JOURNAL OF RESEARCH of the National Bureau of Standards
Vol. 84, No. 1, January–February 1979

Enhancing Fortran to aid Manipulation of Large Structured Matrices*

Harvey J. Greenberg

Office of Analytic Methods, Department of Energy, Washington, DC 20461

James E. Kalan

Department of Management, University of Florida, Gainsville, Florida 32611

September 21, 1978

This paper presents, for wider discussion by the technical community, suggested means for enhancing (ANS) FORTRAN in order to accommodate the needs of operations research analysts in programming tasks involving large, structured or sparse matrices. Such needs frequently arise in connection with large-scale optimization problems. Most of the text deals with fundamental concepts and descriptions of syntax, but related data structures are also treated. Proposed new capabilities include exploitation of repeated values among matrix entries, space-saving "quasi-dynamic storage allocation", and easy set-up for construction of large matrices from smaller ones (with the actual construction deferable until and if the need arises).

Key Words: Data pooling; data structures; mathematical programming; matrices; name generation; operations research; programming languages; sparse matrices.

AMS Subject Classification: Primary 6800; Secondary 90C99; 68A30

1. Introduction

The capabilities of presently available general-purpose mathematical software are often strained by the giant data bases and numerous quantitative relationships characterizing modern large-scale mathematical models. These strains are reflected in inefficient computational procedures, and in awkward programming constraints whose satisfaction is time-consuming and likely to generate errors. There is an increasing demand for software specifically oriented toward these needs of operations-research modelling applications, and a first priority is an improved capability for handling very large sparse or special-structure matrices.

1.1 Purpose

This paper presents the results of a short study on means for enhancing Standard FORTRAN, as specified by the American Standards Association (renamed the American National Standards Institute: ANSI), in order to accommodate the needs of operations research analysts in some of their programming tasks which deal with large, structured and mostly sparse matrices. These needs arise in the widely used techniques of mathematical programming and simulation.

The present study did not extend to the point of trial implementation and experimental testing, and therefore must be understood as representing the investigators' opinions, based upon prior experience and "armchair analysis". In most cases alternatives were considered before deciding upon the syntax presented below. Some of these alternatives are noted because subsequent study may involve their reconsideration. It would be premature to aim here at the precision and completeness appropriate in a full-fledged proposal for a candidate standard. Instead, our goal is to lay out ideas at a level of detail adequate to enable their discussion by the technical community.

We are aware that some of the topics touched upon below have been discussed extensively, for instance, by the ANSI FORTRAN committee on previous occasions. Some array computation capabilities have been

* Based on a report (Computer Science Department, VPI, Blacksburg, VA, September 30, 1975) prepared for the National Bureau of Standards, under Purchase Order #600409 to the Applied Mathematics Division during the Summer of 1975.

implemented in PL/1. Our focus here, however, is on large sparse and structured matrices—a class of data structures which to our knowledge have not been discussed previously from the point of view of language requirements. Last but not least, we aim to identify avenues for much needed research and development, including experimentation (see section 9.1), following this first step.

1.2 American Standard FORTRAN

It is important to emphasize that this study is concerned with enhancing ANS FORTRAN as a standard and not with versions which have been produced commercially or otherwise for different processors. Of course, the recommendations in this report reflect processor considerations, but the primary intent is to comply with the criteria of the American National Standards Institute of which NBS is a member. It is therefore befitting that we briefly review elements of ANS FORTRAN with regard to their effect on our study.

Since the inception of FORTRAN (1956), many versions appeared with different processors. In an effort to provide the first standardized language, the Computer and Business Equipment Manufacturers Association (CBEMA), secretariat for ANSI, formed a committee to establish a standard FORTRAN. Their specifications were published in X3.9-1966 (see [2][1], and we call this ANS FORTRAN. Several errors and misconceptions were rectified in 1969 (see [1]). However, the ANSI FORTRAN Committee (X3J3) recently prepared a detailed proposal for a new standard. That draft is called FORTREV[2] in this report.

Perhaps the most significant enhancement in FORTREV is the allowance of character data types plus associated string processing capability. A second technical change, which appears in FORTREV, is to a PL/1 type of syntax for specifying index ranges. To accomplish this the colon (:) has been added to the character set, and we shall use it for similar index referencing.

The ANSI FORTRAN committee has so far decided against the use of dynamic storage allocation (even to the limited extent found in ALGOL). Their decision stems from a posture on maintaining maximal independence of the processor (viz., the operating system). Our recommendations include a 'quasi-dynamic storage allocation' capability which may be achieved while maintaining the ANSI posture of independence.

The ANSI FORTRAN committee has also adopted the criterion that compilation should be achievable in one pass. The syntax presented in this report complies with this requirement as well.

In general, our enhancement to represent and manipulate large, structured matrices conforms with the former ASA's criteria as listed in X3.9-1966. This includes "interchangeability of FORTRAN programs between processors" and "compatibility with existing practice."

It is important to note that the spirit of ASA's criteria for enhancement included maintenance of language style. This is perhaps a subjective and rather vague criterion, but it has influenced the syntax proposed in this report. At least one less vague derivative from the notion of language style is that of conformance. This is described in FORTREV, at least with respect to upward compatibility (i.e., a correct program must remain so, and an incorrect program must produce the same diagnostics). One exception is noted dealing with optional forms of output, and our enhancement contains a similar exception (see sec. 7).

1.3 Notation

The notation used in this report in describing the form of FORTRAN statements or constructs employs the metalanguage described in FORTREV. This generally agrees with the metalanguages found in most reference manuals. In particular, the following conventions are used:

(1) Special characters from the FORTRAN character set, uppercase letters, and uppercase words are to be written as shown, except where otherwise noted.
(2) Underlined lowercase letters and lowercase words indicate general entities for which specific entities must be substituted in actual statements. Once a given lowercase letter or word is used in a syntactic specification to represent an entity, all subsequent occurrences of that letter or word represent the same entity until that letter or word is used in a subsequent syntactic specification to represent a different entity.

[1] Figures in brackets indicate literature references at the end of this paper.
[2] Now described in ref. 14.

(3) Brackets ([ ]) are used to indicate one or more optional items.

(4) An ellipsis (···) indicates that the preceding optional items may be repeated one or more times in succession.

(5) Blanks are used to improve readability, but unless otherwise noted have no significance.

The *type* of elements refers to the type of variable acceptable in ANSI FORTRAN. Currently, there are two types: (1) arithmetic and (2) logical. (The arithmetic type may be further stratified into integer, real, double precision and complex.) FORTREV admits a third type: (3) character. The construction of the name of a matrix identifies its type of elements in the same manner currently applicable to arrays. Since we require every element of a matrix to be of the same type, we can refer to the 'type of matrix' to mean its type of elements. We use the terms 'row' and 'column' of a matrix in the usual sense, but the manner of accessing a row, column or element is subject to further specification. Before we do so let us define a *trivial element* to be as follows:

Data Type	Meaning of Trivial
arithmetic	zero
logical	false
character	blank

Then, we define a matrix to be trivial if all of its elements are trivial.

Next define a *submatrix* of a matrix to consist of subsets of, respectively, its rows and columns with the associated elements. Thus, each row (or column) of a matrix constitutes an instance of a submatrix. This implies a definition of trivial rows and trivial columns.

A two dimensional array is considered a matrix with an implied manner of access which retains current usage. In particular, trivial elements of arrays are explicitly represented, and we have random access to elements by index numbers.

In the next section we shall describe a syntax for declaring other manners of access, in which the associated (packed) matrices do not explicitly represent trivial elements. Basically we can first specify whether we want to access by row, column or element; then we specify whether the mode is to be sequential or random. The essential difference is in subsequent speed of computation rather than feasibility of references. An operations research programmer can acquire a deeper appreciaton for this distinction upon examining the structures for implementation (see section 8).

In summary, the *manner of access* is by one of the following:

(1) row
(2) column
(3) element

and the *mode* is either sequential or random. We emphasize that the reference to 'sequential' versus 'random' is for efficiency considerations, and affects neither a program's correctness nor its meaning.

We interject the thought that (1) and (2) may be merged into one specification of a dimension when extending the enhancement to higher dimensional arrays. However, there are some difficulties with regard to language style and ease-of-use.

