SPARSE MATRIX INVERSION

by

K. Ratliff

June 1971

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, 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.
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.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Background</td>
<td>1</td>
</tr>
<tr>
<td>2. Compile Time and Execution Time Processes</td>
<td>3</td>
</tr>
<tr>
<td>3. Pivot Selection Algorithm</td>
<td>5</td>
</tr>
<tr>
<td>4. The Inversion Process</td>
<td>8</td>
</tr>
<tr>
<td>5. The Symbolic Inversion</td>
<td>10</td>
</tr>
<tr>
<td>6. Register Optimization</td>
<td>15</td>
</tr>
<tr>
<td>7. Tests</td>
<td>18</td>
</tr>
</tbody>
</table>

### APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Diagram of Subroutine Interaction</td>
<td>20</td>
</tr>
<tr>
<td>II. Description of Arrays</td>
<td>22</td>
</tr>
<tr>
<td>III. Flowcharts of Sparse and Compile</td>
<td>27</td>
</tr>
<tr>
<td>IV. Calls to Compile</td>
<td>34</td>
</tr>
<tr>
<td>V. Code Generated for Various Cases</td>
<td>38</td>
</tr>
</tbody>
</table>
1. Background

In numerical problems one is often presented with an equation of the type

$$Ax = b$$

where $x$ and $b$ are vectors of length $n$ and $A$ is an $n \times n$ matrix. To obtain the value of $x = A^{-1}b$ is an expensive project in terms of computer time and memory if $n$ is large. Typically this computation is incidental to the main problem and $x$ may have to be re-evaluated many times due to changes in $b$ and/or $A$. Therefore the cost of the computation is not proportional to its importance.

Additionally the $A$ matrix may be sparse (i.e. having few nonzero entries). Standard matrix inversion routines using Gauss elimination require that the matrix be stored in its entirety. Further the ordinary pivot selection algorithm will in most cases introduce a large number of new nonzero entries in $A$, which in turn increases the number of arithmetic operations needed to solve for $x$.

The purpose of this report is to describe a program which saves a significant amount of computer time and space in the inversion of sparse matrices.
There are three factors which make the program more efficient for sparse matrices:

1. The pivot selection algorithm is designed to introduce few new nonzero matrix elements.

2. The matrix $A^{-1}$ is never actually computed. A record (in subroutine form) is kept of the operations required to produce the resultant vector $x$.

3. The $A$ matrix is not stored as a whole. Instead various arrays are constructed which give necessary information about the nonzero elements only.
2. Compile Time and Execution Time Processes

Prior to the inversion process the A matrix is analyzed and its nonzero elements are stored along with information about the elements. (See Appendix II). In particular the elements of A are classified according to their type; an integer is type one, a constant type two, and a variable type three.

The procedure used in obtaining the $x$ vector is Gauss elimination-back substitution performed on A and $b$. This procedure is a series of operations of the form:

$$\frac{a}{d} \cdot a$$

$$a - b \cdot c + a$$

The first type is used in the division of row elements by the pivot, while the second is used in row operations. The program, SPARSE, which formulates the sequence of such operations needed to produce $x$, outputs this information in the form of two subroutines, MATINV and MATMUL. The operations involving only elements of A are placed in MATINV; those involving $b$ are put in MATMUL. Thus when $x$ must be re-evaluated due to changes in the $b$ vector, only MATMUL need be called. When A changes, both subroutines must be executed.

Since the elements of A are classified by type, it is possible for SPARSE to recognize operations exclusively
involving elements which never vary (types one and two), and to perform these operations at compile time (during the compilation of MATINV and MATMUL). In such a case no code is compiled so that the compiled subroutines contain fewer instructions. This is quite a saving since SPARSE is executed only once, while MATINV and MATMUL are executed frequently.
3. **Pivot Selection Algorithm**

The pivot selection strategy is an important part of the program, since the choice of different pivots or the same pivots in a different sequence greatly effects the number of arithmetic operations necessary for the inversion. The selection is based on the number of nonzero elements in a row or column in order to avoid introducing new elements during the Gauss elimination.

The following are some of the pivot selection algorithms which were tested.

1. **Minimum column, minimum row:** This strategy compares all columns of $A$ with respect to the number of nonzero elements in the column. The column(s) which is minimum is then searched for the element with the minimum number of nonzero elements in its row. (The nonzero comparisons refer only to that part of $A$ not already involved in pivoting.)

2. **Minimum row, minimum column:** This algorithm is the same as the first applied to the transpose of $A$. 
3. Minimum product: Each matrix element is compared according to the product of the nonzero element count of its row and that of its column.

The basis for judging the efficiency of a method was the number of computer operations required to obtain $x$. (See Chapter 7.) The first two algorithms were found to be about equivalent. The minimum product strategy was much more efficient in some cases, but less efficient in the majority of examples.

