The implementation and use of sparse matrix techniques in general simulation programs

D. M. Brandon, Jr.

Engineering Department, University of Connecticut, Storrs, Connecticut 06268, USA

Classical matrix algebra requires excessive computer assets (core storage and execution time) when applied to the solution of simultaneous linear equations. Most physical systems, whose simulation eventually requires the solution of these equations, produce 'sparse' matrices. The approach described herein exploits this sparseness to minimise core storage, execution time, and round off error while providing a flexible programming base for a variety of related simulation operations.

(Received November 1972)

The developments in computer technology and applied mathematics in the last decade have unified many previously separate fields into a common disciplinary approach called 'systems engineering'. For continuous systems analysis, as in the engineering fields, the trend is towards a unified method of problem analysis as the state variable approach. With this approach (and others like it) a system is characterised in vector and matrix notation. The details of calculating the individual components of these vectors and matrices (which are not usually constant) is temporarily ignored to gain an appreciation for the general characteristics of the system, the relationship of variables one to another, and logical development of a common solution technique.

This type of methodology requires numerical techniques for the solution of systems of equations. Ordinary differential equations are broken down into simultaneous algebraic equations by implicit integration techniques. Partial differential equations can be broken down into ordinary differential equations by differencing methods. Nonlinear equations are broken down into linear equations by Newton-Raphson-like techniques. Thus, at the lowest level, a common solution technique must adequately provide for the solution of linear algebraic equations: $\bar{G}\bar{X} = \bar{H}$.

An 'adequate' solution technique should try to minimise computer execution time, core storage, and round off error. The classic method of solving sets of linear equations is that of direct Gaussian elimination (triangular factorisation). For N equations in N variables the number of operations (multiplications or divisions) required is:

$$\frac{N^3}{3} + N^2 - \frac{N}{3}$$

The minimum amount of core storage required is:

$$N^2 + N$$

There are many variants on this basic approach for full matrices. Some have eliminated needs for intermediate storage and recording requirements (such as the Crout or Doolittle modifications); some have reduced the round off error (as partial or complete pivoting strategies); some have reduced the number of multiplications required (Brent, 1970); and some have reduced the *order* of the number of multiplications (Strassen, 1969). However, as N becomes large the storage and execution time required become very large with all classical type methods that operate on full matrices.

Most physical systems give rise to 'sparse' matrices. That is the matrix containing the coefficients of the simultaneous equations (coefficient matrix) is mostly zeros. Some matrices such as tridiagonal or other 'banded' matrices have special algorithms that only store and operate on the non-zero elements. These algorithms for 'special' systems are fast and require little core. Some 'band' algorithms allow for interchanges with minimal increase in storage and execution time.

What are needed are general methods that will handle any matrix of 'general' sparseness using only the minimum amount of core and execution time necessary and capable of reducing round off error. The approach to be described in this paper is such a technique and represents the basis of the *IMP* system (Brandon, 1972).

There are, in the literature of the last few years, several 'sparse matrice techniques'. These are usually implemented as optimised utility subroutines. Many problems arise with the use of these methods in a simulation system as:

1. The applicability of the sparse indexing scheme to simulation tasks (other than the direct elimination process for which it was designed).
2. The sharing of common resources (scratch storage and intermediate results) with other simulation tasks.
3. The addition of necessary error monitors, controls, and diagnostic messages.
4. The combination of the matrix generation step with the elimination step for maximum efficiency.
5. The use of alternate solution techniques for special systems (such as the iterative solution of simultaneous tasks).

The purpose of this paper is, therefore, to present the basics of our approach, the types of processing to which it has successfully been applied, and some results (computer assets required) of its application. Our discussion will be confined to *sequential* digital computers (without dynamic core allocation) and we will use FORTRAN notation. More detail may be found in the *IMP* general manual which is available upon request. The following symbols will be used:

Nomenclature

A	Coefficient Subarray
$\bar{A}$, a_{ij}	Coefficient Matrix (or Jacobian Matrix)
$\bar{B}$, bi	Input or Constant Vector
CORE	Real Word Array
E	Average Number of Non-Zero Elements in an Equation
G	Solution Subarray
$\bar{G}$, g_{ij}	Solution Matrix
h	Heat Transfer Coefficient
$\bar{H}$	Solution Right Hand Side Vector
$\bar{I}$	Identity Matrix
ICORE	Integer Word Array
ICUMA	Coefficient Matrix Cumulative Subarray
ICUMG	Solution Matrix Cumulative Subarray
IMP	Implicit Solution Software System
JHA	Coefficient Matrix Column Storage
JHG	Solution Matrix Column Storage

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022

k	Thermal Property
N	Number of Equations (and Variables)
NELA	Number of Non-Zero Elements in $\bar{\bar{A}}$
NELG	Number of Non-Zero Elements in $\bar{\bar{G}}$ (Upper Triangular Only)
NEQN	Number of Equations
L	Length of Plate
ONE	Real Word Common Block
t	Time
T	Temperature
T_s	Surface Temperature
TWO	Integer Common Block
$\bar{X}$	State Vector
X	Space Coordinate
Y	Space Coordinate