An example illustrates the metalanguage. Given a description of the form of a statement as:

$$\text{CALL } \underline{\text{sub}} \; [(\underline{\text{a}} \; [\,,\; \underline{\text{a}}] \cdots)]$$

the following forms are allowed:

CALL sub
CALL sub ($\underline{\text{a}}$)
CALL sub ($\underline{\text{a}}$, $\underline{\text{a}}$)
CALL sub ($\underline{\text{a}}$, $\underline{\text{a}}$, $\underline{\text{a}}$)
etc.

2. Extended Terms and Concepts

This section describes the extended terms and concepts associated with the proposed enhancement. Consideraton is given to differences between the current standard and FORTREV.

2.1 Arrays and Matrices

Current (1966) ANSI FORTRAN defines an array as "an ordered set of data of one, two, or three dimensions." FORTREV extends this range to seven dimensions.

Within the context of this study the meaning of "array" is maintained with the following characteristics noted:

(1) storage in linear list form,
(2) references by index numbers,
(3) no dynamic storage allocation.

Our intention is to provide a new type of array, which we may call a *packed array*, differing in characteristics (1) and (2); further, certain syntactical elements and use of intrinsic functions (cf. sec. 5.2) will provide a "quasi-dynamic storage allocation" (which is processor independent with regard to feasibility).

In this study we shall confine ourselves to packed arrays of two dimensions which we call "matrices." As we describe some of the difficulties encountered in achieving the stated purpose, the existence of problems to be resolved for higher dimensions will become noticeable. (The handling of higher dimensional arrays is cited in secton 9 as one of the avenues for further study.)

Formally, a *matrix* is a named collection of data whose elements are referenced by two *index values*. The exact nature of the index values, method of storage, types of elements and manners of reference are subjects described in subsequent sections.

We may note here that the basic attributes of a matrix are:

(1) type of elements
(2) manner of access
(3) effective size.

which we shall mention in secton 3 when introducing the syntax for declaration.

The *effective size* of a matrix is a dynamic entity represented by a couple, say ($\underline{m}$, $\underline{n}$), where $\underline{m}$ and $\underline{n}$ are non-negative integers specifying the number of non-trivial rows and non-trivial columns, respectively. We may thus think of every matrix as infinite dimensional (to have inherent mathematical conformality of dimensions when performing operations) with the effective size determining storage requirements and speed of computation. Note this would mean every matrix is a trivial matrix (i.e. a matrix whose elements are all trivial) when first declared.

2.2 Names

It is often desirable to permit index reference by name rather than by number. This is especially true in operations research applications, notably in mathematical programming or simulation models. For example, a column may correspond to an activity of transporting goods from a supplier to a warehouse, perhaps in a particular time period. In this case it would be desirable to reference the column by the form:

supplier · warehouse · time period.

Specifically, the supplier may be identified by a city name, such as DALLAS, the warehouse may be identified by a region,say SW, and the time period may be a month, say OCT. Thus, in this case a column may be referenced by the *3-field* name

DALLAS · SW · OCT

We shall assume FORTREV becomes accepted, at least the allowance of character variables and expressions. In this case string processing will become a part of ANSI FORTRAN. The only special concern for this enhancement is the reference to a *name* of a column or row. This is a set of character variables associated with every matrix. Specifically, every row and every column has an associated name. Initially, the names are blank strings; sections 4.6 and 7.1 describe how names can be changed.

We point out that at least one alternative was considered for this report but rejected on stylistic grounds. That is, it may be useful to the operations research programmer to permit names consisting of n-tuples (where he specifies the non-negative integer n). This would mean references to a name would appear as n-dimensional arrays, presenting difficulties, for example, with assignment statements. Also, since FORTREV allows any subscript range, it may be burdensome for a one-pass compiler to process character expressions involving names. The decision to represent names as character strings (of one dimension) stems from our desired compliance with ANSI criteria, which include compatibility with current processors as well as language style. At present, operations research programmers use one character string to name an entity (such as an activity) and assume the burden of establishing attribute fields for report writing.

3. Matrix Declaration and Reference

This section describes and illustrates the ways in which matrices are declared and referenced. It includes the syntax for the declarations.

3.1 Matrix Specification Statement

FORTREV defines nine kinds of specification statements, each of which is nonexecutable. We propose a tenth specification statement to specify a (packed) matrix. The syntax is as follows:

[d] MATRIX a [(o [, o]· · ·)] [, ···]

where

d is a data type declaration
a is the name of the matrix
o is an option according to the following:

Option	Meaning
NAME(n)	specifies the names of rows and columns have lengths not exceeding n characters (default of n is 6)
SEQUENTIAL $\left[\frac{\text{BY ROW}}{\text{BY COLUMN}}\right]$	specifies sequential mode of access by row (the default) or by column
RANDOM ACCESS	specifies random access to rows, columns or elements
MAXROW = n	specifies upper limit (n) on row index number
MAXCOL = n	specifies upper limit (n) on column index number

Before we further describe and illustrate these options, consider the simple declaration:

MATRIX A

which declares a matrix whose name is 'A'. Its data type is arithmetic by standard default (in particular, A is REAL because its name begins with 'A'). If this were preceded by the specification:

LOGICAL A

then the data type of A would be logical. The mode of access is random by element (the default). The row and column names would be limited to six characters apiece, and no limit would be placed on the magnitudes of row or column index numbers.

Now consider the options listed. First, the data type may be specified in the matrix specification statement as in declaring arrays. That is, we may write:

```
LOGICAL A
DIMENSION A(100)
```

to declare an array of one dimension having data of type logical, or we may write

```
LOGICAL A(100)
```

to make the same declaration. In our case we may write

```
LOGICAL A
MATRIX A
```

or

```
LOGICAL MATRIX A
```

to declare a matrix whose elements are of the logical data type.

The remaining options may be declared in any order. The first of these (as tabulated above) is used to specify the maximum number of characters (called the *length*) of the row and column names. Thus, we may write

```
MATRIX A(NAME(10))
```

to specify a maximum of 10 characters in the names.

The next option specifies a sequential mode of desired accessibility. This will not limit feasibility of random access, such as referencing an element by a random choice of indices in an expression, but it will affect the consumption of time and space (see section 8). Special input and output statements (see section 7) are affected by use of this option.

If the option specifying a sequential mode is not used, then a random access by element is assumed as default. However, if the option specifying sequential mode is used, then it may be desired to specify random access with respect to row or column identification. This is the purpose of option 3: RANDOM ACCESS. For example, if the operations research programmer uses the matrix by proceeding sequentially from one column to the next, then he should specify SEQUENTIAL BY COLUMN and not specify RANDOM ACCESS. This would be the case in the "ordinary" simplex algorithm. On the other hand, if the program uses the matrix only by accessing a column at a time, but with the selection of the next column dependent on the data, then both options, SEQUENTIAL BY COLUMN and RANDOM ACCESS, should be specified. This is exemplified by the simplex algorithm modified by a data-driven pricing tactic.

The next two options specify automatic testing of effective row and/or column dimensions. If the effective size would violate this limit during execution, then an 'index-out-of-range' error condition would be set.

EXAMPLE 1: MATRIX A(NAME(10)), K

Assuming the program uses no other specification statements, this specifies two matrices. The first has been named 'A'; its data type is arithmetic (REAL), and its mode of access is random by element. The second matrix has been named 'K'; its data type is arithmetic (INTEGER), and its mode of access is random by element. A allows up to 10 characters in a name, but K only allows 6 (the default).

EXAMPLE 2:

```
LOGICAL Y
COMPLEX C
MATRIX Y (SEQUENTIAL BY ROW), C(MAXROW = 100)
```

In this example two matrices are specified. The first has been named 'Y'; its data type is logical, and its mode of access is sequential by row. The second matrix has been named 'C'; its data type is arithmetic (COMPLEX), and its mode of access is random by element; it has an effective row size limit of 100.

3.2 Index and Matrix References

A matrix may be referenced in its entirety by using its name, and this constitutes the simplest matrix reference. To permit references to submatrices, notably to its rows, columns or elements, an *index reference* is used which may be *explicit* or implicit.

An *explicit index reference* is of the following form:

(a[:b], c[:d])

where

a and c are integer or character expressions or '*'
b is the same type of expression as a or is '*'
d is the same type of expression as c or is '*'

The colon (:) is used for *ranging*, and the asterisk (*) is used to denote extremes.

In the case of a name range, comparisons are based on collating sequence (as in FORTREV). For example, if a set of names is {ARK, ALA, LO, TS} then the range 'ALA:TS' would reference the entire set, and the range 'LO:TS' would reference the subset: {LO, TS}. This same subset would be referenced by 'L:TZ' because it is not necessary that a and b be in the set of names.

