.2. A symmetric matrix and its graph.

arrow, as illustrated in Figure 1.2.1. For example, the line from node 1 to node 2 in the digraph corresponds to the entry a_{12} of the matrix. The more general representation of the directed graph includes self-loops on nodes corresponding to diagonal entries (see, for example, Varga (1962), p. 19). To make the visualization simpler, we do not include self loops for the common case where the diagonal coefficients of the matrix are all nonzero or the zeros are to be treated as entries. Where the distinction between zeros and nonzeros on the diagonal is required, self loops are needed to represent the nonzeros on the diagonal.

Formally, $G(\mathbf{A})$, the digraph associated with the matrix $\mathbf{A}$ is not a picture, but a set V of nodes and a set E of edges. An edge is an ordered pair of nodes (v_i, v_j) and is associated with the matrix entry a_{ij}. The associated picture of these nodes and edges is the usual way to visualize the graph.

For a symmetric matrix, a connection from node i to node j implies that there must also be a connection from node j to node i; therefore, a single line may be used and the arrows may be dropped. We obtain an **undirected graph** or **graph**, as illustrated in Figure 1.2.2. We say that a graph is connected if there is a path from any node to any other node.

An important special case occurs when a connected graph contains no closed path (**cycle**). Such a graph is called a **tree**. In a tree, we can pick any node as the root, to give a rooted tree, as illustrated in Figure 1.2.3, where the node labelled 5 is the root. From any node other than the root there is a unique path to the root. If the node is i and the next node on the path to the root is j, then i is called the **child** of j and j is called the **parent** of i. It is conventional to draw trees with the root at the top and with all the children of a node at the same height. A node without a child is called a **leaf**.

A digraph with no closed paths (cycles) is also important in sparse matrix work (see Section 13.7) and is known as a **directed acyclic graph** or **DAG**.

A fundamental operation used in solving equations with matrices involves adding multiples of one row, say the first, to other rows to make all entries in the first column below the diagonal equal to zero. This process is illustrated in

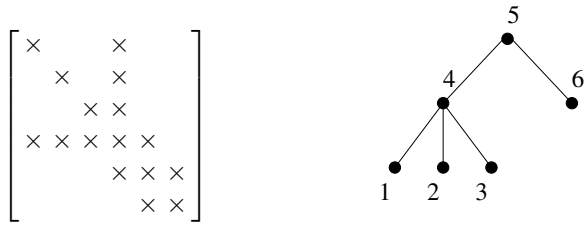

FIG. 1.2.3. A matrix whose graph is a rooted tree.

the next section. Detailed discussion of the algorithm, which is called Gaussian elimination, is found in Chapters 3 and 4. Notice that when this is applied to the matrix of Figure 1.2.1, adding a multiple of the first row to the fourth creates a new entry in position (4,2). This new entry is called a **fill-in**.

Graph theory helps in visualizing the changing pattern of entries as elimination takes place. Corresponding to the graph G, the elimination digraph G_y for node y is obtained by removing node y and adding a new edge (x, z) whenever (x, y) and (y, z) are edges of G, but (x, z) is not. For example, G_1 for the digraph of Figure 1.2.1 would have the representation shown in Figure 1.2.4, with the new edge $(4, 2)$ added. Observe that this is precisely the digraph corresponding to the 3×3 submatrix that results from the elimination of the (4,1) entry by the row transformation discussed in the last paragraph. In the case of a symmetric matrix whose graph is a tree, no extra edges are introduced when a leaf node is eliminated. The corresponding elimination operations introduce no new entries into the matrix.

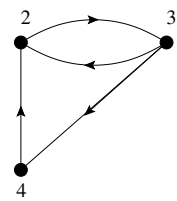

FIG. 1.2.4. The digraph G_1 for the digraph of Figure 1.2.1.

This relationship between graph reduction and Gaussian elimination was first discussed by Parter (1961). It is most often used in connection with symmetric matrices, since a symmetric permutation of a symmetric matrix leaves its graph unchanged except for the numbering of its nodes. For an unsymmetric matrix, it is often necessary to interchange rows without interchanging the corresponding columns. This leads to a different digraph, and the correlation between the digraph and Gaussian elimination is not so apparent (Rose and Tarjan 1978), see Exercise 1.6. If symmetric permutations are made, the digraph remains unchanged apart from the node numbering.

A graph for unsymmetric matrices that remains invariant even when the permutation is not symmetric is the **bipartite graph**. In this graph, the nodes

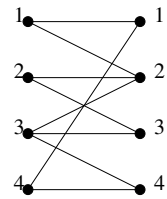

FIG. 1.2.5. The bipartite graph for the matrix in Figure 1.2.1.

are divided into two sets, one representing the rows and the other the columns, so that a row node, i say, is joined to a column node, j say, if and only if a_{ij} is an entry in the matrix. In Figure 1.2.5, we show the bipartite graph for the matrix in Figure 1.2.1.

1.3 Example of a sparse matrix

In the design of safety features in motor cars and aeroplanes, the dynamics of the human body in a crash environment have been studied with the aim of reducing injuries. In early studies, the simple stick figure model of a person illustrated in Figure 1.3.1 was used. Much more sophisticated models with many more variables are used today. However, this small model is ideal for tracing through the ideas behind the manipulation of a sparse matrix, and we use it for that purpose.

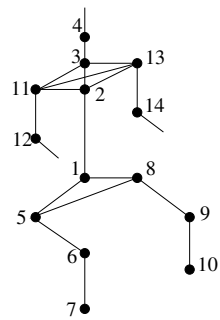

FIG. 1.3.1. Stick figure modelling a person.

The dynamics are modelled by a set of time-dependent differential equations with which we are not concerned here. The body segments are connected at joints (the nodes of the graph), as illustrated in Figure 1.3.1. These segments may not move independently since the position of the end of one segment must match the position of the end of the connecting segment. This leads to 42 equations of constraint (three for each numbered joint), which must be added to the mathematical system to yield a system of algebraic and differential equations to be solved numerically. At each time step of the numerical solution (typically there will be thousands of these), a set of 42 linear algebraic equations must be solved for the reactions at the joints.

