27
10-19-77
5 NTIS

SAND77-1091

Unlimited Release

3151

Basic Row-Column Operations With Orthogonally Linked Sparse Matrices for Use With FORTRAN

MASTER

Richard J. Hanson, John A. Wisniewski

Prepared by Sandia Laboratories, Albuquerque, New Mexico 87115 and Livermore, California 94550 for the United States Energy Research and Development Administration under Contract AT(29-1)-789

Printed October 1977

Sandia Laboratories

SF 2900 Q(7-73)

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Issued by Sandia Laboratories, operated for the United States Energy Research & Development Administration by Sandia Corporation.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research & Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

SAND77-1091
Unlimited Release
Printed October 1977

Basic Row-Column Operations with Orthogonally Linked Sparse Matrices for Use with FORTRAN

Richard J. Hanson
John A. Wisniewski
Sandia Laboratories
Albuquerque, New Mexico 87115

NOTICE
This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research and Development Administration, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

ABSTRACT

An orthogonally linked data structure is used to represent sparse rectangular matrices. This representation requires four type INTEGER locations and one type REAL location per nonzero entry in the matrix.

Operations are presented that perform many of the basic computations required in numerical linear algebra.

The operations are available in portable FORTRAN with the exception of the necessarily operating system sensitive "get" and "put" subprogram for the individual records.

The package of subprograms manages out-of-memory portions of the matrices as well as the insertion, modification and deletion of entries.

Using the package, many of the familiar processes of numerical linear algebra, such as solving sparse linear algebraic equations, can be written with a relatively few calls to these subprograms.

Printed in the United States of America

Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22161
Price: $4.00 Microfiche $3.00

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Table of Contents

Basic Row-Column Operations with Orthogonally Linked Sparse Matrices for Use with Fortran

Introduction

This paper discusses a convenient structure for storing and performing operations on sparse real matrices and vectors. The data structure dealt with here is the orthogonally linked list, (1). A set of fourteen operators is then presented which perform many of the basic operations of numerical linear algebra for row or column vectors stored using this structure. The subprograms which do the operations are written in ANS FORTRAN, and it is intended that they be used with ANS FORTRAN programs. The only primitive which necessarily contains operating system sensitive components is the subprogram TRWVIR( ) which reads from or writes to virtual memory such as extended memory or random access storage devices.

Much of the subprogram and test driver development work was done by Katherine Peters. Her efforts are gratefully acknowledged.

Reasons for Developing the Package

1. Many published algorithms in sparse numerical linear algebra theory describe both a data structure and a computing method. Understanding and implementing the data structure is often more complicated than the computing algorithm itself. By standardizing the data structure much of this preliminary complication can be eliminated. The programmer can move immediately to the implementation of the algorithm.
2. Many published algorithms require both row and column operations on the matrix. The proposed data structure described allows the programmer to do this with ease.
3. The package of FORTRAN subprograms for the basic operations are portable except for the TRWVIR( ) routine which depends on the operating system.

4. The FORTRAN subprograms, in their portable forms, require more high-speed memory than other programs that use more specialized data structures, e.g., Curtis, Reid, (2). One result of this work, however, may be the ability to compare various algorithms on moderate sized problems by virtue of the more general applicability of the operators.
5. The data structure is designed to use low-speed virtual memory for part of the data storage. This can allow an applications programmer to solve certain large linear algebra problems without getting involved in the details of storage management.

There are often compelling reasons why a programmer would develop a separate data structure that exploits features of a specific problem or machine. These include the ability to minimize the storage requirements of the necessary overhead information of sparse matrix problems. This often allows for a more efficient algorithm because the problem data can be kept entirely in high-speed memory.

There are also equally compelling reasons why a programmer should not develop a separate data structure that is tailored to a problem or a machine:

● It requires a lot of human toil. This is particularly true if low-speed storage is utilized as part of the data structure design.

● The package so developed by a programmer will hardly ever be portable, even on the same machine line, if it is tailored to a specific problem and machine.

● A small and insignificant algorithmic change in the problem may require extensive changes in the data structure. This may have an additional high cost in human toil.

● The design of this package represents a compromise between low storage requirements and the availability of a wide variety of permissible operations on the matrices stored using this data structure. By localizing the operating system dependent parts of this package it is possible to achieve portability insofar as this is possible. Many different algorithms can be interfaced using this data structure, and it provides for ease of usage without the programmer being unnecessarily diverted by data handling problems.

A word of caution is in order. It is possible to misuse this package by coding algorithms in such a manner that "page thrashing" occurs. Attention to the order in which elements are stored and processed can prevent this phenomenon. Moler, (5), considers this problem in the context of solving linear algebraic systems having dense matrices using FORTRAN on operating systems with paging.

The Orthogonally Linked Data Structure

We will always be dealing with rectangular matrices in this data structure. Vectors are to be considered as slim matrices. Only the nonzero entries of matrices are stored in this data structure. Note that matrices with less than 20% of their entries nonzero achieve savings in the required storage.

