AN INTRODUCTION TO SPARSE MATRICES

Derek O'Connor

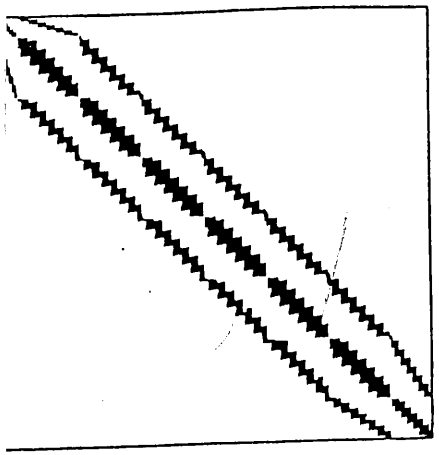
1. INTRODUCTION

There is no strict definition of a sparse matrix, but generally speaking it is an nxn matrix which has O(n) non-zero elements. A full or dense matrix has $O(n^2)$ non-zero elements.

Sparse matrices occur naturally in the solution of many practical problems, e.g. electrical, gas, and water distribution systems; civil and mechanical engineering (structural analysis); production and financial planning (inventory control and portfolio selection); national and local government operations (income tax analysis and scheduling of fire and ambulance services); economics (Input-Output analysis). At a more theoretical level sparse matrices arise in Graph Theory, Linear Programming, Finite Element Methods, and the solution of ordinary and partial differential equations. Duff's excellent survey article [6] contains a long list of application areas, with references.

The interest in sparse matrices comes about because of the need to solve on a computer, linear equations or linear optimization problems that have many thousands of variables (possibly millions) but whose coefficients are mostly zeros. Generally, this sparsity can be exploited, giving large savings in computer time and storage. Indeed, were it not possible to exploit sparsity then many important problems could not be solved on present or future computers.

The sparse matrices that arise in practice are not only sparse but are also highly structured, i.e. the non-zeros form very definite patterns. Figures 1.1 to 1.4 below are taken from Duff [7] and show some typical sparse matrices. This structure can be exploited also, giving even greater savings



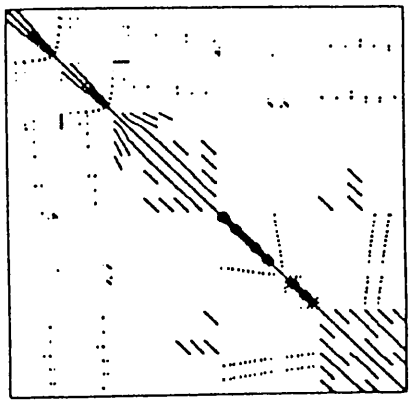
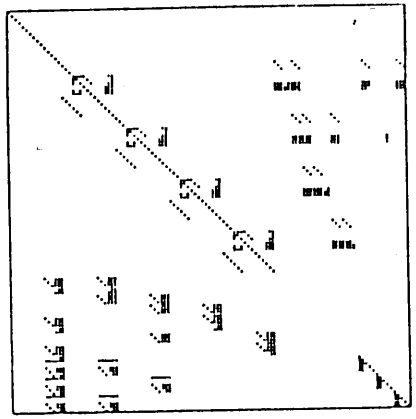


Figure 1.1: Matrix of Order 147 Figure 1.2: Matrix of Order 503



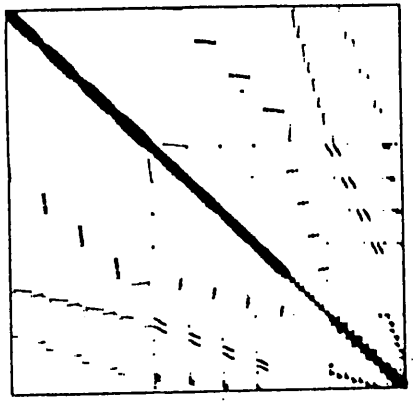


Figure 1.3: Matrix of Order 113 Figure 1.4: Matrix of Order 1005

in space and time.

An outline of this paper is as follows: Section 2 briefly outlines the history and goals of sparse matrix research; Section 3 discusses the standard schemes for sparse matrices in digital computers; Section 4 discusses direct and iterative methods for solving sets of sparse linear equations; Section 5 considers the special but very important class of symmetric positive definite matrices; Section 6 is the conclusion with a brief discussion of extensions to eigenvalue, least squares and optimization problems.

2. THE HISTORY AND GOALS OF SPARSE MATRIX RESEARCH

The interest in sparse matrices seems to have started in the late 1950s when researchers in linear programming and electrical power system analysis began to solve realistic problems on computers. It was noticed that when real problems were modelled by systems of linear equations, the resulting matrices were sparse with highly structured non-zero patterns. It was also noticed that these matrices were large and that the sparsity would need to be exploited if these problems were to be solved on the rather small and slow computers available at that time. Sparse matrices became increasingly important in the 1960s and the first conference on sparse matrices was held at the IBM Research Center, Yorktown Heights, in 1968 [24]. Since then there have been at least six international conferences, with published proceedings, devoted entirely to sparse matrices. The number of papers on sparse matrices is very large. Duff's survey paper [6] lists over 600 references up to the end of 1975 and a casual check of journals since then shows that this number is growing steadily. To date there are at least three textbooks on sparse matrices, while most textbooks on numerical analysis and data structures contain sections or chapters on the subject.