Superscripts

.	Denotes Derivative
D	Denotes Diagonal Vector
–	Denotes Vector
=	Denotes Matrix

Subscripts

i	Denotes Row Index
j	Denotes Column Index
n	Denotes Time or Iteration Step Index

Automatic core allocation

A software system should be capable of expanding its storage requirements to the size and topology of the problem at hand and also to the particular solution option chosen. To use the *IMP* system the user writes a main program which declares two common blocks and sets up their length based upon storage formulas. One common block is for real numbers ('CORE') the other is for integers ('ICORE'). Thus, all information is in the common blocks and not transferred as addresses through subroutine parameters. Most computers allow variable size common blocks provided the longest is loaded first. Since the longest will always be in the user main program (and *IMP* subroutines are on the system library) this arrangement will be effective.

This also allows users to alter their main program declarations, for example, to double precision or half word (or less) integer storage. There are corresponding versions for different computers to handle these different declarations. The manual provides guidelines as to how big a 'CORE' (real number storage) and 'ICORE' (integer number storage) to declare. In addition, the output (after one run) will give the exact amount necessary, so that future runs of the same problem will only have to use the minimum storage required (one or more debugging runs are usually necessary anyway).

The only arrays used in *IMP* are 'CORE' and 'ICORE'. Various portions of these arrays are allocated to subarrays which are designated by cumulative starting address pointers (**Fig. 1**). During the remainder of this paper the vectors referred to are not arrays in the actual program, but are subarrays imbedded in 'CORE' and 'ICORE'. Many subarrays are not used at the same time (different solution options or scratch storage in different subroutines). These are given the same pointer values, thus 'equivalencing' is achieved. For some options some subarrays are not necessary at all. The corresponding pointers are given zero values so that *storage collapses to the minimum required for a particular problem using a particular solution option.*

Coefficient matrix storage allocation and setup

The coefficient matrix, $\bar{\bar{A}}$, in the equation:

$$\dot{\bar{X}} = \bar{\bar{A}}\bar{X} + \bar{B}$$

is stored in a vector (subarray) called 'A' (the coefficient matrix

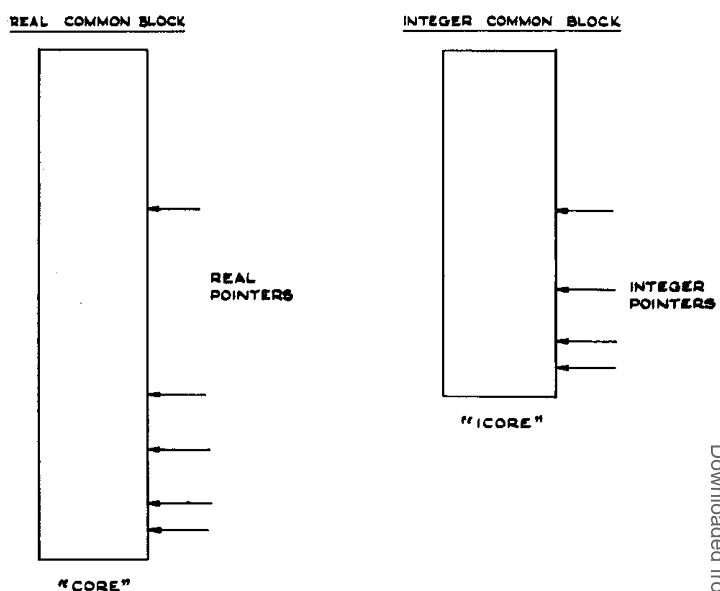

Fig. 1 Data structures

may be a Jacobian matrix, parts of the vector $\dot{\bar{X}}$ may be zero or all zero (totally algebraic system)). The user defines the values of $\bar{\bar{A}}$ and $\bar{B}$ in his user written subroutine 'PARM'. Once a user sets values for $\bar{\bar{A}}$ and $\bar{B}$ they are not altered by the system until the user changes them. Thus, constant elements of $\bar{\bar{A}}$ and $\bar{B}$ need only be setup once (the exception to this rule occurs if a user is using an iterative rather than a direct solution technique).