For each nonzero matrix component a_{ij} we store, (1), a list of records

a_{ij}	i	Location of Next Nonzero in Row	j	Location of Next Nonzero in Column

These and other aspects of the data structure are illustrated with the matrix, used as an illustration in (3),

$$\boxed{\begin{matrix} 4 & 0 & 2 & 0 & 6 \\ 0 & 2 & 3 & 0 & 3 \\ 0 & 0 & 0 & 3 & 7 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 4 \end{matrix}}$$

Figure 1. Sample 5 by 5 Matrix

Using the orthogonally linked data structure to store this matrix yields the following table. The definitions of LA, LP and LM appearing in Table 1 are given in the section <u>Data Structure Specifications.</u>

Table 1. Storage of 5 by 5 matrix.
The "u" indicates undefined quantities.

Location	Matrix Element Values	Subscript Within Row	Next Location of Nonzero in Row	Subscript Within Column	Next Location of Nonzero in Column
1	0.	-5	6	-5	6
2	0.	0	9	0	9
3	0.	0	12	0	7
4	0.	0	14	0	12
5	0.	0	15	0	8
6	4.	1	7	1	14
7	2.	3	8	1	10
8	6.	5	1	1	11
9	2.	2	10	2	15
10	3.	3	11	2	3
11	3.	5	2	2	13
12	3.	4	13	3	4
13	7.	5	3	3	16
14	1.	1	4	4	1
15	1.	2	16	5	2
16	4.	5	5	5	5
17=LA	u	u	18	u	u
18	u	u	22	u	u
19=LP	u	u	u	u	u
20	u	u	u	u	u
21	u	u	u	u	u
22=LM	1.	19	17	1	-1
23	Storage will be in virtual memory from this point onward.				

Examples:

- Find row 2 of the matrix.

  Use location 2 row information to start. The first nonzero is at location 9. From locations 9, 10, and 11 we have nonzero values and corresponding subscripts which yield the row vector

  $$[0 \quad 2. \quad 3. \quad 0 \quad 3.]$$

  as row 2 of the matrix. The end of the row vector is indicated by component 5 in location 11 pointing to location 2 which has an index ≤ 0.

- Find column 5 of the matrix.

  Use location 5 column information to start. The first nonzero is at location 8. From locations 8, 11, 13, and 16 we have nonzero values and corresponding subscripts which yield the column vector

  $$[6. \quad 3. \quad 7. \quad 0. \quad 4.]^T$$

- Change a_{42} from zero to 8.

  A circular linked list of available storage starts in location 17. Thus we must modify the records at locations 9 and 14 to become respectively

2.	2	10	2	17
1.	1	17	4	1

  The record at location 17 is defined as

8.	2	4	4	15

  The value of LA is updated to 18. The record at location LM=22 is modified to become

u	19	18	1	-1

Here we could have used the location LP=19 in place of location 17 to store the new record. The value 19 indicates that record locations 19-21 are free.

In general the data structure for storing each m by n matrix consists of two lists. The first list is for storage of the matrix elements. The second list has four rows and is for storage of the row and column indices and the row and column vector links.

Only rows 1,...,LM of these lists must be in high-speed memory. The remainder of the lists are in some type of virtual memory.

	SX(LM)	IX(4,LM)			
Location in SX( )	Matrix Element Values	Subscript Within Rows	Next Location of Nonzero in Row	Subscript Within Columns	Next Location of Nonzero in Column
1	0. or 1.	-m	k+1	-n	k+1
2	0.	0			
3	0.			⋮	
		0		0	
		⋮			
k	0.	0			
k+1	a_{11}	1	k+2	1	...
k+2	a_{12}	2	...		
⋮					
LA	u				
⋮	⋮				
LP	u				
⋮	⋮				
LM	0. or 1.				

Fig. 2. The High-Speed Memory Portion of the Data Structure for an m x n matrix. Here k = max(m,n).

Data Structure Specifications

1. To find row i of the matrix use IX(2,i) to find the location of the first nonzero in row i. If IX(2,i) = i, then row i is zero.
2. To find column j of the matrix use IX(4,j) to find the location of the first nonzero in column j. If IX(4,j) = j, then column j is zero.
3. The values of IX(*,LM) have a special meaning, and this record must be kept intact.

The value IX(1,LM) $\equiv$ LP defines a pointer such that records LP,...,LM-1 are free. If LP $\geq$ LM, then there is no free and open block of records.

The value IX(2,LM) $\equiv$ LA points to the start of a linked list of free records. This list is circular and ultimately points back to record LM. If LA = LM, this list is empty. The values of IX(3,LM) = MN and IX(4,LM) = $\pm$NP are the matrix number and, up to a sign, the page number. If the page number is negative there is no page beyond this one for that matrix.

The value of SX(LMX) contains a binary flag indicating if this page is available in low-speed memory in exactly the form it had in high-speed memory. If the high-speed copy is not exactly the same as the low-speed copy we call this page an original. If the page is original, SX(LMX) = 1. Otherwise, SX(LMX) = 0.

The following initialization is required of the user for each m x n matrix occupying a distinct data structure of the form shown in Fig. 2.

```
For i = 1,...,m set IX(1,i) = 0, IX(2,i) = i;
                                 IX(1,1) = -m

For j = 1,...,n set IX(3,j) = 0, IX(4,j) = j;
                                 IX(3,1) = -n

Set IX(1,LM) = max(m,n) + 1
    IX(2,LM) = LM
    IX(3,LM) = Matrix Number = MN
    IX(4,LM) = -1 = - Page Number (= -NP) .
 Set SX(LMX) = 1 , SX(i) = 0., i = 1,...,max(m,n)
```

The FORTRAN statement

```
CALL  TINITM(M,N,MN,SX,IX,LMX)
```

allows the user to initialize the data structure representing matrix number MN. The subprogram TINITM( ) is included in the package.