There are two complementary goals in sparse matrix research: (1) reduce the amount of computer memory needed to store sparse matrices, and (2) reduce the amount of computation time needed to solve problems involving sparse matrices. It is a general rule of thumb in computing that SPACE x TIME = CONSTANT, i.e. to save storage space more computation time must be spent and *vice-versa*. We will see that in solving sparse matrix problems it is often possible to make a simultaneous reduction in the amount of time and space required for the dense matrix case.

Some may wonder why it is important to reduce or conserve computer time and space when computers are getting larger and faster at lower costs each year (you can now buy a microcomputer for $5,000 which is as powerful as a mainframe which cost $ millions in the early '60s). Here are three reasons why conserving space and time is important:

* There is no computer available today that can store a 1000 x 1000 matrix in core. Many engineering and business problems have from 100,000 to 25,000,000 variables. Problems of this size are routinely solved but not without great care being taken to reduce their storage and time requirements.

* Sparse linear equation solvers are often a small but critical part of a much larger algorithm. As such, they may be used many thousands of times in solving a single problem. Any inefficiency in the equation solver will be greatly magnified and thus degrade the performance of the otherwise good, larger algorithm.

* Microcomputers are widely used today and in general they have small memories and slow processors. If sparsity techniques are used then these machines can solve realistic problems in the order of 100-1000 variables.

Thus it seems that, despite the increasing size and speed of computers, there is still - and perhaps always will be - a need

to conserve computer time and storage.

3. SPARSE MATRIX STORAGE SCHEMES

We now discuss some general schemes for compactly storing sparse matrices. These schemes should not be viewed in isolation from the algorithms which use them. In general, they must be tailored to suit the problem and the algorithm used to solve the problem. Thus the algorithm and the data structure (storage scheme) are intimately linked.

We start with the standard scheme for dense matrices and then discuss the three main methods for sparse matrices. For simplicity, we consider only square matrices.

3.1 Dense Matrix Storage

The memory of a digital computer can be regarded as a one-dimensional array or vector because each cell or word in memory is indexed or addressed by a single number, called the machine address.

A 2-dimensional array or matrix must be stored as a one-dimensional array and this is done by placing the columns (or rows) one after another in memory. This is shown in Fig. 3.1.

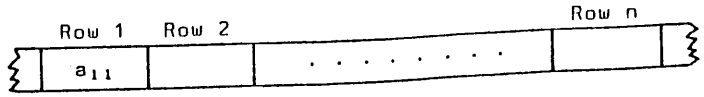


Figure 3.1: A Matrix Stored Row by Row

To access an element a_{ij} of a matrix A we must calculate its position relative to the location of the first element a_{11} of A. Thus,

$loc(a_{ij}) = loc(a_{11}) + n(i-1) + j-1$ Stored row by row, or

$loc(a_{ij}) = loc(a_{11}) + n(j-1) + i-1$ Stored column by column.

It is important to note that these calculations assume that all n^2 elements of the matrix are present and in a predetermined place relative to the first element. Also note that two extra cells are needed to store $loc(a_{11})$ and n and that each access requires one multiplication and two additions. This is called the 'overhead' of this storage scheme.

3.2 Static Sparse Matrix Storage

The dense scheme in the last section is appropriate if the number of non-zeros is high, because very little space is wasted in storing zeros. It should be remembered however, that with current computer technology, we are limited to storing matrices of order 300-500 in dense or full form.

In many practical applications, involving matrices of high order ($n > 500$), there may be only 2 to 20 non-zeros per row. Even if we could store such matrices in dense form we would be wasting most of the allocated space in storing zeros. Worse still, most of the arithmetic calculations would involve addition and multiplication of zeros, and these are *null* calculations, i.e. require no actual computation.

Most storage schemes for sparse matrices store only the non-zeros along with indexing information for each non-zero. This indexing information must be stored explicitly with each non-zero because the non-zeros will not be in predetermined positions relative to the first element of the matrix. This is in contrast to the dense scheme where no indexing information is stored (it is calculated).

We now describe three schemes for storing sparse matrices that are static, i.e. the number and locations of the non-zeros do not change during calculations. If this is not the case then we assume their number and locations at any point in a calculation can be determined before the calculation begins.

Storage Scheme 1:

Consider the 6 x 6 sparse matrix shown in Fig. 3.2.

$$A = \begin{vmatrix} 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 3 & 6 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 5 & 0 & 0 & 0 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

	1	2	3	4	5	6	7	8	9	10
VAL	2	1	4	3	6	3	8	5	9	1
COL	1	4	3	1	2	6	4	1	5	6
ROW	1	1	2	3	4	4	5	6	7	6

Figure 3.2: A Sparse Matrix and Storage Scheme 1

If the matrix has t non-zeros then we use 3 arrays of length t to store the non-zeros (VAL), the row position (ROW) and the column position (COL). This scheme requires 3t cells instead of n^2 for the dense scheme. Thus in a 1000 x 1000 matrix with a non-zero density ($t/n^2 \times 100\%$) = 5%, we would save $1000^2 - 3 \times 50{,}000 = 850{,}000$ cells.