The present program uses the first method. In the search for an optimal pivot candidate, two or more elements may have identical nonzero row and column counts. In such cases the new element, $b$, is tested to see if it is the integer one. If $b$ is not equal to one, its type (see Chapter 2) is compared with that of the former pivot candidate, $a$. If $b$ is of a simpler type than $a$, it is preferred as a pivot. If $a$ and $b$ are both type one (integers), $a$ is retained as the best choice. If $a$ and $b$ are both type two or three (constants or variables), they are compared by size and the larger (in absolute value) is chosen. In summary nonvarying elements, types one and two, are preferred to those
which change when the $A$ matrix changes, and large elements are preferred to small ones.

The motivation for choosing elements of types one and two rather than those of type three is that row operations or pivot divisions involving only nonvarying elements can be performed at compile time as explained in Chapter 2. Integers are preferred over constants, since integer arithmetic not involving division is exact.
4. The Inversion Process

The program consists of a group of routines which are divided into two types; those which are used initially to set up and analyze the $A$ matrix, and those used repeatedly to obtain the $x$ vector. Appendix I gives a diagram of the interaction of these programs.

The routine SETUP codes information about the $A$ matrix for use by SPARSE. It does this by differentiating a series of expressions $E_0$ to form

$$A_{ij} = \frac{\partial E_i}{\partial y_j} + a \frac{\partial E_i}{\partial y_j}$$

SPARSE performs the bulk of the decision making for the inversion. SPARSE calls COMPIL (an assembly language program) to generate the routines (MATINV and MATMUL) which manipulate $A$ and $b$ to obtain $x$.

As was mentioned previously, the inverse of $A$ is never calculated explicitly, rather SPARSE records only those operations which directly effect the final value of $x$. These essential operations are of two types which correspond to forming $A^{-1}$ and finding the product $A^{-1}b$. The former arithmetic is contained in the routine MATINV and the latter in MATMUL. Hence
if \( \mathbf{b} \) changes more often than \( \mathbf{A} \), the \( \mathbf{x} \) vector can be obtained frequently by calling \text{MATMUL} only. When the \( \mathbf{A} \) matrix changes, both \text{MATINV} and \text{MATMUL} must be called in that order.

Since all decisions pertaining to pivot selection, etc., are made by \text{SPARSE}, the generated routines are very simple and execute rapidly. They are straight-line code using only basic machine instructions (\text{LOAD}, \text{STORE}, \text{ADD}, \text{SUBTRACT}, \text{MULTIPLY}, \text{DIVIDE}, \text{LOAD COMPLEMENT}).

The \text{COMPIL} routine is also simple in that its function is to translate three parameters into one instruction and store that instruction in the appropriate location in memory. The parameters to \text{COMPIL} indicate the arithmetic operation, the array location containing the operand, and the subroutine (\text{MATINV} or \text{MATMUL}) which is effected. (See Appendix IV)
5. The Symbolic Inversion

This section will outline in more detail the process by which MATINV and MATMUL are generated. The SETUP routine uses eight arrays to store data about the A matrix. These arrays are described in Appendix II.

Since only nonzero elements are considered, information about the position of these elements must be available. This data, along with pointers to the next elements in the same row and column, information about the simplicity of the element, and the location of its numerical value are passed to the SPARSE routine.

Basically SPARSE analyzes the attributes of the A matrix and tries to determine the best code for MATINV and MATMUL. It can be divided into three sections: selection of a pivot, Gauss elimination and back substitution.

The pivot selection process uses the algorithm described in Chapter 3. The matrices used to store information about A contain information about the inversion process itself at different stages of the analysis of the matrix. The set of columns which has a minimum nonzero count is recorded by means of a chain. The pointers to columns in the chain are stored in the location formerly used for nonzero column counts.
(This space is no longer needed since all columns in the chain have the same nonzero count, which is stored in a single location.)

When all columns have been compared, those in the minimum chain are searched for the element which is the best pivot according to nonzero row count, type, and size. After a column has been searched in this fashion the information about its nonzero count is restored if the column is not used for pivoting. The space for the nonzero count on the column which does contain the pivot is set to zero during pivoting and to minus one after pivoting.

When the pivot element is chosen, the instructions to divide the elements in the pivot row and the b vector element corresponding to the pivot row are compiled (if the pivot equals one no instructions are compiled). As each pivot row element is divided, its row number is set negative to indicate that it is in a pivot row. Then the nonzero count in the column of this element is reduced by one. During the divisions the pivot element is neglected (as are elements in pivoted columns) and at the completion of the pivot division it is removed from the row chain. Finally the b vector element is divided by the pivot.
The elements in the pivot column are then checked to determine which ones are to be eliminated (those not in pivoted rows). When an element to be eliminated is found, its row is searched simultaneously with the pivot row to decide what row operations are necessary. The indicated instructions are compiled to change the elements in the eliminated-element row and the corresponding b entry. (See Appendix V)

During the elimination process the compiled arithmetic is also performed on the variable elements (type three) by SPARSE, since it is necessary to avoid choosing an element as pivot which has value zero or is very small.

