General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

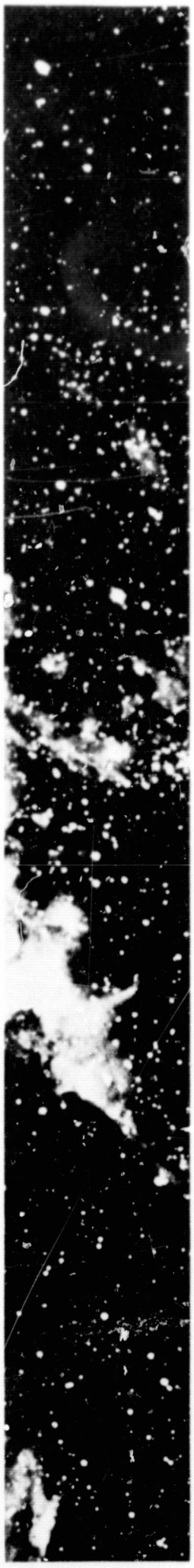
- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

DSRF

SPACE SYSTEMS

Document No. 70SD5478



MATE USER'S MANUAL

FINAL REPORT

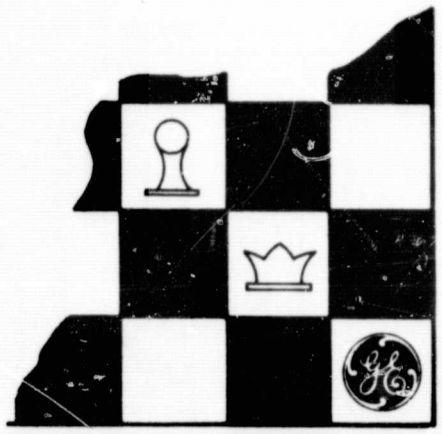
Study-Simulation of Space Station Dynamics

NOV 1970
RECEIVED
NASA STI FACILITY
INPUT BRANCH

N71-15159
(ACCESSION NUMBER)
224
(PAGES)
CR-102971
(NASA CR OR TMX OR AD NUMBER)
(THRU)
03
(CODE)
31
(CATEGORY)
FACILITY FORM 602

GENERAL ELECTRIC

31



DOCUMENT NO. 70SD5478
CONTRACT NO. NAS-8-25710
26 OCTOBER 1970

MATE

USER'S MANUAL

FINAL REPORT
STUDY - SIMULATION OF SPACE STATION DYNAMICS

PREPARED UNDER CONTRACT
NAS 8-25710

FOR

GEORGE C. MARSHALL SPACE FLIGHT CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
MARSHALL SPACE FLIGHT CENTER, ALABAMA 35812

PREPARED BY _____
M.J. GAITENS
PRINCIPAL INVESTIGATOR

APPROVED BY _____
C.V. STAHLE, JR.
PROGRAM MANAGER

GENERAL ELECTRIC
SPACE SYSTEMS ORGANIZATION
Valley Forge Space Center
P. O. Box 8555 • Philadelphia, Penna. 19101

ABSTRACT

This report provides the necessary information for use of the MATE digital computing system. MATE is an acronym for Matrix Algibra Translator and Executor. It is a digital computing system that gives the analyst a rapid and direct means of presenting any problem formulated in terms of matrix, vector and/or scalar arthmetic to the computer for numerical evaluation. Included in this report are complete coding instructions for use by the analyst in solving his problem. This includes input data procedures; coding matrix, vector and scalar expressions and statements; available logical operations, expressions and statements; control statements; and procedures for coding MATE statements and routines. Sample problems are included to demonstrate the operation of the MATE system.

PREFACE

This report presents a manual for use of the MATE computing system initiated under NASA Contract NAS 8-21043, "Dynamics of Air Elastic Space Vehicle" and completed under Contract NAS 8-25710, "Simulation of Space Station Dynamics." A detailed description of the MATE system including the coding of the Translator and Executor portions of the system is presented in the MATE Programer's Manual. This report is limited to the information necessary for the analyst to use the MATE system.

A complete description of the procedures for coding a problem in the MATE language is provided.

The first chapter is a brief introduction to MATE describing the system and its intended use. The second chapter is an introduction to the manual and discusses its use. The subsequent chapters provide detailed coding instructions and define the operations performed by the MATE system. Sample problems are described and solved in Chapter 13 and Appendices F and G. Also included in the Appendices are error messages provided by the system and arrangements to be used in preparing a MATE deck.

ACKNOWLEDGEMENTS

The author wishes to thank: E. H. Worley, Jr., and C. V. Stahle, Jr. for their support of this work; C. G. Petroff for his criticism of this document; and Judy Matthai for her invaluable aid in preparing it. The author also feels especially indebted to J. R. Freelin and R. E. Roach for the inspiration and advice they provided during the initial stages of this project.

TABLE OF CONTENTS

Chapter			Page
1.0	INTRODUCTION		1-1
	1.1	The MATE Translator	1-2
	1.2	The MATE Executor	1-4
2.0	INTRODUCTION TO THIS MANUAL		2-1
	2.1	Glossary	2-2
3.0	DATA INPUT PROCEDURES		3-1
	3.1	Input Data Blocking	3-1
	3.2	Contents of a Standard Data Block	3-2
	3.3	Card Format	3-2
	3.4	Array Header Cards	3-3
	3.5	Number Coding	3-9
	3.6	Repeat Field Indicator	3-10
	3.7	Row and Column Indicators	3-10
	3.8	Matrix Coding	3-11
	3.9	Coding of Logical Arrays	3-13
	3.10	Coding of Alphanumeric Data	3-13
	3.11	Differences Between the Coding of Data and Constants	3-14
	3.12	Summary	3-15
4.0	MATRIX, VECTOR, AND SCALAR EXPRESSIONS AND STATEMENTS		4-1
	4.1	Naming Rules	4-1
	4.2	Arithmetic Expressions	4-1
	4.3	Arithmetic Expressions With Constants	4-4
	4.4	Arithmetic Function References	4-5
	4.5	Hierarchy of Operations	4-6
	4.6	Definition of Operations	4-8
		4.6.1 Scalar-Scalar Operations	4-8
		4.6.2 Scalar-Matrix Operations	4-9
		4.6.3 Matrix-Scalar Operations	4-10
		4.6.4 Matrix-Matrix Operations	4-11
	4.7	Arithmetic Exceptions	4-18
	4.8	Numerical Examples of Basic Expressions	4-19
	4.9	Arithmetic Statements	4-22
	4.10	Number Types	4-23
	4.11	Integer Scalar Expressions	4-24
	4.12	Matrix Name Qualifiers	4-25
	4.13	Subarray Referencing	4-27
		4.13.1 Element Referencing	4-28
		4.13.2 Row Referencing	4-28
		4.13.3 Column Referencing	4-28

TABLE OF CONTENTS (CONTD)

Chapter			Page
		4.13.4 Subarray Referencing	4-29
	4.14	Subarray Referencing in Expressions	4-32
	4.15	Updating a Matrix	4-37
5.0	LOGICAL OPERATIONS, EXPRESSIONS, AND STATEMENTS		5-1
	5.1	Relational Operators	5-2
	5.2	Logical Operators	5-4
	5.3	Logical Constants and Functions	5-11
	5.4	Logical Statements	5-11
6.0	ALPHANUMERIC CONSTANTS, VARIABLES, AND STATEMENTS		6-1
7.0	CODING MATE STATEMENTS		7-1
8.0	CONTROL STATEMENTS		8-1
	8.1	The Unconditional Transfer Statement	8-1
	8.2	The Computer GO TO Statement	8-1
	8.3	The Arithmetic IF Statement	8-2
	8.4	The STOP Statement	8-2
	8.5	The Logical IF Statement	8-3
	8.6	DO Looping Statement	8-3
	8.7	The CONTINUE Statement	8-5
9.0	ROUTINE REFERENCE STATEMENTS		9-1
	9.1	Routine Reference Statement	9-2
	9.2	Function References	9-6
10.0	CREATION OF MATE ROUTINES		10-1
	10.1	Routine Definition Statement	10-1
	10.2	Function Definition Statement	10-5
	10.3	Summary of Restrictions on the Routine Definition Statement	10-6
	10.4	Individual Array Pool Definition	10-7
	10.5	Argument Restrictions	10-9
	10.6	The RETURN Statement	10-11
	10.7	The END Statement	10-11
11.0	CALLING FORTRAN SUBROUTINES		11-1

TABLE OF CONTENTS (CONTD)

Chapter		Page
12.0	MISCELLANEOUS MATE STATEMENTS	12-1
12.1	Data Type Specification Statements	12-1
12.2	The DELETE Statement	12-1
12.3	The READ Statement	12-2
12.4	The REPORT Statement	12-2
13.0	A SAMPLE PROBLEM	13-1
13.1	The Sample Problem as a General-Purpose Routine	
Appendix A	MATE System Functions	A-1
Appendix B	Translator Error Messages	B-1
Appendix C	Mate Executor Control Matrices	C-1
Appendix D	Executor Error Messages	D-1
Appendix E	Restrictions on the IBM 7090/94 Version of Mate	E-1
Appendix F	Sample Problem - Properties of Point Mass System	F-1
Appendix G	Sample Problem - Elastic Two-Body ATM Simulation	G-1
Appendix H	Deck Set-Up for IBM 7094 Version of MATE	H-1

CHAPTER 1.0 INTRODUCTION

MATE is an acronym for Matrix Algebra Translator and Executor. It is a digital computing system that gives the analyst a rapid and direct means of presenting any problem formulated in terms of matrix, vector, and/or scalar arithmetic to the computer for numerical evaluation.

The creation of MATE was prompted by the need to mathematically simulate the structural/control system/environmental interaction problem for large flexible spacecraft. A block diagram of a typical system is shown in Figure 1. Writing a conventional "canned" program, which contains all of the general equations of the system and into which the user loads only the numeric data and controls that specify the options to be taken, was not considered feasible. Sufficiently general equations of such systems do not exist because of the multitude of the possible representations and the uncertainty of the complexity required for a particular configuration.

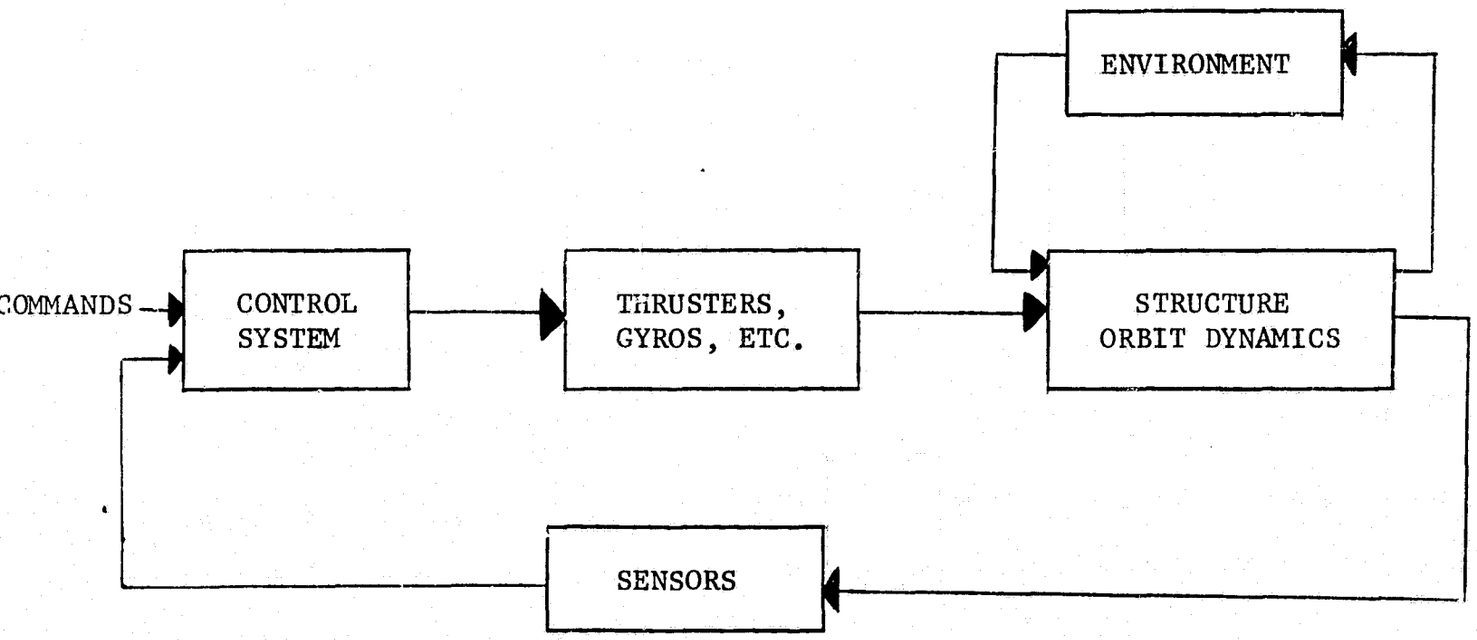


FIGURE 1. BLOCK DIAGRAM OF TYPICAL SYSTEM

This is not to say these spacecraft can't be analyzed. For a given structure, control system, or sensor, etc., the experienced analyst can decide how each component should be modeled and derive the equations which predict its behavior. After determining the equations of each component and defining its interface with the rest of the system, all required information to code a program which will allow study of the system has been obtained.

Unfortunately, there is a great difference between having equations for a program and having the actual program itself. The creation of a conventional program involves a sizable investment of both time and money. Moreover, if the design of the spacecraft system under study should change considerably, it may be found that drastic modifications are required to the program. In fact, substantial changes to the model may mean that the program just written will have to be scrapped and another program written to replace it. MATE was written to combat this unnecessary waste of time and effort by providing a framework into which the analyst loads his equations directly and lets the computer program itself.

The MATE computing system consists of two main programs - the translator and the executor. It is the purpose of the translator to read the user's equations which are usually written in matrix or vector algebra and translate these equations into a digital computer program. The executor provides the bookkeeping support necessary for numerical evaluation of the user's equations. The flow of information between the Translator and Executor is shown in Figure 2.