The non-zero entries of $\bar{\bar{A}}$ are stored in 'A' consecutively by a row wise scan. Thus:

$$\bar{\bar{A}} \quad \begin{vmatrix} 1 & 2 & 1 & 2 \\ 0 & 2 & 4 & 2 \\ 0 & 2 & 4 & 0 \\ 1 & 0 & 0 & 2 \end{vmatrix} \rightarrow \quad 'A' \begin{vmatrix} 1 \\ 2 \\ 1 \\ 2 \\ 2 \\ 4 \\ 2 \\ 2 \\ 4 \\ 1 \\ 2 \end{vmatrix} \qquad (1)$$

Before elements can be stored in 'A' however, the topology of the coefficient matrix must be described to the system. This can be read from a file or set up automatically for certain special matrix structures as tridiagonal or banded. Basically the system finds out the column numbers of non-zero elements for each row. These are put in a vector, 'JHA', as for the previous example (1):

$$\begin{vmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 2 \\ 3 \\ 4 \\ 2 \\ 3 \\ 1 \\ 4 \end{vmatrix}$$

A cumulative vector is also formed to separate the above list into rows. The vector is called 'ICUMA', and would look like (for example (1)):

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022

4
7
9
11

This technique of reducing a sparse matrix, N by N, into three vectors is called 'cumulative indexing'. The length of the vector 'A' and 'JHA' is the number of non-zero elements and the length of 'ICUMA' is N. This type of storage technique is used for the solution matrix '$\bar{G}$' also (this will be discussed later). In addition to the obvious reduction in storage for large sparse matrices, cumulative indexing offers two other major advantages:

1. All secondary N looping operations are removed. This reduces computer processing time quite considerably. As explained earlier for classical Gaussian elimination, there are looping operations of order N^3. For example, to multiply $\bar{A}$ by $\bar{X}$ of the previous example, the code will normally be:

```
     DO 1 I = 1, N, 1
     SUM = 0·0
     DO 2 J = 1, N, 1
2    SUM = SUM + A(I, J) * X(J)
     F(I) = SUM
1    CONTINUE
```

Thus, N^2 operations are involved (a logical check could be inserted to jump around the multiplication if $A(I, J)$ or $X(J)$ was zero, but this takes as much or more processing time as a multiplication). With cumulative indexing the code is:

```
     IL = 1
     DO 1 I = 1, N, 1
     SUM = 0·0
     IT = ICUMA (I)
     DO 2 II = IL, IT, 1
     J = JHA (II)
2    SUM = SUM + A(II) * X(J)
     F(I) = SUM
     IL = IT + 1
1    CONTINUE
```

The cumulative indexing code involves $N \cdot E$ operations and is also faster by virtue of the fact that *only singly subscripted arrays are used*. Elimination algorithms are on the order of NE^2 rather than N^3 (where E is the average number of non-zero elements per equation).

2. *All* operations in *IMP* are *row oriented* and take full advantage of cumulative indexing to reduce the order of calculations. This includes matrix generation, optimal ordering, determination of derivatives and integrating factors, iterative methods, element placing, norms, scaling, error monitors, topology generation, output, retrieval, etc.

Matrix generation

The solution matrix, $\bar{G}$ (resulting algebraic system at each time step), is obtained from the Jacobian matrix $\bar{A}$ and input vector $\bar{B}$ by use of a single step inplicit integration algorithm (Brandon, 1972). A discussion of this algorithm is beyond the scope of this paper and may be found in the manual. The matrix generation step is actually done with the solution procedure and is not a separate step (except in the interative techniques). This saves execution time and only requires the upper triangular portion of $\bar{G}$ (not including diagonal) to be stored.

Iterative solution techniques

For iterative solution, $\bar{G}$ is formed (written over) on $\bar{A}$, and by a series of iterations $\bar{X}$ is obtained. There are many possible iterative schemes (Jacobi, over-relaxation, Gauss-Seidel, successive over-relaxation, gradient, conjugate gradient, etc.). Numerical testing has shown the double sweep version of Gauss-Seidel to be, in general, the best technique for use with the *IMP* integration method on a variety of partial differential equations. This method does not require strict diagonal dominance although radius of convergence is restricted as shown in **Fig. 2.** The convergence is a function of diagonal dominance, which can be achieved (for differential systems) by taking a smaller time step. The automatic integration step size option of *IMP* takes this into account and the two work together very well. The Gauss-Seidel solution option is usually better than direct elimination for partial differential equations in two or more space coordinates or nearly diagonally dominant large sparse matrices of large bandwidth. The example shown in Fig. 2 compares this iterative technique with the direct method of *IMP*, the popular alternating directions method (uses tridiagonal elimination algorithm), and tridiagonal iteration or 'the strongly implicit method' (Weinstein, 1968). For all methods, the integration method of *IMP* was used (if either the classical trapezoidal method or Euler implicit method is used, the computation times are longer for equivalent accuracy). The results are shown for 25 and 225 differential equations (M) approximating the partial differential equation. As M increases not only the number of equations increase but the bandwidth increases and eigenvalues become more distant.