As the pivot column is being searched in the elimination process, the pivot and eliminated elements are removed from the column chain (i.e. the preceding column element is made to point to the element following the one removed). The place in the MX array (see Appendix II) formerly occupied by an eliminated element is placed on a chain of free elements. These spaces are then reused when a new matrix element is generated by the elimination process. The pivot element is placed on a backward pivot chain (i.e. the beginning of the chain is the most recently chosen pivot). Pointers to elements in the free element chain are in the MX location normally used
to give the type of the element, while the back pivot chain
pointers occupy the location for the column chain pointers.

When all elements in the pivot column have been checked
(and eliminated where appropriate) another pivot is selected
and the procedure is repeated. (The last pivot requires little
processing, since it is the only nonzero element in its row.
Instructions are compiled to simply divide the $b$ entry by the
pivot value and store the result in $x$.)

The back pivot chain is used in the back substitution
process in the following way. The column of the first pivot
on the chain is searched for remaining elements and as they
are found, instructions are compiled to alter the $b$ vector in
those entries where the corresponding rows of $A$ have elements
in the pivot column. This continues until the pivot is the
only element in the column. Whenever an element is the last
nonpivot in its row, the $b$ entry for that row is stored in the
$x$ vector in the position corresponding to the column number
of the pivot in the row. This indicates that no further changes
are necessary in the row.

When a pivot column contains only the pivot element,
the pivot is removed from the back pivot chain and the process
is repeated for the next pivot. After all pivots have been
used in the back substitution process, the \( x \) vector is in its final form \( (x = A^{-1}b) \). Flow charts of SPARSE and COMPIL are given in Appendix III for more specific information on the mechanics of the program.
6. **Register Optimization**

The COMPIL subroutine generates the machine code for MATINV and MATMUL in such a way that all computation is performed in floating point registers 0 and 2, with general registers 1 through 15 used as base registers in addressing the arrays PW, XIN, and XOUT. (General register 0 holds the address of the beginning of MATINV or MATMUL at execution time.) MATINV and MATMUL are called with the addresses of the arrays as parameters so the subroutine can use relative addresses in their register-to-storage instructions.

The XIN and XOUT arrays have fixed length, \( n \), and are referenced using registers 14 and 15, respectively, as base registers. These are adequate for all \( n < 2^9 \). The PW array on the other hand has variable length depending on the number of nonintegers in the A matrix. The limitation on its length is imposed by \( n \) (i.e., \#PW entries < \( 2^{18} \)). Since there are only 13 index registers available for PW, at any one time only \( 13 \times 2^{10} \) locations can be accessed (if PW is double precision, this figure is further reduced to \( 13 \times 2^9 \)).

This restriction on the length of PW is circumvented in the following way. Initially the address of PW(1) is in register 1 and those of PW(n*2^{10}), n = 1, 12 in registers 2
through 13. (This is done by taking the PW parameter address as the value of register 1 and adding 4096 to this for each successive register.) Thereafter when a PW location not currently accessible is required, the base register used least recently is reset to contain the needed base address.

The recomputation of the base register value is done within the subroutines by instructions also generated by COMPIL. In order to accomplish this, COMPIL keeps a count of the instructions generated and stores the current count in a table in a position corresponding to the base register used in the instruction. The decision of what base address to discard is made by searching this table.

A second table is needed to indicate which base registers contain various base addresses. For example, if PW(14,000) is needed for an instruction, the entry in this second table corresponding to base address 14 (indicating PW(13312)) is checked. If there is a positive number in that position in the table, it gives the base register to be used for PW(14,000); if there is a negative number, no base register is currently being used for that part of the PW array. In the latter case the last used register is set to the desired base
address and the table entry for its former base address is made negative.

In the matrix inversion computations performed by the generated subroutines, floating point register 2 is used when possible for the values of elements which are used several times in succession (i.e., the value of the pivot during the division of the pivot row, or the value of the eliminated element during the row operations due to the elimination step). Floating point register 0 is used for those values which change with each operation (i.e., the pivot row elements to be divided or the elements in the pivot row used in operations of the type \( a - b \times c + a \)). In this way more economical register-to-register instructions can be used, with register 0 being reloaded for each new element used and register 2 remaining fixed for a period of several operations. These decisions are made within SPARSE.

Elements with special properties also provide the opportunity to cut down on the number of register-to-storage instructions. A comprehensive description of these properties is given in Appendix V.
7. **Tests**

Two types of tests have been employed in the development of the program. The first type checks the efficiency of the pivot selection algorithms and the second type determines the accuracy of the inversion.

The pivot selection algorithm test consists of varying the initial part of the SPARSE program so that different methods of pivot selection are used on the same examples with all procedures following the selection of a pivot remaining fixed. The number of times COMPIL is called (equivalent to the number of instructions generated for the subroutines) is compared for all methods tested. A summary of these tests was given in Chapter 3.