1.1 The MATE Translator

The MATE translator functions much as a compiler in that it reads the user's equations, devises a computing sequence from them, and then generates the machine instructions required to perform these computing sequences. The user's equations and control statements which direct the branching and computation flow within the

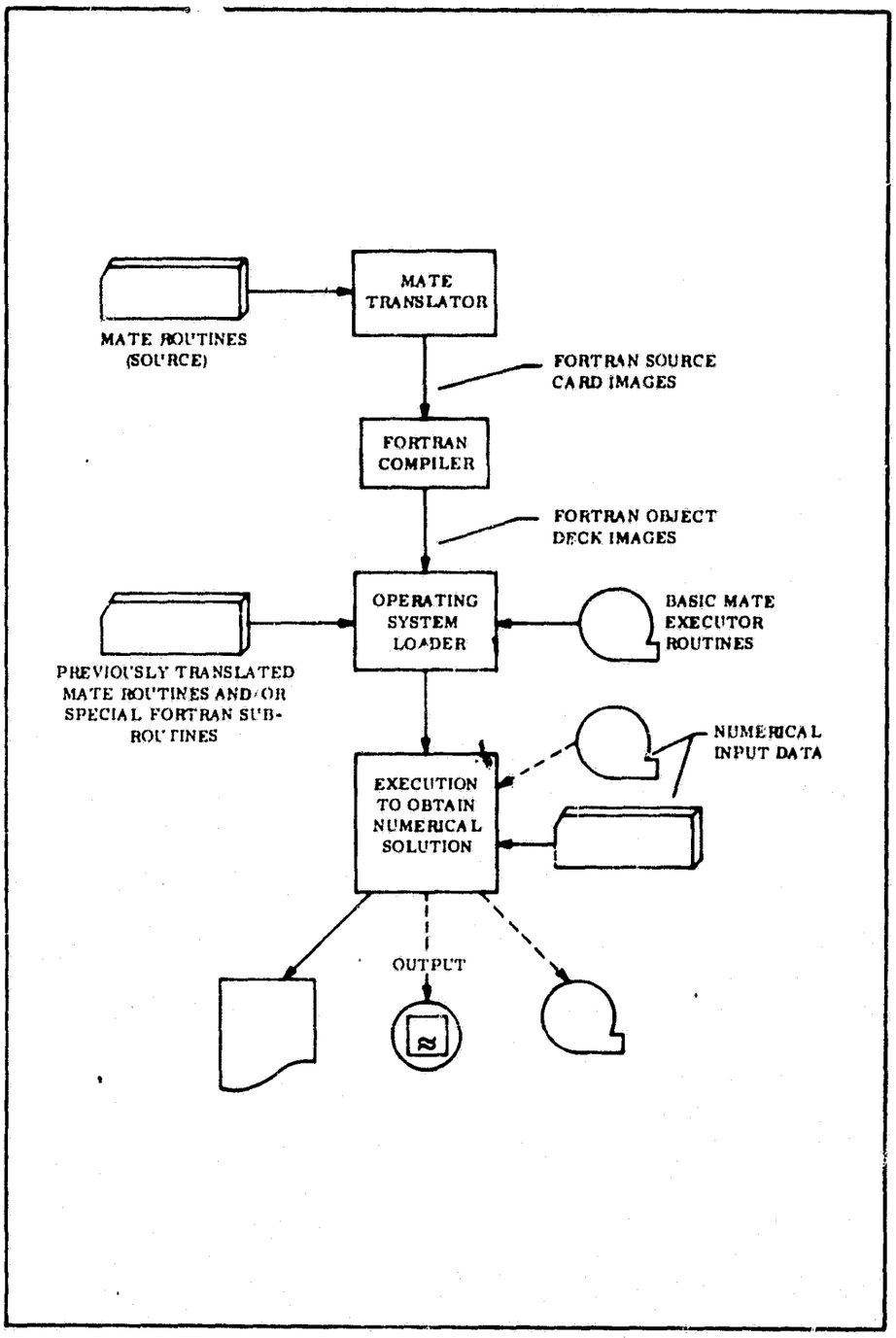


FIGURE 2. MATE INFORMATION FLOW

program are loaded into MATE in a coding language which is structured much like Fortran. In fact, the MATE coding language presents a highly generalized form of Fortran in which the variables may be matrices and vectors as well as scalars. The definition of the Fortran operation symbols has been expanded and new operation symbols for matrix and vector operations and psuedo operations have been defined and the control statements in the language have been brought to the most general form. MATE is upward compatible with Fortran so that the user who already knows some Fortran need not learn new rules for familiar operations. Figure 3 shows a comparison of the coding of a routine which evaluates the highest derivatives of a second-order matrix differential equation in both Fortran and in MATE. The elimination of the need to perform scalar arithmetic greatly simplifies the coding job so that the MATE statements become a direct copy of the matrix equations. Also note that if the Fortran routine were to be capable of processing models so large that all the required matrices would not fit into core, it would require substantial modification whereas the MATE routine is already valid for any size model.

The ease of the linkages between routines in the MATE language encourages modularization of the solution. This has several advantages, but perhaps the most important is that the program may be easily and readily changed to reflect the changes in the model.

1.2 The MATE Executor

The executor program of MATE is part of the MATE system. After the translator has broken the user's equations down to a sequence of elementary matrix and vector operations, it is the MATE executor which performs the actual numerical processing of the data. The executor also performs data management and bookkeeping services. It keeps track of the current size and type of all matrices defined in the problem. It also manages the core resources of the computer compacting the data to make the

$$\{\ddot{X}\} + [C]\{\dot{X}\} + [K]\{X\} = \{F\}$$

LET

$$\{\dot{X}\} = \{Y\}$$

$$\{\dot{Y}\} = -[C]\{\dot{X}\} - [K]\{X\} + \{F\}$$

CODE A ROUTINE TO COMPUTE $\{\dot{X}\}$ AND $\{\dot{Y}\}$ GIVEN $\{X\}, \{Y\}, [C], [K], \{F\}$

FORTRAN

```
      SUBROUTINE DERIVE (X, XDOT, Y, YDOT, C, K, F, N)
      DIMENSION X (N), XDOT(N), Y(N), YDOT(N), C(N,N), K(N,N), F(N)
      REAL K
      DOUBLE PRECISION SUM

      DO 20 I = 1, N
   20 XDOT(I) = Y(I)

      DO 50 I = 1, N
      SUM = 0.0
      DO 40 J = 1, N
   40 SUM = SUM - C(I,J) * XDOT (J) - K(I,J) * X(J)
   50 YDOT(I) = SUM + F(I)

      RETURN
      END
```

MATE

```
      ROUTINE DERIVE (X, XDOT, Y, YDOT, C, K, F)
      XDOT = Y
      YDOT = - C * XDOT - K * X + F
      END
```

FIGURE 3. COMPARISON OF MATE AND FORTRAN CODING

best use of available core storage and even automatically goes to auxiliary external storage devices when the core becomes full. It checks the "reasonableness" of every operation it is to perform. It selects the best approach to the operation considering where the data is and where it will have to be in order to perform the operation. It also considers the type of data involved in the operation and converts this data as required.

This report is one of two volumes which describe the MATE system and it provides the necessary information to enable the analyst to use the MATE system. The use of this manual is described in the next chapter which provides a suggested method for the analyst to become familiar with and utilize MATE.

A detailed description of the MATE system including the Fortran IV coding of both the Translator and Executor are presented in the Programmer's Volume. The information contained in that report is necessary for maintaining the system and converting it to other computers but is not essential to the use of the MATE system by the analyst.

CHAPTER 2.0 INTRODUCTION TO THIS MANUAL

This chapter serves as a foreward to the remainder of this volume. In it the author will suggest how and in what sequence to read this manual. In addition some basic terms will be defined.

In order to use MATE effectively, the engineer/analyst must be able to comfortably formulate his mathematical model primarily in terms of matrix and/or vector arithmetic. Therefore, in this manual, it is presumed the reader is at least somewhat well versed in matrix or vector algebra; however, the presentation does not require previous knowledge or experience in programming the digital computer.

One of the primary goals in the creation of MATE was to endow the digital computer with as much compassion and common sense as was possible for an inanimate machine. In other words, the number of picayune rules (which serve mainly to frustrate the user) have been reduced to a minimum. Consequently, the remainder of this manual is an attempt to give the reader a feeling for the power and convenience of the MATE language, rather than list a long string of "Thou s'lt not's". In fact, the successful use of MATE requires only that what you want done make computational sense and that it be described in the context of the MATE language.

Since the most important concept to be defined in this document is the concept of MATE itself, it is suggested that the reading be done at as fast a pace as comfortable in order to grasp the big picture. Don't stop after every sentence to ponder "now what did he mean by that?" MATE really is an easy language and once the gist of it is learned, the rules of the language will become obvious and natural.

MATE is readily adaptable to any large computer which has a Fortran IV compiler. However, since each manufacturer's machine and compiler are different and each one has its own peculiar restrictions, certain differences will exist in legal MATE on

various machines. The main body of this manual is devoted to universally adaptable aspects of MATE. Machine dependent restrictions are discussed in an appendix.

The rest of this manual is a discussion of how to use MATE. Normally, it should be read in the order presented. However, some short-cuts are possible. For instance, if you only want to use a previously-written MATE program, the loading conventions of the necessary numerical input data is described in the next chapter. Or if you have a simple problem you want to code as quickly as possible, scan these sections: Chapter 4 thru Section 4.10 (if the problem requires submatrices, all of Chapter 4); Chapter 7; Chapter 8 thru Section 8.3; Chapter 9; Chapter 10 Sections 10.1, 10.3, and 10.7; and Chapter 13; then Chapter 3 thru Section 3.8 to code the input data.

This should give you enough information to program small problems. However, some of the best convenience features of the language are in the remaining sections. These should be studied as the need arises or before.

2.1 Glossary

Array - An array is a rectangular arrangement of computer data. Each element of an array requires the same amount of computer storage.

Alphanumeric Data - Computer data which consists of a group of any key-punchable characters. Such data is used for titling.

Digital Computer - An expensive, thoughtless electronic machine. It can be used to create new extraneous problems or to aid in the solution of the problem under study.

Double Precision - An internal computer representation of a floating-point number which utilizes two consecutive computer 'words' for storage and usually yields about 16 decimal digit precision.

Element - A piece of data that resides in an array. In an m row by n column array there are mn elements. A particular element is referenced by specifying the row and column at whose intersection the element is located.

Floating-Point Number - A rational number expressed in decimal notation.

Fortran - A good, fast, and powerful computer language. It is well-suited to scalar problems.

MATE - A good, fast, powerful, and more convenient computer language. It is ideally suited to matrix and vector oriented math models.

Matrix - An array of numeric data.

Operating System - A large package of computer programs normally furnished by the computer manufacturer. This package contains a Fortran compiler, and a monitor/loader which prepares jobs for execution. MATE runs under control of the local operating system.

Scalar - A matrix with but a single element; i.e. a one row by one column matrix.

Single Precision - An internal computer representation of a floating-point number which allocates one computer 'word' for its storage. This usually implies approximately eight decimal digit precision.

Vector - A one column matrix.

CHAPTER 3.0 DATA INPUT PROCEDURES

Often a user's first introduction to a digital computer program is trying to apply it to a particular problem. That is, he has a program deck which reputedly does the processing he requires and has been checked-out. The question he has is how to code his numeric input data for this program. We will devote the rest of this chapter to the answer of that question. Although, since the primary purpose of MATE is to ease the job of writing the program deck, this is like starting at the bottom of an ice cream cone. But, hopefully, it won't get too messy and there's always the best part to look forward to in the following chapters. Note that in this chapter, we are discussing coding of the data upon which the program is to operate, not the coding of the instructions which comprise the program.

3.1 Input Data Blocking

An input data block consists of all cards which are read during one call to a card reading routine. We'll define two types of data blocks - standard and non-standard. A standard data block can define or update any number of arrays or matrices. A non-standard data block, which consists of the cards to be read by a Fortran subroutine, can only update the information in existing arrays; it cannot define new arrays or matrices.

The first data block in every MATE job must be a standard data block. Furthermore, every MATE job requires at least one data block even though it might conceivably be empty, i.e. define no arrays.

The initial standard data block is not read under program control. Indeed, it is read before the program gets control so that by the time the first MATE instruction is executed, it generally already has a flock of matrices to operate upon.

Any remaining data blocks are read under program control and must be stacked in the order in which they will be requested during execution.

The end of a standard data block is denoted by a COMPUTE card. There is no standard card to denote the end of a non-standard data block.

3.2 Contents of a Standard Data Block

A standard data block begins with an array header card. This array header card gives the name, size, and type of data of the first array. The array header card is then followed by the data cards which specify the elements of that array. After these data cards, comes the array header card for the second array which in turn is followed by its data cards, etc. After the data cards for the last array in the block is a COMPUTE card which causes MATE to stop looking for more array header cards and proceed with the computing.

3.3 Card Format

Most cards in a standard data block are in a free-field format, that is a particular piece of information need not start nor end in a certain card column. Hence, blank card columns or commas are used to separate pieces of information. This implies that blanks may be inserted freely before or after a particular item but never within it.

MATE scans only the first 72 card columns for information. The remaining card columns may be used for card sequencing if desired. The 72 card columns we're concerned with can contain either an information field or a comments field or both. These two fields are separated by a slash (/) with the slash and all characters to the right of it being considered as comments. Thus, all information which MATE is to glean from a card occurs between card column one and the first slash. In the absence of a slash, all 72 card columns are assumed to contain information.

3.4 Array Header Cards

Each array in a standard data block is preceded by an array header card. The information field of this card contains four items: the complete name of the array, its size, the type of data it contains, and any options to be used in the reading of the data cards which follow. These items must be given in the order specified. We'll now discuss them in more detail.