Optimal ordering for direct solution

The order in which rows are eliminated in sparse matrices effects the amount of processing time required and the number of non-zero elements generated ('fill-in'). There are many different schemes for 'optimal ordering'. The optimality of the order produced must be judged against the time required to do the ordering in the various methods. There are three general classes of methods:

Break up $\bar{G}$ as:

$$\bar{G} = \bar{L} + \bar{U} + \bar{I}$$
$$\bar{X}^{p+1} = -(\bar{I} + \bar{L})^{-1}\,\bar{U}\bar{X}^{p} + (\bar{I} + \bar{L})^{-1}\,\bar{H}$$
$$\bar{X}^{p+2} = -(\bar{I} + \bar{U})^{-1}\,\bar{L}\bar{X}^{p+1} + (\bar{I} + \bar{U})^{-1}\,\bar{H}$$

($\bar{L}$ = Lower Triangular) ($\bar{U}$ = Upper Triangular)

converges if:

$$\|(\bar{I} + \bar{L})^{-1}\,\bar{U}\| < 1$$

or

$$\|(I + \bar{U})^{-1}\,\bar{L}\| < 1$$

Example problem (square plate)

$$\frac{\partial T}{\partial t} = k\left(\frac{\partial^2 T}{\partial X^2} + \frac{\partial^2 T}{\partial Y^2}\right) - (T - T_s)\,hT$$

$\partial T/\partial X = 0$ @ $X = 0$ & $X = L$
$\partial T/\partial Y = 0$ @ $Y = 0$ & $Y = L$
$T(X, Y) = 100$ @ $t = 0$ FOR ALL X, Y

$T_s = 500$ IN A CENTRE REGION OF PLATE @ $t > 0$
$T_s = 0$ ALL OTHER REGIONS OF PLATE

Results (full transient simulation)

	Total time (*CDC* 6600 *CPU sec.*)
Direct ($M = 25$)	8·17
Gauss Seidel-Double Sweep ($M = 25$)	15·67
Alternating Directions ($M = 25$)	17·14
Strongly Implicit ($M = 25$)	7·81
Direct ($M = 225$)	528·32
Gauss Seidel-Double Sweep ($M = 225$)	74·80
Alternating Directions ($M = 225$)	445·15
Strongly Implicit ($M = 225$)	152·25

Fig. 2 Iterative solution Gauss Seidel (double sweep)

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022

1. The very simple methods—such as eliminating the row with the fewest elements first, minimum degree algorithms, etc.
2. The methods which try every possible order.
3. Methods in between 2 and 3. These are basically heuristic methods such as bandwith minimisation, once through operations count, fixed depth tree searches, etc.

There are five basic ordering options in *IMP*:

1. The original order may be preserved. Thus, the user may pre-order his system as he desires. This is usually the best approach for 'regular' topology problems such as partial differential equations, etc.
2. The order will be read from a user specified file. If complex ordering is done on one run of a problem, it can be preserved (written on a file) to be read in on future runs of the same problem.
3. Bandwidth minimisation (by interchanging rows only) with no check for creation of diagonal element.
4. Bandwidth minimisation with elimination simulation to check for generation of diagonal elements.
5. Elimination simulation with operations count and check for generation of diagonal element.

Methods 4 and 5 are more time-consuming than 3 since the elimination procedure is simulated for each candidate row at each position. This ordering is done during initial processing. No knowledge of the magnitude of the elements is known—only the topology of $\bar{A}$. As will be discussed later, this order may be altered to reduce round off error. Therefore, this initial ordering represents 'preferred order'.

Row operations

Previously, the use of cumulative indexing was illustrated for the multiplication of a vector and a matrix. Most operations of a simulation system are similar to this and their conversion from 'matrix algebra' to 'vector algebra' is straightforward. An exception to this occurs with the use of 'row operations'. This is defined as the linear combination of two rows of a matrix and is used extensively in the direct elimination process. The aim is, of course, to combine the rows using the minimum number of operations and preserving sparsity.

The problem can be thought of as:

$$Ri = Ri - \sum \alpha j\, Rj \quad i \neq j$$

where:

Ri = one row of a matrix
Rj = other row of a matrix
α = constant multiplier

Now Ri and Rj are in compact storage with their associated integer pointers as previously illustrated. The most popular row operation algorithms require the 'fill-in' to be determined in advance and stored (Rose, 1972; Chang, 1968). This is an optional procedure in *IMP*. The data structures G, JHG, and ICUMG (similar to A, JHA, and ICUMA) are used to hold the $\bar{G}$ matrix *including future filled-in locations*.

There are several disadvantages to this 'approach *I*' however:

1. If the fill-in is to be known in advance the order of elimination must be fixed. This has associated numerical difficulties to be discussed later.
2. If it is desired to preserve $\bar{A}$(A, JHA, and ICUMA), then the storage required is larger than minimum.