The accuracy tests refer to the validity of the matrix inversion procedure as well as to the numerical accuracy of the results. A check on the inversion procedure was accomplished by printing all operations performed during the execution of SPARSE and duplicating these by hand on the same matrices. The examples used in this way had eight as a maximum value of n.

The test for numerical accuracy (which of course implies a check on the inversion logic) is a program which
calculates an arbitrary $b$ vector and calls SETUP, MATSET, and SPARSE to produce MATINV and MATMUL for some $A$ matrix. The two generated routines are then executed to produce an $x$ vector. Finally the following is calculated

$$|Ax - b|$$

to measure the accuracy of the output vector, $x$. 
APPENDIX I

DIAGRAM OF SUBROUTINE INTERACTION
SETUP generates the arrays containing info. about A

MATSET evaluates those elements in A which are variables

SPARSE symbolically inverts A and premultiplies b by A⁻¹ (pivot selection, Gauss elimination, and back substitution)

MATINV performs arithmetic related to forming A⁻¹

COMPILE sets each machine instruction in memory in the appropriate subroutine

MATMUL performs arithmetic related to forming A⁻¹b
APPENDIX II

DESCRIPTION OF ARRAYS
MX is an integer array which is the primary source of information about the elements of A. It is dimensioned 6 x K, where K is the upper limit on the number of elements possible in A. (Usually n*n) There are six entries for each element which contain the following data.

MX(1,I) contains the type of the element, initially 1, 2, or 3, meaning integer, constant, or variable, respectively. If the type is 0, the element has been reduced to zero value and will be removed from A. This space is also used for pointers in a chain of free spaces in MX, where the previous element has been removed.

MX(2,I) contains the row number of the element. This is set to minus the row number when the row is pivoted.

MX(3,I) contains the column number of the element.
MX(4,I) points to the location of the next row element in MX.

MX(5,I) points to the location of the next column element in MX. If the element is a pivot this points to the next element in the backward pivot chain.

MX(6,I) contains the numerical value of type 1 elements and the location of this value in PW (see below) for type 2 and 3 elements.

IR1, IC1, IR2, and IC2 are integer arrays dimensioned 1 by n.

IR1(I) gives the location in MX of the first element of row I.

IC1(I) similarly gives the first element of column I.

IR2(I) contains the number of nonzero elements in row I.
column I. During the pivot selection this slot is a pointer to the next column in a chain of columns with minimum nonzero counts. IC2(I) is set to 0 while the I-th column is being pivoted and to -1 after pivoting.

PW is a floating point array dimensioned 1 by K where K is the upper limit on the number of constants or variables possible (usually n*n). It is used to store the numerical values of noninteger elements in A. The values of variables are set into PW by the routine MATSET which computes new values in A. The values of constants are set in PW by the routine, SETUP. The first twenty locations are filled with nonzero integers between -10 and +10. These locations are referenced when computation must be compiled involving an integer.

IQ and IP are integer arrays, dimensioned 1 by n and 1 by n**2, respectively, containing information about the position of elements in PW which are variables. These arrays are used by MATSET to compute the PW entries initially and each time the A matrix changes.

IQ(I) gives the number of elements to be computed in the I-th column of A.
IP(J) gives the row numbers of successive entries in PW which are variables.

For example, if IP(1) = 4 and the first four entries in IP were 6, 8, 11, 15, the nonzero variable elements in column 1 of A would appear in rows 6, 8, 11, 15.

XIN and XOUT are double precision arrays dimensioned 1 by n which correspond to the vectors b and x, respectively. They are used by MATMUL.
APPENDIX III.

FLOWCHARTS OF SPARSE AND COMPIL
SPARSE
GET COL. # J

3

HAS THIS COL. BEEN PIVOTED BEFORE? NO: GO TO 4

-1

-2

-3

GET COL. # J

3

IS THIS THE PIVOT COL.?

NO: MARK THIS COL. AS HAVING BEEN PIVOTED + REMOVE PIVOT FROM ITS ROW CHAIN

IS TYPE OF PIVOT EQUAL TO 1, 2, or 3?

NO: GO TO 4A

YES: CALL DIVISION TO BE COMPUTED

IS L=R?

NO: COMPUTE DIVIDED VALUE OF 1 AS A CONSTANT 4 STORE IN P(l)

YES: CALL DIVISION TO BE COMPUTED

IS THERE A DIVISION STILL BEING SAVED?

NO: MAKE THE SAVED ELT. THE CURRENT ONE

YES: IS THIS THE LAST ROW ELT.?

NO: GO TO 5

YES: PERFORM DIVISION

IS THE VALUE OF PIVOT = 1?

NO: GO TO 4

YES: COME TO DIVISION

IS TYPE OF 3 TO 2 OR 2 TO 3 DIVISION STORED IN P(3)?

NO: PERFORM DIVISION

YES: IS THERE A DIVISION SAVED?

NO: MAKE THE SAVED ELT. THE CURRENT ONE

YES: COME TO DIVISION

IS THE TYPE OF J ELT 1, 2, or 3?