EXAMPLE 1: A(2*I + 1, J)

This references an element of A by evaluating the row number 2*I + 1 and the column number J. (Both are integer expressions.)

EXAMPLE 2: A(1:K, J)

This references a submatrix by specifying a column number (evaluation of J) and a range of row numbers (1 to the value of K).

EXAMPLE 3: A(3HABC, 3)

This references the element identified by the row whose name is 'ABC' and column number 3. (The same reference can be made by A('ABC', 3).)

EXAMPLE 4: A('COST', 'DALNEOCT':'DALSWOCT')

This references the submatrix consisting of the one row named 'COST' and the columns whose names fall in the range 'DALNEOCT' to 'DALSWOCT'. For example, if characters 4 and 5 must be one of the strings, 'NE', 'NW', 'SE' or 'SW', then the four columns referenced are 'DALNEOCT', 'DALNWOCT', 'DALSEOCT' and 'DALSWOCT'.

EXAMPLE 5: A(4:*, *:'DALSWOCT')

This references the submatrix consisting of rows 4 to the largest index value (dynamic) and columns whose names are less than or equal to (in collating sequence) 'DALSWOCT'.

An *implicit index reference* has at least one of the row/column index references (of form a[:b]) given by an asterisk (*) or a period (.) with the following meaning:

*	specifies all rows/columns
.	specifies row/column name

Here are some examples:

A(5, *)	references row 5 (i.e. '*' specifies all columns)
A(*, J + 1)	references column whose index number is the evaluation of 'J + 1'
A(1:10, *)	references the first 10 rows of A
A(*, 'DALNE':'DALSW')	references all rows and those columns whose names lie in the range 'DALNE' to 'DALSW'
A(. , 5)	references name of column 5
A(5, .)	references name of row 5
A(. , 2:J + 1)	references names of columns 2 through the value of 'J + 1'

If an index name cannot be found during execution (i.e. no name match), an 'index-out-of-range' error condition is set. It should be noted that names are treated as character variables; this constitutes a departure from CBEMA's style since the data type of the entire matrix need not be of character data type.

We point out that it may be desirable to reference a collection of submatrices specified by a *list*. This is exemplified by referencing a basis from a matrix with a list of column numbers (or names). Technically, this could be achieved, for example by the form:

$$A(\underline{i}, \underline{j})$$

where $\underline{i}$ and $\underline{j}$ are lists. However, such constructs are not part of ANSI FORTRAN, and arrays would be needed. This would introduce the need for further identifying which elements of the array are to be used. (To always use the entire array raises other problems.) This would cause concern over maintaining the language style. Of course, a subset of columns can be extracted with the use of a DO loop, one column-at-a-time.

4. Matrix Expressions and Matrix Assignment Statements

This section presents an extension of the expression types in FORTREV to permit direct manipulation of matrices (with two-dimensional arrays treated as matrices of special access structure). The types of matrix expressions are arithmetic, character, logical, relational and concatenation. The style of such expressions and associated matrix assignment statements complies with the syntactic style of "scalar expressions" (i.e. expressions in standard FORTRAN). Reasons for certain restrictions are presented; generally these are used to resolve certain ambiguities that arise.

In each case the simplest matrix expression is a matrix reference. More complicated matrix expressions may be formed by use of *operators* and *operands*. These depend upon the data type (arithmetic, logical or character), and all data types (matrices or scalars) must be the same in a matrix expression or a matrix assignment statement. This agrees with standard FORTRAN usage of expressions and assignment statements.

4.1 Arithmetic Matrix Expressions

An *arithmetic matrix expression* is used to express a numeric computation involving one or more (sub) matrices and possibly some scalars. Evaluation of an arithmetic matrix expression must produce an arithmetic matrix.

To form a matrix expression other than a simple matrix reference, operators are used. These may be unary or binary, and the operands may each be scalar or matrix. In ANSI FORTRAN the five arithmetic operators are as follows:

Operator	Representing
**	Exponentiation
/	Division
*	Multiplication
−	Subtraction
+	Addition

The subtraction and addition operators also have unary specifications (e.g. −X is the same as 0−X).

We shall use the same set of operator symbols, but the meaning will depend upon the operands used. Let 'A' and 'B' denote matrix references, and let 'X' denote a scalar reference in the following:

Expression	Meaning
A**X	Exponentiate every element of A by X (convention: 0**0 = 0)
A**B	UNDEFINED
X**A	UNDEFINED
A/X	Divide every element of A by X
A/B	UNDEFINED
X/A	UNDEFINED
A*X	Multiply every element of A by X
A*B	Matrix multiplication
X*A	Multiply every element of A by X
A − X	Subtract X from every element of A
X − A	Add X to the negative of every element of A
A − B	Matrix subtraction
A + X	Add X to every element of A
A + B	Matrix addition
X + A	Add X to every element of A

Further, the subtraction operator is used as a unary operator, so −A means to negate every element of A. Note that since zeroes are not represented, there is no distinction between +0 and −0; this agrees with the specification in FORTREV.

When both operands are matrices (A*B, A + B or A − B), the issue of *conformality* arises. Mathematically, there is a restriction on the relative dimensions. Since a major aspect of this enhancement is to permit effective sizes which vary during execution, it is not possible to check coformality at compilation without degrading performance, particularly in a one-pass compiler. By treating the matrices as infinite dimensional (with trivial rows and columns not explicitly represented) every operation is conformal. If the operations research programmer wants to perform his own conformality check, he may use intrinsic functions (see section 5) to retrieve the effective size and perform an IF test.

We also considered a condition switch of the form

$$\text{CONFORMALITY} = \underline{a}$$

where a specifies the type of conformality test desired in all subsequent matrix expressions. However, this requires further study to consider types of conformality checks and certain storage features (such as blocking).

Notice that A (or B) and X in the above table of binary expressions may be replaced by arithmetic matrix expressions and arithmetic scalar expressions, respectively. Further, a matrix expression may reduce to a scalar value, such as the following inner product between row (I) of A and column (J) of B:

A(I, *) *B(*, J).

The result is a 1 × 1 matrix (different from a scalar) but may be assigned to a scalar variable (cf. sec. 4.5).

4.2 Logical Matrix Expressions

The *logical matrix expression* is formed with logical operators and logical operands (matrix or scalar). Evaluation of a logical matrix expression produces a matrix of type logical.

The logical operators are:

Operator	Representing	Priority
.NOT.	Negation	Highest
.AND.	Conjunction	Intermediate
.OR.	Disjunction	Lowest

The operation is element-by-element. Thus, A.AND.B produces a matrix with element (i, j) equal to the logical value of A(i, j).AND.B(i, j). Similarly, .NOT.A negates A. Use of a logical matrix and a logical scalar is permitted with the following meaning:

A.OR.X (or X.OR.A) = A.
A.AND.X (or X.AND.A) = A if X = .TRUE. and null matrix if X = .FALSE.

4.3 Character Matrix Expressions

The character matrix expression is formed with character operators and character operands (matrix or scalar). Evaluation of a character matrix expression produces a matrix of type character.

The only operator is the concatenation, denoted by two slant characters (//). This is to be performed element-by-element. Thus, A//B produces a matrix whose (i, j) element is A(i, j)//B(i, j). If A//X (or X//A) is used, then the resulting matrix has (i, j) element equal to A(i, j)//X (or X//A(i, j)), respectively.

4.4 Relational Matrix Expressions

A *relational matrix expression* is used to compare two arithmetic expressions or two character expressions. In each case at least one must be a matrix expression.

As in FORTREV, relational matrix expressions may appear only within logical expressions. Evaluation of a relational matrix expression produces a (scalar) result of type logical.

The relational operators are:

Operator	Representing
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Operations are on the elements. Thus, the logical value of A.LT.B. is .TRUE. if A(i, j).LT.B(i, j) for all (i, j) such that either A($\underline{i}$, $\underline{j}$) or B($\underline{i}$, $\underline{j}$) is nontrivial and .FALSE. otherwise.

One of the operands may be a scalar. For example, the logical value of A.LT.O is .TRUE. if every nontrivial element of A is less than zero; otherwise it is .FALSE.

