MATHEMATICS
IN SCIENCE
AND
ENGINEERING

Volume 99

Sparse Matrices

Reginald P. Tewarson

SPARSE MATRICES

REGINALD P. TEWARSON

Department of Applied Mathematics and Statistics
State University of New York
Stony Brook, New York

ACADEMIC PRESS New York and London 1973

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1

LIBRARY OF CONGRESS CATALOG CARD NUMBER: 72-88359

AMS (MOS) 1970 Subject Classifications: 65F05, 65F15, 65F25, 65F30

PRINTED IN THE UNITED STATES OF AMERICA

Preface

The purpose of this book is to present in a unified and coherent form the large amount of research material currently available in various professional journals in the area of computations involving sparse matrices. At present, results in this area are not available in book form, particularly those involving direct methods for computing the inverses and the eigenvalues and eigenvectors of large sparse matrices.

Sparse matrices occur in the solution of many important practical problems, e.g., in structural analyses, network theory and power distribution systems, numerical solution of differential equations, graph theory, as well as in genetic theory, behavioral and social sciences, and computer programming. As our technology increases in complexity, we can expect that large sparse matrices will continue to occur in many future applications involving large systems, e.g., scheduling problems for metropolitan fire departments and ambulances, simulation of traffic lights, pattern recognition, and urban planning.

My interest in the area of sparse matrices dates back to 1962–1964, when I helped design and write a computer code for solving linear programming problems for a major computer manufacturer. The matrices occurring in linear programming problems are generally large and sparse (they have few nonzero elements); therefore, in order to make the code efficient, only the nonzero elements of such matrices are stored and operated on. I found then that very little published material was available on computing the sparse factored form of inverses needed in the linear programming algorithms. This experience led to the publication of a number of research papers.

In the spring of 1968, I was invited to speak at a Symposium on Sparse Matrices and Their Applications held at IBM, Yorktown Heights, New York in September of the same year. Another invitation followed in 1969 to present a paper at a Conference on Large Sparse Sets of Linear Equations at Oxford University in April 1970. In the summer of 1969, I wrote a survey paper on computations with sparse matrices at the request of the editors of the Society of Industrial and Applied Mathematics, which appeared in the September 1970 issue of the SIAM Review. In the same year Professor R. Bellman suggested that I write a book on this subject. It was a happy coincidence when Professor L. Fox asked me to give a graduate seminar on Sparse Matrices during the Hillary term 1970 at Oxford University. Out of these lectures the book grew.

This book is intended for numerical analysts, computer scientists, engineers, applied mathematicians, operations researchers, and others who have occasion to deal with large sparse matrices. It is aimed at graduate–senior level students. It is assumed that the reader has had a course in linear algebra. I have tried to avoid a terse mathematical style at the cost of being at times redundant and attempted to strike a balance between rigor and application. I believe that applications should lead to generalizations and abstractions. As far as is possible, the algorithmic or constructive approach has been followed in the book.

I have given the basic techniques and recent developments in direct methods of computing the inverses and the eigenvalues and eigenvectors of large sparse matrices. I have avoided including material which is readily available in well-known texts in numerical analysis, except that needed as a basis for the material developed in this book.

The organization of the text is as follows.

In Chapter 1, several commonly used schemes for storing large sparse matrices are described, and a method for scaling matrices is given, such that in computations involving the scaled matrices, the round-off errors remain small.

In Chapter 2, a discussion of the well-known Gaussian elimination method is given. It is shown how the Gaussian elimination can be used to express the inverse of a given sparse matrix in a factored form called the Elimination Form of Inverse (EFI). Techniques are given for getting as sparse an EFI of a given sparse matrix as possible. Some methods for minimizing the total number of arithmetical operations in the evaluation of the EFI are also described. The storage and the use of the EFI in practical computations are discussed.

In Chapter 3, several methods are given for obtaining a reasonably sparse EFI. These methods do not require as much work as those in Chapter 2. The permutation of the given sparse matrix to one of the several forms (e.g., the band form) that are desirable for getting a sparse EFI is also discussed.

The Crout, Doolittle, and Choleskey methods, which are closely related to the Gaussian elimination method, are considered in Chapter 4. Techniques for minimizing the number of nonzeros created at each step for these methods are given; these techniques are naturally similar to those given in Chapters 2 and 3 for the Gaussian elimination method.