YES: PERFORM DIVISION USING COEFFS WRITTEN INREW TYPE OF J ELT TO 2 + STORE IN P↓

NO: PERFORM CONSTANT ARITHMETIC

IS THIS ELT THE END OF PIVOT ROW?

NO: SET J = NEXT ROW ELT.

YES: COMPARE SAVED RESULT

IS THERE A DIVISION SAVED?

NO: MARK RESULT AS HAVING NO ELEMENTS BUT THE DIVIDED PIVOT

YES: STORE RESULT IN OUTPUT VECTOR

IS THIS THE LAST PIVOT TO BE CHOSEN?

NO: GO TO 6

YES: SET J = FIRST ELT IN PIVOT COL. J = 0

CALL COMPARE TO PUT VALUE OF RHS, IN R.3 FROM INPUT VECTOR; DIVIDE R.3 BY PIVOT VALUE

STORE RESULT IN OUTPUT VECTOR

GO TO 6

S P A R S E

-2-
THIS PAGE WAS INTENTIONALLY LEFT BLANK
HAVE WE REACHED END OF PIV. COLUMN?

NO  

GO TO 6

INCREMENT COUNTER FOR PIVOT SELECTION

YES

GET PIVOT AT START OF BACK PIVOT CHAIN (IPS) FOR EACH SUBSYSTEM - SET THIS EILT IF IPS NEXT EILT IN CHAIN=IPS

NO

HAVE ALL PIVOTS BEEN CHosen?

NO

GO TO 1

YES

GET PIVOT AT START OF BACK PIVOT CHAIN (IPS) FOR EACH SUBSYSTEM - SET THIS EILT IF IPS NEXT EILT IN CHAIN=IPS

YES

LOAD R.L FROM INPUT VECTOR OF RHS

GO TO 14

LOAD R.L WITH R.H.S. OF ALL ROW FROM OUTPUT VECTOR

NO

NOTE THAT RESULT GOES IN OUTPUT VECTOR

YES

HAVE ALL ELTS IN THIS PIVOT COL BEEN ELIM?

NO

SET N-2 COUNT ON ROW TO -1

SET 1 IN MU COL = SET = 1

GO TO 13

IS THIS THE LAST EILT IN THIS ROW TO BE ELIM?

NO

REDUCE N-2 COUNT IN ECLM. ROW

YES

15

ELIMINATED EILT. AN INTEG.

NO

GO TO 12

14

AS ELIM EILT = 11, -1 OR OTHER

-1

GO TO 14

COMPILE LOAD R.O FROM R.H.S. ADD R.O TO R.L

GET NEXT ELEMENT IN INPUT VECTOR OF ECLM. ROW

LOAD COMPLEMENT OF R.L INTO R.O MULT R.O BY ECLM EILT. AD R.H.S. OF E-ROW TO R.O

STORE RESULT IN INPUT VECTOR OF ECLM. ROW

SIGN OF RESULT GO INTO INPUT VECTOR

GO TO 14

COMPILE END OF SUBROUTINES AND RETURN

SPARSE

- 5 -
COMPILE (I,J,K)

=0

GET ADDRESS OF TABLE OF INSTRUCTIONS + PUT IT IN R.2

TEST K TO SEE IF Pw REFERES TO Pw XIN, OR XOUT

IS Pw SINGLE OR DOUBLE PRECISION

DIVIDE THE BASE AND DISPLACEMENT
BASE PART -> R.9
DISPL -> R.5

LOOK IN TABLE TO FIND WHICH R. TO BE USED AS BASE FOR THE VALUE IN R.9

IS THERE A REG. CURRENTLY MATCHING THE BASE NEEDED

FIND BASE REG. WHICH WAS USED LEAST RECENTLY + COMPILE INSTR. TO PUT VALUE CORRESPONDING TO Pw INTO IT.

GO TO 1

CONFIRM INITIAL INSTRUCTIONS FOR SUBRTS (TO STORE GEN. REG + RELOAD + BRANCH TO 1ST INSTR.)
NOTE IF Pw IS SINGLE OR DOUBLE PRECISION

I:0

>0

PUT INSTR. CORRESPONDING TO I IN R.5

IS THIS INSTR. FOR MATLAB OR MATINV

MATUL

MATINV

CHECK IF ON HALF OR FULL WORD BOUNDARY
STORE INSTR. IN MATLAB + INCREASE PTR TO END OF SUBRT BY 4

RETURN

CHECK IF ON HALF OR FULL WORD BOUNDARY
STORE INSTR. IN MATINV + INCREASE PTR TO END OF SUBRT BY 4

RETURN

IS INSTR. FOR MATLAB OR MATINV

MATUL

MATINV

CHECK IF ON HALF OR FULL WORD BOUNDARY
STORE INSTR. IN MATLAB + INCREASE PTR TO CURRENT END OF SUBRT BY 4

RETURN

CHECK IF ON HALF OR FULL WORD BOUNDARY
STORE INSTR. IN MATINV + INCREASE PTR TO CURRENT END OF SUBRT BY 4

RETURN

PUT DISPLAY IN ARRAY (I+8) IN R.5 + COMBINE BR. I4 WITH IT

PUT DISPLAY IN ARRAY (I+8) IN R.5 + COMBINE BR. I5 WITH IT

PUT BASE + DISPL. IN ARRAY IN R.5 (I+8)
PUT BASE + DISPL. IN ARRAY IN R.5 (I+8)

COBINE INSTR. IN R.5 WITH THE BASE REG. NOTE THAT THIS INSTR. USED THIS BASE REG.
APPENDIX IV

CALLS TO COMPIL
The form of the call to COMPIL is

Call COMPIL (I, J, K).