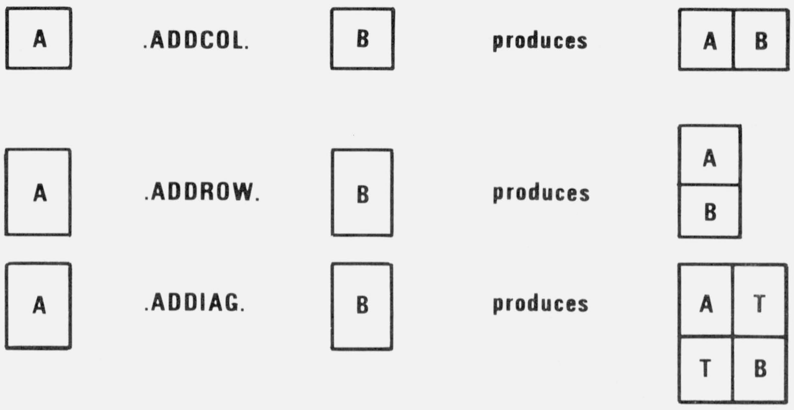
4.5 Concatenation Matrix Expressions

A *concatenation expression* is used to concatenate two or more matrices of the same type. This is the basic method by which matrices may be built from smaller matrices. There are three forms of concatenation: row, column and diagonal.

The concatenation operators are:

Operator	Representing
.ADDCOL.	Add new column(s)
.ADDROW.	Add new row(s)
.ADDIAG.	Add new (block) diagonal

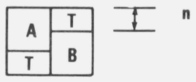
Both operands must be matrices. A schematic is as follows:



where T is a trivial matrix. The rows and/or columns of B are renumbered as appropriate in forming the new matrix.

Notice that conformality always prevails with missing elements interpreted as trivial (0 for arithmetic, .FALSE. for logical and blank for character). This stems from viewing the matrix as quadrant IV with index numbers for rows and columns running from 1 to infinity. We now describe how negative indices may be similarly interpreted, resulting in a more flexible construct.

Suppose we wish to form a staircase of the form



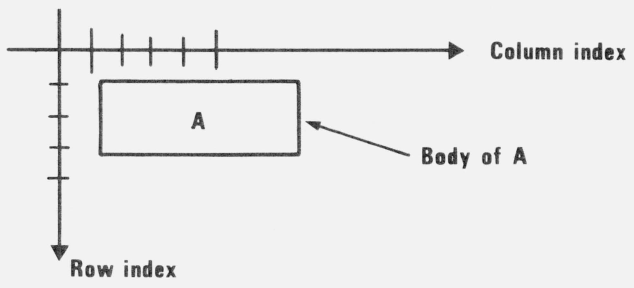
where T is a trivial matrix, and n is a positive integer. Let us further suppose we know the contents of N is n. Then, we could perform

A.ADDCOL.B(−N:*, *)

The matrix reference, B(−N:*, *), says we want the submatrix of B consisting of rows numbered from −N to the highest row index number and all columns. By interpreting negative row and column values as containing trivial elements only, this forces renumbering to produce the desired result.

This places some burden on the processor because a negative (numerical) index value ordinarily results in an "index-out-of-range" error condition (although FORTREV allows negative index numbers in arrays). For this use in the concatenation operators, negative index values do hold meaning.

We view the "body" of the matrix as having finite effective size, but the matrix implicity has infinite size from minus infinity to plus infinity in both dimensions. A schematic of this view is as follows:



Outside the body all elements are trivial.

Thus, the elements in quadrants I, II and III may be referenced in the manner just described even though their values are trivial.

4.6 Matrix Assignment Statement

The matrix assignment statement is of the form

$$\underline{a} = \underline{e}$$

where $\underline{a}$ is a scalar variable or a matrix reference and $\underline{e}$ is a matrix expression. The data types of $\underline{a}$ and $\underline{e}$ must conform as in FORTREV assignment statements.

The matrix assignment statement may be used to convert from one structure to another. For example, if A is a matrix and X is an array (of two dimensions), then

$$A = X$$

packs X (eliminates trivial elements). The reverse is possible with

$$X = A$$

Similarly, if X has been declared as a column by

DIMENSION X(100, 1)

then this could be augmented to A by

A = A.ADDCOL.X

If $\underline{a}$ is a scalar variable, then the evaluation of $\underline{e}$ must produce a matrix whose effective size does not exceed (1, 1). For example, we may compute the inner product between a row array declared by

DIMENSION PI(1, 100)

and column (J) of matrix A by

DJ = PI(1, *)*A(*, J)

In a less obvious situation we may have

DJ = PI*X

At execution time, after PI*X is evaluated, it is determined if this assignment is correct. If the evaluation of e produces a matrix whose effective size exceeds (1, 1), then an error condition is set by the processor.

The names of the rows and columns of a (when a is a matrix reference) are not determined by e. This reflects the fact that the operations are on bodies of matrices, and no naming is inherited. Of course, a simple assignment statement of the form:

A(.,*) = B(.,*)

assigns the column names of B to the column names of A.

It is appropriate to point out why a matrix expression was defined to require at least one matrix reference. Consider the assignment

A = B*C

where A is a matrix. If B and C are both scalars, we could interpret the result as a 1 × 1 matrix. However, in keeping with ANSI FORTRAN style this would *overwrite* A. We believe this poses a danger in that the programmer probably did not intend to perform such an overwrite.

One way the programmer could achieve the intended result is to reinitialize A as trivial (see, for example, sec. 6.2). Then he could name the first row and column, thereby making its effective size (1, 1). Finally, he could write:

A = A + B*C

5. Functions and Subroutines

FORTREV identifies four categories of procedures: statement functions, intrinsic functions, external functions and subroutines. Our proposed enhancement extends the references made in FORTREV to 'variable' and 'expression' to permit 'matrix' and 'matrix expression' plus some added features.

5.1 Statement Functions

A *statement function* is defined internally to the (sub) program unit in which it is referenced, and it is of the following form:

f(a[, a]···) = e

where f is the function name, a is an argument and e is an expression. In the enhancement f, a or e may be matrices with the stipulation that if f is a matrix, then e must be a matrix expression (i.e. must contain a matrix reference).

We also introduce another form of function definition similar to the statement function. Its form is as follows:

NOTE: f[(a[, a]···)] = e

where f is the function name, a is an argument and e is an expression. The form following 'NOTE:' is the same as a statement function, except it is not necessary to have any arguments. However, the meaning differs as follows.

In the NOTE statement (just defined) matrix references and variables appearing in e need not appear as arguments. The processor merely uses e wherever f[(a[, a]···)] is referenced in the source program.

For example, to use a statement function that expresses a matrix product we would write

PROD(A, B) = A*B

where the arguments are necessary. However, in the NOTE statement we may write

NOTE: PROD = A*B

Then, wherever ‘PROD’ appears, ‘A*B’ will be used.

5.2 Intrinsic Functions

Certain functions of repeated use by many programmers have been specified by FORTREV to be intrinsic. There are two ways this idea is exploited in our proposed enhancement: (1) extension of scalar functions to allow matrix arguments, and (2) specification of new intrinsic functions for matrices.

5.2.1 Extension of Scalar Functions

If $\underline{f}$ is a function of a scalar, then this extends to $\underline{F}(\underline{A})$, where A is a matrix reference, and $\underline{F}$ now becomes a matrix reference. This is to be interpreted by defining the ($\underline{i}$, $\underline{j}$) element of $\underline{F}$(A) by $\underline{f}$(A($\underline{i}$, $\underline{j}$)). For example, ABS(A) defines the matrix (of arithmetic type) whose elements are the absolute values of the associated elements of A.

5.2.2 Matrix Intrinsic Functions

In the table below a list of matrix intrinsic functions is given with associated meanings. ‘A’ and ‘B’ are used to denote matrix references, ‘X’ denotes an arithmetic scalar, and ‘N’ deontes an integer scalar.

Table of Intrinsic Matrix Functions

name	input	output	meaning
TRANS	A	B	B = transpose of A
DIAG	A	B	B = diagonal of A
TRACE	A	X	X = trace of A for arithmetic type
IMAGE	A	B	B = boolean image of A (non-trivial elements replaced by 1)
MIN	A	X	X = maximum value for arithmetic type (inside body of A)
MAX	A	X	X = minimum value for arithmetic type (inside body of A)
NCOL	A	N	N = number of non-trivial columns of A
NROW	A	N	N = number of non-trivial rows of A
MAXCOL	A	N	N = maximum column index number
MAXROW	A	N	N = maximum row index number
NWORDS	A	N	N = number of words of storage used to represent A
INDEX	A	N	N = latest key index number of A referenced for sequential mode

Some illustrative programs using these intrinsic matrix functions are as follows.