In Chapter 5, the well-known Gauss–Jordan elimination method is investigated, and it is shown how another factored form of inverse, called the Product Form of Inverse (PFI) can be obtained. The relation between the PFI and the EFI, as well as the techniques for finding a sparse PFI are also given.

The orthonormalization of a given set of sparse vectors by using the Gram–Schmidt, the Householder, or the Givens method is discussed in Chapter 6. The last two methods are also used in Chapter 7 for evaluating the eigenvalues and eigenvectors of sparse matrices. Another method in Chapter 7 makes use of a technique similar to the Gaussian elimination method to transform the given matrix. In both chapters, techniques are described which tend to keep the total number of new nonzeros (created during the computational process) to a minimum.

Finally, in Chapter 8, the relevant changes in the EFI or the PFI, when one or more columns of the given matrix are changed, are described. This happens in many applications, e.g., linear programming. Another factored form of the inverse, which is similar to the EFI, is also given.

A comprehensive bibliography on sparse matrices follows Chapter 8.

Acknowledgments

I would like to express my thanks to the following: Professor L. Fox, for inviting me to spend my sabbatical year at Oxford University, where most of this book was written; Professor R. Bellman, for his initiative in suggesting that this book be written, as well as for his advice and encouragement; Professor R. Joseph and my Ph.D. students—Mr. I. Duff (Oxford), Mr. Y. T. Chen, and Mr. K. Y. Cheng —for reading and suggesting various improvements in the manuscript.

Contents

CHAPTER

I

Preliminary Considerations

1.1. Introduction

In this introductory chapter, we shall first mention some of the areas of application in which sparse matrices occur and then describe some commonly used schemes for storing such large sparse matrices in the computer (internal and/or external storage). A simple method of scaling matrices in order to keep round-off errors small is also given. The chapter ends with a bibliography and related comments.

1.2. Sparse Matrices

A matrix having only a small percentage of nonzero elements is said to be *sparse*. In a practical sense an $n \times n$ matrix is classified as sparse if it has order of n nonzero elements, say two to ten nonzero elements in each row, for large n. The matrices associated with a large class of man-made systems are sparse. For example, the matrix representing the communication paths of the employees in a large organization is sparse, provided that the ith row and the jth column element of the matrix is nonzero if and only if employees i and j interact. Sparse matrices appear in linear programming, structural analyses, network theory and power distribution systems, numerical solution of differential equations, graph theory, genetic theory, social and behavioral sciences, and computer programming.

The current interest in, and attempts at the formulation and solution of problems in the social, behavioral, and environmental sciences (in particular as such problems arise in large urban areas; see, for example, Rogers, 1971) will in many cases lead to large sparse systems. If such systems are nonlinear, then their linearization—often the first step towards the solution—will result in still larger sparse systems.

Often, interesting and important problems cannot be solved because they lead to large matrices which either are impossible to invert on available computer storage or are very expensive to invert. Since such matrices are generally sparse it is useful to know the techniques currently available for dealing with sparse matrices. This allows one to choose the best technique for the type of sparse matrix he encounters. The time and effort required to develop the various techniques for handling sparse matrices is especially justified when several matrices having the same zero–nonzero structures but differing numerical values have to be handled. This occurs in many of the application areas already mentioned.

1.3. Packed Form of Storage

Large sparse matrices are generally stored in the computers in *packed form*; in other words, only the nonzero elements of such matrices with the necessary indexing information are stored. There are four reasons for utilizing the packed form of storage. First, larger matrices can be stored and handled in the internal storage of the computer than is otherwise possible. Second, there are cases when the matrix even in packed form does not fit in the internal storage (for example, in time sharing) and external storage (for example, tapes or discs) must be used. Generally, getting the data from the external storage is much slower than internal computations involving such data, therefore, the packed form is preferred for the external storage also. Third, a substantial amount of time is saved if operations involving zeros are not performed; this is done by symbolic processing in which only the nontrivial operations are carried out. This is often the only way in which large matrices can be reasonably handled. Fourth, it turns out that the inverse of a given matrix expressed as a product of elementary matrices (only the nontrivial elements of such matrices are stored in packed form) usually needs less storage than the explicit inverse of the matrix in packed form. Such factored forms of inverses are particularly advantageous when they are later used for multiplying several row and column vectors, in linear programming, for example.