The names of the input arrays were assigned by the author of the particular program which you are trying to use. The names on the array header cards must exactly duplicate the names he chose so that when MATE looks for a particular array to perform an operation, its search won't be fruitless. Ordinarily, an array name will consist of only a string of one to six letters and/or digits. However, sometimes one or two integers, known as array name qualifiers, may be appended to the name. Again the values of these integers, or perhaps the range they are to span, must be furnished by the author of the particular MATE program you are using. These array name qualifying integers are coded after the array name and are enclosed in double apostrophes both fore and aft. Although two array name qualifiers appended to a name may be separated with a blank on the header card , it is recommended that a comma be used as the separator since the comma is required when coding MATE instructions and it's unwise to develop bad habit patterns. Occasionally, an array known as an INTERNAL array will be input. These arrays appear to have a hyphenated array name. Actually, the first name given is the array name while the second name is the name of the routine to which the array belongs. The hyphen is key punched as a minus sign. If an INTERNAL name is also qualified, the qualifier comes after the routine name. Some examples of array names are:

```
ALPHA
A12
MASS''3''
TORQUE''1,4''
XYZ-SETUP
KEYS-STMCMP''7''
```

After the array name, which can take any of the above forms, there must be two integers which are the number of rows and columns, respectively, in the array. Note that the number of rows is always given first.

Following the size specification comes the type specification. This specification consists of a key letter or word. (If a word is given only the first letter of it is used to determine the type.) The defined type letters and their meanings are given in the table below.

Key Letter	Suggested Key Word	Each Element of the Array is to be:
R	REAL	A single precision floating point number
I	INTEGER	A signed whole number
D	DOUBLE or DP	A double precision floating point number
C	COMPLEX	A complex number whose real and imaginary parts are represented by a pair of single precision floating point numbers
L	LOGICAL	A logical quantity which has either the value .TRUE. or .FALSE.
T	TITLE	A string of 72 characters to be used as a title on the output of the program
H	HEADING	A string of 12 characters normally used to identify rows/columns of a printout
X		A complex number whose real part only is given in the data which follows. The imaginary part is to be assumed to be zero
Y		A pure imaginary number. The real part of each number is assumed to be zero while the imaginary part only is given on the data cards

Again, any word or any spelling of the given key word which has the same starting letter may be used. So that the array header card

```
A 5,6 D
```

is identical to the header card

```
A 5 6 DOUBLEPRECISION
```

To review, these header cards specify that a five row by six column matrix named A is about to be read, and that its elements are to be stored in the machine in double precision format. Also, note the use of blanks in the above examples. Since a blank is a separator and only one word is allowed to indicate the type, a blank cannot appear between the words DOUBLE and PRECISION. However, since it is necessary to separate the name and type from the number of rows and columns, the other blanks (or commas) are mandatory.

Finally, we have the option subfield. Again key words, of which only the first letter is important, are used to specify the options.

The absence of a key word indicates that the standard option, which incidently does not have a key word associated with it, is to be taken. The currently defined key words are:

ROWS
DIAGONALS

and

SYMMETRIC
UPDATE
NOPRINT

The MATE input data reading routine presumes that all arrays are loaded by columns. For instance, given a string of six numbers such as 1,2,3,4,5,6 and told that these are the elements of a two row by three column matrix, MATE would assume the matrix is:

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

However, if the key word ROWS appears after the type indicator, MATE expects the array elements to be given by rows. Hence, in our two by three example the elements 1,2,3,4,5,6 are interpreted as:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Now for the hard one. When the key word DIAGONALS is given after the type indicator, MATE places the elements along the diagonals of the matrix. To define this procedure more precisely, let r be the number of rows in the array and s be the number of columns. Further, let d be the minimum of r and s, and let e equal the maximum of r and s. Then we say that the array contains d diagonals, each one with e elements. If r is greater than s, these diagonals start consecutively across the first row. If r is less than or equal to s, the first element of each diagonal appears consecutively in the first column. To demonstrate this, consider the following matrices in which the tens digit of each element is the number of the diagonal and the units digit is the number of the element within that diagonal. These matrices could be coded by giving the elements shown in straight increasing order.

$$\begin{bmatrix} 11 & 32 & 23 & 14 & 35 & 26 \\ 21 & 12 & 33 & 24 & 15 & 36 \\ 31 & 22 & 13 & 34 & 25 & 16 \end{bmatrix}$$

$$\begin{bmatrix} 11 & 42 & 33 & 24 & 15 \\ 21 & 12 & 43 & 34 & 25 \\ 31 & 22 & 13 & 44 & 35 \\ 41 & 32 & 23 & 14 & 45 \end{bmatrix}$$

$$\begin{bmatrix} 11 & 52 & 43 & 34 & 25 \\ 21 & 12 & 53 & 44 & 35 \\ 31 & 22 & 13 & 54 & 45 \\ 41 & 32 & 23 & 14 & 55 \\ 51 & 42 & 33 & 24 & 15 \end{bmatrix}$$

$$\begin{bmatrix} 11 & 21 & 31 & 41 \\ 42 & 12 & 22 & 32 \\ 33 & 43 & 13 & 23 \\ 24 & 34 & 44 & 14 \\ 15 & 25 & 35 & 45 \end{bmatrix}$$

$$\begin{bmatrix} 11 & 21 & 31 \\ 32 & 12 & 22 \\ 23 & 33 & 13 \\ 14 & 24 & 34 \\ 35 & 15 & 25 \\ 26 & 36 & 16 \end{bmatrix}$$

The pattern, in case you haven't seen it, is a sawtooth along the horizontal axis for $r \leq s$, and along the vertical axis for $r > s$. The applications for coding in this manner are much harder to see, but a few are presented at the end of the matrix coding section.

Note that the key words ROWS and DIAGONALS are mutually exclusive. You can't load an array under two different sets of ground rules at the same time.

If the array contains only one row and/or only one column, it makes no difference whether the ROWS or DIAGONALS or neither (i.e. columns) key word is specified.

The next optional key word we will consider is SYMMETRIC. This key word will be ignored unless the array is square. If the number of rows does equal the number of columns (i.e. it's square), then the SYMMETRIC key word causes any element loaded into the lower triangular portion to be copied into the corresponding location in the upper triangular portion and vice versa. Moreover, upon completing a column (or row), the next element is placed on the principal diagonal in the following column (or row). Therefore, the element string 1,2,3,4,5,6 for a three row and three column matrix would generate:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

using the SYMMETRIC option. The very same matrix would be generated using the SYMMETRIC ROWS options. The combination DIAGONALS SYMMETRIC is processed somewhat differently. The diagonals are started in the same fashion as under the DIAGONALS option but are considered complete as soon as the last row is hit. Thus, with our element string 1,2,3,4,5,6 for the three by three and using SYMMETRIC DIAGONALS options, we obtain:

$$\begin{bmatrix} 1 & 4 & 6 \\ 4 & 2 & 5 \\ 6 & 5 & 3 \end{bmatrix}$$

So whenever the SYMMETRIC option is given for a square matrix, a symmetric matrix results.

Up to this point, we've implied that to load arrays you list all the elements in a string and hope you have correctly anticipated where the read routine is going to put them. Although this is a primary method of coding arrays, it is certainly not the only way. Other schemes exist whereby the row and column into which an element is to be put may be specified directly. With this device, you can hopscotch freely around the array to load elements in any order. Further since a large percentage of practical matrices are sparse, this method becomes particularly useful if you don't have to load zeros. Therefore, the MATE read routine initializes every element of an array before reading the data cards. Matrix elements are initialized to zero.

The UPDATE option will prevent the automatic initialization of array elements if at the time the array header card is being processed MATE already has an array of the same name, size, and type. The data on the cards which follow the header card then become corrections to the existing matrix, and elements not mentioned in the data are left unchanged. The standard option, implied by the absence of UPDATE, is to treat every array being read as a new array and delete any existing old array with the same name.

Every array read is automatically printed so that a record of the input will appear with the output. Option NOPRINT will inhibit this printing. The user is advised to think twice before employing this option.

To summarize the array header card: the name of the array, the number of rows, and the number of columns must be given in that order. This information is followed by key letters or words which specify the type of the data then any non-standard options. The type specifications must always be given unless the type is to be REAL and no options are specified in which case it could be omitted and the number of columns would be the last piece of information on the header card. The order among

the option key words is immaterial.

Once again, remember that these items must be separated by blanks or commas. Moreover, blanks can be used freely on the card except in the middle of a keyword, number, or alphanumeric part of a name (the punctuation around the name qualifiers may be riddled with blanks); however, all the header information must appear on one physical card.

3.5 Number Coding

Numbers on MATE data cards can be coded in a variety of formats, for instance the number seven thousand could be coded:

7000	7+3	7E3	7D3
7000.	7.+3	7.E3	7.D3
7000.0	.7+04	7E+3	7D+3
+7000	7000-0	7E+03	7D+03

Which brings us to the following rules:

1. A number can have a leading sign. If the sign is omitted, it is assumed positive.

2. A number can contain a decimal point. If the decimal point is omitted, it is assumed to be after the last digit in the fractional part.

3. A number can be expressed as a factor times the appropriate integral power of ten. The power of ten, or decimal exponent, is written as a one or two digit integer and must be preceded by a sign and/or the letter E or D.

4. A number can contain only digits and the characters: a plus sign, a minus sign, a decimal point, and the letters D or E.

5. Blanks are not allowed within a number with the following exceptions: before or after the letter D or E, and after the leading or exponent sign. On the array header card, blanks are not allowed before the letter D or E.

Since the type of the data has been specified on the array header card, the read routine makes no distinction between the decimal exponent letters D and E. In the case integer data has been specified, the given numbers are converted to integers by rounding, e.g. 0.26+1 becomes the integer 3.

3.6 Repeat Field Indicator

Sometimes the same entry is to appear in several consecutive locations. The read routine contains a feature which allows a data item to be preceded by a count of the number of the times the data entry is to be repeated. The count and the data item are separated by an asterisk. For instance, coding 3*1.75 is equivalent to coding 1.75, 1.75, 1.75. The count number coding does not follow the usual rules in that this number can contain only digits and may not have a sign, decimal point or decimal exponent.

3.7 Row and Column Indicators

As we have previously mentioned, consecutive items in a data string are placed in consecutive locations in the array. These locations are either down the columns, across the rows, or along the diagonals of the array as specified by the array header card. Ordinarily, the first item in the data string is assumed to belong in the first row and first column of the array and any other items follow as prescribed. However, the row and column counters may be reset at any point in the data list by using row and column indicators. These indicators consist of an integer pair enclosed in parentheses. Unless the array contains but one row or column, both the row and the column numbers must be given in that order. If the array is a column or row vector, only one number need be given and the missing column or row subscript, respectively, will be understood as one. Note that the indicators reset the read routine's counters so that the data item immediately following the indicators goes into the specified position and the remaining items go into subsequent rows, columns, or diagonals, until either the end of the data or another row or column indicator pair is encountered.

3.8 Matrix Coding

Since a matrix is an array of numbers, a matrix is characterized by a type word which begins with the letter R, I, D, C, X, or Y and by the data items being all numbers. The coding of matrix elements may incorporate repeat field and row and column indicators as desired.

Because the coding of a matrix can extend over as many cards as desired and since all elements need not be coded, a special designator is required to mark the end of a matrix. The designator chosen is the dollar sign. As is the case with the slash, any characters to the right of a dollar sign are treated as comments. The card following a card which has a dollar sign as the last character in its information field must be either an array header card of the next array or a COMPUTE card.

Some examples of card images which define matrices follow:

1. Code a twelve row by twelve column identity matrix named IDENT.

```
IDENT   12   12   REAL DIAGONAL

12*1$   SINCE ONLY NON-ZERO TERMS NEED BE GIVEN
```

2. Code the following twelve row by three column matrix.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix is named SUMER.

```
SUMER,12,3,REAL,DIAG
/FOUR 3X3 IDENTITY SUBMATRICES STACKED
12*1$
```

3. The vector R is to be:

$$\begin{Bmatrix} 7.47863 \\ 0.009635 \\ 30.416912 \end{Bmatrix} \times 10^{+12}$$

It can be coded by:

```
R 3,1
7.47863+12,9635E6,.30416912D 14$
```

4. Code the vector A which is:

$$\begin{Bmatrix} 0 \\ 4 \\ 0 \\ 0 \\ 3 \end{Bmatrix}$$

Any of the following forms could be used:

```
A,5,1/COMMAS AS SEPARATORS
0,4,0,0,3$
```

```
A,5,1/BLANKS AS SEPARATORS
0 4 0 0 3 $
```

```
A,5,1/USING ROW/COL INDICATORS
(2,1) 4 (5),3 $
```

```
A,5,1/USING MULTIPLE COMMAS
,,4,,,3 $
```

Note that (n+1) consecutive commas imply that n consecutive locations are to be skipped.

5. Code matrix DISMAS''2,2'' which is:

$$\begin{bmatrix} 4 & 0 & -2 & 3 \\ 0 & 8 & 0 & -2 \\ -2 & 0 & 8 & 0 \\ 3 & -2 & 0 & 4 \end{bmatrix}$$

by:

```
DISMAS''2,2'' 4 4 R SYM DIAG
4*4, (3,1) 2*-2,3
(2,2) 2*8./REDEFINE 2,2 AND 3,3 TERMS
$ END OF MATRIX
```

3.9 Coding of Logical Arrays

Logical arrays are coded in an identical fashion to matrices with the following exceptions:

1. The values .TRUE. and .FALSE., which are the only values permitted in logical arrays, are abbreviated by the letters T and F respectively in the data strings.

2. The array elements are initialized to the value .FALSE.; hence, in the absence of the UPDATE option only the .TRUE. values need be mentioned.

As an example consider the coding of the logical array KEY which is to be:

$$\begin{bmatrix} .\text{FALSE.} & .\text{FALSE.} & .\text{TRUE.} \\ .\text{FALSE.} & .\text{FALSE.} & .\text{FALSE.} \\ .\text{TRUE.} & .\text{TRUE.} & .\text{FALSE.} \end{bmatrix}$$

The coding could be:

```
KEY 3,3, LOGICAL,ROWS
(1,3) T, (3,1)T T $
```

3.10 Coding of Alphanumeric Data

The coding of data typed HEADING is substantially different and much more restrictive than the coding of matrices and logical arrays. First, each data card contains six 12-character elements. Thus, all 72 card columns are reserved for

information and comment fields are not allowed. Furthermore, since the items are fixed-field, they are not separated by blanks or commas and repeat field or row and column indicators may not be used. Finally, since all array elements must be explicitly given, the end of the array need not and should not be marked by a dollar sign.