The following tables give the values of the three parameters and the corresponding interpretation.

The first parameter may assume 25 different values, most indicating a machine instruction involving floating point registers.

<table>
<thead>
<tr>
<th>I</th>
<th>If K=0 when I=0</th>
<th>If K=1 when I=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Compile end of subrts</td>
<td>same</td>
</tr>
<tr>
<td>0</td>
<td>Compile beginning of subrts</td>
<td>same</td>
</tr>
<tr>
<td>1</td>
<td>LE 2,-</td>
<td>LD 2,-</td>
</tr>
<tr>
<td>2</td>
<td>LE 0,-</td>
<td>LD 0,-</td>
</tr>
<tr>
<td>3</td>
<td>LD 0,-</td>
<td>same</td>
</tr>
<tr>
<td>4</td>
<td>STE 0,-</td>
<td>STD 0,-</td>
</tr>
<tr>
<td>5</td>
<td>STD 0,-</td>
<td>same</td>
</tr>
<tr>
<td>6</td>
<td>LCER 2,2</td>
<td>LCDR 2,2</td>
</tr>
<tr>
<td>7</td>
<td>LCDR 0,0</td>
<td>same</td>
</tr>
<tr>
<td>8</td>
<td>DER 0,2</td>
<td>DDR 0,2</td>
</tr>
<tr>
<td>9</td>
<td>MDR 0,2</td>
<td>MDR 0,2</td>
</tr>
<tr>
<td>10</td>
<td>DE 0,-</td>
<td>DD 0,-</td>
</tr>
<tr>
<td>11</td>
<td>ME 0,-</td>
<td>MD 0,-</td>
</tr>
<tr>
<td>12</td>
<td>DD 0,-</td>
<td>same</td>
</tr>
<tr>
<td>13</td>
<td>MD 0,-</td>
<td>same</td>
</tr>
<tr>
<td>14</td>
<td>AE 0,-</td>
<td>AD 0,-</td>
</tr>
<tr>
<td>15</td>
<td>AD 0,-</td>
<td>same</td>
</tr>
<tr>
<td>16</td>
<td>LCER 0,2</td>
<td>LCDR 0,2</td>
</tr>
<tr>
<td>17</td>
<td>LER 0,2</td>
<td>LDR 0,2</td>
</tr>
<tr>
<td>18</td>
<td>LCER 0,0</td>
<td>LCDR 0,0</td>
</tr>
<tr>
<td>19</td>
<td>SD 0,-</td>
<td>same</td>
</tr>
<tr>
<td>20</td>
<td>LD 2,-</td>
<td>same</td>
</tr>
<tr>
<td>21</td>
<td>LCDR 0,2</td>
<td>same</td>
</tr>
<tr>
<td>22</td>
<td>SDR 0,2</td>
<td>same</td>
</tr>
<tr>
<td>23</td>
<td>ADR 0,2</td>
<td>same</td>
</tr>
</tbody>
</table>
The second parameter gives the array index of the element to be used in register-to-indexed-storage instructions. In register-to-register instructions, it is ignored.

The third parameter may take on four different values which in general indicate the array referenced in r-x instructions and the subroutine (MATMUL or MATINV) which is being added to. The following table gives the values of $k$ and their meaning.

<table>
<thead>
<tr>
<th>$k$</th>
<th>When $I \neq 0$</th>
<th>When $I = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>array index refers to PW, instruction is for</td>
<td>PW array is double</td>
</tr>
<tr>
<td></td>
<td>MATINV</td>
<td>precision</td>
</tr>
<tr>
<td>1</td>
<td>array index refers to PW, instruction is for</td>
<td>PW array is single</td>
</tr>
<tr>
<td></td>
<td>MATMUL</td>
<td>precision</td>
</tr>
<tr>
<td>2</td>
<td>array index refers to XIN, instruction is for</td>
<td>not to be used</td>
</tr>
<tr>
<td></td>
<td>MATMUL</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>array index refers to XOUT, instruction is for</td>
<td>not to be used</td>
</tr>
<tr>
<td></td>
<td>MATMUL</td>
<td></td>
</tr>
</tbody>
</table>
To illustrate the manner in which instructions are generated for MATINV and MATMUL, take the following example.