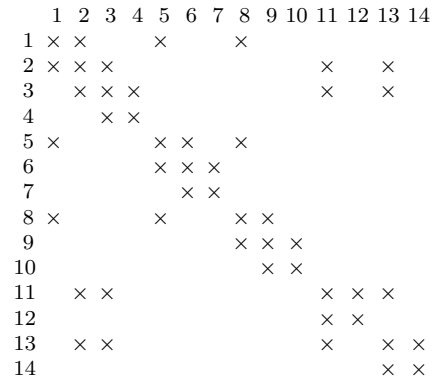

FIG. 1.3.2. The pattern of the matrix associated with the stick person of Figure 1.3.1.

Since there are 14 numbered joints, with three constraints for each joint, the matrix representing the algebraic equations may be considered as a 14×14 block matrix with entries that are 3×3 submatrices. This is the pattern shown in Figure 1.3.2, where each $\times$ represents a dense 3×3 submatrix. The pattern of this block matrix may be developed from Figure 1.3.1 by associating the given joint numbers with block numbers. For example, since joint 6 is connected by body segments to joints 5 and 7, there is a relationship between the corresponding reactions, and block row 6 has entries in block columns 5, 6, and 7.

Referring to the discussion of the previous section on the relationship between sparse matrices and graph theory, note that the stick person shown in Figure 1.3.1 corresponds to the graph of the matrix in Figure 1.3.2.

In the remainder of this section, we demonstrate that a careful utilization of the structure of a matrix as in Figure 1.3.2 leads to a significant saving in the computational cost of the solution of the algebraic equations. Since these equations must be solved at each time step, with the numerical values changing from step to step while the sparsity pattern remains fixed, cost savings can become significant over the whole problem. In Section 3.12, the relationship between block and single variable equations is established. Here it is sufficient to say that an efficient solution of linear equations whose coefficient matrix has the sparsity pattern of Figure 1.3.2 can readily be adapted to generate an efficient solution of the block equations. We therefore examine the solution of the 14 simultaneous equations whose sparsity pattern is shown in Figure 1.3.2.

Methods for solving systems of n linear equations

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \qquad i = 1, 2, ..., n, \tag{1.3.1}$$

or, in matrix notation,

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.3.2}$$

are discussed and compared in Chapters 3 and 4. For the purposes of this example, where $n = 14$, we simply solve the equations by a sequence of row operations. Multiples of equation 1 are first added to the other equations to eliminate their dependence on x_1. This leaves a revised system of equations of which only the first involves x_1. If no account is taken of the zeros, these calculations will require 29 floating-point operations for the elimination of x_1 from each of 13 equations, making 29×13 operations in all. Then multiples of the second equation are added to the later equations to eliminate their dependence on x_2. This requires 27 operations for each of 12 equations, making 27×12 operations in total. We continue until we reach the fourteenth equation, when we find that it depends only on x_{14} and may be solved directly. The total number of operations to perform this transformation (including operations on zeros) is

$$29\times13 + 27\times12 + 25\times11 + \cdots + 5\times1 = 1911. \tag{1.3.3}$$

The solution of the resulting trivial fourteenth equation requires only one division. Then the thirteenth equation may be solved for x_{13}, using the computed x_{14} (requiring three operations). Similarly $x_{12}, x_{11}, \cdots, x_1$ are computed in turn using the previously found components of $\mathbf{x}$. Thus, the calculation of the solution from the transformed equations requires

$$1 + 3 + \cdots + 27 = 196 \tag{1.3.4}$$

operations. The process of eliminating variables from each equation in turn and then solving for the components in reverse order is called **Gaussian elimination**. Its variants are discussed further in Chapter 3.

Following the same approach, but operating only on the entries, requires far less work. For instance, in the first step, multiples of equation 1 need only be added to equations 2, 5, and 8, since none of the others contain x_1. Furthermore, after a multiple has been calculated, only 4 multiplications (including one for the right-hand side) are required to multiply the first equation by it. Therefore 3 divisions, 12 multiplications, and 12 additions are needed to eliminate x_1. Note, however, that when a multiple of equation 1 is added to equation 2, new entries are created in positions 5 and 8. Figure 1.3.3 shows by bullets all the fill-ins that are created. The elimination of x_2 from each of the five other equations in which it now appears requires 1 division and 7 multiplications and causes fill-ins to equations 3, 5, 8, 11, and 13. The number of floating-point operations needed for the complete elimination is

$$2\times(13\times5) + 3\times(11\times4) + 4\times(9\times3) + 2\times(7\times2) + 2\times(5\times1) = 408. \tag{1.3.5}$$

The corresponding number of operations required to solve the transformed equations is

$$7 + 2\times11 + 3\times9 + 3\times7 + 2\times5 + 2\times3 + 1 = 94. \tag{1.3.6}$$

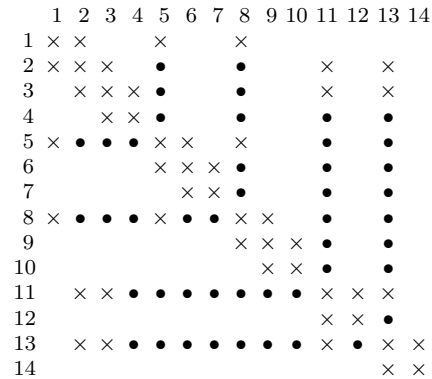

FIG. 1.3.3. The pattern of the matrix and its fill-ins. Fill-ins are marked •.

Thus, utilizing the sparsity in the solution process reduces the required number of operations from 2 107 to 502, a factor of more than four. For a large matrix, the factor will be far greater. This becomes particularly significant when the saving is repeated thousands of times. To realize these savings in practice requires a specially adapted algorithm that operates only on the entries. For a large matrix, it is also desirable to store only the entries. The development of data structures and special algorithms for achieving these savings is the major topic of this book.

Before leaving this example we ask another natural question. Can the equations be reordered to permit their solution with even fewer operations? One of the principal ordering strategies that we discuss in Chapter 7, when applied to the symmetric matrix of Figure 1.3.2, results in the reordered pattern of Figure 1.3.4. Here, the original numbering is indicated by the numbering on the

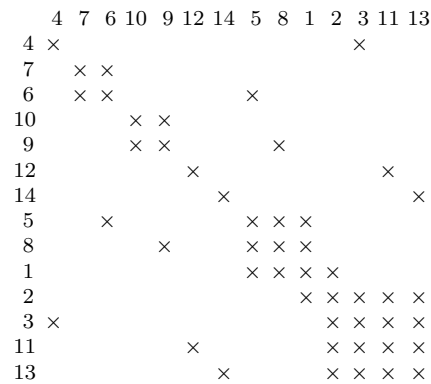

FIG. 1.3.4. The pattern of the reordered matrix.

rows and columns. The symmetry of the matrix is preserved because rows and columns are reordered in the same way.