EXAMPLE 1: Pricing routine for simplex method obtaining first candidate.

```
           SUBROUTINE PRICE (M, PI, A, *)                    1
           DIMENSION PI(1, M)                                2
           MATRIX A (SEQUENTIAL BY COLUMN)                   3
           COMMON/TOL/DJTOL                                  4
           N = MAXCOL (A)                                    5
           DØ 10 J = 1, N                                    6
           DJ = PI(1, *)*A(*, J)                             7
           IF (DJ.GT.DJTOL)RETURN                            8
    10     CONTINUE                                          9
           RETURN 1                                         10
           END                                              11
```

EXPLANATION:

Line 1: Subroutine identification passing the number of rows (M), price vector (PI), matrix (A) and statement number (*) for transfer of control if there are no candidates.
Line 2: Standard specification statement.
Line 3: Matrix specification statement.
Line 4: Passes a tolerance through named COMMON.
Line 5: Use of intrinsic function MAXCOL to obtain largest column index referenced (i.e. number of activities in problem).
Line 6: DO loop over column numbers.
Line 7: Inner product between the price vector and column (J) which is assigned to 'DJ' (denoting reduced cost).
Line 8: Test for candidacy. If the value of DJ exceeds the tolerance (DJTOL), a RETURN to calling program unit occurs.
Line 9: End of DO loop.
Line 10: Alternate RETURN (if no candidates are found).

Now suppose the calling program is re-entered after the CALL statement, and a candidate is found (normal RETURN). To identify the index number the intrinsic matrix function, INDEX, is used as

J = INDEX (A)

This retrieves the latest column number referenced. To expand column (J) in preparation for a row selection routine, the programmer merely writes

ALPHA = A(*, J)

where ALPHA has appeared in a specification statement such as

DIMENSION ALPHA (M, 1)

with M equal to the row dimension.

EXAMPLE 2: The scalar product of two matrices is:

$$\sum_{(i,j)} a_{ij}b_{ij}$$

It is not difficult to show this is equal to

$$trace(a^*(b^T)).$$

Thus, if C is a matrix of costs and X is a matrix of activity levels, the total cost,

$$\sum_{(i,j)} C_{ij}X_{ij},$$

may be computed by specifying:

MATRIX X, C

COST = TRACE(C*TRANS(X))

where COST is a scalar.
Additional examples will be given in section 7 to illustrate input and output statements.

5.3 External Functions

An *external function* is defined externally to the (sub) program unit that references it, and its form is:

[t] FUNCTION f(a[, a]· · ·)

where:

t is a data type,
f is the name of the function,
a is a dummy argument.

In the enhanced FORTRAN we allow t to be MATRIX (in addition to INTEGER, REAL, etc.).
The basic external functions described in X3.9–1966 are not part of FORTREV, which placed them into the category of intrinsic functions.

5.4 Subroutines

A subroutine is defined externally to the (sub) program unit that references it and is of the form:

SUBROUTINE s[(a[, a]···)]

where

s is the name of the subroutine
a is a dummy argument.

We require no ammendments of FORTREV with respect to an (external) subroutine, except that matrix constructs may be passed as dummy arguments.

6. Storage Control

Since the effective sizes of matrices are dynamic, it is possible to consume all available space, yet attempt to execute a statement which requires additional storage. (This includes scratch space needed for intermediate matrix generation during the evalution of a matrix expression.) When there is not sufficient storage to execute such statements, a state of OVERFLOW is said to exist. The operations research programmer may be able to take certain remedial action such as permanently or temporarily removing a matrix to make storage available. This section describes the "OVERFLOW statement" and statements to achieve a degree of control over storage. In this sense the enhancement admits a quasidynamic storage allocation capability, not dependent on whether the processor uses dynamic storage allocation.

6.1 OVERFLOW and RESUME Statements

The form of an overflow statement is:

```
OVERFLOW = s
```

where s is a statement label of an executable statement that appears in the same program unit as the overflow statement. This defines the transfer of control when an overflow condition is reached.

It should be noted that this form is analogous to the "error specifier" in FORTREV, except the OVERFLOW statement may be used at any time as an executable statement.

After the control transfers, which results from an overflow condition, the (sub) program unit may take certain remedial action (to be described shortly). If the programmer has included an appropriately-placed statement,

```
RESUME,
```

this will then transfer control back to the place overflow occurred.

6.2 ERASE Statement

The ERASE statement is used to delete a matrix or a submatrix. The name of the matrix remains valid and may be reconstructed or still used in any matrix expression (treated as a trivial matrix). Its form is:

```
ERASE a [, a]···
```

where a is a matrix reference.

For example, a mathematical programming routine may declare two matrices as:

```
MATRIX A(NAME(15)), B(SEQUENTIAL BY COLUMN)
```

The first matrix, named 'A', is to be used for the matrix generation phase, facilitating declaration of nonzeroes randomly (by name up to 15 characters). The second matrix, named 'B', represents the same mathematical model, but it is stored by column to facilitate execution of the primal simplex method.

Once A is constructed, we may convert by the assignment statement:

```
B = A
```

At this point both A and B are nontrivial, and it may be necessary to make room for another matrix in order to represent the basis (as another matrix, possibly in a factored form). We may thus reach an overflow condition.

In anticipation of that possibility, the program unit may contain the OVERFLOW statement:

```
OVERFLOW = 100
```

and there may appear the statements:

```
100 ERASE A
    OVERFLOW = 200
    RESUME
200 CALL EXIT
```

Another use of the ERASE statement is to remove a submatrix. For example, suppose we have

```
MATRIX A, B, C
B = A(1:10, 1:10).ADDCOL.A(1:10, 100:200)
```

If the statement,

ERASE B(1:10, 1:10),

is executed, then B equals A(1:10, 100:200), except that the columns would be renumbered.

Notice that the ERASE statement complements the assignment statement for matrix reduction. For example, if we write

ERASE A(11:*, 11:*)

then the (10, 10) principal minor of A is retained, and the remainder of A becomes trivial. On the other hand, if we write

A = A(11:*, 11:*)

then the (10, 10) principal minor is removed from A, and the remaining nontrivial portion is renumbered.

6.3 DEACTIVATE and ACTIVATE Statements

The DEACTIVATE statement removes a matrix from working storage and saves it on an auxiliary memory device. Its form is:

DEACTIVATE a

where a is a matrix name.

In our former example it may be desirable to save the matrix A (e.g. for report writing) rather than delete it entirely. One way is to replace the ERASE statement by:

100 DEACTIVATE A

The ACTIVATE statement retrieves a matrix previously deactivated. Its form is:

ACTIVATE a

where a is a matrix name.

In our example we may execute the following statements after the simplex algorithm terminates:

ERASE B
ACTIVATE A

It is interesting to see how these statements permit both row and column forms to be used efficiently. For example, consider the specification:

MATRIX C(SEQUENTIAL BY COLUMN), R(SEQUENTIAL BY ROW)

Let a part of one program unit build a matrix by rows using R. Later it is desired to process sequentially by column (as in the primal simplex method). Then we simply convert by the assignment

C = R

and may use the DEACTIVATE or ERASE statements if R is no longer needed. In fact, whenever a row-driven algorithm is to be executed, we can

DEACTIVATE C
ACTIVATE R

and proceed to use R. In this manner (use of ACTIVATE and DEACTIVATE statements) the efficiency of row versus column data structures could be employed in each segment of the master algorithm.

6.4 PRESERVE and RESTORE Statements

The PRESERVE statement saves a matrix by writing it to a specified unit. (The unit's disposition is under the control of the operations research programmer.)

Its form is:

PRESERVE (cilist) a [, a]′ · · ·

where cilist is a control information list, and a is a matrix name.

The RESTORE statement restores a matrix which was preserved. Its form is:

RESTORE (cilist) a [, a,]· · ·

In our former example we may wish permanently to save the A matrix. Then we could write

```
100 PRESERVE (6) A
    ERASE A
```

where '6' identifies the disposition. Later, if A is needed (e.g. for report writing), then we could write

```
RESTORE (6) A
```

7. Data Transfer Input/Output Statements

This enhancement specifies a special READ interpretation when referencing a matrix. The declared mode of access determines the meaning of the statement. Further, an output statement, PICTURE (which is available in mathematical programming systems, e.g. MPSX), is introduced.

7.1 READ Statement

The READ statement is the data transfer input statement. Its proposed special form for matrices is:

READ (cilist) a

where cilist is a control information list and a is a matrix name.