An example of the construction of a matrix is given in the section entitled Constructing a Matrix.

The portion of the high-speed storage in locations k+1,...,LM may be overwritten with pages of the same length that are stored in auxiliary random access storage. The subprograms keep track of all paging requirements.

The page size itself is of length LPG = LM - k. Frequently this value must be chosen to reflect operating system parameters such as block size for data transfers.

The locations of the pages for each matrix are kept in a page location table updated in the "get" and "put" subprogram TRWPGE( ). A value of zero in this table indicates that this page has not yet been written. Otherwise, the entry gives the location of the pages in the auxiliary storage. The package allows one to have 5 matrices, each with 100 pages. This restriction can be removed by recompiling TRWPGE( ) after modification of the variables IPTBLE(,), MAXMN and MAXPGE.

NP →

Page Number

MN ↓ Matrix Number	1	2	3	4	5	6
1	LOC	LOC	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Table 2. A Sample random access location table of pages for 5 matrices each having up to 6 pages; one matrix and 2 pages are now in use.

A Discussion of Virtual Memory and the Orthogonally Linked Sparse Matrix Data Structure

When solving sparse matrix problems, we often solve large sparse problems. These problems can become so large so as to require more storage than that available in high speed central memory. Any general package which is intended to perform operations on sparse matrices should implement a secondary, mass storage scheme to be used when the available storage in central memory is exhausted.

A secondary, mass storage scheme should have a number of properties. The switch from high speed memory to secondary memory should be automatic. The management of secondary, as well as high speed memory, should be done entirely by the package without the need for direct user intervention. The amount of high speed memory allocated should be under user control, and the user should have some indirect means of controlling the number of accesses to secondary memory.

What we shall describe is a secondary mass storage scheme which meets the above requirements. The method employed is that of virtual memory with demand paging (9), which is often used in an operating systems context.

In our implementation of secondary storage, the orthogonally linked list data structure is broken into two parts. If we let k = max(m,n), where m and n are matrix dimensions, the first segment or master page consists of the first k entries in the SX(*) and IX(4,*) arrays. The data associated with these entries point to the starting locations of a given row or column. Since these data items are frequently accessed, they are always kept in high speed memory. The remaining LPG = LM - k entries constitute a virtual memory page.

Since the user specifies the quantities m, n and LM, control is maintained over the length of a page and certainly the amount of high speed memory available to the package. As far as the user is concerned, the arrays SX(*),IX(4,*) are essentially unbounded. Since only one virtual memory page resides in high speed memory at a time, the larger the page length, the better. If sufficient high speed memory is provided for a given sparse matrix no secondary memory will be used.

Every virtual address (pointer value) is decoded into the quantities NP = (page number) and IOFF = (offset within the page), where NP = integer part $[(\text{LOC}-k-1)/\text{LPG}] + 1$ with LOC being the virtual address, and IOFF = remainder $[(\text{LOC}-k-1)/\text{LPG}] + k + 1$. If the current page number $|\text{IX}(4,\text{LMX})|$ is not equal to NP, we write page $|\text{IX}(4,\text{LMX})|$ to secondary memory if it was an original and read page NP. This decoding function is performed by the Function Subprogram ITLOC( ).

Since the virtual addresses range from 1 to some large number ℓ, the arrays SX(*) and IX(4,*) have the appearance of being of length ℓ rather than of length LM. This property preserves the updating techniques required for the orthogonally linked list data structure regardless of the amount of high speed memory available. The code switches from high speed memory to a secondary memory scheme as soon as the first page is exhausted. Since all virtual addresses are decoded into page numbers and offsets, with paging being performed for the user, the management of secondary memory is done entirely by the package.

In addition, the package manages the dynamic allocation and deallocation of virtual memory locations. This is done by subprogram TCHNGE( ), depending on the old and new values of the element to be changed. If the old and new values of the element are both zero or nonzero, no allocation or deallocation is required. When both the new and old values are nonzero, only the value of the matrix element is changed and no pointers are modified.

When the old value of an element is zero, but the new value is nonzero, a virtual memory location is allocated and the orthogonal links are updated to reflect the creation of the new matrix element.

When the old value of an element is nonzero, but the new value is zero, the virtual memory location associated with the element is deallocated and the orthogonal links are updated to reflect the deletion of the matrix element. Deallocation of a virtual memory location is accomplished by updating the circular list of free storage, for the given page, as was described in the section "The Orthogonally Linked Data Structure". Each page of virtual memory has its own linked list of free storage which is maintained independently of other pages. So, the pointers stored in the lists of free storage represent addresses within the page. They do not represent virtual addresses.

The allocation of a virtual memory location is accomplished according to the following algorithm.

Algorithm Allocate:

Scan the virtual memory pages for free storage in the following order.

1. check the current page.
2. check the page where we last found a free storage location.
3. check all pages from the current page +1 to the last page.
4. check all pages from the first page to the current page -1.

If a free memory location is found in any page, terminate the algorithm and return the virtual memory location of the free storage location.

If all pages were full, generate a new page of virtual memory and return the virtual address of the first free memory location of the newly generated page.

End allocate.

Within each page locations from the free and open block are first assigned. When this is exhausted the linked list of available storage is assigned on a last in - first out basis.