The standard procedure of *IMP* called 'approach II' uses a 'dynamic' technique to allocate filled-in storage as necessary. A real number workspace vector of length N is filled with a row of $\bar{A}$ (using A, ICUMA, and JHA). As new rows ($\alpha j\, Rj$) are numerically combined with the Ri workspace (using G, ICUMG, and JHG, which have been set up for rows above the current row i) an unordered integer list is maintained containing active columns of the workspace. One scan through this integer list indicates the next row Rj to be combined with Ri since the integers in the list are also flagged when their corresponding row Rj has already been processed. After row Ri has been combined with all necessary Rj, the integer list is ordered and used to set up G, ICUMG, and JHG for that row. Thus, the storage allocation for $\bar{G}$ is dynamic and a row i may be rejected at any point for numerical considerations. Only the upper triangular portion of $\bar{G}$ is saved with fill-in (no diagonal elements either)' In addition, only partial row operations are carried out since any location of G corresponding to an eliminated position (a numerical result of 0·0 (or 1·0 on the diagonal)) is not operated upon. In addition to the above savings in storage and execution time, other important advantages are:

1. Since knowledge of the fill-in is not needed the elimination order may be varied (this will be discussed later). Also an 'elimination simulation' to determine fill-in ahead of time is not required.
2. There is much more flexibility to add optional matrix and vector scaling, element rejection, etc.
3. $\bar{A}$ may be preserved since only the upper triangular portion of $\bar{G}$ is kept. This is important since elements of $\bar{A}$ may be expensive to calculate or cumbersome to reinitialise.

With its advantages this 'approach II' does have two drawbacks:

1. All the factors resulting from elimination (the full $\bar{G}$) are not saved thus not permitting the standard method of 'iterative improvement'. This disadvantage is circumvented by a number of procedures to reduce round off error which are discussed later. If one were going to do the standard iterative refinement, certain other numerical information would have to be stored as elimination proceeded (in double precision).
2. If all of $\bar{G}$ were saved, the solution to:

$$\bar{G}\bar{X} = \bar{H}$$

could be quickly obtained for new $\bar{H}$'s, if $\bar{A}(\bar{G})$ did not change 'too much'. However, usually $\bar{A}$ and $\bar{B}$ change constantly during a transient (and even during time steps) due to the integration algorithms; also $\bar{A}$ is in general a Jacobian matrix.

Both 'approach *I*' and 'approach *II*' are available to *IMP*. A more accurate solution is usually obtained by 'approach *II*' and the advantages of this approach outweigh the disadvantages. Both approaches require about the same execution time.

Direct solution

The direct solution of linear equations involves an elimination step followed by back substitution. Since the order is variable, the generation, storage allocation, and elimination are all done simultaneously at each iteration. Since the storage allocation (for $\bar{G}$) is variable *the actual solution algorithms change as the ordering changes*. A modified Crout algorithm is used with the list processing techniques previously described. *A 'check column' is used to monitor round off error and report this to the user if it becomes significant.*

The area of sparse matrix techniques is very recent and relatively little work is found in the literature on general methods (Reid, 1971; Rose, 1972). The opinions discussed in this paper are the result of studying other approaches; indeed several of these previously published methods were tried in early versions of the *IMP* system and the disadvantages discussed were found. This subject is quite controversial, however, and 'hard' comparisons are difficult due to different types of problems considered, different criterion for effectiveness, and different functions requiring different computer assets on different types of hardware.

One of the earliest methods to solve general sparse matrix systems (other than completely iterative methods) was that of

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022

decomposition and tearing. This method combines ordering, decomposition, direct solution by full matrix methods and iteration (Steward, 1965). It is not applicable to large systems as the decomposition and tearing execution time becomes very large. The iterative methods of recombining the decomposed subsystems may often have convergence difficulties.

Several techniques using storage similar to cumulative indexing have been used for specific classes of simulations (Tinney and Walker, 1967; Chang, 1968; Rose, 1972). The first of these uses chained list indexing which requires more storage and is slower than the *IMP* methods. The others use 'approach *I*' which has the disadvantages discussed previously.

The indexing necessary in *IMP* (and the above methods) has some overhead of computer execution time associated with it. An alternative would be to first go through an optimal ordering step and then symbolically generate a non-looping code to solve the equations in that fixed order (Gustavson, Liniger, Willoughby, 1970; Hatchel, Brayton, Gustavson, 1971). This would be faster than the previous methods (if all the code were in core), but has many disadvantages as:

1. Once optimal ordering is accomplished and the code generated, the order of elimination is fixed. This could lead to significant round off error. (However, *only if* $\bar{A}$ *were constant*, ordering could initially be done using the magnitude and location of $\bar{A}$ elements.)
2. The symbolic generation of the code takes a *long* time. A problem has to be run many times to justify the cost of generating the code.
3. The generated code requires a lot of core for large problems. It can be buffered in when needed, but this substantially decreases efficiency.
4. For a general system (such as *IMP*) indexing (or added code generation) would have to be added for integration algorithms and non-linear programming anyway.
5. The technique cannot be entirely implemented in FORTRAN and thus conversion from one computer to another would be difficult.

In summary, the vector algebra approach described here provides for the general solution of systems of equations, exploiting sparseness (core storage and execution time minimisation), minimising round off error, and providing a flexible programming base for other simulation tasks. The storage formulas for *IMP* are:

Real Words = NELA + NELG + 11*NEQN
Integer Words = NELA + NELG + 5*NEQN

where:

NELA = Number of elements in coefficient matrix, $\bar{A}$.
NELG* = Number of elements in solution matrix, $\bar{G}$. (Upper triangular portion only—not including diagonal)
NEQN = Number of equations.