There are various packing schemes available, some of which are described below; these have been found efficient and are incorporated in computer codes.

Let A be a square matrix of order n with τ nonzero elements, where $\tau \ll n^2$, then A is clearly sparse. Let the ith row and the jth column element of A be denoted by a_{ij}. In order to store only the nonzero elements $a_{ij} \neq 0$, we need to store i, j and a_{ij}. If one cell of the storage is used for each of these quantities, then a total of 3τ cells will be needed to store all the nonzero elements of A. Evidently, 3τ should be substantially less than n^2 to make it worthwhile to spend the extra effort and computing time involved in packing.

In many algorithms that transform A to some other desirable form, additional nonzero elements are created in the various steps of the computations. Therefore, in the packed storage some provision has to be made to add new nonzero elements to the various columns (or rows) of A as the computation proceeds and the elements get changed. The ideal storage would be one which minimizes both the total storage used and the total computation time. In general, the two requirements, minimum storage and minimum time, are incompatible and a trade-off must be made.

UTILIZATION OF LINKED LISTS IN PACKING

One way of storing the nonzero elements of the given sparse matrix A is by making use of *linked lists* as follows. Each nonzero element a_{ij} is stored as an *item* in its column j (see Fig. 1.3.1). An item is an ordered

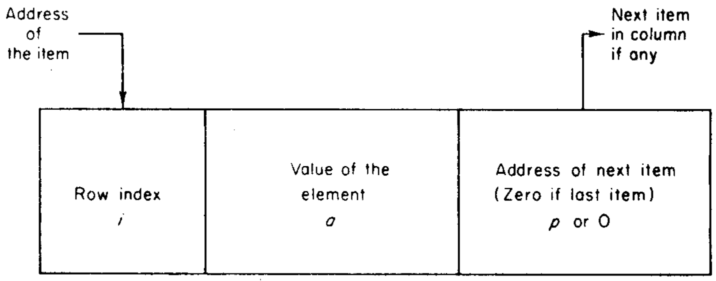

Fig. 1.3.1. The item corresponding to a_{ij}.

triple (i, a, p), where i is the row index, a the value of the element a_{ij} and p is the address of the next nonzero element of column j. The address p is zero if the item corresponds to the last nonzero element of the column. The total storage consists of two parts, BC (*Beginning of Column address*) and SI (*Storage for Items*). The first part BC has n contiguous locations, each of which contains the starting address of the first item of the corresponding column. For example, the jth cell of BC has the starting address SI(α) of the item associated with the first nonzero element in column j (see Fig. 1.3.2). SI, the second part, consists of all items associated with the nonzero elements of A. Since A has τ nonzero elements and to each element there corresponds an item which is three

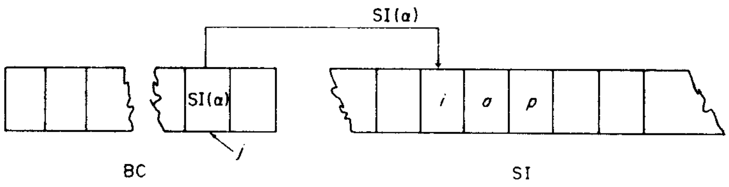

Fig. 1.3.2. Linked lists packed storage.

elements long, SI will require 3τ storage locations, which need not be necessarily contiguous. Therefore, if we use linked lists, a total of $n + 3\tau$ storage locations is required to store the given matrix A in packed form.

The principal advantage of this storage scheme is that during the computations nonzero elements created in the columns can be easily stored in SI; there is no need to move down all the following elements, as would be the case in the usual storage scheme when a new element is inserted. Furthermore, the cells in SI need not be contiguous, as long as they are in groups which are divisible by 3.

Let us give a simple example showing how the creation of a new nonzero element affects BC and SI. Suppose, $a_{13} = a_{33} = 0, a_{23} = 0.5$, and $a_{43} = 1.5$; the storage for BC begins at location 101, and the items corresponding to a_{23} and a_{43} begin at locations 200 and 203 respectively. If later on a_{33} changes from zero to nonzero (say 2.5), and the corresponding item is to be stored starting at location 300, then the relevant changes can be exhibited as follows:

Location	103	200	201	202	203	204	300	301	302
Present contents	200	2	0.5	203	4	1.5	—	—	—
New contents	200	2	0.5	300	4	1.5	3	2.5	203

Thus, in the existing matrix storage only the contents of location 202 had to be modified in order to add a new nonzero element. However, if instead of a_{33}, a_{13} became nonzero (say 3.5) and the corresponding item was stored (as before) starting at location 300, then we would have the following.