Storage Scheme 2:

This scheme is obtained from the first scheme by observing that some of the row index numbers are repeated and hence redundant. This is because the elements of any single row are stored contiguously in a block of memory cells. As a result we need only know where each row block begins. This scheme is shown in Fig. 3.3 for the matrix A above.

	1	2	3	4	5	6	7	8	9	10
VAL	2	1	4	3	6	3	8	5	9	1
COL	1	4	3	1	2	6	4	1	5	6
ROW START	1	3	4	7	8	10				

Figure 3.3: A Sparse Matrix and Storage Scheme 2

This scheme requires two arrays of length t and one array of length n, giving a total of 2t+n cells. An alternative method is to store the number of non-zeros in each row instead of the row-start positions. This alternative scheme is used in the program given in the appendix.

Storage Scheme 3:

The third scheme saves storage by *packing* each row and column index of the first scheme into one cell, using the following calculation:

$$N_{ij} = n \times (i-1) + j$$

To retrieve or *unpack* the values of i and j we use the following calculation:

$$i = [(N_{ij}-1)/n] + 1,$$

$$j = N_{ij} - n \times (i-1).$$

This scheme requires only 2t cells but it involves more computation to pack and unpack the indices. An example of this storage scheme is shown in Fig. 3.4 again using the matrix A.

	1	2	3	4	5	6	7	8	9	10
VAL	2	1	4	3	6	3	8	5	9	1
N_{ij}	1	4	9	13	14	18	22	25	29	36

Figure 3.4: A Sparse Matrix and Storage Scheme 3

3.3 Dynamic Sparse Matrix Storage

In many algorithms involving matrices the number of non-zeros varies during the course of the computation. In addition to this the locations of newly created non-zeros or zeros are not known in advance. These two difficulties require that we be able to insert and delete elements of the matrix data structure (storage scheme).

The static storage schemes above do not allow efficient implementation of the 'insert' and 'delete' operations. This is because all non-zero elements are stored contiguously and inserting a new element in its appropriate place requires a movement of half the elements, on average.

The most common way of storing sparse matrices whose elements vary dynamically is to use some form of *linked list* scheme. Linked lists are fundamental to, and widely used in, computer science. In the context of sparse matrix storage, a linked list scheme requires that for each non-zero, we store its value (a_{ij}), its indices (i,j), and the address in memory (m) of the next non-zero element in the same row or column. This is in contrast to the static scheme in which the value and the indices are stored, and the dense scheme in which only the value is stored. The advantage of the linked list scheme is that each non-zero can be stored anywhere in memory, i.e. the non-zeros do not need to be stored contiguously, as in the static and dense schemes. Hence, inserting or deleting a non-zero does not disturb the other non-zeros.

To implement a linked list storage scheme we need to 'create' new storage cells and 'destroy' old cells as the number of non-zeros varies. We will not discuss where new storage cells come from or what is done with old cells except to say that these are 'house-keeping' functions which are handled by a 'memory manager'. Such memory managers may be part of the language (e.g. in Pascal) or may need to be programmed by the user/designer (e.g. in FORTRAN). Suffice it to say that the

memory manager must efficiently recycle old cells and make available new cells. For a good discussion of memory management see Knuth [14] or Horowitz and Sahni [12].

3.4 Operations on Sparse Matrices

Writing software to perform the standard matrix operations is easy when matrices are stored in dense form. All scientific programming languages allow the user to declare matrices as 2-dimensional arrays and to perform arithmetic operations on the individual matrix elements in a direct way. Thus in Pascal, the sum of two nxn real matrices A, B, is computed as follows:

```
Var A, B, C : Array [1..n, 1..n] of Real;
        i, j : Integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            C[i,j] := A[i,j] + B[i,j]
end;
```

The above program segment is essentially a direct translation of the mathematical definition of matrix addition. Furthermore, it is clear from the program that matrix addition is being performed. Writing the equivalent program for sparse matrices compactly stored is a much more difficult job. Also, reading and understanding such a program is difficult because the matrices are no longer represented in a direct and obvious way. These comments are true in general for sparse matrix software and for this reason such software is written only by those who have a sufficient knowledge of both mathematics and computer science. A list of the best-known sparse matrix software is given in the appendix.

3.5 Literature on Sparse Matrix Storage

The literature on sparse matrix storage schemes is scattered throughout books and journals in engineering, numerical linear algebra, computer science and operations research. The book by Jennings [13], and the survey article by Pooch and Nieder [15] are good starting points. Also, many computer science texts on Data Structures contain sections on sparse matrix storage (see [12], [14] and [23]).

4. SOLVING SPARSE LINEAR EQUATIONS

We now consider the solution of the equation

$$Ax = b,$$

where x, b are vectors in R^n and A is an n x n sparse matrix.

The algorithms for solving this equation fall into two broad categories, viz., Direct and Iterative. Iterative algorithms (Gauss-Seidel, Jacobi, etc.) are easy to implement (program) but may suffer from slow convergence. Direct algorithms (Gaussian Elimination, LUP Decomposition, etc.) give a solution in a finite number of steps but are difficult to implement, especially when the sparsity of A is exploited.