The format of data cards is free except that the first column contains an asterisk (*) or a blank. The first data card is one of the following:

Mode	Form of Card 1
BY COLUMN	*[m] [, NAME = n]
BY ROW	*[m] [, NAME = n]
BY ELEMENT	*i, j [NAME = n] [, NAME = n]

where m, i and j are positive integers and n is a name.

If the name options are not used, the current name (possibly none) remains in effect. If the index numbers (e.g. m is the column number in mode BY COLUMN) are not used, then the next highest index number is used. Thus, to simply create two columns named 'C1' and 'C2' respectively we may write

```
MATRIX A(SEQUENTIAL BY COLUMN)
READ A
READ A
```

The data cards would appear as

```
*, NAME = C1
*, NAME = C2
```

The column numbers 1 and 2 would automatically be used.

On the other hand, to create only column 2 (before column 1) we may have the statement 'READ A' appear once, and the data card would be:

```
*2, NAME = C2
```

After card 1 the values appear. In the case of BY ELEMENT the input consists of one index pair (by name, number or both) and the value. (No real gain in ease-of-use for input appears in the BY ELEMENT manner of access.) However, for BY COLUMN and BY ROW manners of access, the nontrivial elements are declared with column 1 blank and card k (for (k-1)st element) as follows:

$$\begin{bmatrix} \underline{n} = \underline{v} \\ (\underline{m}, \underline{v}) \end{bmatrix} [, \cdots]$$

where $\underline{n}$ is a name, $\underline{v}$ is a value conforming to data type, and $\underline{m}$ is an index number. For example, suppose we have:

```
   MATRIX A(SEQUENTIAL BY COLUMN)
   DØ 10 I = 1, 3
10 READ A
```

Then, 3 columns of A will be read in. Suppose the data cards are as follows:

CASE 1:

$$\left.\begin{array}{l}\text{*, NAME = C1}\\ \text{R1 = 1.0, R3 = 5.0}\end{array}\right\}\text{column 1}$$
$$\left.\begin{array}{l}\text{*, NAME = C2}\\ \text{R1 = −20.0, R2 = 4.0}\end{array}\right\}\text{column 2}$$
$$\left.\begin{array}{l}\text{*, NAME = C3}\\ \text{R2 = 3.0, R3 = 6.0}\end{array}\right\}\text{column 3}$$

The matrix will be:

$$\begin{array}{cc|ccc} & & \multicolumn{3}{c}{\overbrace{\hspace{5cm}}^{\text{column names}}} \\ & A & C1 & C2 & C3 \\ \text{row names} & \begin{cases} R1 \\ R3 \\ R2 \end{cases} & \begin{matrix} 1.0 \\ 5.0 \\ \ \end{matrix} & \begin{matrix} -20.0 \\ \ \\ 4.0 \end{matrix} & \begin{matrix} \ \\ 6.0 \\ 3.0 \end{matrix} \end{array}$$

CASE 2:

* (1, 1.0), (2, 5.0)	column 1
* (1, −20.0), (3, 4.0)	column 2
* (3, 3.0), (2, 6.0)	column 3

This produces the same matrix except without names for rows and columns.

We see that the last column read needs an end-of-data marker to separate it from the next input. Thus, we add that the last card must be an asterisk(*). In general, the contents of a column lies between 2 asterisked cards, so there will be $k + 1$ asterisks if k columns are to be read. All but the first and last asterisked cards serve to separate columns and specify end-of-column-record simultaneously with top-of-column-record for successive columns.

7.2 PICTURE Statement

The PICTURE statement is used to output matrix structure. Its form is:

PICTURE a [(o[, o]· · ·)]

where a is a matrix reference and o is an option. The list of options are:

Option	Meaning
NOSORT	Use index numbers, not names, to determine order of printing rows and columns.
SORT(i)	Sort and print names for first i characters only (where i is a nonnegative integer).
NOLEGEND	Suppress printing of legend.
HEADING = c	Print heading described by the character expression c.

Before illustrating the use of the PICTURE statement and the options, let us comment on the defaults.

If the NOSORT option is not used, the rows and columns are printed in their name sort order. The SORT option allows the name sorting and printing to be truncated. (For example, the name may have maximum length of 15 characters, while every name used has length not exceeding 5. In that case the 10 trailing blanks would be outputted unless the option SORT (5) were used. This would produce a neater printed output; however, permitting the retention of all characters, including trailing blanks, is needed to allow the output to be read by a program with format field lengths data independent.) It should be noted that both NOSORT and SORT (i) options may be used. This would mean the index numbers are to be used, but when printed, the names are truncated.

If the NOLEGEND option is not used, the output includes a legend to describe the output (see example 1 below).

If the HEADING option is not used, the heading will be:

MATRIX: a

where a is the matrix reference with values of the index reference (if used) printed. For example, if we write

$I = 1$
$J = 10$
PICTURE A(I:J, *)

then the heading will be:

MATRIX: A(1:10, *)

To illustrate the PICTURE statement let us consider the following arithmetic type matrix used earlier:

$$\begin{array}{cccc}
 & C1 & C2 & C3 \\
R1 & 1.0 & -20.0 & \\
R3 & 5.0 & & 6.0 \\
R2 & & 4.0 & 3.0
\end{array}$$

EXAMPLE 1: PICTURE A

```
Output:
          MATRIX: A
          C  C  C
          1  2  3
                      }4 trailing blanks in column names
R1        1— B
R2           A  A
R3        A     A
       ⏟
       4 trailing
          blanks    in
          row    names
```

Legend	
Symbol	Range
A	(1, 10]
B	(10, 100]
C	(100, 1000]
D	(1000, 10000]
E	.GT.10000
V	[.1, 1)
W	[.01, .1)
X	[.001, .01)
Y	[.0001, 001)
Z	.LT.0001

EXAMPLE 2: PICTURE A(SORT(2), NOLEGEND)

Output:

```
        MATRIX:  A
          C    C  C
          1    2  3
      R1  1 —  B
      R2       A  A
      R3  A       A
```

EXAMPLE 3: B = IMAGE(A)
PICTURE B(SORT)(0), NOLEGEND, HEADING='IMAGE OF A')
Output:

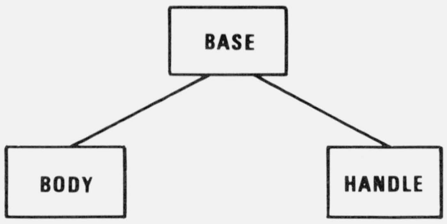
```
      MATRIX: IMAGE OF A
            1  2  3
         1  1  1
         2  1     1
         3     1  1
```

8. Data Structures

This section describes internal data structures subordinate to the features of the ANSI FORTRAN enhancement, notably to the features described in sections 3 and 4. The purpose is to provide a more complete understanding of matrix operations. Further, the data structures are designed to: (1) effect economic utilization of space and time in handling matrices, and (2) insure room for future extensions in the language as well as in structure.

8.1 Explicit Structure

A matrix is represented internally as a base linked to a body and a handle:



The body is equivalent, regarding information content, to an array (cf. sec. 2.1). The handle contains the row and column names. The base not only connects the handle and body, but also contains other information about the matrix important to processing.

8.1.1 Base

The base of a matrix is a contiguous, fixed-length list containing data, including the following six items:

1. matrix name
2. matrix type and access mode
3. effective and maximum row size
4. effective and maximum column size
5. links to handle and body
6. reference value

The matrix name, type, access mode, maximum row size and maximum column size are declared by the matrix specification statement (see section 3.1). (It should be noted that PICTURE uses the matrix name in the default heading (see section 7.2), so its presence is desirable. Further, the processor may use the matrix name and other characteristics for debugging utility.)

The effective row and column sizes and the links (item 5) are initially null and may change during execution. The reference value (item 6) depends on the access mode. For sequential access, it contains the key (column or row) index number most recently referenced; for random access it contains a pointer to the most recently referenced element.

8.1.2 Body Substructure

The body of a matrix is a substructure which logically consists of two entities: indicial information and elemental information. Each is a list with a correspondence between them. The reason for this structure is to provide a variety of compact forms frugal with space and time against a background of expected demands.

The basic compact form is based on the supersparsity [8] that prevails in operations research models which involve large, structured matrices. Supersparsity pertains to replication of nontrivial elements plus a distinguished set of element values.

For logical data types only .TRUE. is nontrivial, so supersparsity abounds, and the need for elemental information disappears.

For arithmetic types, the distinguished values are +1 and − 1; other nonzeroes are put into a list without replication. The idea is to reference them in the indicial information list, and it takes less space to store an index (pointer) than to replicate the nonzero value. This list of nonzeroes is called the literal pool.