To some extent, the user has control over the page in which an element is stored. For example, when a matrix is first created using TCHNGE( ), the first LPG - 1 data items which are stored will reside on page number 1, the second LPG - 1 on page number 2, etc. Similarly, if the user accesses the data in the same order in which it was created, a reduction in the number of page faults associated with a given matrix operation can be achieved.

A number of 'overhead' items are required in order to implement paging on a variety of machines. The most important item is the page table. The page table has already been described in the previous section. In the package, we use a single subprogram TRWVIR( ) to implement the machine dependent parts of the implementation associated with paging. We intend to have versions of TRWVIR( ) for the CDC 6000-7000 series, UNIVAC 1100 series, and IBM 360/370 series. In order to implement this package on a virtual memory operating system, all the user needs to do is to declare the arrays SX(*) and IX(4,*) to be at least k + 1 plus the number of nonzeros in the computation.

When we implement paging on a system using random access disk for secondary memory, the address of a page is uniquely determined from the page number and matrix number. In this instance, the page table is of little use. It is then used only for error checking purposes. On the other hand, if paging is implemented using some type of low speed, word addressable memory device, such as extended core storage on the CDC 6600-7000 series, then the page table is required. The page table provides the offset in low-speed memory of a given page. This offset, and the length of the given page are required to retrieve the page.

Paging, although convenient, may not be the best implementation of secondary memory for the orthogonally linked sparse matrix data structure. This was discussed in the introduction. Due to the existence of the orthogonal links it is quite possible to get page thrashing regardless of the manner in which the matrix is accessed or modified. Paging, however, is a natural way to implement secondary memory and significantly reduces the complexity of the implementation since memory is treated as lengthened versions of the SX(*) and IX(4,*) arrays. The best way a user can reduce page thrashing is to make the arrays SX(*) and IX(4,*) as large as possible. This will reduce the total number of pages and perhaps limit the number of pages to a single page. Thrashing can be further reduced by zeroing array elements once they are no longer needed. More complicated methods, such as transferring data to additional arrays, may also reduce thrashing.

Specification of the Operations

The package of subprograms performs computations on row or column vectors of a matrix in single precision. The categories of operations performed are array or matrix element manipulations, vector arithmetic, norms and transmission, and an initialization subprogram.

Dimensioning Information

```
      REAL SX(LMX),SY(LMY),SW,SA,SM(4)
      INTEGER IX(4,LMX),LMX.IY(4,LMY),LMY,IP,IRCX,IRCY,IPLACE(2)
```

Matrix Initialization

```
      CALL TINITM(M,N,MN,SX,IX,LMX)
```

Initialize the M by N matrix numbered MN in the data structure represented by SX( ), IX(,) and LMX. One call to TINITM( ) is required for each matrix in the problem.

Assign the matrices consecutive integer numbers beginning at the value one. Each matrix requires a unique set of arrays SX( ), IX(,) and a value LMX. A call to TINITM( ) is equivalent to zeroing the entire matrix.

Subscripting Conventions for Rows or Columns

In each of the operations 1. - 13. the sign of IRCX or IRCY determines whether a row or column of the matrices is being used.

If IRCX ≤ 0, then use row vector -IRCX in the operation

If IRCX ≥ 0, then use column vector IRCX in the operation.

Analogous remarks hold for IRCY.

Array or Matrix Element Manipulation

In subprograms 1. - 3. the scan to locate element I of the vector is started from the virtual location in IPLACE(1) if IPLACE(1) $\neq 0$. The scan starts from the beginning of the vector if IPLACE(1) = 0. In case IPLACE(1) $\neq$ 0 it's necessary that this value point to an element with the next index greater than I or to a nonpositive index.

1. CALL TNNZR(I,XVAL,IPLACE,SX,IX,LMX,IRCX)

   Get the next nonzero in the vector with an index $\ge$ I. The values of I, XVAL, and IPLACE(*) may be updated. The value itself and its index are returned in XVAL and I. An input value of I ≤ 0 starts the scan from the beginning of the vector. An output value of I = 0 denotes that all components with an index $\ge$ the input value of I are zero.

2. CALL TGET(I,XVAL,IPLACE,SX,IX,LMX,IRCX)

   Get element I in the vector and put its value in XVAL. A value of XVAL = 0 denotes that the element was zero and that it is not stored.

3. CALL TCHNGE(I,XVAL,IPLACE,SX,IX,LMX,IRCX)

   Change element I in the vector to XVAL. All of the details of creation or deletion of elements are done in this routine.

In the operations 4. - 13., operations are performed on the subvectors $\underset{\sim}{x}$ or $\underset{\sim}{y}$ of the partitioned row or column vectors. The beginning and length of these vectors is determined by the sign of a partitioning parameter IP.

If IP ≤ 0, then use indices from both vectors with values $\le$ -IP in the operation.

If IP ≥ 0, then use indices from both vectors with values $\ge$ IP in the operation.