If we carry through the transformation on the matrix of Figure 1.3.4, we observe that no new entries are created. Using cost formulae as before, the number of operations required is 105. Solving the transformed equations adds another 48 operations. The results of the three approaches are summarized in Table 1.3.1. The gain is more dramatic on large practical problems and, in Table 1.3.2, we show the same data for a problem from the Florida Sparse Matrix Collection (see Section 1.7).

Table 1.3.1 *Numbers of operations for Gaussian elimination on matrix of Figure 1.3.2.*

Treating matrix as	dense	sparse	
		unordered	ordered
Transformation cost	1 911	408	105
Solution cost	196	94	48

Table 1.3.2 *Operations for Gaussian elimination on* `onetone1`*, which has order 36 057 and 341 088 entries.*

Treating matrix as	dense	sparse	
		unordered	ordered
Transformation cost ($\times 10^9$)	31 251	17.6	3.6
Solution cost ($\times 10^6$)	1 300	33.4	5.3

Several comments on this reordering are appropriate. First, it produced an ordering where no new entries were generated; this is not typical. Secondly, the ordering is optimal for this problem (in the sense that another ordering could not produce fewer new entries) which also is not typical, because finding an optimal ordering is very expensive for a genuine problem. Thirdly, that this ordering is based on a practical approach and is able to produce a significant saving is typical. Fourthly, we assumed one ordering was better than another because it required fewer arithmetic operations. This is not necessarily true. The time required to solve a problem on a modern computer depends on much more than the amount of arithmetic to be done. This is discussed in Section 1.4.

In this discussion, the symmetry of the matrix has been ignored (apart from the use of a symmetric reordering). In Chapter 3, we introduce the Cholesky method, which exploits the symmetry and approximately halves the work for the elimination operations on the matrix. Also, we did not consider the effects of computer rounding in this illustration. These effects are discussed in Chapter 4.

This example gives an indication that exploiting sparsity can produce dramatic computational savings and that reordering can also be significant in producing savings. Methods of reordering sparse equations to preserve sparsity are the major topic of Chapters 7–9. The order of magnitude cost reduction

reflected in Table 1.3.1 for this example ***is not unusual***, and indeed much greater gains can be obtained on large problems, as illustrated by Table 1.3.2.

1.4 Modern computer architectures

Until the early 1970s, most computers were strictly serial in nature. That is, one arithmetic operation at a time ran to completion before the next commenced. Although some machines allowed operands to be prefetched (fetched before they are needed) and allowed some overlapping of instructions, the execution time of a program implementing a numerical algorithm could be well approximated by the formula

$$\text{time} = (\text{number of operations})/K, \tag{1.4.1}$$

where K is the theoretical peak performance in floating-point operations per second. The work in moving data between memory and functional units was either low or proportional to the computation time for the arithmetic.

Over the years, raw speed has dramatically increased, and has been measured in **Mflops** (millions of floating-point operations per second), **Gflops** (10^9 floating-point operations per second), **Tflops** (10^{12} floating-point operations per second) and even occasionally **Pflops** (10^{15} floating-point operations per second)[1]. Furthermore, computer architectures have evolved to multiple hierarchies of memory and perform parallel operations, sometimes at a massive scale, making such simple relationships no longer valid.

Arithmetic operations may be segmented into several (usually four or five) distinct phases and functional units designed so that different operands can be in different segments of the same operation at the same time. This technique, called **pipelining**, is an integral component of vector processing, since it is particularly useful when performing calculations with vectors. Nearly all current chips are able in some way to catch the essence of vector processing: they are able to execute operations on vectors of length n much faster than they can execute those operations on sets of n scalars. Vector processing was widely in use on high-performance computers when we wrote the first edition of this book and we gave it prominence.

Current computer architectures employ a memory hierarchy with several layers. Fastest is the level-1 cache, commonly denoted by ℓ_1, next the ℓ_2 cache, then the ℓ_3 cache and then the main memory. Data is moved between these levels in cache lines, typically 64 bytes, and special hardware is used to map main memory addresses to addresses in the cache and to move wanted data into the caches to replace data not needed at the time. In Table 1.4.1, we show figures for the Intel Sandy Bridge chip that is used in the desktop machine that we use for most of the runs in this book. Latency is the delay in the movement of data and is usually measured in terms of the number of clock periods needed. In the

[1]These are pronounced Megaflops, Gigaflops, Teraflops, and Petaflops, respectively. The production of an Exaflop computer (10^{18} floating-point operations per second) is expected in the early 2020s.

table, we give rounded numbers. In practice there is often a range, depending on the status of the data concerned. For example, if it has been referenced in the recent past, access might be faster. We skip these complications and just give an indication of the relative speeds and latencies in the table to show the importance of good cache management in algorithm design.

Table 1.4.1 *Cache sizes and speeds for the 8-core Intel Xeon E5-2687W. Clock speed is 3.1 GHz. Speed and latency are for transfer to ℓ_1 cache and the figures for ℓ_1 cache are for transfer to the CPU.*

Cache level	Size (bytes)	Speed (GBytes/s)	Latency clocks
1	8×32 KB	367	5
2	8×256 KB	86	12
3	20 MB	41	28
M	256 GB	10	189

If a computation involves data that is scattered in memory and few operations are performed on each datum, it will run far slower than the theoretical peak because most of the time will be spent in moving cache lines between different levels of memory. To obtain high performance, it is important to amortize the cost of moving data to the ℓ_1 cache by ensuring that once data are in the ℓ_1 cache, many operations are performed on them.