For character types the application of supersparsity requires further study, and we not give details here. We shall focus on the arithmetic type, particularly with mathematical programming models in mind.

One design is to have one master literal pool shared by all matrices; another is to have a separate literal pool for each matrix. While the latter may result in more space consumption, we shall use it to facilitate certain operations. For example, the storage control statements (see section 6) function better with separate literal pools. Moreover, a processor is better able to manage storage by relocating a literal pool (and change only its base address, thereby keeping the index pointers valid).

8.1.2.1 Sequential Access Mode

A matrix declared sequential is maintained in a form optimal for sequential access by column, by row or by element. In many operations research models the matrix information, once completely specified, is used in precisely the declared manner. For example, a conventional pricing scheme for implementing the simplex algorithm is to proceed column-by-column in their natural order.

The BY COLUMN and BY ROW sequential access modes have a natural interchangeability so we shall only describe the data structure for SEQUENTIAL BY COLUMN.

Each column is composed of a header and an indicial list. The header contains, contiguously, the following items:

1. link to next column
2. number of $+1$ coefficients
3. number of -1 coefficients
4. number of non-zero, non-unit values.

(Alternative schemes are possible but their relative merits are computer dependent. This is an avenue for further study.)

The column links (item 1 of each column header) form a circular list. This implies that sequential processing wraps around to the first column after the last column is referenced.

The indicial list immediately follows the header and contains $p + m + 2c$ items, where p, m and c, respectively, are the numbers defined in 2, 3 and 4 above. The first p items in this list are the row index numbers with elemental value of $+1$; the next m items are the row index numbers with elemental value of -1; the last $2c$ items are, alternatingly, row index numbers and pointers to the literal pool to identify their associated elemental value.

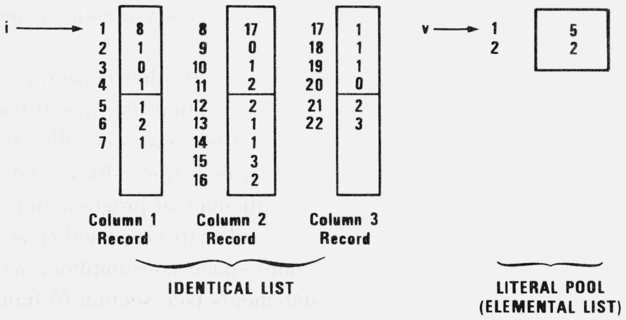
To illustrate the data structures described so far, let us consider the following matrix:

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 5 & \\ 5 & -1 & 1 \\ & 2 & -1 \end{bmatrix} \end{array}$$

Assume the following characteristics:

1. no names have been assigned
2. it is sequential by column
3. no row or column maximum has been specified
4. A was read in column order.

The body would have the following two lists:



The base would appear as follows (after the third READ statement):

A →	1	A	name
	2	a s c	arithmetic, sequential, by column
	3	3 0	effective row size = 3, no maximum
	4	3 0	effective column size = 3, no maximum
	5	0 i + 16 v	no handle, pointer to column 3, base of literal pool
	6	3	latest column number reference

The use of the body pointer and the reference value in the base facilitates sequential processing. For example, consider a DO loop:

```
      DØ     10 J = 1, N
         ⋮
      DJ  = PI(1, *)*A(*, J)
         ⋮
10    CONTINUE
```

(cf. section 4.6). We shall suppose this is the only reference to A in the DO loop. Then, to execute the assignment statement, the processor uses the fact that the pointer to the body is already positioned on the previous column, and its advance becomes simple.

8.1.2.2 Random Access Mode

If a matrix specification statement includes the RANDOM ACCESS option, this results in additional space consumption to set up efficient methods to access any column, row or element at random. When used in conjunction with sequential mode, the data structure is an indexed sequential file. For example, if we have

MATRIX A(SEQUENTIAL BY ROW, RANDOM ACCESS)

then this sets up one pointer per row to avoid a search through the rows until the desired row number is reached.

If access is to be completely random (i.e. by element), then both row and column pointers are established as well as links in the body to chain columns and rows. (See for example [9]).

8.1.3 Handle Substructure

The handle substructure is to permit efficient reference by name. Given by index reference (cf. section 3.2) in the form of names (or name ranges), the associated index number(s) are needed to correspond to the body information (viz., indicial list). Conversely, the name of a particular column number, for example, may be requested.

One basic structure uses binary search trees (see for example Knuth [9]). The success that has been reported in the use of AVL trees make them a plausible alternative. However, the conditions under which AVL trees perform best typify a more general-purpose data base management environment, where references do not adhere to any predictable structure. In many operations research models the programming, such as in matrix generation for a mathematical program, can exploit special structures. The study of different structures well suited for operations research applications is to be considered an avenue for further study.

8.2 Implicit Structure

In executing certain assignment statements it is desirable to procrastinate explicit construction of a matrix in the hope it will never be necessary, thereby saving space. Implicit structures are intended to permit such procrastination.

An implicit structure has no explicit handle or body. The links in the base are used to specify the implicit structure. For example, the simple concatenation matrix assignment statement

A = B.ADDCOL.C

produces an implicit structure with the link items in the base used to specify:

B
C
.ADDCOL.

Notice that a processor can use an implicit structure with no degradation in performance. In the above example of concatenation, columns can still be processed sequentially with the pointer simply moving to C when the columns of B are exhausted.

It should be noted that the programmer may want to use a simple matrix assignment statement (e.g. A = B) for conversion. Therefore, such assignments do not produce implicit structures. At the other extreme, it may be costly to permit complicated matrix assignment statements to be represented with an implicit structure. Therefore, we require that an implicit structure be permitted only in the form:

$$\underline{a} = \underline{b}\ \underline{op}\ \underline{c}$$

where $\underline{a}$ is a matrix name, $\underline{b}$ and $\underline{c}$ are matrix names different from $\underline{a}$, and $\underline{op}$ is an operation. Alternatively, we can have:

$$\underline{a} = \underline{x}\ \underline{op}\ \underline{c}$$

or

$$\underline{a} = \underline{c}\ \underline{op}\ \underline{x}$$

where $\underline{x}$ is a scalar.

Examples of matrix assignment statements which can be represented with an implicit structure are

A = B + C
A = B*C
A = B.ADDCOL.C
A = 5*B
A = 0 −B
A = 5*B + C

Examples which cannot are:

A = A + B
A = B

8.3 Examples

In this section two examples are presented together with comments on the behavior of the data structure.

Example 1: Generation of a transportation-problem polytope.

```
SUBROUTINE TGEN(COST, A, *)                            1
MATRIX COST, A(NAME(12), SEQUENTIAL BY COLUMN)         2
OVERFLOW = 300                                         3
NS = MAXROW(COST)                                      4
ND = MAXCOL(COST)                                      5
NR = NS + ND + 1                                       6
ERASE A                                                7
NC = 0                                                 8
DO 10 J = 1, ND                                        9
```

```
10    A(NS + J, .) = COST(., J)                          10
      A(NR, .) = 'COST'                                  11
      DO 200 I = 1, NS                                   12
      A(I, .) = COST(I, .)                               13
      DO 100 J = 1, ND                                   14
      NC = NC + 1                                        15
      A(., NC) = COST(I, .)//COST(., J)                  16
      A(I, NC) = 1                                       17
      A(NS + J, NC) = 1                                  18
      A(NR, NC) = COST(I, J)                             19
100   CONTINUE                                           20
200   CONTINUE                                           21
      RETURN                                             22
300   RETURN 1                                           23
      END                                                24
```

Explanation:

Statement 1 declares the subroutine (named 'TGEN') and passes the matrix named 'COST', whose effective size is number of suppliers by number of demand points; the matrix named 'A' is to be generated; control transfers to (*) if overflow occurs.

Statement 2 is the matrix specification statement declaring COST (random access) and A (sequential by column).

Statement 3 sets the statement label (300) for transfer if overflow is reached.

Statement 4 uses the intrinsic function MAXROW to retrieve the number of suppliers and statement 5 uses the intrinsic function MAXCOL to retrieve the number of demand points. Statement 6 sets the number of rows (one row per supplier plus one row per demand point and an objective row).

Statement 7 initializes A to a trivial matrix.

Statement 8 initializes a counter (NC) for the number of activities (i.e. number of columns in A).

The DO loop, in statements 9 and 10, sets the row names for demand rows. Statement 11 assigns the last row of A to be named 'COST'.