TITLE typed data is coded under the same restrictions as HEADING data except each element of the array contains 72 characters and occupies one data card.

3.11 Differences Between the Coding of Data and Constants

In the following chapters, we will introduce MATE instructions which may use explicit constants. Now although these constants appear similar to the data items discussed here, there are several important differences in their coding which are required by the change in context. In an effort to eliminate some of the anguish generated by these differences, we present the following tabulation of forewarnings.

In Array Data Coding	In MATE Instruction Coding
The letters E and D before a decimal exponent are optional and interchangeable.	The letter E or D must precede a decimal exponent. Furthermore, a distinction is made between the two letters.
The logical values .TRUE. and .FALSE. are abbreviated as T and F, respectively.	The logical values can only be written as .TRUE. and .FALSE. with the enclosing dots being mandatory.
Blanks or commas may separate items in a string.	Only commas may separate items in a string.
(n+1) consecutive commas are required to imply n locations are to be skipped.	(n+1) consecutive commas imply n items are omitted from the string except a single comma before the first explicit item or a single comma after the last explicit item implies the first or last item in the list, respectively, has been omitted.
Only the first letter of key words is used.	All letters up thru the seventh are checked on key words.

3.12 Summary

Usually the data for a program written with MATE will be contained in a single standard data block. This block will define a number of arrays in an arbitrary order. Each array definition consists of an array header card followed by some data cards. The last card in a standard data block must be a COMPUTE card. This card consists of the word COMPUTE as the first word in the information field of that card.

CHAPTER 4.0 MATRIX, VECTOR, AND SCALAR EXPRESSIONS AND STATEMENTS

This section discusses the coding of scalar, vector, and matrix equations. Now since we have defined scalars and vectors to be matrices subject to certain size restrictions, points made about matrices are, of course, also valid for scalars and vectors.

4.1 Naming Rules

Each matrix must have a unique name to identify it. The matrix name consists of from 1 to 6 characters. The first character must be a letter of the alphabet. The remaining characters may be alphabetic or numeric. The name may not contain any special characters or imbedded blanks. As a further restriction, if the name consists of 6 characters, both the 5th and 6th characters may not be the letter Z. Such names are reserved for use by the MATE system. These naming rules hold not only for matrices, but also for arrays of non-numeric data as well as function and routine names which will be discussed in later chapters. Some examples of legal and illegal names are given below:

Legal	Illegal
AB	A B (Contains an imbedded blank)
ALFA2	2ALFA (1st character not alphabetic)
RADIUS	DIAMETER (More than 6 characters)
J3754	J37.54 (Contains a special character)
WXYZZ	VWXYZZ (Both 5th and 6th character Z)

4.2 Arithmetic Expressions

An arithmetic expression consists of a sequence of numeric constants, known matrices, and arithmetic functions which are separated by arithmetic operation symbols. This sequence can always be reduced to a single matrix result by performing arithmetic. The key word in this definition is "known". That is, a matrix must

have been previously defined, either by having been read-in or obtained by previous calculations, before it can be used in the evaluation of an expression.

There are nine arithmetic operation symbols defined in MATE. These are:

+	Addition
-	Subtraction
*	Scalar and matrix multiplication
**	Element exponentiation
/	Element division
.*.	Element multiplication
...	Dot (inner) product
.X.	Cross (vector) product
.D.	Multiplication by diagonalized vector

The most common form of an expression consists of two matrices separated by one of the arithmetic operation symbols above. Some examples of the coding of standard operations is given below.

Consider the following basic expressions where F and G are matrices, U and V are vectors (column matrices), and R and S are scalars or constants. Then the coding for the operations commonly defined in ordinary, vector, and matrix algebra would appear as:

	Operation Name	Mathematical Symbolism	Coding
Scalar Operations	addition	$r+s$	R + S
	subtraction	$r-s$	R - S
	multiplication	rs	R * S
	division	r/s	R / S
	exponentiation	r^s	R**S

	Operation Name	Mathematical Symbolism	Coding
Vector Operations	vector addition	$\vec{U} + \vec{V}$	U + V
	vector subtraction	$\vec{U} - \vec{V}$	U - V
	vector magnification	$a\vec{V}$	S * V
	dot product	$\vec{U} \cdot \vec{V}$	U ... V
	cross product	$\vec{U} \times \vec{V}$	U .X. V
Matrix Operations	matrix addition	$F + G$	F + G
	matrix subtraction	$F - G$	F - G
	scalar multiplication	aG	S * G
	matrix multiplication	FG	F * G
	transpose multiplication	$F^T G$	F ... G

where in the last example the symbol F^T denotes transpose (or transposed conjugate) of the matrix F. The examples presume that the usual conformability restrictions are met. We shall defer a rigorous discussion of the operations and conformability as well as the definition of the .*. and .D. operators to a later section.

Rather than go into a formal discussion of what constitutes a legal expression, we will point out that any matrix name in the above list could be replaced by a matrix name proceeded by a plus or minus sign, and any of the expressions above could be enclosed in parentheses and then treated as a matrix name and combined with other arithmetic operators and matrices to form expressions of the desired complexity. Some examples of legal expressions are given below:

A	A known matrix is also an expression
+A	Same as A
-A	Result is a matrix whose elements are the negative of those in A

```
A**-B
-B/A
C+(A/B)
-(A+B)*C
-((A.X.B)...(C.X.-D)
```

Thus, except for the dots which surround some arithmetic operation symbols, and the fact that the operation symbols must be explicitly given (e.g. AB does not imply A*B), the coding of matrix, vector, and scalar equations in MATE is very similar to the way the equations are written during their derivation.

4.3 Arithmetic Expressions With Constants

A scalar constant may also be used in an expression in place of a matrix name. These constants consist of a string of digits and may contain a decimal point or a decimal exponent. If the decimal point is omitted, a whole number is assumed. The decimal exponent field, if given, must start with the letter E or D and contain one or two digits. These digits may be preceded by a sign and represent the power of 10 by which the number is to be multiplied. Some examples of scalar constants are:

```
 0.723
91
7000
7.0 E+0          (i.e. 7.0 x 10^0 = 7)
19761.26 E+1     (i.e. 19761.26 x 10^1 = 197612.6)
7E3
7.E3
7.0E3            (i.e. 7.0 x 10^3 = 7000)
7.E+3
7E03
7.0E+03
.7E4             (i.e. 0.7 x 10^4 = 7000)
7E-3             (i.e. 7.0 x 10^-3 = 0.007)
```

Note that there are several important differences between the coding of constants in MATE expressions and writing numbers for input matrices as discussed in the previous chapter. First, the exponent must always be preceded by the letter E or D. Since in the instructions 7-2 is interpreted as the number 5, while 7E-2 becomes 0.07.

Further, there is a distinction between the letters E and D in that the letter E implies that the number is a single precision number while D implies the number is double precision. That is, the letter sets the type of the number. We'll discuss number types later in this chapter. Finally, since blanks are not used as separators in a MATE expression, a blank may appear anywhere in the constant. An example of an expression which uses scalar constants is given below.

```
-3.14159*RADIUS**2+1/3*BASE*13.35E-4
```

4.4 Arithmetic Function References

We define a function to be a sequence of operations which yield a single array as a result. This result is called the value of the function. If the value of the function is a matrix, that is an array of numbers, it may be used in an expression in the place of a matrix name.

A function is referenced by coding the function name preceded by a dollar sign. If the function requires arguments, they are enclosed in parentheses which follow the function name. These arguments are expressions and are separated by commas if there is more than one. Some examples of function references are given below.

```
$COS (ALPHA)
$INVERS (A+B)
$PBN (2*B,A,C-D,3)
```

Then, using arithmetic function references as matrix names can result in expressions like:

```
2.3*$J2 (CQ-M,$ABC(3)*C)
```

In the above example, the function ABC is called using as its argument the number 3. As a result of the operations in function ABC, a matrix known as its value is generated. This value is then post-multiplied by the matrix C to form the

second argument to the function J2. After the difference of matrices CQ and M has been computed to form the first argument of J2, function J2 is called and during its execution generates another matrix as its value. This final value is multiplied by 2.3 to form the result of the expression.

Note that before a function may be used, the user must know what arguments are required, what the restrictions on those arguments are, and at least roughly the type of matrix that will result from the function given those arguments. After this is known, the arithmetic function reference becomes nothing more than an exotic matrix name as far as expressions are concerned. The definition of functions, that is how to write them, will be taken up in a later chapter.

4.5 Hierarchy of Operations

The parentheses used in the preceding construction of arithmetic expressions may be omitted except when the mathematical meaning of the expression would be changed or the parentheses enclose the arguments of a function. Within a given level of parentheses, or where parentheses are omitted, the order of computation is understood to be:

1) Function reference
2) **
3) .D.
4) *,/,.*.
5) .X.
6) ...
7) +,-

If several operators of the same level appear in the same term, the calculations proceed from left to right.

Consider the following illustrations of equivalent (⇔) and non-equivalent (⇎) expression coding.

(A+B) + C ⇔ A + (B+C) ⇔ A + B + C

A * (B+C) ⇎ A * B + C

A / (-B) ⇔ -A/B ⇔ -(A)/B ⇔ (-A)/B ⇔ A/-B

A...B.X.C ⇔ A...(B.X.C) ⇎ (A...B).X.C

A/B/C ⇔ (A/B) /C ⇔ A/ (B.*.C)

A ** (1/4) ⇔ A ** 0.25

A ** 1/4 ⇔ (1/4) * A ⇔ 0.25 * A

A ** (1/4) ⇎ A ** 1/4

A ** -1/4 ⇔ 0.25/A

There exist two cases where parentheses are required mathematically due to a lack of associativity. However, because of the left to right rule, the parentheses could be omitted in the coding. These cases are:

A ** B ** C = (A**B) ** C ≠ A ** (B**C)

A.X.B.X.C = (A.X.B) .X.C ≠ A.X. (B.X.C)

These examples should have impressed upon the reader that while short-cuts are all well and good, if doubt exists about the possible meaning of an expression, parentheses should be used for clarity and to avoid communications problems between the user and the MATE translator.

Finally, when forming expressions remember that:

1. Every matrix mentioned in an expression must exist at the time the expression is to be evaluated.
2. That an expression may always be reduced to a single matrix by performing arithmetic.

3. That the grouping of terms in an expression follows the same conventions as defined in algebra.

4.6 Definition of Operations

Now, if only the operation symbols were a little better defined, the reader should be able to write expressions with abandon. However, in an expression like A*B, the size and shape of A and B grossly affect the result of the expression as well as the interpretation and "legality" of the operation. That is, the first thing to notice about an expression like A*B in MATE is that it is not discernible whether A and B are scalars, vectors, or matrices. This lack of discernibility is not really a problem, since the analyst who originally codes the MATE expressions knows what the matrix names represent. (However, it does point out the need for adequate comment cards within the routine to enhance its readability to other users.)

The following will define the meaning of the MATE arithmetic operation symbols for the normal (conformable) cases. Let the matrix C equal the result of a basic arithmetic operation involving the matrices A and B; i.e. C = A (operation) B. We will denote the sizes of these matrices by letting r_A, r_B, r_C be the number of rows and s_A, s_B, s_C be the number of columns in A, B, and C, respectively. Further, let the element in the ith row and jth column of the matrices A, B, and C be denoted by a_{ij}, b_{ij}, and c_{ij}.

4.6.1 Scalar-Scalar Operations

When both A and B are scalars, then C will be a scalar, i.e.

$$r_A = s_A = r_B = s_B = r_C = s_C = 1$$

Addition

$$C = A + B \qquad \text{implies} \qquad c_{11} = a_{11} + b_{11}$$

Subtraction

C = A - B implies $c_{11} = a_{11} - b_{11}$

Multiplication

C = A * B implies $c_{11} = a_{11}b_{11}$

Division

C = A/B implies $c_{11} = a_{11} / b_{11}$

Exponentiation

C = A**B implies $c_{11} = a_{11}^{b_{11}}$

C = A.*.B
C = A...B
C = A.D.B
C = A.X.B

} Treated as C = A*B

4.6.2 Scalar-Matrix Operations

When A is scalar and B is not, the result C is a matrix of the same size as B, i.e.

$$r_A = s_A = 1$$

$$r_C = r_B$$

$$s_C = s_B$$

Addition (Note this operation is not defined classically)

C = A + B implies $c_{ij} = a_{11} + b_{ij}$ for all i,j such that $1 \leq i \leq r_B$, $1 \leq j \leq s_B$

Subtraction

C = A - B implies $c_{ij} = a_{11} - b_{ij}$ for all i,j such that $1 \leq i \leq r_B$, $1 \leq j \leq s_B$

Multiplication $a[B]$

C = A * B	implies	$c_{ij} = a_{11}b_{ij}$	for all i,j such that $1 \leq i \leq r_B$, $1 \leq j \leq s_B$

Division

C = A/B	implies	$c_{ij} = a_{11}/b_{ij}$	for all i,j such that $1 \leq i \leq r_B$, $1 \leq j \leq s_B$

Exponentiation

C = A**B	implies	$c_{ij} = a_{11}^{b_{ij}}$	for all i,j such that $1 \leq i \leq r_B$, $1 \leq j \leq s_B$

C = A.*.B
C = A...B
C = A.D.B
C = A.X.B

} Treated as C = A*B

4.6.3 Matrix-Scalar Operations

When B is a scalar and A is not, the result C is a matrix of the same size as A, i.e.

$$r_B = s_B = 1$$

$$r_C = r_A$$

$$s_C = s_A$$

Addition

C = A + B	implies	$c_{ij} = a_{ij} + b_{11}$	for all i,j such that $1 \leq i \leq r_A$, $1 \leq j \leq s_A$

Subtraction

C = A-B	implies	$c_{ij} = a_{ij} - b_{11}$	for all i,j such that $1 \leq i \leq r_A$, $1 \leq j \leq s_A$