As complex as this may appear, it is simply illustrated with everyday examples. To illustrate hierarchical memory, think about the way sugar is used in a home. There is a bowl of sugar on the table which makes small amounts of sugar readily available. But if larger amounts of sugar are needed, then it takes a trip to the pantry to get a larger store of sugar. More time is required, but more sugar is available. To get sugar in even larger amounts, you could drive to the local market and buy some. The time for retrieval is much higher, but the available quantity is much greater. The computer model assumes that data must pass through the various stages, so the sugar from the pantry would be first used to refill the sugar bowl on the table, and the sugar would still be taken from the bowl, for example.

Pipelining is readily illustrated by a manufacturing assembly line for automobiles. Creating a single automobile in this environment would take a great deal of time for it to pass through all of the stages of production. However, by filling the assembly line (pipeline) with automobiles, after the first one comes out, the second one follows in just the time it takes to perform each stage of production.

For parallel computers, the situation with data access becomes even more complicated. In shared memory processing, the unit of execution is the **thread** and any item in memory is directly addressable from every thread. Conflicts occur

when two threads wish to access the same item. When there are many threads, the hardware to maintain shared access becomes very complicated and makes an accurate determination of the cost of fetching data essentially impossible. Note that a single silicon chip now normally has many cores and each core supports one or more threads.

In distributed memory machines, the unit of execution is the **process** where each process has its own memory, and access to its own memory is much faster than to the memory of another process. To complicate things further, many architectures have a hybrid structure where data is exchanged between processes by message passing, but each process has several threads accessing shared memory. We note that the word 'process' is also used within this book in its normal English sense. Which meaning is intended is clear from the context and should not cause any confusion.

Distributed memory parallelism can be illustrated by the challenge of writing this book. Each author has his own brain corresponding to a process, and work is parcelled out so that each can work independently on his own piece. However, to create the whole, from time to time the work must be brought together through message passing (email) or brought into a common memory (a personal visit or a Skype session). Results are achieved most effectively when the independent work is large, the message passing is small, and the meetings represent a small percentage of the overall project.

1.5 Computational performance

A key operation in linear algebra is the multiplication of two dense matrices, or more generally, the operation

$$\mathbf{C} := \mathbf{C} + \alpha\mathbf{A}\mathbf{B}, \tag{1.5.1}$$

for a scalar α and matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ of orders (say) $m\times k$, $k\times n$, and $m\times n$. This can be performed very effectively if $\mathbf{B}$ and $\mathbf{C}$ are stored by columns (the Fortran way) and there is room in the ℓ_1 cache for $\mathbf{A}$ and a few columns of $\mathbf{B}$ and $\mathbf{C}$. $\mathbf{C}$ is computed column by column. For each column, $2m$ reals are moved into the cache, $(2k+1)m$ floating-point operations are performed, and m real results are stored to memory and are no longer required in the cache. If k is reasonably large, say 32, the data movement time will be small compared with the arithmetic time. There is no need for the cache to hold $\mathbf{B}$ or $\mathbf{C}$—indeed there is no limit on the size of n. The system will bring the columns into cache as they are needed and may indeed be able to overlap the computation and data movement. This is known as **streaming**.

For the highest computational performance, it is necessary to exploit any parallelism available in the computer. We will make use of it at various levels. For the first, the computation is arranged to involve fully independent subproblems for which code can execute independently. We might permute a sparse set of equations to the form

$$\begin{bmatrix} \mathbf{A}_{11} & & & & \mathbf{A}_{15} \\ & \mathbf{A}_{22} & & & \mathbf{A}_{25} \\ & & \mathbf{A}_{33} & & \mathbf{A}_{35} \\ & & & \mathbf{A}_{44} & \mathbf{A}_{45} \\ \mathbf{A}_{51} & \mathbf{A}_{52} & \mathbf{A}_{53} & \mathbf{A}_{54} & \mathbf{A}_{55} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \\ \mathbf{b}_4 \\ \mathbf{b}_5 \end{bmatrix}, \tag{1.5.2}$$

which allows us to begin by working independently on the four subproblems

$$\begin{bmatrix} \mathbf{A}_{ii} & \mathbf{A}_{i5} \\ \mathbf{A}_{5i} & \mathbf{A}_{55}^{i} \end{bmatrix} \begin{bmatrix} \mathbf{y}_i \\ \mathbf{z}_i \end{bmatrix} = \begin{bmatrix} \mathbf{b}_i \\ \mathbf{b}_5^{i} \end{bmatrix}, \ i = 1, \ldots, 4. \tag{1.5.3}$$

where $\sum_{i=1}^{5} \mathbf{A}_{55}^{i} = \mathbf{A}_{55}$ and $\sum_{i=1}^{5} \mathbf{b}_5^{i} = \mathbf{b}_5$.

Such a block structure can occur though the use of a nested dissection ordering, as discussed in Chapter 9, or when domain decomposition is used in the numerical solution of partial differential equations. Parallelism can also be exploited at the level of the still sparse submatrices, $\mathbf{A}_{ij}$. Finally, we show in Chapters 11–13 how dense kernels can be used at the heart of a sparse code. For example, consider the matrix multiplication problem of the previous paragraph. If the problem is large enough, the data may be redistributed across the processes so that they share the work and each works on submatrices for which streaming occurs.

When discussing the merits of various algorithms throughout the remainder of this book, we will use the formula (1.4.1) where appropriate. K represents the theoretical peak performance of the computer. Attaining the peak performance would require maximally efficient use of the memory hierarchy, which is rarely possible for sparse problems. The data structures in a sparse problem are unlikely to use the pipelines, buffers, and memory hierarchies without inefficiencies. For this reason, we compare performance on different computer architectures and different problems using cases from the Florida Sparse Matrix Collection (see Section 1.7) and elsewhere. In the case of dense vectors and matrices, the outlook is more predictable. Here the basic manipulations have been tuned to the various architectures using the BLAS (Basic Linear Algebra Subprograms). The Level 1 BLAS (Lawson, Hanson, Kincaid, and Krogh 1979) are for vector operations, the Level 2 BLAS (Dongarra, Du Croz, Hammarling, and Hanson 1988) are for matrix-vector operations, and the Level 3 BLAS (Dongarra, Du Croz, Duff, and Hammarling 1990) are for matrix-matrix operations. Optimized versions of these subroutines are available from vendors. Of particular importance are the Level 3 BLAS because they involve many arithmetic operations for each value that is moved through caches to the registers. Although the BLAS are crucial in the efficient implementation of dense matrix computation, they also play an important role in sparse matrix computation because it is often possible to group much of the computation into dense blocks.