4.1 Iterative Algorithms

Iterative algorithms require that the equation Ax = b be transformed into the *'Fixed Point'* form

$$x = Cx + d = T(x),$$

which is then solved by *successive approximations* using the iteration formula

$$x^{(k+1)} = Cx^{(k)} + d, \quad x^{(0)} = 0.$$

This iteration formula (or algorithm) generates a sequence of

ectors $(x^{(1)}, x^{(2)}, \ldots, x^{(k)}, \ldots)$, which, under suitable onditions on C, converges to x, the solution of Ax = b.

This general iterative algorithm has very nice features nd it is easy to implement sparse versions of it. These eatures are:

1. Transforming A to C is simple and if A is sparse then C is sparse.
2. The matrix C does not change during the iteration process and hence an efficient static data structure can be used.
3. Programs for iterative algorithms tend to be short and simple.
4. Roundoff error is not generally a problem.

'he main difficulty with iterative algorithms is that they may ıave very slow linear convergence. This is why direct methods ;end to be preferred. Nonetheless iterative methods are use- 'ul for extremely large problems or where good initial solut- ons are available. In fact it is good practice to 'polish')r refine a direct solution with one or two iterations of an ıterative algorithm (see Rice [17] and Forsythe and Moler [9]).

The program shown in the appendix is a straightforward ORTRAN implementation of the Gauss-Seidel iterative method sing the static sparse storage scheme No. 2 given above. It an be seen that the heart of the program requires only 10 ines of code (lines 30-39). This program was run on a Z80, -bit, 64 K Byte microcomputer and solved a 1200 variable prob- em in 2.5 minutes. The A matrix had 3 non-zeros per row and 6 iterations were required to get 7-digit accuracy. Thus we ee that even very slow and small microcomputers can solve ealistic problems.

4.2 Direct Algorithms

Direct algorithms usually use some variant of the classic Gaussian Elimination method. This method transforms the system $Ax = b$ into $Ux = y$, where U is upper-triangular. $x = U^{-1}y$ is then found by *back-substitution*.

A more general statement of the Gaussian Elimination Algorithm is as follows:

LU - Decomposition to Solve $Ax = b$

Step 1: Decompose (Factorize) A, i.e. find L and U such that $LU = A$, where U is upper triangular and L is unit lower triangular.

Step 2: Solve $LUx = b$ as follows:

a. Solve $Ly = b$ by Forward-Substitution.
b. Solve $Ux = y$ by Back-Substitution.

Roundoff errors can accumulate in Step 1, and if A is *ill-conditioned* these errors will be greatly magnified. To control these errors, some form of *pivoting* (row or column interchanges or both) is necessary. Hence we normally obtain an LU factorization of PA or PAQ where P and Q are *permutation matrices*. These permutation matrices are determined *during the factorization*.

Exploiting the sparsity of the A matrix is difficult when using direct algorithms. This is because L and U may be dense, even though A is sparse. This is called the *fill-in problem*. Fill-in can be controlled by permuting rows and columns but these permutations may affect the pivoting scheme and hence increase the roundoff. Thus, in choosing P and Q we must balance the competing requirements of roundoff-error and fill-in control. An additional complication is that P and Q cannot be determined before the factorization process begins. Therefore we cannot predict where fill-in will occur. Thus we are

forced to use a *dynamic* storage scheme which allocates storage as fill-in occurs.

5. SPARSE, SYMMETRIC POSITIVE DEFINITE MATRICES

Symmetric positive definite (SPD) matrices do not suffer from the problems outlined in Section 4 above. This is fortuitous because many real, physical systems are modelled by very large, sparse SPD matrices.

5.1 Some Computational Properties of SPD Matrices

1. Gaussian Elimination becomes Cholesky's Method and yields the triangular factorization

$$A = LL^t,$$

where L is lower triangular with positive diagonal elements.

2. No pivoting is required to control roundoff errors, i.e. the diagonal elements, a_{ii}, are stable pivots.

3. A symmetric permutation of A is symmetric, positive definite, i.e. if A is SPD then PAP^t is SPD, where P is a permutation matrix.

Properties 2 and 3 allow us to find a P which reduces fill-in *before* we begin the factorizing, without worrying about pivoting to control roundoff error. Thus we can predict fill-in and allocate storage accordingly.

We can now break a direct algorithm for SPD matrices into four independent steps as shown below in Fig. 5.1.

The ability to break the direct algorithm into four independent steps has enormous practical implications. We are free to look for the best possible ordering P to minimize fill-in.