Multiplication $[A]*[B]$

C = A*B	implies	$c_{ij} = a_{ij}b_{11}$	for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Division $[A]/[B]$

C = A/B	implies	$c_{ij} = a_{ij}/b_{11}$	for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Exponentiation

C = A**B	implies	$c_{ij} = a_{ij}^{b_{11}}$	for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

C = A.*.B
C = A...B
C = A.D.B
C = A.X.B

} Treated as C = A*B

4.6.4 Matrix-Matrix Operations

The first five operations listed below are element-by-element operations. For conformability, both given matrices, hence also the result matrix, must be of the same size. That is:

$$r_C = r_A = r_B$$

$$s_C = s_A = s_B$$

Addition $[A]+[B]$

C = A + B	implies	$c_{ij} = a_{ij} + b_{ij}$	for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Subtraction $[A]-[B]$

C = A - B	implies	$c_{ij} = a_{ij} - b_{ij}$	for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Element Multiplication (Note the asterisk is enclosed in dots)

C = A.*.B implies $c_{ij} = a_{ij}b_{ij}$ for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Element Division

C = A/B implies $c_{ij} = a_{ij}/b_{ij}$ for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Element Exponentiation

C = A**B implies $c_{ij} = a_{ij}^{b_{ij}}$ for all i,j such that $1 \le i \le r_A$, $1 \le j \le s_A$

Matrix Multiplication - Conformability for matrix multiplication requires that the number of columns in the first matrix equals the number of rows in the second. That is:

$[A][B]$

$$s_A = r_B$$

$$r_C = r_A$$

$$s_C = s_B$$

$C = A*B$ implies $c_{ij} = \sum_{k=1}^{s_A} a_{ik}b_{kj}$ for all i, j such that $1 \leq i \leq r_A$, $1 \leq j \leq s_B$

Dot or Inner Product - Assuming that vectors are defined as column matrices, this instruction obtains the dot product (also known as the inner or scalar product) of two vectors. In addition, it may also be viewed as a matrix multiplication in which the transpose of the first matrix is post-multiplied by the second. For conformability both matrices must have the same number of rows. So that:

$\vec{a} \cdot \vec{b}$

$[A]^T[B]$

$$r_A = r_B$$

$$r_C = s_A$$

$$s_C = s_B$$

$C = A...B$ implies $c_{ij} = \sum_{k=1}^{r_A} a_{ki}b_{kj}$ for all i, j such that $1 \leq i \leq s_A$, $1 \leq j \leq s_B$

Multiplication by a Diagonalized Vector - This operation results in a matrix multiplication in which vectors are treated as diagonal matrices. There are four cases possible.

$\{a\}, \{b\}$ as $[A][B]$

C = A.D.B where both A and B are vectors.

Conformability requires the same number of rows in both A and B, so

$$s_A = s_B = 1$$

$$r_C = s_C = r_A = r_B$$

implies $c_{ii} = a_{i1}b_{i1}$ for all i such that $1 \leq i \leq r_A$

$c_{ij} = 0$, for $j \neq i$ and $1 \leq j \leq r_A$

C = A.D.B where A is a vector and B is a matrix.

$\{a\}, [B]$ as $[A][B]$

Conformability requires the same number of rows in both A and B. Thus,

$$s_A = 1$$

$$r_C = r_A = r_B$$

$$s_C = s_B$$

implies $c_{ij} = a_{i1}b_{ij}$ for all i, j such that $1 \leq i \leq r_A$, $1 \leq j \leq s_B$

C = A.D.B where A is a matrix and B is a vector.

$[A], \{b\}$ as $[A][B]$

Conformability requires the number of columns in the first matrix equals the number of rows in the second.

$$s_B = 1$$

$$r_C = r_A$$

$$s_C = s_A = r_B$$

implies $c_{ij} = a_{ij} b_{j1}$ for all i, j such that $1 \leq i \leq r_A$, $1 \leq j \leq s_A$

C = A.D.B where neither A nor B is a vector is treated as if C = A*B were written.

Cross Product - This operation forms the cross or vector product of two vectors. It is the least general of all the operations in that it not only requires that the row size of both vectors be the same, but also that the row size be two or three. Again, we will consider four cases.

C = A.X.B where both A and B are vectors. That is: $\vec{a} \times \vec{b}$

$$s_C = s_A = s_B = 1$$

and for the 3-dimensional case

$$r_C = r_A = r_B = 3$$

implies in vector notation $\vec{c} = \vec{a} \times \vec{b}$

or in terms of the elements

$$c_{11} = a_{21}\, b_{31} - a_{31}\, b_{21}$$

$$c_{21} = a_{31}\, b_{11} - a_{11}\, b_{31}$$

$$c_{31} = a_{11}\, b_{21} - a_{21}\, b_{11}$$

Now for the planar case

$$r_A = r_B = 2$$

$$r_C = 1 \quad \text{(that is, C is a scalar)}$$

with

$$c_{11} = a_{11}\, b_{21} - a_{21}\, b_{11}$$

C = A.X.B where A is a matrix and B is a vector. In this case, each column of the matrix A is envisioned as a separate vector. Pictorally:

$$A = \left[\vec{a}_1 \quad \vec{a}_2 \quad \vec{a}_3 \text{ - - - } \vec{a}_{s_A} \right]$$

Since $s_B = 1$, the operation consists of crossing each column of A with B separately and stacking the answers columnwise to form the matrix C. More precisely:

$$\vec{c}_j = \vec{a}_j \times \vec{b} \qquad \text{for all } j \text{ such that } 1 \leq j \leq s_A$$

with the arithmetic being identical to that used in the case above. The sizes for the 3-dimensional case are:

$$r_C = r_A = r_B = 3$$

$$s_B = 1$$

$$s_C = s_A$$

and for the planar case:

$$r_A = r_B = 2$$

$$r_C = 1 \qquad \text{(C becomes a row matrix)}$$

$$s_B = 1$$

$$s_C = s_A$$

C = A.X.B where A is a vector and B is a matrix. In this case, it is the matrix B which is considered to contain a vector in each column. Here we cross A with each column of B to obtain the columns of C, i.e.:

$$\vec{c}_j = \vec{a} \times \vec{b}_j \qquad \text{for all } j \text{ such that } 1 \leq j \leq s_B$$

The sizes are the same as given in the discussion of the previous case if s_A and s_B are interchanged. In fact, this case is equivalent to C = -B.X.A which was covered above.

C = A.X.B where both A and B are matrices. In order to make this operation meaningful, we shall apply a conformability restriction that the number of columns in A and B are the same. Then we can consider A and B as:

$$A = \left[\vec{a}_1 \quad \vec{a}_2 \quad - - - \quad \vec{a}_{s_A} \right]$$

$$B = \left[\vec{b}_1 \quad \vec{b}_2 \quad - - - \quad \vec{b}_{s_A} \right]$$

and form the cross product of corresponding vectors on a one-for-one basis. That is:

$$c_j = a_j \times b_j \qquad \text{for all } j \text{ such that } 1 \leq j \leq s_A$$

For the 3-dimensional case, the sizes are:

$$r_C = r_A = r_B = 3$$

$$s_C = s_A = s_B$$

while for the planar case:

$$r_A = r_B = 2$$

$$r_C = 1$$

$$s_C = s_A = s_B$$

4.7 Arithmetic Exceptions

The rules governing division and exponentiation when one of the numbers involved is zero will now be discussed. Let x be any non-zero real number, then:

the result of	$0/x$	is 0
the result of	$x/0$	is 0
the result of	$0/0$	is 0
the result of	0^x	is 0
the result of	x^0	is 1
the result of	0^0	is 1

that is, any number divided by zero is zero; and any number to the zeroth power is one. Only the second one of the above (x/0) is considered an error. If it should occur during the evaluation of an expression, a warning message is printed after evaluation has been completed.

When evaluating A...B with A containing complex numbers, the conjugate of the elements of A are used in the computation. This holds even if one (or both) of the matrices is a scalar. Hence, the dot operation can provide the true inner product of vectors in complex space.

4.8 Numerical Examples of Basic Expressions

Some numerical examples of the results of basic expression evaluation are in order. Let

$$S = 4$$

$$Q = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 0 & 5 \end{bmatrix},$$

$$VA = \begin{Bmatrix} 3 \\ 1 \\ 6 \end{Bmatrix}, \qquad VB = \begin{Bmatrix} 8 \\ 5 \\ 2 \end{Bmatrix}$$

$$T = \begin{bmatrix} 3 & 2 & 9 \\ 7 & 8 & -3 \\ 1 & 4 & 0 \end{bmatrix}$$

Then:

S + Q (or Q + S) results in

$$\begin{bmatrix} 5 & 6 & 8 \\ 7 & 4 & 9 \end{bmatrix}$$

S * Q (or Q * S, or Q...S, or S.D.Q , etc.)

$$\begin{bmatrix} 4 & 8 & 16 \\ 12 & 0 & 20 \end{bmatrix}$$

Q/S yields

$$\begin{bmatrix} 0.25 & 0.5 & 1 \\ 0.75 & 0 & 1.25 \end{bmatrix}$$

S/Q yields

$$\begin{bmatrix} 4 & 2 & 1 \\ 1.333 & 0 & 0.8 \end{bmatrix}$$

and a division by zero warning

S**Q yields

$$\begin{bmatrix} 4 & 16 & 256 \\ 64 & 1 & 1024 \end{bmatrix}$$

Q**S yields

$$\begin{bmatrix} 1 & 16 & 256 \\ 81 & 0 & 625 \end{bmatrix}$$

VA + VB (or VB + VA) yields

$$\begin{Bmatrix} 11 \\ 6 \\ 8 \end{Bmatrix}$$

VB - VA yields

$$\begin{Bmatrix} 5 \\ 4 \\ -4 \end{Bmatrix}$$

VA.*.VB (or VB.*.VA) results in

$$\begin{Bmatrix} 24 \\ 5 \\ 12 \end{Bmatrix}$$

VA.D.VB (or VB.D.VA) is

$$\begin{bmatrix} 24 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 12 \end{bmatrix}$$

VA/VB yields

$$\begin{Bmatrix} 0.375 \\ 0.2 \\ 3 \end{Bmatrix}$$

VA...VB yields

41

VA.X.VB yields

$$\begin{Bmatrix} -28 \\ 42 \\ 7 \end{Bmatrix}$$

Q*VA is

$$\begin{Bmatrix} 29 \\ 39 \end{Bmatrix}$$

Q.D.VA gives

$$\begin{bmatrix} 3 & 2 & 24 \\ 9 & 0 & 30 \end{bmatrix}$$

VA...T is

$$\begin{bmatrix} 22 & 38 & 24 \end{bmatrix}$$

T...VA is

$$\begin{Bmatrix} 22 \\ 38 \\ 24 \end{Bmatrix}$$

T*VA is

$$\begin{Bmatrix} 65 \\ 11 \\ 7 \end{Bmatrix}$$

T.*.T (or T**2) is

$$\begin{bmatrix} 9 & 4 & 81 \\ 49 & 64 & 9 \\ 1 & 16 & 0 \end{bmatrix}$$

T*T gives

$$\begin{bmatrix} 32 & 58 & 21 \\ 74 & 66 & 39 \\ 31 & 34 & -3 \end{bmatrix}$$

T...T is

$$\begin{bmatrix} 94 & 10 & 11 \\ 10 & 122 & 39 \\ 11 & 39 & 17 \end{bmatrix}$$

Q/Q is

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

Q**0 is

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

4.9 Arithmetic Statements

After spending so much time on the generation of arithmetic expressions, the reader will be glad to know we're finally going to put them to use. The first application will be the arithmetic statement. In its simplest form, this statement consists of a matrix name, an equal sign, and an arbitrary arithmetic expression in that order.

For example:

```
A = B
```

or

```
FORCE = -K*X + FAP
```

The statement is processed by evaluating the expression (which, of course, always boils down to a single matrix); then deleting the matrix on the left of the equal sign if it currently exists; and finally (re)defining the matrix on the left of the equal sign to be a matrix of the same size, shape, and type, and containing the numbers that result from the expression. In other words, the equal sign may be interpreted as meaning "the matrix on the left is replaced by the result of the expression on the right". Since the equal sign does not imply algebraic equivalence, statements of the form:

```
I = I + 1
```

or

```
Q = P*Q
```

are perfectly acceptable in MATE. Note that when the same name is used on both sides of the equal sign, the old original value is used in the evaluation of the expression; then it is deleted from memory and finally is replaced by the result of the entire expression. In this sequence, it is quite possible to change the size of a matrix, as will happen in our second example if the matrix P is neither a scalar nor a square matrix.

Remember: The matrix name goes on the left of the equal sign, the expression on the right.

4.10 Number Types

The reader with a great deal of previous programming experience may be wondering when we're going to discuss number types which can prove to be an annoyance in most other programming languages. The reasons for postponing this discussion till now are

simply that number types aren't critical in most of MATE work, and that the user shouldn't have to worry about the differences between 2 and 2.0 (which amounts to just a different bit pattern in the machine).

The elements of a matrix in MATE are numbers all of the same type. In the most complete version of MATE, the type may be:

1. INTEGER
2. REAL or single-precision floating point
3. DOUBLE PRECISION
4. COMPLEX
5. DPCOMPLEX - double precision complex

The type of numbers in the result of an arithmetic expression will be the same type as those in the quantity with the highest type number within the expression. For example, if an expression contains integer, double precision and complex quantities, the result of the expression will be a matrix of complex numbers. The sole exception to this rule is a basic expression which involves division or exponentiation of integers. In these cases, the results of the expression are REAL numbers. Therefore, the expression (5/8) has a result of 0.625 rather than zero which would be the result of this expression in FORTRAN.

The remaining sections in this chapter will show how matrix names themselves may be made variable (that is, changed during execution of the program), discuss the use of submatrices in expressions, and demonstrate updating a portion of a matrix.