1.6 Problem formulation

If the potential gains of sparse matrix computation indicated in this chapter are to be realized, it is necessary to consider both efficient implementation and basic

problem formulation. For the dense matrix problem, the order n of the matrix controls the requirements for both the storage and solution time to solve a linear system. In fact, quite precise predictions of solution time ($\mathcal{O}(n^3)$) and storage ($\mathcal{O}(n^2)$) may be made for a properly formulated algorithm (see Section 3.11).

This type of dependence on n becomes ***totally invalid*** for sparse problems. This point was demonstrated for the example in Section 1.3, but it should be emphasized that it is true in general. The number of entries in the matrix, τ, is a more reliable indicator of work and storage requirements, but even using τ, precise predictions similar to those of the dense case are not possible.

For much of the book, we will be concerned with the implementation of the solution to the set of sparse linear equations

$$\mathbf{Ax} = \mathbf{b}, \tag{1.6.1}$$

where $\mathbf{A}$ is an $n \times n$, nonsingular matrix.

However, equation (1.6.1) arises from the formulation of the solution to a mathematical model. How that model is formulated can result in varying amounts of sparsity. Equations like (1.6.1) come from linear least squares problems, circuit simulation problems, control systems problems, and many other sources. In each case, seemingly innocent 'simplifications', which may be very helpful in the dense case, can destroy sparsity and make the solution of (1.6.1) either much more costly or infeasible.

It is the authors' belief that most very complicated physical systems have a mathematical model with a sparse representation. Thus, anyone concerned with the solution of such models must pay careful attention to the way the model is formulated. We will illustrate, though certainly not exhaustively, some examples from diverse applications in Chapter 15. The real benefits of sparsity have depended upon the formulation of the model as well as the choice of solution algorithm.

1.7 Sparse matrix test collections

Comparisons between sparse matrix strategies and computer programs are difficult because of the enormous dependence on implementation details and because the various ordering methods (introduced in Section 5.3 and discussed in detail in Chapters 7–9 are heuristic. This means that comparisons between them will be problem dependent. These concerns led to the development of a set of test matrices by Duff and Reid (1979) extended by Duff, Grimes, and Lewis (1989). A nice interface for accessing both the problems and associated statistics has been designed at NIST (MatrixMarket 2000) and the original Harwell-Boeing format was extended to the Rutherford-Boeing format by Duff, Grimes, and Lewis (1997). The GRID-TLSE project in Toulouse offers a web-friendly interface to direct sparse solvers and sparse matrix test problems (Amestoy, Duff, Giraud, L'Excellent, and Puglisi 2004).

Currently, the most extensive collection of sparse matrices is the University of Florida Sparse Matrix Collection (Davis and Hu 2011). Not only are the matrices

available in several formats, but there is a substantial discussion of each with several pictures, not just of the matrix. Additionally, there is software to enable the extraction of matrices with various characteristics including the application domain.

A major objective of the test collections has been to represent important features of practical problems. Sparse matrix characteristics (such as matrix size, number of entries, and matrix pattern including closeness to symmetry) can differ among matrices arising from, for example, structural analysis, circuit design, or linear programming. The test problems from these different application areas vary widely in their characteristics and often have very distinctive patterns. Pictures of some of these patterns are included in Appendix B. While their patterns are often not regular, they are certainly not random.

Throughout this book, we will use matrices to illustrate our discussion, to demonstrate the performance of our algorithms, and to show the effect of varying parameters. The default is that these will be matrices from the Florida Collection. Those not from this collection will be flagged by a footnote and the accompanying text will indicate their origin. Where we show the numbers of entries in a table, we count explicit zeros because the analysis performed may be required later for another matrix having the same pattern where some of these entries are now nonzero. We count the entries in both the upper and lower triangular parts of a symmetric matrix, although some algorithms require only one of the triangles. These test matrices will be symmetric, unsymmetric, or even rectangular determined by the algorithm or code being considered. As some approaches exploit patterns that are nearly symmetric, we will sometimes display the **symmetry index**, as defined by Erisman, Grimes, Lewis, Poole Jr., and Simon (1987), which is the proportion of off-diagonal entries for which there is a corresponding entry in the transpose, so that a symmetric matrix has a symmetry index of 1.0.

Another group of matrices that we use arises from solving partial differential equations on a square $q \times q$ grid, pictured for $q = 5$ in Figure 1.7.1. The simplest finite-element problem has square bilinear elements. Here, there is a one-to-one correspondence between nodes in the grid and variables in the system of equations. The picture of the grid is also a picture of the graph. The resulting symmetric matrix $\mathbf{A}$ has order $n = (q + 1)^2$, and each row has entries in nine columns corresponding to a node and its eight nearest neighbours. Such a pattern can also arise from a 9-point finite-difference discretization on the same grid. Another test case arises from the 5-point finite-difference discretization, in which case each row has off-diagonal entries in columns corresponding to nodes connected directly to the corresponding node by a grid line. Further regular problems arise from the discretization of three-dimensional problems using 7-point, 11-point, or 27-point approximations. These matrices are important because they typify matrices that occur when solving partial differential equations and because very large test problems can be generated easily.

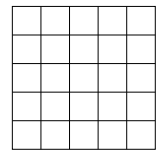

FIG. 1.7.1. A 5×5 discretization.

Another way of generating large sets of matrices is to use random number generators to create both the pattern and the nonzero values. Early testing of sparse matrix algorithms was done in this way. While small random test cases are useful for testing sparse matrix codes, we do not recommend their use for performance testing because of their lack of correlation to real problems.

When using an existing sparse matrix code for particular applications, the user will be confronted with a variety of choices as will be discussed in the rest of the book. Since many of the algorithms discussed here are based on heuristics, we recommend that users experiment with these choices for their own applications. The test problems provide an invaluable source for this purpose. Other readers may be more interested in extending research: which algorithms work best depending on size of matrix, application area, computer architecture, and many other factors? The test problems are useful for this purpose as well.

The Florida Collection will also be used in several of the Research Exercises that we have included in many of the chapters.

Exercises

1.1 For the machine described in Section 1.4, calculate the maximum performance of a code that moves data in 10 KB blocks and reuses it three times when in the ℓ_1 cache.