<u>Arithmetic</u>	<u>Operation</u>
4. SW = TDOT(IP,SX,IX,LMX,IRCX,SY,IY,LMY,IRCY)	Dot Product, $w := \underset{\sim}{x}^T \underset{\sim}{y}$
5. CALL TAXPY(IP,SA,SX,IX,LMX,IRCX,SY,IY,LMY, IRCY)	Elementary Vector Operation, $\underset{\sim}{y} := a\underset{\sim}{x} + \underset{\sim}{y}$
6. CALL TSCAL(IP,SA,SX,IX,LMX,IRCX)	Vector Scaling, $\underset{\sim}{x} := a\underset{\sim}{x}$
7. CALL TMPRO(IP,SM,SX,IX,LMX,IRCX,SY,IY,LMY, IRCY) On input have SM(1) = a, SM(2) = b, SM(3) = c, and SM(4) = d.	Matrix Product, $\begin{pmatrix} \underset{\sim}{x}^T \\ \underset{\sim}{y}^T \end{pmatrix} := \begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} \underset{\sim}{x}^T \\ \underset{\sim}{y}^T \end{pmatrix}$
<u>Norms</u>	
8. IW = ITAMAX(IP,SX,IX,LMX,IRCX)	Smallest Index of the Component with Maximum Magnitude
9. SW = TNRM2(IP,SX,IX,LMX,IRCX)	Euclidean Length
10. SW = TASUM(IP,SX,IX,LMX,IRCX)	Sum of Magnitudes

Transmission

11.	CALL TSWAP(IP,SX,IX,LMX,IRCX,SY,IY,LMY, IRCY)	Interchange Vectors, $\underset{\sim}{x} :=: \underset{\sim}{y}$
12.	CALL TCOPY(IP,SX,IX,LMX,IRCX,SY,IY,LMY, IRCY)	Copy Vectors, $\underset{\sim}{y} := \underset{\sim}{x}$
13.	CALL TMOVE(IP,SX,IX,LMX,IRCX,SY,IY,LMY, IRCY)	Move Vectors, $\left((\underset{\sim}{y} := \underset{\sim}{x}), \underset{\sim}{x} := \underset{\sim}{0}.\right)$

Supporting Subroutine Descriptions

Subprograms 0. - 13. are part of the package. Subprograms 14. - 18. have the following ancillary uses.

14. INTEGER FUNCTION ITFIND( )

Find a free location to store a new record, or matrix element. The location returned is a virtual location. This routine may read and write pages.

15. INTEGER FUNCTION ITLOC( )

Compute a relative or present page address given a virtual location for a matrix element. This routine may read or write pages.

16. SUBROUTINE TRWPGE( )

Read or write a page to random access or virtual memory. This routine controls the locations for the pages. It does not actually write on a device per se.

17. SUBROUTINE TRWVIR( )

Read or write the pages on a virtual memory device. This is the only nonportable FORTRAN subprogram. It is operating system sensitive and must usually be rewritten at the user installation.

18. SUBROUTINE TERROR( )

This is the error processor and status keeping subprogram. The usage of this subprogram and the interpretation of errors and status of the computation is given in Appendix 1.

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.
0. TINITM																	x		x
1. TNNZR																x			x
2. TGET																x			x
3. TCHNGE			x												x	x			
4. TDOT		x																	
5. TAXPY		x		x															
6. TSCAL		x		x															
7. TMPRO		x		x															
8. ITAMAX		x																	
9. TNRM2		x																	
10. TASUM		x																	
11. TSWAP		x		x															
12. TCOPY		x		x															
13. TMOVE		x		x															
14. ITFIND																	x		
15. ITLOC																	x		x
16. TRWPGE																		x	x
17. TRWVIR																			x
18. TERROR																			

Table 3. Subroutine Heiarchy Array

An x in entry at row i and column j denotes that subroutine number i calls subroutine number j.

Constructing a Matrix

In this section an example is presented for storing the problem data involved in the computation of the solution to a sparse N by N linear algebraic system $A\underset{\sim}{x} = \underset{\sim}{b}$ using the Fortran preprocessor language FLECS, (6). We will store the matrix A in sparse form and the vector $\underset{\sim}{b}$ as a linear FORTRAN array.

The single most important point in defining a matrix is the usage of the TCHNGE( ) subprogram. This subprogram manages the details of the orthogonally linked data structure. This eliminates any need for the user to directly manipulate this data structure.

Here we provide one example of a method for defining the elements of a matrix A in an arbitrary order.

The user provides a subprogram

TENTRY(I,J,AIJ,IFLAG)

to define A and $\underset{\sim}{b}$. When this subprogram is called with IFLAG = 1 the user must define those row and column indices and nonzero values of the matrix and right hand side vector which occur. This data can be returned in any convenient order. The nonzero matrix elements will have $1 \leq I \leq N$, $1 \leq J \leq N$. The right hand side vector, $\underset{\sim}{b}$, will have each nonzero b_I denoted by the output values of $J = N + 1$ and $1 \leq I \leq N$. When the matrix and right side vector entries that are nonzero have been defined, the user sets IFLAG = 2 as an output variable.