4.11 Integer Scalar Expressions

Before we can proceed with the remaining options in arithmetic expression and statement writing, we'll need a general way to write integers. Since we've just developed a most general way of writing expressions, we will define an Equivalent

Integer Expression (EIE) by the following procedure:

1. Take an arbitrary MATE expression and evaluate it by the rules already given.
2. If the result is not a scalar, take the element in the first row and first column of the matrix and consider it as a scalar.
3. If this scalar is a complex number, discard its imaginary part and treat only its real part.
4. Set the sign positive.
5. Convert the number to an integer by rounding from the nearest half.

by this definition then, we can take an arbitrary MATE expression and reduce it to either a positive integer or zero. Steps 2 and 3 in the above procedure are included only to allow processing of an "absurd" condition since we will see that EIE's are used only in places where it would be the intent of the user to write integer scalar expressions. It should be noted that MATE always converts to an integer by rounding, whereas FORTRAN truncates.

We will not present any examples of EIE's since they look like any other arithmetic expression. It is the context in which an expression occurs which determines whether or not it is interpreted as an EIE.

4.12 Matrix Name Qualifiers

Suppose it is desired to perform a sequence of operations, say twenty times, to define twenty different matrices. Assuming we want access to all twenty matrices at the same time, it will be necessary to give each matrix a unique name. Now of course, we could code the operations twenty times each time using a different matrix name on the left of the equal sign, but it would probably be a lot easier to set-up some sort of loop to perform the operation the required number of times if we had a method of distinguishing the names of the produced matrices. Naturally, we have

such a method and we shall call it Array Name Qualifying.

We now expand the definition of a matrix name. A matrix or array name not only consists of the 1 to 6 alphabetic and/or numeric characters, but also may contain an appendage of one or two non-negative integers. These integers are called array name qualifiers. They are coded after the fixed (alphabetic) part of the name and enclosed in double apostrophes both fore and aft. If two qualifiers are used, they are separated by a comma. For example:

```
A''9''
Q27''3,5''
```

Note that the digits 2 and 7 in the last example are <u>not</u> array name qualifiers, but are a part of the fixed (alphabetic) array name. The numbers 3 and 5 constitute the qualifiers on the name Q27.

Now, if the qualifier numbers were restricted to constants, nothing would be added to the flexibility of the language. So we allow the qualifiers to be any Equivalent Integer Expressions (EIE) as defined in the preceding section. Then qualified array names could appear as:

```
A''3+J''
PSZ''K*(LEVEL-2*P),BODYNO''
```

Array names with qualifiers can be used anywhere an unqualified array name may be used. (The sole exception to this rule is given in Chapter 10.) However, if qualified array names are used in EIE's which are qualifiers themselves, the inner qualified names must be within parenthesis. For example, the MATE translator can <u>not</u> handle:

```
A''B''C''''          Illegal
```

the coding should be:

```
A''(B''C'')''
```

Other examples of array names qualified with other qualified array names are:

```
S''1,GR-(T''J''-T''J-1'')*P''
R''(S''(U''V'',W''X'')'')''
```

The above examples are not to be construed as typical, but merely legal.

When a qualifier number is zero, it is treated as if no qualifier had been given. That is, MATE makes no distinction between the names A, A''0'', and A''0,0''. However, MATE makes a great distinction between the names A, A''1'', and A''2''. The same distinction, in fact, it would make between the names D, E, and F. That is, the existence of matrix A''2'' does not imply there exists the matrix A''1''. Moreover, if both A''1'' and A''2'' do exist, they are not necessarily the same size or type.

So although the user may visualize array name qualifying as providing an "array of arrays" capability; the program takes the more pragmatic view that it has several arrays with the same name and so it needs the qualifiers to tell them apart.

4.13 Subarray Referencing

Often the user may want to reference a single element, or perhaps a row or column, or even a submatrix of a matrix. This reference could occur either to use the portion of the matrix in an expression or to redefine its contents. The procedure for subarray referencing consists of using EIE's to denote which row(s) and column(s) are to be taken. We shall call EIE's used for this purpose array limiters since they limit the region of interest in an array. (Although these limiters are identical

to matrix element subscripts, we won't use the word 'subscript' to avoid confusion with the array name qualifiers which may be thought of as matrix subscripts.) The array limiters are coded after the array name and are enclosed in parentheses and separated by commas. We will consider four cases of array limiting.

In the following discussions, let I, J, K, and L stand for arbitrary EIE's which have the values i, j, k and l, respectively. That is, i, j, k, and l are the integers I, J, K, and L reduce to.

4.13.1 Element Referencing

The element in the ith row and jth column of the array A is referenced by:

```
A(I,J)
```

Of course, if A is a matrix then A(I,J) is a scalar. Note that the row number is always given before the column number.

4.13.2 Row Referencing

The entire ith row of the array A may be referenced by any of the following forms:

```
      A(I)
      A(I,)
      A(I,0)
or    A(I,J)    with j = 0
```

If A is a rectangular matrix, then A(I) is a one row matrix, while if A is a vector, then A(I) is a scalar.

4.13.3 Column Referencing

The entire jth column of the array A can be referenced by:

```
      A( ,J)
      A(0,J)
or    A(I,J)    with i = 0
```

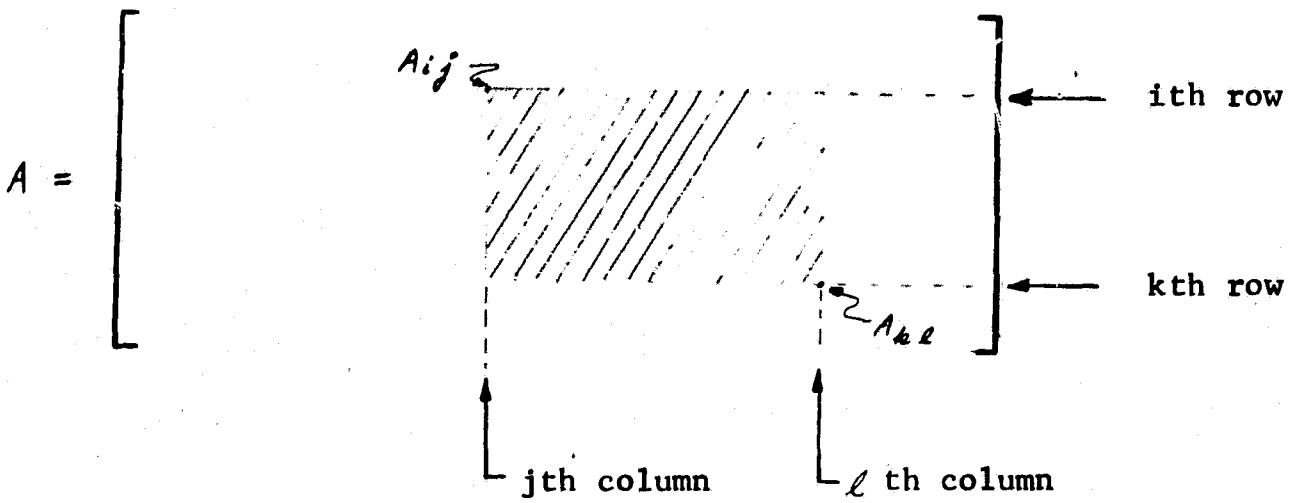
If A is a rectangular matrix, A(,J) is a vector.

4.13.4 Subarray Referencing

We reference a rectangular subarray of the array A which extends from the ith thru the kth rows and the jth thru the lth columns, inclusively, by:

A(I,J,K,L)

This specification references the submatrix shown shaded.



Thus, A(I,J,K,L) is equivalent to an (k-i+1) row by (ℓ-j+1) column matrix whose (1,1) term equals a_{ij}, (1,2) term is a_{ij+1}, etc.

For example, let D be the matrix:

$$D = \begin{bmatrix} 1 & 0 & 4 & 8 & 0 \\ 2 & 2 & 3 & 6 & 8 \\ 2 & 4 & 1 & 3 & 0 \\ 4 & 2 & 1 & 6 & 7 \\ 3 & 7 & 5 & 7 & 9 \end{bmatrix}$$

then:

$$D(1,1,3,3) = \begin{bmatrix} 1 & 0 & 4 \\ 2 & 2 & 3 \\ 2 & 4 & 1 \end{bmatrix}$$

$$D(2,3,4,5) = \begin{bmatrix} 3 & 6 & 8 \\ 1 & 3 & 0 \\ 1 & 6 & 7 \end{bmatrix}$$

$$D(5,1,5,4) = \begin{bmatrix} 3 & 7 & 5 & 7 \end{bmatrix}$$

$$D(1,1,5,2) = \begin{bmatrix} 1 & 0 \\ 2 & 2 \\ 2 & 4 \\ 4 & 2 \\ 3 & 7 \end{bmatrix}$$

furthermore:

both D(2) and D(2,1,2,5) = $\begin{bmatrix} 2 & 2 & 3 & 6 & 8 \end{bmatrix}$

both D(,2) and D(1,2,5,2) = $\begin{Bmatrix} 0 \\ 2 \\ 4 \\ 2 \\ 7 \end{Bmatrix}$

both D(5,3) and D(5,3,5,3) = 5

Again, the limiters may be arbitrary EIE's. For instance, to reference a 4 row by 7 column submatrix of the matrix PNB which is to start in the row and column numbers contained in the variables names F and G, respectively; we could code:

```
PNB (F,G,F+3,G+6)
```

So far we have implicitly assumed that the limiters are within the defined bounds of the matrix and that they are greater than zero except for the cases where two limiters are used, and it is desired to reference an entire row or column; in these cases one limiter was zero while the other was non-zero. We will now introduce some rather exotic concepts to give meaning to the otherwise ridiculous situation of referencing the zeroth row or an element outside of the array. And in the course of this development, we will simplify the use of subarray referencing.

The greatest simplifications in the use of subarray referencing occur when some of the limiters are given as zero. A zero limiter is a signal to MATE that it is to compute the value of the limiter to be used. Thus, the user is spared the time-consuming and error-prone job of writing correct and consistent limiters everywhere. There are three ways to code a zero limiter: (1) an arbitrary EIE which upon evaluation is found to have a value of zero; (2) the simplest zero EIE, namely the scalar constant zero; or (3) to omit the limiter entirely and merely imply its presence with the separating commas. Some examples of implied zero limiters for both two and four element limiter strings are given below:

A ( )	implies	A (0,0)
A (,)		A (0,0)
A (I)		A (I,0)
A (,I)		A (0,I)

A (I,J,)	implies	A (I,J,0,0)
A (I,J,,)		A (I,J,0,0)
A (,I,J)		A (0,I,J,0)
A (,I,J,)		A (0,I,J,0)
A (I,,J)		A (I,0,J,0)
A (,I,,J)		A (0,I,0,J)
A (,,I,J)		A (0,0,I,J)
A (,,)		A (0,0,0,0)
A (,,,)		A (0,0,0,0)
A (I,,)		A (I,0,0,0)
A (,I,)		A (0,I,0,0)
A (,,I)		A (0,0,I,0)
A (,,,I)		A (0,0,0,I)
A (I,J,,K)		A (I,J,0,K)
A (I,J,K)		A (I,J,K,0)

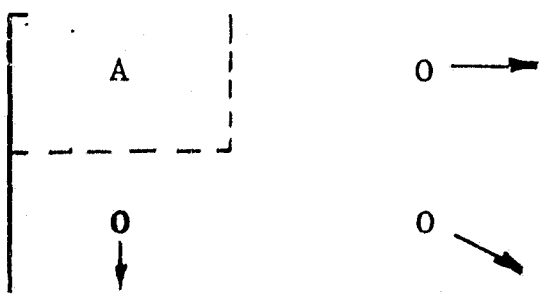
The first two examples in the list are eventually interpreted as a scalar from the first row and column of the matrix, i.e. A(1,1). The third and fourth examples reference the ith row and the ith column of A, respectively, as we have already seen. The interpretation of the zeros in the four element limiter strings will be seen to depend on the context in which the submatrix appears. Remember that in the four element limiter strings, the first two entries specify the row and column subscripts of the element which is to become the first element of the submatrix, while the last pair of limiters define the row and column subscripts of the element which will be the last number in the submatrix.

4.14 Subarray Referencing in Expressions

Subarray referencing in expressions will now be discussed. Since the earmark of a matrix used in an arithmetic expression is that the matrix has been previously defined, any submatrix of that matrix must certainly be defined. Or is it? What if from the 5 by 5 matrix named D, which was used in the numerical examples earlier in this section, the submatrix D (4,4,7,7) were referenced? We answer that question with the following definition:

When constructing a subarray of an existing array, as many additional rows and/or columns as required to satisfy the last two limiters may be appended to the existing array. These additional rows and columns contain all zeros.

Pictorally, then we can envision any array A as:



where the dotted lines denote the "defined" region of the array. Since this "expanded" array has no bounds, all subarrays defined by non-zero limiters exist. Therefore, returning to our numerical 5 by 5 matrix D, we have:

$$D\ (4,4,7,7) = \begin{bmatrix} 6 & 7 & 0 & 0 \\ 7 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and

$$D\ (2,6,3,8) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

while

$$D\ (27,36) = 0$$

and

$$D\ (6) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The next topic for consideration is the assumptions made when a limiter is given as zero for a matrix used in an expression. Let A be a 3 by 3 matrix whose elements we will denote by:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

and let B be the 5 by 4 matrix:

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \\ b_{51} & b_{52} & b_{53} & b_{54} \end{bmatrix}$$

Now ordinarily A and B could not be joined by any arithmetic operator since their sizes are such that they are not conformable for any operation. However, just because these operations are not defined in the classical sense, does not imply that it would not be desirable and necessary at times to join these matrices in an operation. To perform such operations, subarray referencing is required and the easiest way to accomplish conformability is to let MATE compute the limiters. Consider a matrix and a submatrix joined by an arithmetic operator, for instance:

```
H (1,3,6,) * W
```

What value should be used for the missing subscript? Why, the value which provides conformability for the operation, of course. In this case, it would be the number of rows in the matrix W plus two. This brings us to the first rule:

> Whenever possible, the conformability restrictions determine the values to be assumed for limiters which were given zero.