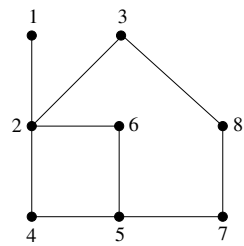

FIG. E1.1. A graph with 8 nodes.

1.2 For the graph shown in Figure E1.1, write down the corresponding sparse matrix pattern.

1.3 For the graph in Figure E1.1, re-label the nodes starting on the left working from top to bottom (1,2,4,3,6,5,8,7) and write down the corresponding matrix pattern. Compare this pattern with the one from Exercise 1.2, and comment on the differences and similarities.

1.4 Using the matrix of Exercise 1.2, find the pattern of the matrix that results from the elimination of the subdiagonal entries in columns one and two by adding multiples of rows one and two to appropriate later rows.

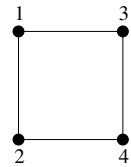

FIG. E1.2. A graph with 4 nodes.

1.5 Can you find a reordering for the sparse matrix corresponding to the graph shown in Figure E1.2 for which no new entries are created when solving equations with this as coefficient matrix?

1.6 For the matrix in Figure 1.2.1, reorder it by swapping rows 1 and 4. Draw its associated digraph, and compare it with the digraph in Figure 1.2.1. Comment on the difference and the similarities.

1.7 From Exercises 1.3 and 1.6, how does the relationship between the digraph of an unsymmetric matrix differ from the relationship between the symmetric matrix and its graph?

2

SPARSE MATRICES: STORAGE SCHEMES AND SIMPLE OPERATIONS

We consider how best to store sparse vectors, to add a multiple of one to another, and to form inner products. We also discuss the effective use of a full array even when the vector is sparse. We compare the different ways that sparse matrices can be held and discuss the use of linked lists. Sorting may be needed before or during a sparse matrix computation, so we discuss two sorting methods that we have found particularly useful: the counting sort and the heap sort. We consider grouping variables into supervariables, and we discuss matrix-vector and matrix-matrix products.

2.1 Introduction

The aim of the chapter is to examine data structures suitable for holding sparse matrices and vectors. We can only do this in conjunction with a consideration of the operations we wish to perform on these matrices and vectors, so we include a discussion of the simple kinds of operations that we need.

An important distinction is between **static** structures that remain fixed and **dynamic** structures that are adjusted to accommodate fill-ins as they occur. Naturally, the overheads of adjusting a dynamic structure can be significant. Furthermore, the amount of space needed for a static structure will be known in advance, but this is not the case for a dynamic structure. Both types are widely used in the implementation of sparse Gaussian elimination. We discuss the use of them in detail in Chapters 10–14.

Usually what is required is a very compact representation that permits easy manipulation. There is no one best data structure; most practical computer codes use different storage patterns at different stages.

2.2 Sparse vector storage

For simplicity, we begin by considering the storage of sparse vectors of order n. Some of our remarks generalize at once to sparse matrices whose columns (or rows) may be regarded as sets of sparse vectors. Furthermore, vector operations play an important role in Gaussian elimination (see Chapters 3 and 4).

A sparse vector may be held in a full-length vector of storage (length n). When the vector is very sparse, this is rather wasteful of storage, but is often used because of the simplicity and speed with which it may then be manipulated. For instance, the i-th component of a vector may be found directly.

To economize in storage we may pack the vector by holding the entries as real, integer pairs (x_i, i), one for each entry. In Fortran, we normally use a real array

Direct Methods for Sparse Matrices, second edition. I. S. Duff, A. M. Erisman, and J. K. Reid.

(for example, `value` in Figure 2.2.1) and a separate integer array (for example, `index` in Figure 2.2.1), each of length at least the number of entries. This is called the **packed** form. We call the operation of transforming from full-length to packed form a **gather** and the reverse a **scatter**.

```
product = 0.0
do k = 1,tau
   product = product + value(k)*w(index(k))
end do
```

FIG. 2.2.1. Code for the inner product between packed vector (`value`, `index`) with `tau` entries and full-length vector `w`.

To compare the storage requirements for the two methods, we first look at two examples. For a vector of length 10 000 with value 3.7 in location 90, and –4.2 in location 5 008, the vector would be represented by:

```
value = (3.7, -4.2)
index = (90, 5008)
```

Only four storage locations are required for this vector compared with 10 000 representing it as a full vector, a dramatic saving.

Now consider the case of the vector (3.2, 4.1, –5.3, 0). This requires four storage locations, but the corresponding vector in packed form would be represented by

```
value = (3.2, 4.1, -5.3)
index = (1, 2, 3)
```

In this case, the packed form requires more storage (six units) than the full-length vector (four units).

In general, it is easy to see that the packed form requires less storage when the vector is at least 50% sparse. When the numerical values in the vector are held in double-precision arrays, or if the values are single-precision complex numbers, then the break-even point will drop to 25%, and will drop even further in the extended-precision real or double-precision complex case. Thus, the packed form generally requires far less storage in practical computations, where the vectors, at least at the beginning of the computation, are far less dense than 25%.

Applying an elementary operation, say adding a multiple of component i to component j, is not convenient on this packed form because we need to search for x_i and x_j. A sequence of such operations may be performed efficiently if just one full-length vector of storage is used. For each packed vector, we perform the following steps:

(i) Place the nonzero entries in a full-length vector known to be set to zero.
(ii) For each operation:
   (a) Revise the full-length vector by applying the operations to it, and
   (b) if this changes a zero to a nonzero, alter the integer part of the packed form to correspond.

(iii) By scanning the integers of the packed form, place the modified entries back in the packed form while resetting the full-length vector to zero.

Notice that the work performed depends only on the number of entries and not on n. It is very important in manipulating sparse data structures to avoid complete scans of full-length vectors.[1] However, the above scheme does require the presence of a full-length real array and does not distinguish between default zeros and entries that have the value zero. One remedy is to use a full-length integer vector that is zero except for those components that correspond to entries. These hold the position of the entry in the sparse representation. This makes it almost as cheap to access entries and is equally cheap at identifying whether an entry is present. We illustrate the use of a full-length integer vector in Section 2.4.