This is only one of many ways that a user could define a sparse matrix.

```
C
C      DEFINE A SPARSE N BY N MATRIX A AND A VECTOR B.
C      STORE A USING THE ORTHOGONALLY LINKED LIST
C      STRUCTURE IN SX( ),IX(4, ), AND LMX.  STORE
C      THE RIGHT HAND SIDE B IN THE LINEAR ARRAY B( ).
C
       DIMENSION SX(LMX),IX(4,LMX),B(N),IPLACE(2)
C
C      INITIALIZE THE DATA STRUCTURE TO STORE THE MATRIX A.
       CALL TINITM(N,N,1,SX,IX,LMX)
C
C      SET B( ) TO ALL ZEROS.
       CALL SCOPY(N,0.E0,0,B,1)
C
C      INITIALIZE IFLAG TO ASK TENTRY( )
C      SUBPROGRAM FOR PROBLEM DATA.
       IFLAG=1
       CALL TENTRY(I,J,AIJ,IFLAG)
       UNTIL(IFLAG.EQ.2)
C      .  THE USER DEFINES THE NONZERO MATRIX ELEMENTS (INSIDE TENTRY( ))
C      .  IN ANY ORDER.  THE CASE J=N+1 DENOTES
C      .  THAT THE VALUE AIJ IS THE I-TH ELEMENT OF B.
       .  IPLACE(1)=0
C      .
       .  WHEN(J.LE.N) CALL TCHNGE(I,AIJ,IPLACE,SX,IX,LMX,J)
       .  ELSE B(I)=AIJ
C      .
       .  CALL TENTRY(I,J,AIJ,IFLAG)
       ...FIN
C       .
C       . (PROBLEM COMPUTATION.)
C       .
       STOP
       END

   (FLECS VERSION 22.34)
```

Operations with Vectors in Linear Arrays

Utilizing subprograms in the package one can perform computations involving sparse vectors stored in this orthogonally linked data structure and vectors stored as linear arrays in the usual FORTRAN convention.

For instance in the example used here the right hand side is stored as a linear array. During the process of decomposition we must perform operations of the form

$$b_I := a*x_I + b_I\ ,\quad I = J,\ldots,N$$

This can be done using the loop of the following form:

```
IPLACE(1)=0
I=J
CALL TNNZR(I,XVAL,IPLACE,SX,IX,LMX,J)

WHILE(I.GT.0)
|B(I)=A*XVAL+B(I)
|I=I+1
|CALL TNNZR(I,XVAL,IPLACE,SX,IX,LMX,J)
```

The vector $(x_J,\dots,x_N)^T$ is sparse in this example while the vector $(b_J,\dots,b_N)^T$ is not sparse. Subprogram TNNZR( ) returns the next nonzero with an index greater than or equal to I, and returns in the variable I the index of that next nonzero. Thus only the nonzero components x_I cause b_I to be updated. The value of I is incremented in the above loop solely to step past the last nonzero which was obtained. When there are no nonzero values with an index greater than or equal to I, subprogram TNNZR( ) returns with the value of I = 0. This is the condition which terminates the above loop.

Relationship to Other Packages

There are some similarities in the naming convention for this sparse package and the BLAS package of (4). This was done intentionally because the names used in designing the BLAS were subjected to about four years of peer discussion and review. No claim is made that this sparse package is as easy to use as the BLAS package. However, there are enough similarities that a user who is familiar with the BLAS package should have little difficulty adjusting to the usage of this sparse package.

Other packages for sparse matrix work, using low-level routines, have appeared. McNamee, (7), wrote a series of subprograms that perform many of the same operations as those of this package. His code was written for IBM 360/370 systems under the "WATFOR FORTRAN" compiler.

McNamee's subroutines are not strictly portable, but hints on necessary changes for some other machine lines are given. The central difficulty with his package, as reported in (8), is that his storage scheme is essentially a static one. It is quite difficult to create and delete elements of a matrix. There is no such restriction in the package proposed here, thanks to the fact that we have used the orthogonally linked list structure.

There are a number of serious problems that await further research. These concern the location of the matrix elements to optimize the number of distinct page references. Each algorithm that dynamically creates and deletes matrix elements will probably require a storage management algorithm to function effectively on large-scale problems.

Bibliography

1. Knuth, D. E., The Art of Computer Programming, Vol. I, p. 295-302 (1968).

2. Curtis, A. R., Reid, J. K., "The Solution of Large Sparse Unsymmetric Systems of Linear Equations," J. Inst. Maths. Applics. 8, pp. 544-353, (1971).

3. Gentleman, W. M., George, A., "Sparse Matrix Software," Sparse Matrix Computations, Eds. J. Bunch, D. Rose, p. 243-261, (1976)

4. Lawson, C. L., Hanson, R. J., Kincaid, D. R., Krogh, F. T., "Basic Linear Algebra Subprograms for Use with Fortran," Submitted to Trans. Math. Software, July, 1977.

5. Moler, C. B., "Matrix Computations with FORTRAN and Paging," Comm. ACM, Vol. 15, No. 4, p. 268, April, (1972).

6. Beyer, T., FLECS-Fortran Language with Extended Control Structures. User's Manual. University of Oregon Computing Center, Eugene, Oregon, Sept., (1974).

7. McNamee, J. M., "Algorithm 408. A Sparse Matrix Package (Part 1)," Comm. ACM, Vol. 14, p. 265-273, (1971).

8. Duff, I. S., "A Survey of Sparse Matrix Research," Proc. IEEE, Vol. 65, No. 4, p. 500-535, April, (1977).

9. Madnick, S. E., Donovan, J. J., Operating Systems, McGraw-Hill, p. 139-165, (1974).

Appendix 1

Description of Subprogram TERROR( ), Error Processing and Status Indication Subprogram

Subprogram TERROR( ) performs the error and status monitoring functions for the sparse matrix package. This appendix will describe the usage of Subprogram TERROR( ) and a list of error messages and probable causes. The method used by the package to monitor demand paging is described. In addition, modifications to the routine necessary to add a new error message or to incorporate a new status indication feature are outlined.