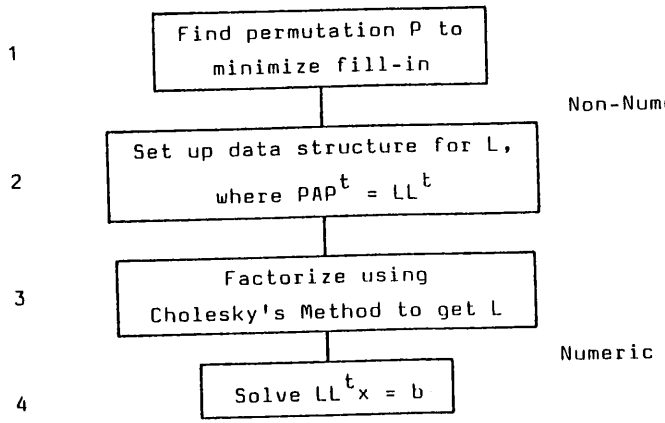


Figure 5.1: Direct Algorithm for Sparse, SPD Systems

A static storage scheme can be used because the factorization will cause only that fill-in predicted in Step 1. Different problems, with the same non-zero structure can be solved using only the numeric Steps 3 and 4. Thus the cost of the non-numeric Steps 1 and 2 can be amortized over a set of problems. The software for each step can be designed and developed independently. This helps to 'modularize' the complete program and yields a more reliable, useful, and versatile software package.

5.2 Ordering an SPD Matrix to Reduce Fill-In

The focus of much research in the last 10 years has been on finding P to minimize fill-in. The problem of finding a good ordering or permutation P is called the *Ordering Problem*. A very thorough exposition is given in the book by George and Liu [11].

We illustrate the ordering problem with the 6 x 6 symmetric matrix shown in Fig. 5.2(a), where the non-zeros are indicated by asterisks. It is assumed that the matrix is positive definite and hence pivoting on the diagonal elements will not cause roundoff problems.

Using standard Gaussian Elimination, variable 1 is eliminated from rows 2,...,6 by subtracting suitable multiples of row 1 from rows 2,...,6. Variable 2 is eliminated from rows 3,...,6 by subtracting suitable multiples of row 2 from rows 3,...,6. Eventually the upper triangular matrix U is obtained, which is shown in Fig. 5.2(b). We note that the number of non-zeros in A and U are 16 and 21 respectively. In general, for this type of matrix, the numbers of non-zeros will increase from 3n-2 to $(n^2+n)/2$, using the same pivot order. Now, if we symmetrically permute the A matrix so that row 1 becomes row 6, var 1 becomes var 6, etc., then we get the matrix PAP^t shown in Fig. 5.2(c). Performing the elimination process on this matrix gives the matrix U' shown in Fig. 5.2(d), which has only 2n-1 non-zeros. Thus the number of additional non-zeros generated by the elimination process has been drastically reduced.

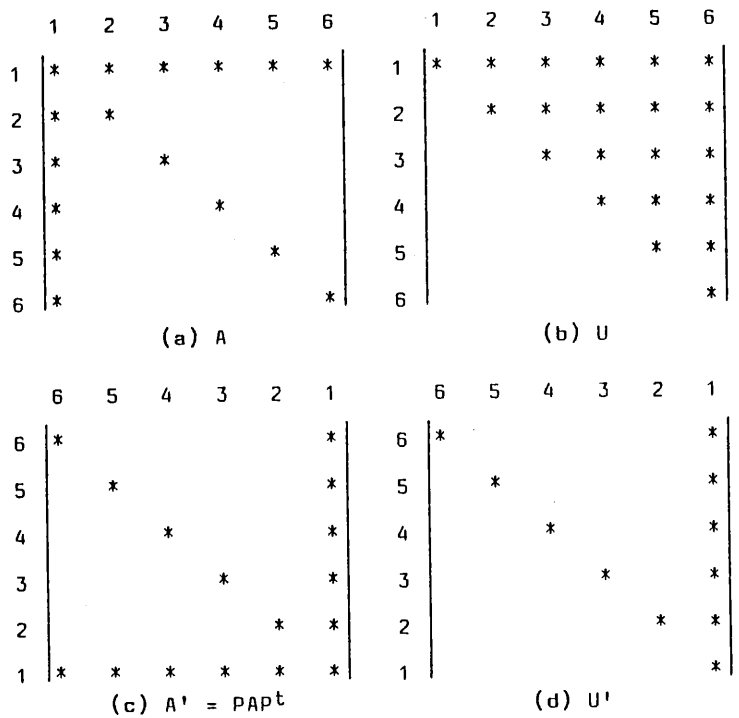


Figure 5.2: Elimination with Different Pivot Orders

The example above shows that a judicious ordering of pivots (choice of P) leads to no increase in the number of non-zeros during elimination. In general it is not possible to find a P so that no fill-in occurs. It is possible to find a P to minimize fill-in, but even this is difficult because it has been shown [19] that this problem is *NP-Complete* (see [10]) and so is essentially intractable.

Although we cannot hope to minimize fill-in there are practical algorithms that tend to reduce fill-in. At present there are four general types of ordering algorithms:

1. Band and Envelope Methods.
2. Mimimum Degree Methods.
3. Quotient Tree Methods.
4. Dissection Methods.

We will not describe the methods because they require a good knowledge of graph algorithms and theory. The first method is designed for band-like SPD matrices; the second for general SPD matrices; and the third and fourth for SPD matrices arising in finite element problems. Figs 5.3, 5.4 and 5.5 show the effects of the first two methods on a 35 x 35 SPD matrix. These figures are reproduced from [11] which gives a complete description and analysis of all four methods.