2.3 Inner product of two packed vectors

Taking an inner product of a packed vector with a full-length vector is very simply and economically achieved by scanning the packed vector, as shown in the Fortran code of Figure 2.2.1. A fast inner product between two packed vectors may be obtained by first expanding one of them into a full-length vector. An alternative, if they both have their components in order, is to scan the two in phase as shown in the Fortran code of Figure 2.3.1.

```
product = 0.0;  kx = 1; ky = 1
do while (kx<=taux .and. ky<=tauy)
   if (index_y(ky)==index_x(kx)) then
      product = product + value_x(kx)*value_y(ky)
      kx = kx+1; ky = ky+1
   else if (ky>kx) then
      kx = kx+1
   else
      ky = ky+1
   end if
end do
```

FIG. 2.3.1. Code for the inner product between two packed ordered vectors (`value_x, index_x`) and (`value_y, index_y`) with `taux` and `tauy` entries.

2.4 Adding packed vectors

Another very important operation is that of adding a multiple of one vector to another, say

$$\mathbf{x} := \mathbf{x} + \alpha\mathbf{y} \tag{2.4.1}$$

If $\mathbf{x}$ and $\mathbf{y}$ are held as packed vectors with their components unordered, a satisfactory technique is again to use a full-length integer vector $\mathbf{p}$ known initially

[1] If this is done n times while factorizing a sparse matrix of order n, $\mathcal{O}(n^2)$ operations would be performed, which is likely to dominate the operation count.

to be zero. We then modify the packed form of $\mathbf{x}$ using $\mathbf{p}$ as a working vector but restoring it to zero as follows:

(i) For each entry y_k, place its position in the packed vector in p_k. For example, if the vector $\mathbf{y}$ had three entries, $y_{10},\ y_{15},\ y_1$ say, then entries 10, 15, and 1 in $\mathbf{p}$ would be set to 1, 2, and 3, respectively.

(ii) Scan $\mathbf{x}$. For each entry x_i check p_i. If it is nonzero, use it to find the value of y_i, modify x_i, and reset p_i to zero.

(iii) Scan $\mathbf{y}$. For each entry y_i check p_i. If it is nonzero, add a new component with value αy_i to the packed form of $\mathbf{x}$ (we have a fill-in). Reset p_i to zero.

Fortran code for these operations is shown in Figure 2.4.1. If their components

```
! Flag the entries.
  do k = 1,tauy
     p(index_y(k)) = k
  end do
! Modify the values that change, flagging each one.
  do k = 1,taux
     i = index_x(k)
     if (p(i)>0) then
        value_x(k) = value_x(k) + alpha*value_y(p(i))
        p(i) = 0
      end if
  end do
! Add the fill-ins at the end.
  do k = 1,tauy
     i = index_y(k)
     if(p(i)>0) then
        taux = taux + 1
        value_x(taux) = alpha*value_y(k)
        index_x(taux) = i
        p(i) = 0
      end if
  end do
```

FIG. 2.4.1. Code for the operation $\mathbf{x} := \mathbf{x} + \alpha\mathbf{y}$.

are in order, an in-phase scan of the two packed vectors, analogous to that shown in Figure 2.3.1, may be made. Note that fill-ins need not occur at the end of the packed form, so we can be sure that the order is preserved only if a fresh packed form is constructed (so that the operation is of the form $\mathbf{z} := \mathbf{x} + \alpha\mathbf{y}$). We set the writing of such code as Exercise 2.1. Perhaps surprisingly, it is more complicated than the unordered code. In fact, there appears to be little advantage in using ordered packed vectors except to avoid needing the full-length vector $\mathbf{p}$, but even this may be avoided at the expense of slightly more complicated code (see Exercise 2.2).

An alternative approach is as follows:

(i) For each entry x_k, place its position in the packed vector in p_k.

(ii) Scan $\mathbf{y}$. For each entry y_i, check p_i. If it is nonzero, use it to find the position of x_i and modify it, $x_i := x_i + \alpha y_i$; otherwise, add i to the packed form of $\mathbf{x}$ and set $x_i := \alpha y_i$.

(iii) Scan the revised packed form of $\mathbf{x}$. For each i, set $p_i := 0$.

While it is slightly more expensive than the first approach, since $\mathbf{x}$ is likely to have more entries than $\mathbf{y}$, it offers one basic advantage. It permits the sequence of operations

$$\mathbf{x} := \mathbf{x} + \sum_j \alpha_j \mathbf{y}^{(j)} \tag{2.4.2}$$

to be performed with only one execution of steps (i) and (iii). Step (ii) is simply repeated with $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, ...$, while the indices for $\mathbf{x}$ remain in $\mathbf{p}$. Step (ii) may be simplified if it is known in advance that the sparsity pattern of $\mathbf{x}$ contains that of $\mathbf{y}$.

We will see later (Chapter 10) that there are times during the solution of sparse equations when each of these approaches is preferable.

2.5 Use of full-sized arrays

In the last three sections, we have considered the temporary use of a full-length vector of integer storage to aid computations with packed sparse vectors. Provided sufficient storage is available, a simpler possibility is to use full-length real vectors all the time. Whether this involves more computer time will depend on the degree of sparsity, the operations to be performed and the details of the hardware. Recent tests, see Table 10.7.1, suggest that indirect addressing is only about 30% slower.

A further possibility is to use a full-length vector of storage spanning between the first and final entry. This allows execution that is almost as fast and may result in a very worthwhile saving in storage (and computation) compared with holding the whole vector in full-length storage.

A sparse matrix, too, may be held in full-length storage, either as a two-dimensional array or as an array of one-dimensional arrays. Similar considerations apply as for vectors. For a typical large sparse problem, where n is in the tens of thousands or even hundreds of thousands, most of the computation is done with vectors that are far less than 25% dense and these sparse storage schemes offer very significant advantages. Nevertheless, at some stages of the computation, we may encounter relatively dense subproblems for which these trade-offs are important.