For some types of computers the integer words are packed two or more to a real word. The above formulae include *all storage* for numerical integration, non-linear programming, system book-keeping, etc. Little storage is added by the *IMP* code which occupies about 4K words.

Examples

These examples concern the use of *IMP* to solve large sets of algebraic systems by direct elimination. An equation generation program was used to produce random equations to be solved. The input to the generation program was:

1. Number of equations to be generated.
2. Number of non-zero elements to be created per row.
3. Bandwidth in which random elements would be generated.

The generation program first picked a random integer row centre number indicating a position in a list of row centres which have not been picked yet. It then generated different random integer numbers within the bandwidth of the row centre. This information was written to a file (topology file). Next the program generated the random numbers (between minus twenty and plus twenty rounded to nearest tenth) associated with the previously generated topology. This was also written to a file.

As an example, suppose the number of equations to be generated is one hundred. The number of elements per equation is five, and the bandwidth is twenty. The first row to be generated might be centred around 35 say. Position 35 and four other random positions between 25 and 45 would be given non-zero elements. Many of these such random matrices were generated as *IMP* was being developed. For testing purposes right hand sides were generated also. These were the sum of the row

*This is zero for an iterative solution.

Table 1[1]

	Total Time[4]	*Elimination Time*[4]	*Total Dimensioned Words*
Classical method[2]	17·26	12·11	10,500
IMP[3]	3·05	0·68	3,776

[1]100 equations, five elements per row, bandwidth generation of twenty, determinant = −0·744E + 84
[2]Gauss elimination with partial pivoting
[3]Variable order elimination (EPS = 1·0 E-10)
[4]CDC 6600—CPU sec; single precision IMP version used (60 bits/word).

Table 2 Effect of number of equations

Number of Equations[1]	$\bar{G}$ *Storage*	*Total Time*[2] *(CDC 6600-CPU Sec)*	*Elimination Time*[3] *(CDC 6600-CPU Sec)*
100	576	3·05	0·68
250	1,483	6·61	1·60
500	2,971	16·87	3·51
1,000	5,955	52·54	7·61

[1]Five elements per row, randomly generated within a bandwidth of twenty
[2]Includes:
(*a*) Finding IMP on system library
(*b*) Compiling MAIN and PARM
(*c*) Linking and loading
(*d*) Reading topology data from cards
(*e*) Optimal ordering by bandwidth minimisation
(*f*) Reading in matrix values from cards
(*g*) Elimination and solution output
[3]Time for elimination (variable order) and solution output

Table 3 Effect of bandwidth

Bandwidth[4] *of Generation*	$\bar{G}$ *Storage*	*(CDC 6600-CPU Sec)* *Total Time*	*Elimination Time*
10	850	5·87	0·88
20	1,483	6·61	1·60
30	2,008	8·02	3·04
40	2,668	10·46	5·47

[4]250 equations, five randomly generated elements per row

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022

elements such that the solution to:

$$0 = \bar{A}\bar{X} + \bar{B}$$

should always be $\bar{X} = -1$.

When a program such as this is executed, the computer time shown in the printout will contain:

1. Time to find *IMP* on system library.
2. Compiling MAIN and PARM.
3. Linking and loading.
4. Reading topology data.
5. Optimal ordering by bandwidth minimisation.
6. Reading in matrix values.
7. Elimination (along with matrix generation, scaling, pivoting, and $\bar{G}$ storage allocation).
8. Solution output to line printer.

The elimination and output time alone is the statistic we seek so each generated matrix problem will be run twice. The first run will give the total time and the second run will give the time for initial processing only (items one through six). Thus, the computer time for elimination and output will be isolated.

Table 1 compares the solution of a 100 equation sparse matrix of random generation using *IMP* versus a classical Gauss elimination with partial pivoting program (as might be found in a scientific subroutine package). As the number of equations increases, the core storage and computer time savings of *IMP* over classical methods increases faster. This execution time is about the same order as other techniques reported in the literature. For example, Curtis (1971) reports a solution time of about 3·7 IBM 360/75 seconds for 100 equations with 505 random non-zeros. Since *IMP* handles differential as well as algebraic systems, there is some overhead embedded in the computer times given for totally algebraic systems. **Table 2** shows computer time and $\bar{G}$ matrix storage as a function of the number of equations. Notice that both core storage and computer elimination time are *LINEAR* in the number of equations. **Table 3** shows computer time and $\bar{G}$ matrix storage as a function of bandwidth of generation. **Table 4** shows computer time and $\bar{G}$ matrix storage as a function of the number of non-zero elements per row. Comparison of these last two tables shows that $\bar{G}$ storage and elimination time is more sensitive to the location of elements in the coefficient (or Jacobian) matrix, $\bar{A}$, than to the number of elements. Initial processing is more sensitive to the number of non-zero elements. **Table 5** shows initial processing times as a function of ordering option. Ordering by operations count is much more expensive.