Some examples of this rule which use the A and B matrices above are:

A + B (2,2,)

This is interpreted as A + B (2,2,4,4) so that conformability for addition is maintained and we have:

$$A + B\ (2,2,) = \begin{bmatrix} a_{11}+b_{22} & a_{12}+b_{23} & a_{13}+b_{24} \\ a_{21}+b_{32} & a_{22}+b_{33} & a_{23}+b_{34} \\ a_{31}+b_{42} & a_{32}+b_{43} & a_{33}+b_{44} \end{bmatrix}$$

B(1,2,3,) . * . B(,4,5,4)

This is interpreted as B (1,2,3,2).*.B (3,4,5,4) since row size of 3 is set by the first submatrix and the column size of one by the second set of limiters. In terms of the elements, we have:

$$B(1,2,3,).*.B(,4,5,4) = \begin{Bmatrix} b_{12} & b_{34} \\ b_{22} & b_{44} \\ b_{32} & b_{54} \end{Bmatrix}$$

It is quite possible that there are no conformability restrictions to specify what is to be assumed for a limiter given as zero. When this occurs, MATE proceeds along the following lines:

> For the starting row or column subscripts, the value of one is usually assumed. The final subscripts are generally assumed to be the number of rows or columns in the matrix.

More precisely, we have for the second rule:

> In the absence of conformability restrictions, the values chosen for limiters given as zero are such that as many as possible of the defined rows and columns are included in the submatrix. However, if it becomes necessary to venture outside of the defined region of the array to define the subarray, append no more null rows and/or columns than required.

Perhaps this rule is best illustrated by an expression which consists of a single matrix with a limiter string. Using our 3 by 3 A matrix, then:

A(,2,,) is interpreted as A(1,2,3,3), i.e.

$$A(,2,,) = \begin{bmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix}$$

A(,,,) is interpreted as A(1,1,3,3) or simply as A.

A(7,2,) is interpreted as A(7,2,7,3). Since A would have to expand to have a seventh row, we have:

$$A(7,2,) = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

A(4,4,) is interpreted as A(4,4,4,4) which is a zero scalar. Because the first element is already in the expanded region and no conformability requirements exist, MATE assumes there is no reason to expand the submatrix further.

In general, the assignment of values to be used for zero limiters involves a blending of both the "keep conformable" and "maximum defined/minimum expansion" rules above. Again, using the 3 by 3 A matrix and the 5 by 4 B matrix, we have the examples:

A(,,,2)+B(,,,4) is interpreted as A(1,1,3,2) + B(1,3,3,4)

A(,,,)*B(,,,) is interpreted as A(1,1,3,3) * B(1,1,3,4)

A(2,3,,)-B(4,2,) is interpreted as A(2,3,3,3) - B(4,2,5,2)

A.X.B(2,4,) is interpreted as A.X.B(2,4,4,4)

In summary, the values assumed for array limiters which were given as zero or

omitted are such that (1) conformability is obtained and (2) the largest submatrix which contains only defined elements is incorporated.

4.15 Updating a Matrix

If you will remember earlier in this chapter, we introduced the arithmetic statement which took the form:

$$A = E$$

where A is a matrix name and E symbolizes an arbitrary arithmetic expression. These statements are processed by:

1. Evaluating the expression E
2. Deleting the matrix A from memory if it was previously defined
3. Defining A to be a matrix which is identical to the result of the evaluation of the expression E

Now, of course, the name A could be qualified with arbitrary EIE's, which we'll denote by I and J, to form arithmetic statements of the form:

```
A''I'' = E
A''I,J'' = E
```

Such statements are processed as outlined above and redefine the qualified name. That is, the execution of the statement A''I'' = E where I reduces to the number one has no effect on the matrices A, A''2'', A''3'', etc.; while the old matrix A''1'' is destroyed and redefined.

Suppose, however, that instead of (re)defining an entire matrix, we wish to change only a portion of a matrix. We accomplish this by using limiters with the name on the left of the equal sign. These limiters specify the region to be changed. For instance, again let A be a matrix name and E an arbitrary expression, and let

I, J, M, N be arbitrary equivalent integer expressions (EIE's). Then, an arithmetic statement of the form:

$$A(I,J,M,N) = E$$

is processed by:

1. Evaluating the expression E
2. Evaluating the EIE's I, J, M, and N to reduce them to integers which we will denote by i, j, m, and n. For now, we will assume all of these integers are greater than zero.
3. Examining the table of currently defined matrices for the matrix named A. If the matrix already exists, procede to step 4a; if not, jump to step 4b.

4a. Compare the size of the previously defined A matrix with m and n. If the number of rows and columns in the existing A matrix are greater than or equal to m and n respectively, procede to step 5. If not, append additional rows and/or columns of zeros to the matrix as required so that it becomes at least an m row by n column matrix. Then perform step 5.

4b. Define the matrix A to be an m row by n column matrix and set every element to zero. Then perform step 5.

5. Copy the elements from the result of the expression E into A starting with ith row and jth column to fill the region of A denoted by the limiters. Should the result of E be too small to fill the region, put zeros in the trailing rows and columns.

Therefore, an arithmetic statement of the form A(I,J,M,N) = E:

(1) Completely redefines the region specified by the limiters.

(2) Does not change any previously defined elements outside this region.

(3) May increase the size of a matrix.

As in the case of subarray referencing in expressions, MATE will assume values for any limiters which reduce to zero or are omitted. The computations in this case are greatly simplified however. The assumed values are chosen such that they are the minimum limiters which will allow the entire result of the expression to be copied into the matrix on the left of the equal sign. For instance, if the result of the expression is an r_E row by s_E column matrix, the missing limiters in the statement

```
A(I,J,,) = E
```

would be taken as $(i+r_E-1)$ and $(j+s_E-1)$, respectively.

In addition, a limiter string of two elements may be used to change a single element, row, or column of an array. Using the same nomenclature as above we would have:

to set the element in the ith row and jth column

```
A(I,J) = E
```

to redefine the ith row

```
A(I) = E
```

and to reset the jth column

```
A(,J) = E
```

CHAPTER 5.0 LOGICAL OPERATIONS, EXPRESSIONS, AND STATEMENTS

So far we have dealt only with matrices, that is arrays of numbers. But arrays of non-numeric items are also possible in MATE. One such non-numeric item, known as a logical variable, is the subject of discussion in this chapter.

In order to provide the user some incentive to read this chapter (other than "because it is there"), we note that the logical expressions we will discuss allow a most general way to compare quantities and evaluate even involved relationships in a concise and straightforward manner. The reader who is familiar with Fortran IV will already appreciate the power and convenience of the logical expression. In MATE, the logical expressions have been expanded to operate on an entire array and two new logical operations have been defined. The material in this chapter closely parallels that in the previous chapter, and the presentation here will assume familiarity with the concepts developed in the later sections of that chapter.

Logical variables are binary quantities. That is, there are only two possible values a logical variable can assume. We will denote these values by the symbols .TRUE. and .FALSE. . Since MATE is an array oriented program, we can have an array of logical variables in which each element of the array has either the value .TRUE. or the value .FALSE. . The naming rules for logical arrays are the same as those for matrices (reference Section 4.1).

Furthermore, array name qualifying and/or subarray referencing may be used with logical arrays names, just as they were used with matrix names. Note that the qualifiers and limiters themselves are still given as integer expressions (EIE's) and cannot contain logical variables. Consider, for instance:

```
D''10'' (2,2,3,4)
```

which we know is a 2 row by 3 column subarray of the array named D''10''. Now as before, we could have:

$$\text{D''10''}\ (2,2,3,4) = \begin{bmatrix} 2.1 & 3.5 & -7.9 \\ 4.2 & 5.1 & -6.3 \end{bmatrix}$$

in which case D''10'' (2,2,3,4) represents a submatrix; or perhaps:

$$\text{D''10''}\ (2,2,3,4) = \begin{bmatrix} .\text{TRUE.} & .\text{TRUE.} & .\text{FALSE.} \\ .\text{FALSE.} & .\text{TRUE.} & .\text{FALSE.} \end{bmatrix}$$

in which case D''10'' (2,2,3,4) is a logical subarray which implies every element of D''10'' must contain either the value .TRUE. or .FALSE. .

5.1 Relational Operators

One of the most useful methods of generating a logical variable involves the use of two numbers combined with a relational operator. The relational operator symbols and their meaning are:

.GT.	greater than
.GE.	greater than or equal to
.EQ.	equal to
.NE.	not equal to
.LE.	less than or equal to
.LT.	less than

Just as the arithmetic operators of the previous chapter took two matrices and performed arithmetic to generate a new matrix, these relational operators take two matrices and compare their elements to generate a logical array. For instance if:

$$A = \begin{bmatrix} 2 & 0 & -3 \\ 5 & 2 & 1 \end{bmatrix}$$

and the matrix B has been defined as:

$$B = \begin{bmatrix} 2 & -2 & -7 \\ 5 & 3 & -1 \end{bmatrix}$$

then the result of the expression A.GT.B is:

$$\begin{bmatrix} .FALSE. & .TRUE. & .TRUE. \\ .FALSE. & .FALSE. & .TRUE. \end{bmatrix}$$

while the expression A.LE.B would have the result:

$$\begin{bmatrix} .TRUE. & .FALSE. & .FALSE. \\ .TRUE. & .TRUE. & .FALSE. \end{bmatrix}$$

Now the quantities being compared by the relational operator are not restricted to matrix names, but may be constants or any arithmetic expressions. To wit:

```
A + 2.7*C...E.LT.$SQRT (F/G)
```

In the evaluation of such expressions, the results of the arithmetic expressions are computed first, then are compared via the relational operator. To emphasize this computing sequence, we could have written:

```
(A + 2.7*C...E).LT.($SQRT(F/G))
```

The reason for performing the arithmetic first becomes apparent when one considers the expression A+B.EQ.C (which is evaluated as if it were written (A+B).EQ.C). Note that if the "expression" were written A+(B.EQ.C), we would have nonsense since it implies that A is to be added to the logical array (B.EQ.C) and addition of logical quantities is not defined.

Since ordering of complex numbers is not defined, if either or both of matrices being joined by a relational operator are of the complex type, the only permissible operators are .EQ. and .NE. .

The conformability requirements and handling procedures for the relational operators are identical to those of the element-by-element arithmetic operators, i.e.

(+,-,.*.,/, and **). Therefore, an expression like (A.GT..5) which combines a matrix and a scalar would generate a logical array which is the same size as the matrix and contains the value .TRUE. wherever the corresponding element of A is greater than one-half. The assumptions made for limiters which are given as zero also follow exactly the same procedures used for the + arithmetic operator, and as such will not be presented here. The reader who is unsure of the conformability restrictions is advised to scan the appropriate sections (4.6 and 4.14) of the previous chapter mentally substituting the relational operators .GT. , .GE. , .NE. , .LE. , and .LT. for the operator +.

At the risk of belaboring a point, let us go over the difference between:

```
A = B
```

and

```
A.EQ.B
```

The top line is an arithmetic statement in which the computer is told to destroy the existing A matrix and redefine the A matrix to be a copy of the matrix B. The second line is a logical expression in which the matrices A and B are to be compared and the results of this comparison define an unnamed temporary logical array. Neither A nor B are changed by their use in the expression.

5.2 Logical Operators

We have just discussed the relational operators which compare two matrices to form logical arrays. We now introduce other operators, named logical operators, which combine the logical arrays to form other logical arrays. MATE contains five logical operators: the complementary operator .NOT. and the four connective operators .AND. , .OR. , .EQ. , and .NE. . To illustrate these operators, let A and B be logical variables or corresponding elements of logical arrays (as generated by a relational

operator, for instance). That is, both A and B have been previously defined and contain values of either .TRUE. or .FALSE. . Then the values of expressions which use logical operators are given below.

.NOT. A	This has the value .TRUE. only if A is .FALSE. ; it has the value .FALSE. only if A is .TRUE.
A .AND. B	This has the value .TRUE. only if A and B are both .TRUE. ; it has the value .FALSE. if either A or B is .FALSE.
A .OR. B	(Inclusive OR) This has the value .TRUE. if either A or B is .TRUE. ; it has the value .FALSE. only if both A and B are .FALSE.
(A) .EQ. (B)	This has the value .TRUE. if either both A and B are .TRUE. or if both A and B are .FALSE. ; it has the value .FALSE. if A and B differ
(A) .NE. (B)	(Exclusive OR) This has the value .TRUE. if A and B differ; it has the value .FALSE. if they are the same

A complete list of outcomes of two logical elements joined by one of the above is given in the Truth Table on the following page. Note that two logical operators may be adjacent only if the second one is .NOT. .

The logical operators have the same conformability restrictions and use the same handling procedures as the relational operators and the element-by-element arithmetic operators such as the + operator.

TRUTH TABLE

Let A and B be logical variables or elements of logical expressions; and denote .TRUE. and .FALSE. by T and F respectively.

If A has the value →	T	T	F	F
and B has the value →	T	F	T	F
then				
A .AND. B	T	F	F	F
.NOT.A.AND.B	F	F	T	F
A.AND..NOT.B	F	T	F	F
.NOT.A.AND..NOT.B	F	F	F	T
.NOT.(A.AND.B)	F	T	T	T
A.OR.B	T	T	T	F
.NOT.A.OR.B	T	F	T	T
A.OR..NOT.B	T	T	F	T
.NOT.A.OR..NOT.B	F	T	T	T
.NOT.(A.OR.B)	F	F	F	T
A.EQ.B	T	F	F	T
.NOT.A.EQ.B	F	T	T	F
A.EQ..NOT.B	F	T	T	F
.NOT.A.EQ..NOT.B	T	F	F	T
.NOT.(A.EQ.B)	F	T	T	F
A.NE.B	F	T	T	F
.NOT.A.NE.B	T	F	F	T
A.NE..NOT.B	T	F	F	T
.NOT.A.NE..NOT.B	F	T	T	F
.NOT.(A.NE.B)	T	F	F	T

Since the most common way to form the logical arrays to be used involves the relational operators, a typical logical expression might be:

A.GE.-10..AND.A.LE.-2

which is .TRUE. for each element of A (a_{ij}) such that:

$$-10 \leq a_{ij} \leq -2$$

As in arithmetic expressions, the liberal use of parentheses to control the sequencing of operations and to eliminate doubt as to 'what goes with what' is encouraged. Within a given level of parentheses, the order of computations is:

1. Function references
2. Arithmetic expressions
3. .GT. , .GE. , .EQ. , .NE. , .LE. , .LT.
4. .NOT.
5. .AND.
6. .OR.

In the event of more than one operator of the same rank within a level of parentheses, the processing occurs from left to right.