The pivot element throughout the example is A(6,4); its value is in PW(13), which indicates it has value +3. There is a non-zero element in A(6,7) with value in PW(25). The element being eliminated in the Gauss elimination process is A(8,4), stored in PW(30). There is an element A(8,7) stored in PW(32). During the back substitution process there is an element A(2,4), located in PW(35); it is the last non-pivot in row 2, and the pivot for row 2 is in column 5.

<table>
<thead>
<tr>
<th>CALLS TO COMPIL</th>
<th>CODE FOR MATINV</th>
<th>CODE FOR MATMUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL COMPIL (1,13,0)</td>
<td>LE 2, PW(13)</td>
<td>More pivot row elements</td>
</tr>
<tr>
<td>CALL COMPIL (2,25,0)</td>
<td>LE 0, PW(25)</td>
<td>being divided will follow</td>
</tr>
<tr>
<td>CALL COMPIL (8,0,0)</td>
<td>DR 0,2</td>
<td>this repeating the last</td>
</tr>
<tr>
<td>CALL COMPIL (4,25,0)</td>
<td>STE 0, PW(25)</td>
<td>three instructions</td>
</tr>
<tr>
<td>CALL COMPIL (3,6,2)</td>
<td>The b element</td>
<td>LD 0, XIN(6)</td>
</tr>
<tr>
<td>CALL COMPIL (10,13,1)</td>
<td>is being</td>
<td>DE 0, PW(13)</td>
</tr>
<tr>
<td>CALL COMPIL (5,6,2)</td>
<td>divided</td>
<td>STD 0, XIN(6)</td>
</tr>
<tr>
<td>CALL COMPIL (1,30,0)</td>
<td>LE 2, PW(30)</td>
<td>More elements in the</td>
</tr>
<tr>
<td>CALL COMPIL (6,0,0)</td>
<td>LCER 2,2</td>
<td>eliminated row will be</td>
</tr>
<tr>
<td>CALL COMPIL (2,25,0)</td>
<td>LE 0, PW(25)</td>
<td>changed following this,</td>
</tr>
<tr>
<td>CALL COMPIL (9,0,0)</td>
<td>MER 0,2</td>
<td>repeating the last</td>
</tr>
<tr>
<td>CALL COMPIL (14,32,0)</td>
<td>AE 0, PW(32)</td>
<td>four instructions</td>
</tr>
<tr>
<td>CALL COMPIL (4,32,0)</td>
<td>STE 0, PW(32)</td>
<td></td>
</tr>
<tr>
<td>CALL COMPIL (2,30,1)</td>
<td>The b element</td>
<td>LE 0, PW(30)</td>
</tr>
<tr>
<td>CALL COMPIL (7,0,1)</td>
<td>in the row of</td>
<td>LCER 0,0</td>
</tr>
<tr>
<td>CALL COMPIL (13,6,2)</td>
<td>the elim elem</td>
<td>ME 0, XIN(6)</td>
</tr>
<tr>
<td>CALL COMPIL (15,8,2)</td>
<td>is being changed</td>
<td>AD 0, XIN(8)</td>
</tr>
<tr>
<td>CALL COMPIL (5,8,2)</td>
<td>ed due to elim</td>
<td>STD 0, XIN(8)</td>
</tr>
<tr>
<td>CALL COMPIL (20,6,2)</td>
<td>This is the</td>
<td></td>
</tr>
<tr>
<td>CALL COMPIL (21,0,1)</td>
<td>back subst</td>
<td>XIN(6)</td>
</tr>
<tr>
<td>CALL COMPIL (11,35,1)</td>
<td>proc which</td>
<td></td>
</tr>
<tr>
<td>CALL COMPIL (15,2,2)</td>
<td>puts the</td>
<td></td>
</tr>
<tr>
<td>CALL COMPIL (5,4,3)</td>
<td>result in x</td>
<td></td>
</tr>
</tbody>
</table>

This is the result in x
APPENDIX V

CODE GENERATED FOR VARIOUS CASES
This Appendix shows various cases of

\[ \frac{a}{d} + a \] and \[ a - b \cdot c + a \]

which occur in SPARSE, and the resulting computation. The following abbreviations are used:

- **PV**: Pivot element
- **P**: Pivot row element (≠ PV)
- **XINi**: Entry in \( b \) vector corresponding to row of \( i \) element
- **XOUTi**: Entry in \( x \) corresponding to column of pivot in \( i \) element row
- **T\#:** Type of element, \( T1 = \) Type one, etc.
- **E**: Eliminated element
- **X**: Eliminated row element (≠ E)

These are shown in the figure below.

```
ARRAY A:
```

```
[--- E --- P --- X ---]
```

Pivot Column
I. Operations of the Type $a/d \rightarrow d$ (if $PV = 1$, no operations required)

\[
P = T_1, PV = T_1 \text{ or } T_2
\]

form $P/PV \rightarrow P$ (if new $P = T_2$ get new space in $PW$)

\[
P = T_2, PV = T_1 \text{ or } T_2
\]

form $P/PV \rightarrow P$

\[
P = T_3 \text{ or } PV = T_3
\]

compile $*LE 2, PV$ (compiled only once for each pivot)