The two DO loops, beginning with statement 12, are used to generate the activities, with supplier the outer loop (statement 12) and demand point the inner loop (statement 14). Statement 13 assigns the supplier name to the associated supply row in A.

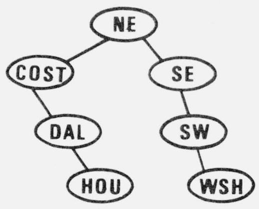
Statement 15 advances the column count, and statement 16 assigns the column name by concatenating the supplier name with the demand point name. Statements 17, 18 and 19 assign the ones and cost to the column. (The processor can store the column contiguously even though the assignments are by element.)

The body is constructed efficiently with the data structures previously described. There will be two +1's per column plus one general nonzero. Therefore, the indicial list will contain 4 items plus the 4 items in the header. This yields a total storage requirement of 8 m n for the indicial list, where m = number of suppliers and n = number of demand points. The size of the literal pool depends on the supersparsity of COST. An advanced processor may take advantage of the fact that the literal pool of A is the same as that of COST; there may be merit in considering user control over such declaration of sharing a literal pool.

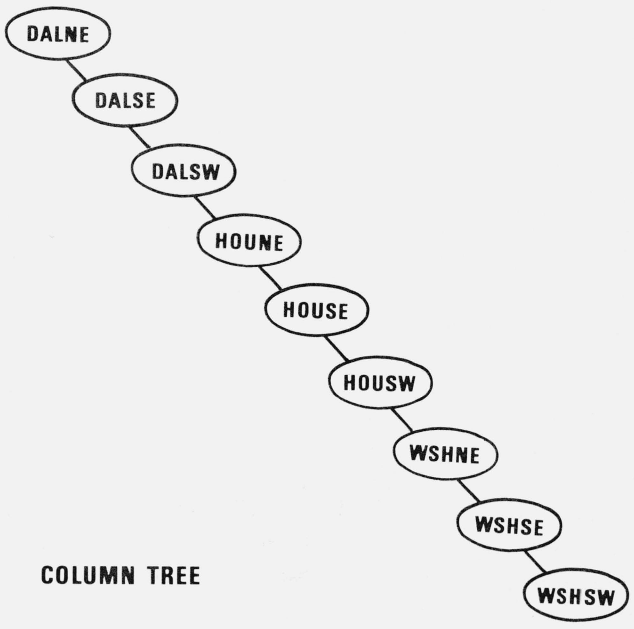
If the handle uses a binary search tree structure, then the (arbitrary) nesting of the DO loops in statements 12 and 14 would affect that tree's shape and hence the efficiency of subsequent searches. For example, suppose the names are:

Suppliers	Demand Points
DAL	NE
HOU	SE
WSH	SW

Then, upon executing TGEN, the tree structure in the handle would be:

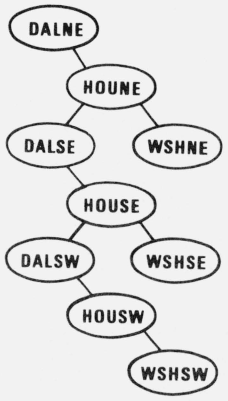


Row Tree



COLUMN TREE

If we interchange the DO loops, the column tree would then be:



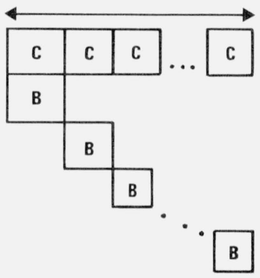
The processor must consider when and how to restructure the trees for efficient processing. This is an avenue for further study.

EXAMPLE 22.: Build a simple dynamic model having block angular structure.

```
      SUBROUTINE TIME (NT, B, C, A, *)
      MATRIX B, C, A(SEQUENTIAL), A1(SEQUENTIAL), A2(SEQUENTIAL)
      OVERFLOW = 20
      DO 10 I = 1, NT
      A1 = A1.ADDCOL. C
      A2 = A2.ADDIAG. B
   10 CONTINUE
      A = A1. ADDROW. A2
      ERASE A1, A2
      RETURN
   20 ERASE A
      OVERFLOW = 30
      RESUME
   30 RETURN 1
      END
```

Result:

NT = number of replications = number of time periods



9. Avenues for Further Study

This paper has described key elements of an enhancement of ANSI FORTRAN in order to accommodate the needs of the operations research analyst/programmer. Several avenues for further study have been cited in this report (e.g. those of data structures). In this section four additional avenues for further study are noted. They have been selected both for their importance and for their promise of successful completion.

9.1 Processor Development

In order to test and develop this enhancement a processor is needed. A first effort might consist of developing a translator, perhaps by extending the NBS precompiler [11]. This should include a capability for experimenting with different data structures. Further, since testing should be performed on large problems, such as arise in the intended operations research applications, efficiency of the translation should be on a par with that of a high quality compiler.

A grander endeavor would permit specification of the computing environment (e.g. paging, dynamic storage allocation, time-sharing, etc.). Then, the processor could provide simulated results to facilitate experimentation with different structures or syntaxes in a variety of environments.

9.2 Environmental Effects

The ANSI FORTRAN Committee has rejected dynamic storage allocation and urges that enhancements conform to the one-pass compilation stipulation. These two restrictions exemplify the Committee's posture of maintaining processor independence. One reason for this posture is cost. It would therefore be useful to measure the gains from relaxing such restrictions. Thus, one avenue for further study is to develop a measure of gain, at least relative to this enhancement, for such relaxation.

Further, one can explore environmental effects without attempting to measure gain. For example, an extension of [10] would be appropriate to study the effects of paging on this enhancement.

If the processor, described in section 9.1, were developed, the study of environmental effects could include experimentation with different scenarios.

9.3. Dynamic Attribute Recognition

To save time and space it may be desirable to identify certain attributes for matrices (e.g. symmetry). The data structure could be extended to keep an attribute identity in the base. Further, the matrix specification statement could be extended to permit declaration of special types of matrices (e.g. diagonal).

Once this notion of attributes, or special matrices, is introduced, it then becomes interesting to see if the processor could be designed to recognize dynamically certain attributes when they prevail. The question of whether such recognition is worth the cost would, of course, require study.

9.4. General packed Arrays

Most of the concepts and data structures described in section 8 can easily be extended to higher dimensional tensors. The problem rests with a development of syntax conformal to ANSI criteria.

10. References

[1] Clarification of Fortran Standards—Initial Progress, (1969), Comm. ACM, 7, 10, 590.

[2] USA Standard FORTRAN (USAS X3.9-1966), USA Standards Institute, NY.

[3] R. Bayer and C. Witzgall (1979), Index Ranges for Matrix Calculi, Comm. ACM, **15**(1972), 1033–1039.

[4] R. Bayer and C. Witzgall (1970), Some Complete Calculi for Matrices, Comm. ACM, **13**, 223–237.

[5] D. M. Brandon, Jr. (1974), The Implementation and Use of Sparse Matrix Techniques in General Simulation Programs, The Comp. J., **17**, 165–170.

[6] M. J. Dillion, P. M. Jenkins and M. J. O'Brien (1970), An Approach to Matrix Generation and Report Writing for a Class of Large-Scale Linear Programming Models, in Applications of Mathematical Programming Techniques, E. M. Beale (ed.), English University Press, London.

[7] W. P. Heising (1964), History and Summary of FORTRAN Standardization Development for the ASA, Comm. ACM, 7, 10, 590.

[8] J. E. Kalan (1971), Aspects of Large-Scale In-Core Linear Programming, Proc. ACM, 304–313.

[9] D. Knuth (1972), The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 2nd ed., Addison-Wesley.

[10] C. B. Moler (1972), Matrix Computations with Fortran and Paging, Comm. ACM, 15, 4, 268–270.

[11] D. J. Orser (1974), A Portable General Purpose Fortran Processor, Workshop on Numerical Preprocessors for Numerical Software, Jet Propulsion Laboratory, Pasadena, Ca.

[12] D. L. Ulery and H. M. Khalil (1974), A Survey of Language-Oriented Systems for Numerical Linear Algebra, The Comp. J., **17**, 82–88.

[13] M. H. Wilson (1973), Flexible Subarray Facilities for Classical Programming Languages, IBM Houston Scientific Center, Tech. Rept. No. 320-2426.

[14] American National Standard Programming Language FORTRAN (ANSI X3.9-1978), American National Standards Institute, Inc., New York, N.Y.