Location	103	200	201	202	203	204	300	301	302
Present contents	200	2	0.5	203	4	1.5	—	—	—
New contents	300	2	0.5	203	4	1.5	1	3.5	200

In either case, it is evident that the contents of only one location in the original linked list have to be modified to insert a new nonzero element.

If during the computations some nonzero element becomes zero, then the storage so released by the corresponding item can be used for storing the items associated with new nonzero elements. The starting addresses of such items which are available for storage can be maintained as a chained list by using the third cell of each item. Only the address of the first available item storage has to be noted elsewhere. The third cell of each available item storage contains the starting address of the next available storage item. If it is the last available item storage, then its third cell is zero. When a new item becomes available for storage it is added to the top of the list. Similarly, available items from the top of the list are used for storing new items.

Let us consider two simple examples to illustrate the above techniques. Suppose two items for storage were available, their starting addresses were, respectively, 101 and 201, and we want to add another available item storage starting at 301 to this list. If location 50 contains the address of the first available item storage, then the appropriate changes are shown below.

Location	50	101	102	103	201	202	203	301	302	303
Present contents	101	—	—	201	—	—	0	—	—	—
New contents	301	—	—	201	—	—	0	—	—	101

On the other hand, to store a new item, we use the first available item storage in the above list, namely, locations 301, 302, and 303, and then change the contents of the various locations in the preceding table to the line labeled as present contents.

Sometimes methods of packing which do not use linked lists are useful. They use less storage, but additional nonzero elements can be introduced only by relocating all the succeeding elements, that is, by moving them down. These schemes are suitable when only a small portion of the matrix can be stored in the internal storage of the computer at one time and a large amount of time would therefore be required to transfer the data to and from the external storage. We shall now describe four such schemes and show how a matrix A_5, whose nonzero elements are $a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24}$, and a_{45}, is stored according to the first three schemes. The last scheme is for symmetric matrices and therefore another matrix is utilized there. In the first three schemes the matrix is stored by columns but in the last one it is stored by rows.

SCHEME I

To each nonzero element of the matrix there corresponds an item of two storage cells. The first storage cell contains the row index and the second the value of the element. A zero row index in an item denotes the end of the current column. The second cell of such an item contains the index of the next column. Zeros in both cells of an item denote the end of the matrix storage. Thus there are $n + \tau + 1$ items in all, n for the columns, τ for the nonzero elements of A, and 1 to denote the end of the matrix storage. As each item uses two storage locations, a total of $2(n + \tau + 1)$ locations will be required to store A.

The matrix A_5 for which $\tau = 7$ and $n = 5$ is stored as the array

$(0, 1;\ 2, a_{21};\ 4, a_{41};\ 0, 2;\ 5, a_{52};\ 0, 3;\ 1, a_{13};\ 3, a_{33};\ 0, 4;\ 2, a_{24};\ 0, 5;$
$4, a_{45};\ 0, 0).$

SCHEME II

The information about the given matrix is stored in three arrays: VE (Value of Elements), RI (Row Indices), and CIP (Column Index Pointer). RI(α), the αth element of RI, contains the row index of the corresponding element VE(α) of VE. If the first nonzero element of the βth column of the given matrix is in VE(t_β) then t_β is stored in the βth element of CIP, namely, CIP(β) $= t_\beta$. It is evident that VE and RI each has τ elements but CIP has n elements. Thus $2\tau + n$ storage cells will be required in this scheme.

The storage for A_5 is as follows:

$$\begin{aligned} \mathrm{VE} &= (a_{21}, a_{41}, a_{52}, a_{13}, a_{33}, a_{24}, a_{45}), \\ \mathrm{RI} &= (2,\quad 4,\quad 5,\quad 1,\quad 3,\quad 2,\quad 4), \\ \mathrm{CIP} &= (1,\quad 3,\quad 4,\quad 6,\quad 7). \end{aligned}$$

The above storage scheme is easy to use. For example, a_{33} can be recovered as follows. Since CIP(3) = 4, RI(4) gives the row index of the first nonzero element of column 3. Then RI(4), or one of the succeeding elements of RI prior to the first nonzero element of column