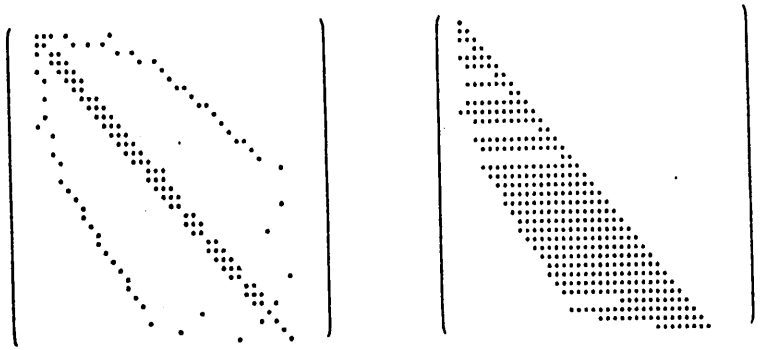


Figure 5.3: Unordered Matrix A and its Factor L

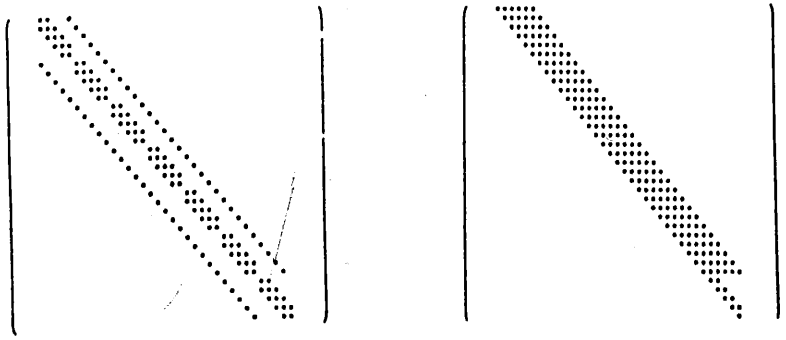


Figure 5.4: Band-Ordered Matrix A' and its Factor L'

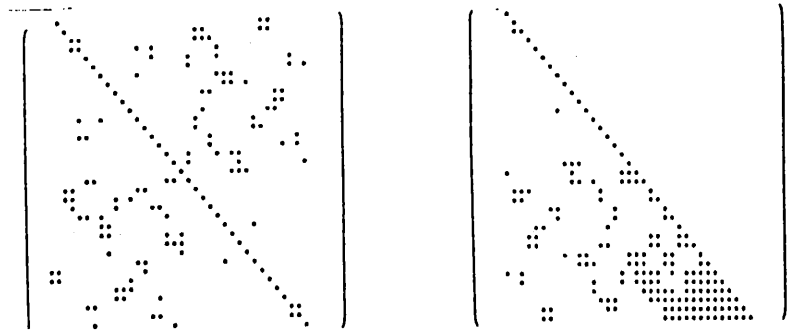


Figure 5.5: Minimum Degree-Ordered Matrix A" and its Factor L"

6. EXTENSIONS AND CONCLUSION

In this paper we have outlined the problems in storing and solving large sparse sets of linear equations. We have seen that substantial reductions in both storage and computation time can be achieved by the proper application of sparsity techniques. Many real problems give rise to matrices that are large and sparse, and these problems cannot be solved by the naive application of standard (dense) matrix methods. Hence it is important that both theoretical and applied scientists be aware of the large body of research, experience, and software that exists in this area of applied mathematics.

Although 'sparse matrices' may seem a rather narrow and specialized topic, it has developed into a very active and broad area of research, with contributions from chemists, computer scientists, engineers, mathematicians, operations researchers and physicists. It draws on topics such as linear algebra, numerical analysis, graph theory, combinatorics, data structures and software design. Research in the area ranges from the theoretical (see [3]) to the practical (see [7]).

The general techniques of sparse matrix storage and factorization are also applied to eigenvalue problems, least squares and linear programming. Special techniques for linear programs of the minimum-cost-flow type have been particularly successful (see [5]), with Barr and Turner [4] solving a network flow problem containing 25,000,000 variables. Sparse matrix techniques are also important in differential, integral, and non-linear equation-solving and in optimization problems. This is because most solution methods use some form of local or piecewise linearization which gives rise to sparse matrices.

Software for sparse matrix problems is not easy to design and develop. In general a sparse matrix subroutine in FORTRAN is 3 to 10 times longer than its dense equivalent and is difficult to understand and modify. For this reason we have not discussed sparse matrix software in any detail. Instead we give a list of some of the available software packages in the appendix.

APPENDIX

Sources of Sparse Matrix Software

Most general collections of mathematical software contain some sparse matrix routines. However, the best and most up-to-date software comes from those institutions and universities that have active research programs in sparse matrices and related areas. These are indicated by an (*) below.

1. ACM - Association of Computing Machinery, publishes *Trans. on Mathematical Software* and the *Collected Algorithms from ACM*. All software is available in machine-readable form from IMSL below.

2. AERE (*) - Atomic Energy Research Establishment, Harwell, Didcot, Oxfordshire, England. One of the leading places for sparse matrix research and software. Reports published by H.M. Stationery Office.

3. Boeing Computer Services Co., Seattle, Washington 98124.

4. IMSL - International Mathematical and Statistical Library, Inc., NBC Building, 7500 Bellaire Blvd, Houston, Texas 77036. This company develops, maintains and sells the general IMSL library and the Collected Algorithms from ACM. The IMSL library is available on UCD's DEC20 and IBM machines.