Description of Usage

```
    CALL TERROR(IFATAL,IERROR,IFMT)
```

Input:

IFATAL is an integer flag indicating the degree of seriousness of the error.

If IFATAL = 1	the program is terminated via a STOP statement following an analysis of the value of IERROR, and a summary of all errors for the run is printed.
Otherwise	This routine returns to the calling program.

IERROR is an error number. A histogram of all errors encountered is kept by this subprogram.

If IERROR < 0	No message is printed, only the status histogram is updated.
IF IERROR = 0	The status histogram for this run is printed.
If IERROR > 0	The message passed in IFMT is printed, and the status histogram is updated.

IFMT(*) is a variable FORMAT which contains the message to be printed. Example: 32H(27HOTHIS MATRIX IS NOT SQUARE.) is a literal argument for a variable FORMAT and results in the printing of: THIS MATRIX IS NOT SQUARE.

Subprogram TERROR( ) restricts $|\text{IERROR}| \le 40$. At present, the maximum value of $|\text{IERROR}|$ used by the package is 18. The variable FORMAT IFMT is used to avoid packing a variable number of characters per word, depending on the machine. Although the variable FORMAT feature is not ANSI standard, it is implemented on the major machine lines, and will be in the new (1977) ANSI standard.

Description of Error Messages and Likely Causes

This section describes the 18 error messages and status indicators that have been implemented in the sparse matrix package. Error checking for a given matrix SX(*), IX(4,*), LMX, can be suppressed in the routines TGET( ) and TNNZR( ) by setting the value SX(1) = 1 for that matrix. This will suppress error numbers 1, 17, and 18 for that matrix in those routines. This error monitoring suppression can lead to run time improvements on the order of 16%, but at the loss of this important debugging aid. The status summary and any error messages are written to the FORTRAN logical unit number 6.

Error number 1,
In TGET( ) or TNNZR( ), row or column number was not compatible with complementary indices.
The data pointed to by the array IPLACE(*) points to data which is not in the desired row or column. Make sure that the IPLACE(*) array is either initialized properly, or that it is currently accessing data in the desired row or column. For example, in table 1, row 2 is stored in memory locations

9, 10 and 11 with subscripts stored in the third column of the table. For row 2, the complementary indices appear in column 5 of the table and all must have a value of 2. If any of these complementary indices were not 2, this message would be printed.

Error number 2,
In ITLOC( ) a value of LOC (first argument) .LE.1 was encountered.
This error could occur from an uninitialized IPLACE(*) value in a higher level routine, or may occur in the presence of an overwrite problem (the array IX(4,*) being overwritten by garbage).

Error number 3,
In TRWPGE( ) the value of MN (matrix number) was not in the range
1.LE.MN.LE.MAXMN.
Currently, MAXMN = 5. The user has declared more than MAXMN matrices to the sparse matrix package.

Error number 4,
In TRWPGE( ) the value of IPAGE (page number) was not in the range
1.LE.IPAGE.LE.MAXPGE.
Currently, MAXPGE = 100. Most likely, the given matrix has exceeded MAXPGE pages of memory.

Error number 5,
In TRWPGE( ) the value of LPG (page length) was nonpositive.
This error is unlikely to occur. Most likely the value of LMX was less than max(m,n) + 2.

Error number 6,

In TRWPGE( ) the value of KEY (read-write flag) was not 0, 1, or 2.

The user has called this routine directly with a bad value of KEY, or an overwrite problem exists.

Error number 7,

In TRWVIR( ) a nonpositive virtual address was encountered.

Bad pointer information exists in IX(4,*) most likely due to an overwrite problem.

Error number 8,

In TRWVIR( ) limits of virtual memory were exceeded.

On the CDC 6000-7000 series extended core memory is being used for virtual memory exclusively, and this memory device has no more available space. To fix, use a mixed strategy of extended core storage and disk for virtual memory. (Set Method = 3 in subprogram TRWVIR( ); See Appendix 2.)

Error number 9,

In TERROR( ) an invalid error number was encountered.

Currently, MAXERR = 40. Subprogram TERROR( ) was called with |IERROR| > MAXERR. If more than MAXERR error numbers are required, recompile TERROR( ) increasing MAXERR and dimension array IERR to be MAXERR long.

Error number 10,

In TINITM( ) one of the matrix dimensions (m or N) was less than one.

Rectangular matrices must have positive dimensions.

Error number 11,

In TINITM( ) the value of MN (matrix number) was less than one.

This error is similar to error number 4. Matrix numbers must be positive integers.

Error number 12,

In TINITM( ) the array dimension LMX was less than max(m,n) + 2.

The array dimensions of SX(LMX) and IX(4,LMX) are too small. Increase the value LMX and the dimensions of SX(*),IX(4,*) to meet or preferably exceed the above requirement.

Status numbers 13, 14, 15 and 16,

These status numbers are reserved for the monitoring of demand paging. Their function and use are described in the next section.

Error number 17,

In TGET( ) or TNNZR( ) subscripts for array element to be accessed were out of range.

A call to subprogram TGET( ) or TNNZR( ) attempted to access an element with row subscript greater than m or column subscript greater than n. All references must be to elements with subscripts within the declared matrix dimensions.

Error number 18,

In TGET( ) or TNNZR( ) a negative value for IPLACE(1) was discovered.

Since the values of IPLACE(*) represent virtual addresses, they are required to be non-negative.