Table 4 Effect of sparsity

Number of[1] *Elements per row*	$\bar{G}$ *Storage*	(*CDC* 6600-*CPU Sec*) *Total Time*	*Elimination Time*
5	1,483	6·61	1·60
7	1,775	8·20	2·21
10	2,078	10·41	2·83

(See notes on Table 2).
[1]250 equations with elements generated randomly within a bandwidth of twenty

Table 5 Ordering expense

Ordering Option[1]	*Initial Processing Time*[2]
Original order	2·34
Order specified by user (read in)	2·39
Bandwidth minimisation	2·37
Operations count	23·12

[1]100 equations, five elements per row, bandwidth generation of twenty
[2]Does not include elimination and solution output time

Table 6[1,2] Effect of pivot magnitude and scaling

EPS	*ICOND*	*Elements in* $\bar{G}$	# *Bad*[3]	*Max. Error*
1.OE-6	0	2,006	245	1·22 E-1
1.OE-4	0	2,008	77	9·86 E-3
1.OE-2	2	2,013	7	8·52 E-3
5.OE-2	2	2,007	27	5·42 E-3

[1]250 equations, five elements randomly generated per row in bandwidth of thirty.
[2]Single precision IMP version for IBM 360/65 used (32 bits/word).
[3]Number of solution values with round off greater than 1·0 E-3 (maximum of 250).

Numerical difficulties

The major difficulty that may arise in the solution of differential systems with an implicit integration algorithm is the round off error in the solution of the resulting simultaneous equations. The equations to be solved are:

$$\bar{G}\bar{X} = \bar{H}$$

It can be shown that the determining factor in the error accumulation is the condition of the coefficient solution matrix $\bar{G}$ of COND ($\bar{G}$). Let:

$$\bar{P} = \bar{G}^T\bar{G}$$

and let τ_{max} be the maximum eigenvalue of $\bar{P}$, and let τ_{min} be the minimum eigenvalue of $\bar{P}$. Then:

$$\text{COND}(\bar{G}) = (\tau_{max}/\tau_{min})^{\frac{1}{2}}$$

or

$$\text{COND}(\bar{G}) = \|\bar{G}\| . \|\bar{G}^{-1}\|$$

Thus, the value of row interchanges and matrix scaling is apparent.

In the classical matrix solution of linear equations, a complete pivoting or partial pivoting approach is usually taken. This usually reduces round off error quite significantly, although for some systems it may increase error. In the solution of systems by a sparse matrix approach, a partial pivoting strategy greatly compromises the sparseness of the final matrix (fill-in and number of operations is increased). With the variable elimination scheme of *IMP* a minimum diagonal pivot size may be specified. This reduces round off error, eliminates the difficulty of finding true zeros on the diagonal, and does not compromise sparseness nearly as much as partial pivoting. The minimum pivot value is specified by the user (EPS) and is a function of the word length being used in the calculations and the accuracy desired.

Three variations of the basic strategy may be followed (depending upon the user specified value for the flag (ICOND)). The first option (ICOND = 0) will swap out rows that produce final diagonal pivots less than EPS. It will also disregard elements to the left of the diagonal that are less than EPS. This has two advantages:

1. Very small contributions of one state variable to another at a certain integration step are ignored resulting in reduced solution time.
2. This removal of small elements from the row prevents them from harming the condition of the remainder of the row. With this option, too small a value for EPS will result in little or no row swapping and build up of round off error; and too large a value for EPS will result in the disregard of

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022

significant terms or the finding of no suitable row at a certain elimination step (as well as increased execution time for swapping and increased fill-in).

The second option (ICOND = 1) is the same as the first except that each row will be scaled so that:

$$.1 < \max_{1 \le j \le n} |g_{ij}| \le 1 \text{ (for each } i\text{)}$$

Thus, each row of $\bar{G}$ will be of nearly equal length in the sense of $\|g_i\|$. This will usually reduce round off over option one. The effect of changes in EPS will be the same as option one. The third option also includes scaling except that only very small elements (less than EPS/100), to the left of diagonal are ignored. Another approach is to reject rows based upon a comparison of the pivot value with other elements on the row (Curtis, 1971). This will produce a more accurate solution but at the expense of many logical operations. It may be economically preferable to switch to a double precision version than to use this approach.

As an example consider one of the previous problems: A random set of 250 equations, five elements per equation, and a bandwidth for generation of 30. When this was solved previously, a single precision version of *IMP* was used on a computer with a 60 bit word. EPS was 1·0 E-10. Round off error was less than 1·0 E-6. The problem was re-run using different values of EPS and ICOND with a single precision version of *IMP* on a computer with 32 bit words. The condition number of this matrix is very high.