5. NAG - National Algorithms Group, Oxford University. Available on UCD's IBM machine,

6. SPARSPAK (*) - Prof. A. George, University of Waterloo, Waterloo, Ontario, Canada.

7. Yale Sparse Matrix Package (*) - Prof. S. Eisenstat, Yale University, New Haven, Connecticut 06520.

```
C=======================================================================
      PROGRAM SSOLVE
C=======================================================================
C This is a driver program and problem generator to test the Gauss-Seidel
C Method for Sparse Matrices
C
C
C-----------------------------------------------------------------------
C
      INTEGER IDXCOL(2400),NONZER(1200),
     +        N, KOUNT, SCREEN, ANSWER
      REAL CMAT(2400), D(1200), X(1200),
     +     EPSIL, A, B, C
      DATA SCREEN /5/

C     WRITE (SCREEN,900)
  900 FORMAT (' INPUT A, B, C, EPSIL (4F5.0), N (I4) :')
      READ (SCREEN,800)A, B, C, EPSIL, N
  800 FORMAT (4F5.0,I4)
      WRITE (SCREEN, 800)A, B, C, EPSIL, N

C-----START CONSTRUCTION OF BAND MATRIX TEST PROBLEM-----------------
C a(i1,i) = A, a(i,i) = B, a(i+1,i) = C
C

      CMAT(1) = -C/B
      IDXCOL(1) = 2
      NONZER(1) = 1
      D(1) = 1.0/B
      IPOINT = 2

C     NM1 = N-1

C     DO 10 IROW = 2, NM1
         CMAT(IPOINT) = -A/B
         IDXCOL(IPOINT) = IROW - 1
         IPOINT = IPOINT + 1
         CMAT(IPOINT) = -C/B
         IDXCOL(IPOINT) = IROW + 1
         IPOINT = IPOINT + 1
         D(IROW) = 1.0/B
         NONZER(IROW) = 2
   10 CONTINUE
      CMAT(IPOINT) = -A/B
      IDXCOL(IPOINT) = N-1
      NONZER(N) = 1
      D(N) = 1.0/B

C-----------------------------------------------------------------------
C
      CALL SEIDEL(CMAT, D, IDXCOL, NONZER, N, EPSIL, X, KOUNT)

C
```



```
      WRITE (SCREEN, 920) N, KOUNT, EPSIL
  920 FORMAT (' N = ', I4,' NO. ITERS. = ', I5,' EPSIL = ', F10.8)
C
C-----WRITE OUT SOLUTION VECTOR--------------------------------------
C
      WRITE (SCREEN, 930) (I, X(I), I=1,N)
  930 FORMAT (' ',5(I4,2X,F10.4))
      STOP
      END

C
C=======================================================================
      SUBROUTINE SEIDEL (CMAT, D, IDXCOL, NONZER, N, EPSIL, X, KOUNT)
C=======================================================================
C This routine solves the linear system of equations x = Cx + d, using the
C Gauss Seidel Method.  The matrix C is sparse and is stored to take advan-
C tage of this sparsity.
C-----------------------------------------------------------------------
C
      INTEGER IDXCOL(2400),NONZER(1200),
     +        N, KOUNT, IPOINT, IROW, JCOL, NUMNZI, SCREEN
      REAL CMAT(2400), D(1200), X(1200),
     +     EPSIL, MAXOLD, MAXNEW, MAXDIF, DELTAX, RELERR, INFNTY
      DOUBLE PRECISION SUM

C     INFNTY = 1.0E20
      KOUNT = 0
      MAXOLD = 1.0

C     DO 5 IROW = 1, N
        X(IROW) = D(IROW)
    5 CONTINUE
C
C-----START OF MAIN ITERATION LOOP -----------------------------------
C
   10 KOUNT = KOUNT + 1
      MAXDIF = -INFNTY
      MAXNEW = -INFNTY
      IPOINT = 1

C     DO 200 IROW = 1, N
         SUM = 0.0
         NUMNZI = NONZER(IROW)
         DO 100 J=1, NUMNZI
            JCOL = IDXCOL(IPOINT)
            SUM = SUM + CMAT(IPOINT) * X(JCOL)
            IPOINT = IPOINT + 1
  100    CONTINUE
C
C-----------CALCULATE INFINITY NORMS---------------------------------
C
         XNEW = SUM + D(IROW)
         DELTAX = ABS( XNEW - X(IROW) )
         IF( DELTAX .GT. MAXDIF ) MAXDIF = DELTAX
         ABXNEW = ABS(XNEW)
         IF( ABXNEW .GT. MAXNEW ) MAXNEW = ABXNEW
         X(IROW) = XNEW
  200 CONTINUE
```

```
C
C-------CONVERGENCE CHECK-------------------------------------------------
C
      RELERR = MAXDIF/MAXOLD
      MAXOLD = MAXNEW
      WRITE (5,900) KOUNT,MAXDIF,MAXOLD,RELERR
  900 FORMAT (' ITER. NO. =', I4,5X, 3F20.7)
      IF(RELERR .GT. EPSIL) GOTO 10
C
      RETURN
      END
C
C============================END OF SPARSE SEIDEL=======================
```

REFERENCES

1. BRAMELLER, A. and HAMAM, Y.M.
'Sparsity: Its Practical Application to Systems Analysis', Pitman, 1976.

2. BUNCH, J.R. and ROSE, D.J. (Eds)
'Sparse Matrix Computations', Academic Press, 1976.

3. BRUALDI, R. and SCHNEIDER, H. (Eds)
'Linear Algebra and its Applications', Vol. 34, Dec. 1980.

4. BARR, R.S. and TURNER, J.S.
"Microdata File Merging Through Large-Scale Network Technology", *Mathematical Programming Study*, 15, 1-22, North-Holland, 1981.