LE 0, P

DER 0,2

STE 0, P

When $a$ is in the $b$ vector

 MATINV CODE

compile $LD 0, XIN_{PV}$

DE 0, PV

**STD 0, XIN_{PV}

** If last pivot compile $STD 0, XOUT_{PV}$
II. Operations of the Type $a - b*c + a$

$$P = T_1, E = T_1, X = T_1$$  \text{NO CODE COMPILED}

form $X - E*P + X$ (if new $X$ is too large make $X = T_2$)

$$P = T_1 \text{ or } T_2, E = T_1 \text{ or } T_2, X = T_1 \text{ or } T_2$$  \text{NO CODE COMPILED}

form $X - E*P + X$ (change $X$ to $T_2$ if it was $T_1$, get space for it in PW array)

$$P = T_1 \text{ or } T_2, E = T_1 \text{ or } T_2, X = T_3$$  \text{MATINV CODE}

form $y = -P*E$

compile LE 0, $y$

AE 0, $X$ (if $X \neq 0$)

STE 0, $X$

$$P = T_3, E = \pm 1, X = T_1, T_2 \text{ or } T_3$$  \text{MATINV CODE}

compile LE 0, $P$

LCER 0, 0 (if $E = 1$)

AE 0, $X$ (if $X \neq 0$)

STE 0, $X$

\text{NOTE: In the following two cases the instructions}

LE 2, $E$

LCER 2,2 (omitted if $E = T_1$, since complement already formed) are compiled on the first row operation and are not repeated for subsequent operations for this $E$. Alternate code is given in footnotes if there is only one operation for this $E$. If $X = T_1$, it is changed to $T_2$ (floating point) for computation and space in PW is reserved for the new value. If $X = T_2$ space for the new $X$ is also reserved. If $E = T_1$ (not $\pm 1$) it is put in PW complemented.
\[ P = \pm 1, \ E = T3, \ X = E1, \ T2 \text{ or } T3 \]

**compile:**
- \( \text{LER } 0, \ 2 \) (P = 1) or \( \text{LCER } 0, \ 2 \) (P = -1)
- \( \text{AE } 0, \ X \) (if \( X \neq 0 \))
- \( \text{STE } 0, \ X \)

\[ P = T3, \ E \neq \pm 1, \ X = T1, \ T2 \text{ or } T3 \]

**compile:**
- \( \text{LE } 0, \ P \)
- \( \text{MER } 0, \ 2 \)
- \( \text{AE } 0, \ X \) (if \( X \neq 0 \))
- \( \text{STE } 0, \ X \)

* If only one row operation compile \( \text{LE } 0, \ E \)
  - \( \text{LCER } 0, \ 0 \) (if \( P = +1 \))
  - \( \text{AE } 0, \ X \) (if \( X \neq 0 \))
  - \( \text{STE } 0, \ X \)

** If only one row operation compile \( \text{LE } 0, \ P \)
  - \( \text{LCER } 0, \ 0 \) (if \( E \neq T1 \))
  - \( \text{ME } 0, \ E \)
  - \( \text{AE } 0, \ X \) (if \( X \neq 0 \))
  - \( \text{STE } 0, \ X \)
When a is in $a$ vector

\[
\text{MATMUL CODE}
\]

\[
\begin{align*}
\text{if } E \neq \pm 1 & \text{ compile } LE 0, E \\
& \quad LCDR 0, 0 \ (\text{if } E \neq T1) \\
& \quad MD 0, XINP \\
& \quad AD 0, XINE \\
& \quad STD 0, XINE \\
\text{if } E = \pm 1 & \text{ compile } LD 0, XINE \\
& \quad SD (\text{if } E = 1) \text{ or } AD (\text{if } E = -1) \ 0, XINP \\
& \quad STD 0, XINE \\
\end{align*}
\]

The back substitution process operations are of the form

\[a - b \cdot c + a\]

Whether a value is in XIN or XOUT depends on if the $x$ vector entry for the pivot row has reached its final value.

For each pivot initially compile

\[
\begin{align*}
\text{LD } 2, \ XINPV \text{ or } XOUTPV
\end{align*}
\]

\[
\begin{array}{ccc}
E = +1 & E = -1 & E \neq \pm 1 \\
LD 0, XINE & LD 0, XINE & LCDR 0, 2' \text{ or } XOUTE \\
SDR 0, 2 & ADR 0, 2 & MD 0, E \\
STD 0, XINE \text{ or } XOUTE & STD 0, XINE \text{ or } XOUTE & AD 0, XINE \text{ or } XOUTE \\
\end{array}
\]

NOTE: If a pivot is the only element in its row and the $x$ entry for its row has not been placed in XOUT by the above process

compile LD 0, XINPV

STD 0, XOUTPV