Monitoring of Demand Paging

Status numbers 13, 14, 15, and 16 are used to monitor demand paging. Status numbers 13 and 14 keep track of page reads from low speed memory and page writes to low speed memory respectively. Whenever a page read or write occurs, the package automatically updates the counts for status numbers 13 or 14 by a call to subprogram TERROR( ). This call is a non-fatal call with error message printing suppressed. Thus, only the histogram count for these status numbers is incremented.

The user can return the histogram counts for status numbers 13 and 14 via a call to subprogram TERROR( ) with a value of IERROR = -15 or -16. A call to TERROR( ) with IERROR = -15 returns in IERROR the histogram count for status number 13, while a call to TERROR with IERROR = -16 returns in IERROR, the histogram count for status number 14. Since the value of IERROR is changed by subprogram TERROR( ), it must be a variable and not a constant. The histogram count represents the total number of page reads or page writes which have occurred during the course of this run for all matrices. A sample calling sequence to return the histogram count for the number of page reads is now illustrated.

```
      IERROR = -15
      IFATAL = 0
      IFMT = 0
      CALL TERROR(IFATAL,IERROR,IFMT)
```

Since no message is to be printed, IERROR < 0, the variable IFMT is set to zero because it is never accessed for the above call. Upon return from TERROR( ), the variable IERROR contains the histogram count for the number of page reads.

Modifications Required to Add a New Error Message or Status Features

Subprogram TERROR( ) is written to allow the easy addition of error messages to the sparse matrix package. The user can add an error message to the package by defining a new error number and variable format, and by making a call to subprogram TERROR( ) to perform the printing and tabulation functions. If an error number greater than 40 is required recompile TERROR( ) with the changes indicated in the description of error number 9.

To add a new status number similar to those described in the previous section, code must be added to subprogram TERROR( ). The existing code in TERROR( ) for error numbers 15 and 16 must be mimicked.

Appendix 2

Usage and Further Implementation of Demand Paging

Using Demand Paging on the CDC 6000-7000 Series

Subprogram TRWVIR( ) implements three methods of demand paging on the CDC 6000-7000 series. The method used is chosen by setting the variable METHOD equal to 1, 2, or 3 in subprogram TRWVIR( ).

For METHOD = 1, we use extended core storage or large core memory exclusively. The user must supply up to 131072_{10} words of extended core storage to the package. The system takes its storage from the start of extended core storage, using as much as it needs. Error number 8 occurs when more than 131072_{10} words of extended core storage are required.

For METHOD = 2, demand paging is implemented using random access disk alone. To use this method, the user must declare TAPE1 as a file on the PROGRAM card. The file TAPE1 corresponds to FORTRAN logical unit number 1.

For METHOD = 3, a combination of methods 1 and 2 is utilized. The user must supply up to the full 131072_{10} words of extended core storage to the package. The user must also declare the file TAPE1 on the program card. The package first uses the 131072_{10} word of extended core memory, and when that is exhausted, it uses random access disk for the overflow pages.

Implementing Demand Paging for a New Machine

In order to implement demand paging on a new machine, subprogram TRWVIR( ) must be modified or rewritten. This can be accomplished by mimicking method 1 or method 2 for the CDC 6000-7000 series. Most often, demand paging will be implemented using random access disk storage and hence method 2 will be mimicked.

Method 2 accomplishes three main tasks. It opens a mass storage file when the routine is first called. It computes a unique random access disk address for a given page number and matrix number, and it performs either the random access read or write depending on the input value of the subprogram argument KEY. A unique random access disk address is computable using the formula

$$\text{IADDR} = (\text{IPAGE} - 1)*\text{MAXMN} + \text{MN}$$

where IPAGE is the page number, MAXMN is the maximum allowed matrix number and MN is the matrix number associated with the given page being accessed. In Method 2, we write the real data and integer data as separate records so to generate a unique disk address for each record, we store the real data at random access address 2*IADDR - 1, and the integer data at random access address 2*IADDR. The record lengths for these two records are LPG and 4*LPG respectively.

Method 1 should be mimicked when some type of low-speed word addressable memory is to be utilized for demand paging. The offset in low-speed memory of matrix MN, page IPAGE is stored in the page table IOFF = IPTBLE(MN,IPAGE). The SX(*) array is stored in locations IOFF through IOFF + LPG - 1, and the IX(4,*) array is stored in locations IOFF + LPG through IOFF + 5*LPG - 1, of low-speed memory.

Distribution:

2613	K. Haskell
2613	R. E. Jones
2613	M. R. Scott
2613	H. R. Watts
5000	A. Narath
5100	J. K. Galt
5110	F. L. Vook
5120	G. J. Simmons
5121	H. T. Davis
5121	G. P. Steck
5121	R. J. Thompson
5122	D. E. Amos
5122	R. J. Hanson (40)
5122	L. F. Shampine
5122	J. A. Wisniewski (10)
5130	G. A. Samara
5150	J. E. Schirber
5160	W. Herrmann
5200	E. H. Beckner
5400	A. W. Snyder
5700	J. H. Scott
5800	R. S. Claassen
8266	E. A. Aas (2)
8325	R. E. Huddleston
8325	T. H. Jefferson
3151	W. L. Garner (3) For ERDA/TIC (Unlim. Release)
3171-1	R. Campbell (25) For ERDA/TIC