5. BRADLEY, G.H., BROWN, G.G. and GRAVES, G.W.
"Design and Implementation of Large-Scale Transshipment Algorithms", *Management Science*, Vol. 24, No. 1 (1977).

6. DUFF, I.S.
"A Survey of Sparse Matrix Research", *Proc. IEEE*, Vol. 65 (1977) 500-535.

7. DUFF, I.S. (Ed.)
'Sparse Matrices and Their Uses', Academic Press, 1981.

8. DUFF, I.S. and STEWART, G.Q. (Eds)
'Sparse Matrix Proceedings 1978, SIAM Publications, 1979.

9. FORSYTHE, G. and MOLER, C.
'Computer Solution of Linear Algebraic Systems', Prentice-Hall, 1967.

10. GAREY, M. and JOHNSON, D.
'Computers and Intractability: A Guide to the Theory of NP-Completeness', Freeman, 1979.

11. GEORGE, A. and LIU, J.W.
'Computer Solution of Large Sparse Positive Definite Systems', Prentice-Hall, 1981.

12. HOROWITZ, E. and SAHNI, S.
'Fundamentals of Data Structures in Pascal', Pitman, 1984.

13. JENNINGS, A.
'Matrix Computations for Engineers and Scientists', Wiley, 1977.

14. KNUTH, D.E.
'The Art of Computer Programming', Vol. 1, 2nd ed., Addison-Wesley, 1973.

15. POOCH, U. and NIEDER, A.
"A Survey of Indexing Techniques for Sparse Matrices", *'Computing Surveys*, Vol. 5, No. 2, 1973.

16. REID, J.K.
'Large Sparse Sets of Linear Equations', Academic Press, 1971.

17. RICE, J.
'Numerical Methods, Software, and Analysis', McGraw-Hill, 1983.

18. ROSE, D.J.
"A Graph-Theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations",

in READ, R. (ed.), *'Graph Theory and Computing'*, Academic Press, 1973.

19. ROSE, D.J. and TARJAN, R.E.
"Algorithmic Aspects of Vertex Elimination on Directed Graphs", *SIAM Journal on Applied Mathematics*, Vol. 34 (1978) 176-197.

20. ROSE, D.J. and WILLOUGHBY, R.A. (Eds)
'Sparse Matrices and their Applications', Plenum Press, 1972.

21. TARJAN, R.E.
"Graph Theory and Gaussian Elimination", in [2] (1976) 3-22.

22. VARGA, R.S.
'Matrix Iterative Analysis', Prentice-Hall, 1962.

23. WELSH, J., ELDER, J. and BUSTARD, D.
'Sequential Program Structures', Prentice-Hall, 1984.

24. WILLOUGHBY, R. (Ed.)
'Sparse Matrix Proceedings', IBM Research, 1968.

Management Information Systems Department,
University College, Dublin

AN INTRODUCTION TO NONSTANDARD ANALYSIS

Christopher Thompson

1. Introduction

This note describes the axiomatic approach to nonstandard analysis developed by Nelson and illustrates it by proving the fundamental theorem of algebra and a form of the spectral theorem in finite dimensions. It is based in part on a talk given at the DIAS in April, 1985.

2. The Axioms

We shall be working in a mathematical universe that contains all the usual familiar objects (e.g. numbers 0, 1, $\sqrt{2}$, π etc., sets $\mathbb{N}$, $\mathbb{R}$, $\mathbb{C}$ etc., function spaces ℓ^2, $C[0,1]$ etc.) and in addition contains new and unfamiliar objects such as infinitely large natural numbers and infinitely small positive real numbers. To ensure the presence of the familiar objects we adopt the usual axioms of set theory, for example the Zermelo-Fraenkel axioms together with the axiom of choice. To make visible the unfamiliar objects we adopt a new undefined unary predicate *standard* and axioms (I), (S) and (T) to govern its use. The resulting theory is called *internal set theory* (IST) and is due to Nelson [7]. Just as the binary predicate "$\in$" of ZFC has the informal interpretation "is a member of" (although strictly speaking it is undefined and therefore meaningless), so also has the unary predicate "standard" of IST an informal meaning: 'x standard' has the interpretation 'x is a familiar object of classical mathematics'. It is a consequence of the axioms that 0, 1, $\sqrt{2}$, π, $\mathbb{N}$, $\mathbb{R}$ etc. are indeed standard, as we shall see.

A formula of IST may or may not contain the predicate "standard". If it does then it is called an *external* formula, otherwise it is called *internal*. Speaking informally, internal