Table 6 shows the effect of 'EPS' and ICOND on round off error. For many ill-conditioned problems, round off error may be reduced to a tolerable level without using a double precision version of *IMP*. In addition to matrix scaling, *IMP* provides an option to scale the state vector so that each internal value of a state variable will be closer to unity while external values remain user oriented.

Acknowledgements

The vector algebra approach described here was developed as part of the author's Ph.D. thesis. The author is most grateful for the support and guidance of his major advisor, Dr. L. F. Stutzman, and to the Control Data Corporation for provision of computer time.

References

BRANDON, D. (1973). Digital Simulation of Continuous Systems and the New IMP Program, *Simuletter* (ACM), April.

BRANDON, D. (1972). *IMP*—General Manual, A General Simulator for Multivariate Differential or Algebraic Systems, Chemical Engineering Department, University of Connecticut, Storrs, Connecticut.

BRANIN, F. H., HOGSETT, G. R., LUNDE, R. L., and KUGEL, L. E. (1971). ECAP II—An Electronic Circuit Analysis Program, *IEEE Spectrum*, June, p. 14.

BRENT, R. P. (1970). Error Analysis of Algorithms for Matrix Multiplication and Triangular Decomposition Using Winograd's Identity, *Numer. Math.* Vol. 16, pp. 145-156.

CHANG, A. (1969). Applications of Sparse Matrix Methods in Electric Power Systems Analysis, *Sparse Matrix Proceedings*, pp. 113-117.

CURTIS, A. R., and REID, J. K. (1971). The Solution of Large Sparse Sets of Unsymmetric Systems of Linear Equations, *J. Inst. Maths. Applics.*, Vol. 8, pp. 344-353.

GUSTAVSON, F. G., LINIGER, W., and WILLOUGHBY, R. (1970). Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations, *J.A.C.M.*, Vol. 17, No. 1, January, p. 87.

HACHTEL, G. D., BRAYTON, R. K., and GUSTAVSON, F. G. (1971). The Sparse Tableau Approach to Network Analysis and Design, *IEEE Trans. Cir. Theory*, Vol. CT-18, No. 1, January, p. 101.

REID, J. K. (1971). *Large Sparse Sets of Linear Equations*, Academic Press.

ROSE, D. J. and WILLOUGHBY, R. A. (1972). *Sparse Matrices and Their Application*, Plenum Press.

STEWARD, D. V. (1965). Partitioning and Tearing Systems of Equations, *J. Siam Numer. Anal.*, Vol. 2, No. 2, p. 345.

STRASSEN, J. (1969). Gaussian Elimination is Not Optimal, *Numer. Math.*, Vol. 13, pp. 354-356.

TEWARSON, R. P. (1966). On the Product Form of Inverses of Sparse Matrices, *Siam Review*, Vol. 8, No. 3, July, p. 336.

TINNEY, W. F. and WALKER, J. W. (1967). Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization, *Proc. of IEEE*, Vol. 55, No. 11, November, p. 1801.

YOUNG, D. M. (1971). *Iterative Solution of Large Linear Systems*, Academic Press.

WEINSTEIN, H. G. (1968). Iteration Procedure for Solving Systems of Elliptic Partial Differential Equations, *Proc. Sparse Matrix Symp.* IBM RA 1 (No. 11707).

Book review

Principles of Digital Computer Operation, by M. D. Freedman, 1972; 223 pages. (*John Wiley and Sons, Inc.*, £3·75)

It is stated in the author's preface that the book is written 'so that no previous knowledge of computers is required'. It is therefore no mean achievement that he has managed to begin by defining the meaning of 'digital' and progress, within two hundred pages, to brief discussions of such subjects as pipeline processors and fault-tolerant machines.

The first half of the book is concerned with the basic concepts of digital computers, namely, their fundamental block structure, language, arithmetic facilities and number representations. The various subsystems described in the first chapter on block structure are later covered in a more detailed way. A wide variety of input/output devices are described. The final quarter of the book deals with a mixed bag of topics, including chapters on Software, Applications, Interrupt and Time-Sharing.

The author's style is clear and concise and this, coupled with good diagrams and illustrative examples, contributes much to the book's undoubted value. The difficulty of writing a book which attempts, as this one does, both to introduce to the subject a reader with no background experience and also to serve as an overview of principles to a professional, is that it may fall between the two aims. One cannot help feeling that the plain number of ideas presented may be slightly overwhelming to a first-year undergraduate and yet leave a second-year student with a rather superficial view of any given topic, particularly those touched on in the last part of the book. One might also be forgiven for supposing that computers are calculators rather than data processors, such is the weight placed on arithmetic operations.

These are minor points, however, to set against the good features of a book which fills a rather noticeable gap in the existing literature. Its appearance is to be welcomed.

N. M. BROOKE (Lancaster)

Downloaded from https://academic.oup.com/comjnl/article/17/2/165/525505 by U.S. Department of Justice user on 17 August 2022