2.6 Coordinate scheme for storing sparse matrices

Perhaps the most convenient way to specify a sparse matrix is as its set of entries in the form of an unordered set of triples (a_{ij}, i, j). In Fortran, these may be held

```
type entry
  integer :: row_index, col_index
  real    :: value
end type
```

FIG. 2.6.1. Derived type for the coordinate scheme.

as an array of derived type such as that in Figure 2.6.1 or as one real array and two integer arrays, all of length the number of entries. The matrix

$$\mathbf{A} = \begin{pmatrix} 1. & 2. & 0. & 0. & 5. \\ 0. & 0. & -3. & 4. & 0. \\ 0. & -2. & 0. & 0. & -5. \\ -1 & 0. & 0. & -4. & 0. \\ 0. & 3. & 0. & 0. & 6. \end{pmatrix}, \tag{2.6.1}$$

for example, might have the representation in Table 2.6.1. This matrix is 44% dense (11 entries in a 5×5 matrix). It is interesting that if reals and integers require the same amount of storage, less total storage is required by the full-sized matrix (25 words) than by the packed form (33 words). The position is reversed for double-length reals to the ratio 25:22 and an even bigger change (to 25:16.5) takes place with extended reals. The situation for complex-valued matrices is analogous to that of double-length reals.

Table 2.6.1 *The matrix (2.6.1) stored in the coordinate scheme.*

Subscripts	1	2	3	4	5	6	7	8	9	10	11
`row_index`	4	5	1	1	5	2	4	3	3	2	1
`col_index`	1	2	2	1	5	3	4	5	2	4	5
`value`	−1.	3.	2.	1.	6.	−3.	−4.	−5.	−2.	4.	5.

The major difficulty with this data structure lies in the inconvenience of accessing it by rows or columns. It is perfectly suitable if we wish to multiply by a vector in full-length storage mode to give a result also in full-length storage mode. However, the direct solution of a set of linear equations, for example, may involve a sequence of operations on the columns of the coefficient matrix. There are two principal storage schemes that provide ready access to this information: the collection of sparse vectors and the linked list. These we now describe.

2.7 Sparse matrix as a collection of sparse vectors

Our first alternative to the coordinate scheme is to hold a collection of packed sparse vectors of the kind described in Section 2.2, one for each column (or row). The components of each vector may be ordered or not. Since our conclusion in Section 2.4 was that there is little advantage in ordering them, we take them to be unordered.

The collection may be held as an array of a derived type with allocatable components, for example, `type sparse_vector` in Figure 2.7.1 or Figure 2.7.2.

```
type vector_entry
   integer :: index
   real    :: value
end type
type sparse_vector
  type(vector_entry), allocatable :: entry(:)
end type
```

FIG. 2.7.1. Derived type for sparse vectors, using a derived type for the entries.

```
type sparse_vector
  integer, allocatable :: index(:)
  real, allocatable    :: value(:)
end type
```

FIG. 2.7.2. Derived type for sparse vectors, using integers and reals.

The disadvantage is that the sizes of the columns vary during the execution. Each change of size will involve allocating a new array, copying the data, and deallocating the old array. It is therefore usual to pack the data into an integer and a real array or into an array of derived type.

For each member of the collection, we store the position of its start (in either array) and the number of entries. Thus, for example, the matrix (2.6.1) may be stored as shown in Table 2.7.1.

Table 2.7.1 *The matrix (2.6.1) stored as a collection of sparse column vectors.*

Subscripts	1	2	3	4	5	6	7	8	9	10	11
`len_col`	2	3	1	2	3						
`col_start`	1	3	6	7	9	12					
`row_index`	4	1	5	1	3	2	4	2	3	1	5
`value`	−1.	1.	3.	2.	−2.	−3.	−4.	4.	−5.	5.	6.

In this representation, the columns are stored in order (column 1, followed by column 2, followed by column 3, etc.). If we hold the start of an imagined extra column (column 6 in Table 2.7.1), it is not necessary to store both `col_start` and `len_col`, since `col_start` can be calculated from `len_col` or `len_col` can be calculated from `col_start`. If it is only necessary to access the matrix forwards or backwards, as is the case when a triangular factor is being held (see Section 10.5), then holding just `len_col` is satisfactory and has the advantage that the integers are smaller and so may fit into a shorter computer word. Where the columns may need to be accessed in arbitrary sequence, we may dispense with `len_col` and hold just `col_start`; holding just `len_col` would lead to a costly extra computation for finding entries in a particular column.

A basic difficulty with this structure is associated with inserting new entries. This arises when a multiple of one column is added to another, since the

new column may be longer, as discussed in Section 2.4. It is usual to waste (temporarily) the space presently occupied by the column and add a fresh copy at the end of the structure. Once the columns have become disordered because of this, both `len_col` and `col_start` are needed (in this case, the location of an extra imagined column is not needed). After a sequence of such operations, we may have insufficient room at the end of the structure for the new column, although there is plenty of room inside the structure. Here, we should 'compress' it by moving all the columns forward to become adjacent once more. It is clear that this data structure demands some 'elbow room' if an unreasonable number of compresses are not to be made.

2.8 Sherman's compressed index scheme

Some economy of integer storage is possible with the compressed index scheme of Sherman (1975), which we now describe. If we look at the matrix of Figure 2.8.1, our usual storage scheme would hold the off-diagonal entries as a collection of

$$\begin{array}{ccccc} \times & \times & 0 & \times & \times \\ & \times & 0 & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{array}$$

FIG. 2.8.1. A matrix pattern.

packed vectors (Section 2.7) as shown in Table 2.8.1. The compressed index scheme makes use of the fact that the tail of one row often has the same pattern as the head of the next row (particularly so for the factors after fill-in has occurred) and so the same indices can be reused. In order to do this, the length of each row (that is the number of off-diagonal entries) must be kept and we show the resulting data structure in Table 2.8.2. A similar gain is obtained automatically when using multifrontal schemes. Referring to Figure 14.2.1, we see that `NCOL*NPIV - NPIV*(NPIV-1)/2` entries will require only `NPIV+NCOL` indices.

Table 2.8.1 *Matrix pattern of Figure 2.8.1 stored as a collection of sparse row vectors.*

Subscript	1	2	3	4	5	6	7	8
`lenrow`	3	2	2	1				
`irowst`	1	4	6	8	9			
`jcn`	2	4	5	4	5	4	5	5

We indicate some of the gains obtained on practical problems in Table 2.8.3 where, if compressed storage were not used, the number of indices would be equal to the number of entries in the factors. Indeed, on the matrix from 5-point discretizaton on a square, see Section 1.7, the storage for the reals increases as