Note that the same symbols are used both for two relational and logical operators, namely .EQ. and .NE. . Now the MATE translator is unable to distinguish the sense in which these symbols are used, and since the relational operations must occur before the logical ones, the user must generally enclose the relational expressions within parentheses whenever these relational expressions are to be joined by the logical operators .EQ. and .NE. . Hopefully, this will become clearer after study of the following examples.

Some examples of logical expressions, in which A, B and C are matrices and L, M, and N are logical arrays, follow.

1. Test that A equals B within the tolerance C (where C is assumed to be a small positive number). This form is recommended whenever both A and B are not integer type matrices since round-off error may occasionally cause an erroneous result of the simpler form A.EQ.B.

```
B-C.LT.A.AND.A.LT.B+C
```

2. Write a logical expression that has the value .TRUE. whenever any of the corresponding elements in A, B, and C are less than or equal to zero.

```
A.LE.0.OR.B.LE.0.OR.C.LE.0
```

   This result could also be obtained from the expression:

```
.NOT.(A.GT.0.AND.B.GT.0.AND.C.GT.0)
```

3. Write an expression which is .TRUE. whenever any of the corresponding elements in L, M, and N are .TRUE.

```
L.OR.M.OR.N
```

4. Write an expression which is .TRUE. whenever the corresponding elements in L, M and N are all .TRUE.

```
L.AND.M.AND.N
```

5. Write an expression which is .TRUE. whenever either A or B elements or both are greater than the corresponding elements in C:

```
A.GT.C.OR.B.GT.C
```

Note that this expression may <u>not</u> be written:

```
A.OR.B.GT.C                    Illegal
```

since A is numeric and thus cannot be joined by a logical operator.

6. Write an expression which is .TRUE. when the elements of L are .TRUE. or when the corresponding elements of B are greater than those of C:

```
L.OR.B.GT.C
```

Note that is exactly the same form that was considered an error in the previous example. It is quite acceptable here since the .OR. combines two logical quantities namely L and (B.GT.C). Also, it is probably well to note here that the MATE translator would accept the erroneous expression in the preceeding example since it is of a legitimate form. The MATE executor would balk at the data type incompatibility when it went to perform the operation.

7. Write an expression which is .TRUE. when the corresponding elements of A, B and C equal each other:

```
A.EQ.B.AND.B.EQ.C
```

8. Write an expression which is .TRUE. when the corresponding elements of A, B and C either equal each other or are all different; .FALSE. if two of the three are the same.

```
A.EQ.B.AND.B.EQ.C.OR.A.NE.B.AND.B.NE.C.AND.A.NE.C
```

9. Write an expression which is .TRUE. whenever L is .TRUE. and the corresponding elements of B and C are equal, or when the element of L is .FALSE. and B and C are not equal:

```
L.EQ.(B.EQ.C)
```

Note the parentheses here are mandatory since without them the MATE translator would interpret the expression as ((L.EQ.B).EQ.C) and cause an error in execution when attempt is made to compare the logical variable from L against the number from B.

10. Write an expression which is .TRUE. whenever corresponding elements from L, M, and N are the same, i.e., either all .TRUE. or all .FALSE.

```
(L.EQ.M) .AND. (M.EQ.N)
```

Letting
```
L = A.EQ.B
M = B.EQ.C
N = A.EQ.C
```

implies we could write the solution of the example No. 8 above as:

```
(A.EQ.B).EQ.(B.EQ.C).AND.((B.EQ.C).EQ.(A.EQ.C))
```

where all parentheses used are required.

5.3 Logical Constants and Functions

A logical expression can consist of any of the following, either alone or in combination.

1. A logical array name
2. An expression which combines matrices via the relational operators, i.e. .GE. , .GT. , .EQ. , .NE. , .LE. , .LT.
3. An expression which uses logical arrays and the logical operators, i.e. .NOT. , .AND. , .OR. , .EQ. , .NE.
4. A logical constant
5. A logical function reference

We have already discussed the first three items. The logical constants are simply coded as .TRUE. and .FALSE. . A logical function is merely a function whose value is a logical array. The appearance of logical functions is identical to that of arithmetic functions (see Section 4.4). Their use also parallels that of arithmetic functions in that the function reference may either stand alone or be used as a logical array name in an expression. For instance:

```
$EXIST(A)
```

or

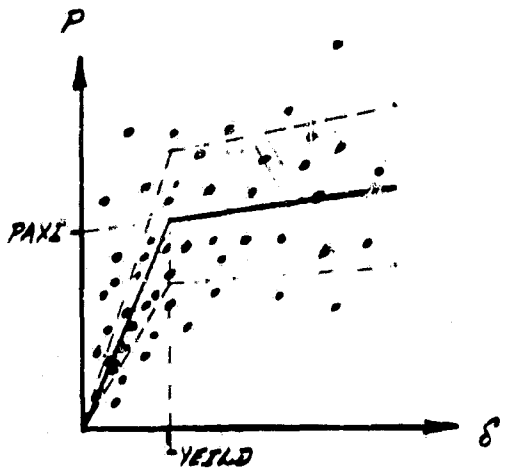
```
$EXIST(A).AND.CASENO.GT.0
```

5.4 Logical Statements

The processing of logical statements is again almost identical to that of arithmetic statements. A logical statement consists of an array name, either with or without qualifiers and limiters, an equal sign, and a logical expression in that order. If the name on the left of the equal sign is given with limiters and already contains some data, that data must be logical, since MATE cannot convert the .TRUE. and .FALSE. values in the expression into any other type of data. Also, if the

limiters require that the array on the left be enlarged, the additional rows and/or columns are filled with the value .FALSE. before the copying.

An example of a logical statement follows. Suppose we are given two large vectors P and DELTA which contain the results of force/deflection tests on a number of similar specimens. The corresponding rows of P and DELTA define one reading, thus their size is the same. It is desired to construct a logical array named GOODPT which will be the same size as P and contain the value .TRUE. in the rows which correspond to a point which is within ± 20% of the theoretical value. In this case, the theoretical value will be defined by two straight lines which have slopes of SLOPE1 and SLOPE2 and P axis intercepts of zero and PAXI, respectively. The second straight line with SLOPE2 is to be used for deflections greater than a deflection called YIELD.



The following logical statement defines the logical array GOODPT.

```
GOODPT  =  DELTA.LE.YIELD.AND.P.GE.0.8*
           SLOPE1*DELTA.AND.P.LE.1.2*SLOPE1
           *DELTA
           .OR.DELTA.GT.YIELD.AND.
           P.GE.0.8*(PAXI+SLOPE2*DELTA)
           .AND.P.LE.1.2*(PAXI+SLOPE2*DELTA)
```

A statement which could use the elements in GOODPT to control further processing is introduced in Section 8.5.

CHAPTER 6.0 ALPHANUMERIC CONSTANTS, VARIABLES, AND STATEMENTS

Another type of non-numeric data which can reside in a MATE array is alphanumeric or title data. Each element of such an array contains a string of printable characters. The names of these arrays must conform to the standard MATE naming rules given in Section 4.1. Again, array name qualifying and limiter strings are permitted. Note, however, that the subarray TITLE (1,1) consists of all the characters in the first element of the array named TITLE; not just the first character or the character in the first computer word of TITLE.

There are two forms of alphanumeric constants which can be coded as part of a MATE statement. In the first form, the alphanumeric information is enclosed in single apostrophes, e.g.:

'THIS IS A SAMPLE OF AN ALPHANUMERIC CONSTANT'

There are two important restrictions when using this form of the constant: the character string cannot contain an apostrophe nor can it be all blank.

To bypass these restrictions, we have the second form of an alphanumeric constant in which the number of characters (including any blanks) are counted, then the letter H is placed directly in front of the constant, and the H in turn is preceded by the character count, e.g.:

45HTHIS IS A SAMPLE OF AN ALPHANUMERIC CONSTANT.

Note that when counting the characters, the place immediately following the H contains the first character.

No operations are defined for alphanumeric arrays or constants. However, there is an alphanumeric statement of the form:

A = B

which allows copying of alphanumeric information; and where A is an array name while B is an array name or an alphanumeric constant. Both A and B may have array name qualifiers or limiter strings. But, if the statement is to update only a portion of an existing A array, the existing elements must also be alphanumeric. If an array is enlarged by this statement, any additional elements which are undefined are set all blank. For example, the statement:

```
HEADNG (2) = 'CMG PROBLEM - CASE 2'
```

redefines the element in the second row, first column of HEADNG so that it contains the given characters. If HEADNG exists before this statement, it must contain alphanumeric characters. If it was not defined before, the above statement will also set the first element all blank.

In addition to their use in alphanumeric statements, alphanumeric arrays and constants may also be used as arguments in routine and function references which are discussed in Chapter 9.

CHAPTER 7.0 CODING MATE STATEMENTS

The user's instruction list, i.e. the MATE statements, may be coded on the standard Fortran coding form. This has been chosen so that no special forms will be required. These cards are subject to the following rules:

1) A "C" in card column 1 indicates that the remainder of the card, columns 2-72, contains comments that are not to be translated. Every comment card must have a "C" in column 1. Comment cards may be placed before any statement.

2) MATE statements are punched within card columns 7-72. If the statement is completed on one card, column 6 must contain a zero or blank. If the statement is too long for one card, it may be continued within columns 7-72 of a continuation card. Each continuation card must have a character other than a blank or zero in column 6 and only blanks in columns 1-5. There may be as many as 5 consecutive continuation cards used to contain a single statement. Each MATE statement must start on a new card.

3) Any MATE statement (except the ROUTINE statement introduced in Chapter 10) may have a statement number. A statement number is entered in columns 1-5 of a card and must be an unsigned integer constant from 1 to 9999. Decimal points or decimal exponents are not allowed on a statement number. Within a given routine each statement number must be unique. Statement numbers need not be right justified, i.e. the units digit need not be in column 5.

4) Within columns 7-72 blanks may be placed wherever desired to improve readability, except a blank may not be imbedded within a name. The liberal use of blanks is recommended and indeed, unlike Fortran, there are places in which we shall require at least one blank. These places are denoted by a lower case 'b' with a slash thru it (ƀ).

5) Card columns 73 and beyond are ignored by the translator and may be used to contain deck identification and card sequence information.

6) Every punch on a card must be a member of our character set. The character set contains the 26 letters of the English alphabet, the numerals zero through nine, the blank, and the following special characters:

+	plus sign
-	minus sign
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
.	period or decimal point
,	comma
=	equal sign
$	dollar sign
'	apostrophe

An example of MATE coding is presented in Chapter 12.

CHAPTER 8.0 CONTROL STATEMENTS

The control statements enable the user to control the flow of processing in his program.

8.1 The Unconditional Transfer Statement

General Form:

GO TO n

where n is a statement number

ex: GO TO 25

This statement causes control to be transferred to the statement numbered 25.

8.2 The Computed GO TO Statement

General Form:

GO TO (n_1, n_2, ..., n_m), i

where n_1, n_2, ..., n_m are statement numbers

i is an equivalent integer expression (EIE).

This statement causes control to be transferred to the statement numbered n_i (that is, to n_1, n_2, ..., n_m depending on whether the value of i is 1, 2, ..., m, respectively, at the time of execution). If the statement number is omitted, control is transferred to the next sequential statement. If the value of i is not within the range (one to m), an error comment is made and control is transferred to the next sequential statement.

ex: GO TO (30, 75, 50, 9), I - 2

If I is 5 at the time of execution, a transfer to the third statement in the list, i.e. statement 50 will occur.

ex: GO TO (, 20, , 40) I

If I is 1, 3, or greater than 4, a transfer to the next statement of the list will occur. If I is 2 or 4, a transfer to statements 20 or 40, respectively, is made.

8.3 The Arithmetic IF Statement

General Form:

IF (e) n_1, n_2, n_3

where n_1, n_2, n_3 are statement numbers, e is an arithmetic expression.

This statement causes control to be transferred to the statement numbered n_1, n_2, or n_3 if the value of the first element in e is less than, equal to, or greater than zero, respectively. If e is COMPLEX, only the real part of e_{11} is considered.

ex: IF (A (J,K) - B * B) 10, 4, 30

Omitted statement numbers imply a transfer to the next sequential statement.

ex: IF (A), 30, 40

This statement causes control to be transferred to the next sequential statement if A is negative or to statement 30 or 40 if A is zero or greater than zero, respectively.

8.4 The STOP Statement

General Form:

STOP

This causes termination of execution

8.5 The Logical IF Statement

General Form:

IF (b) s

where b is a logical expression containing one element and s is a statement.

If the logical expression b is .TRUE. statement s is executed. Control is then transferred to the next sequential statement unless s is a transfer statement. If the logical expression b is .FALSE. statement s is ignored and control is transferred to the next sequential statement.

Restrictions: The statement s may not be one of the following:

IF
DO
ROUTINE
END

If a logical array is given in place of b, only the first element of the array is used in the test. An example of a logical IF statement follows.

```
IF (A.LT.LWRLMT.OR.A.GT.UPRLMT) A = 0
```

This statement sets A to zero if it is less than LWRLMT or if it is greater than UPRLMT.

8.6 DO Looping Statement

General Form:

DO n i = m_1, m_2, m_3

where n is a statement number of a statement which must follow the DO statement, i is a scalar, m_1, m_2, m_3 are arithmetic expressions.

Only the first element of these expressions are used. They may not be COMPLEX type. If m_3 is omitted, it is assumed to be plus or minus one depending on whether m_1 is less than or greater than m_2, respectively.

Define the range of a DO to be from the statement which immediately follows the DO down thru and including the statement with the statement number n.

In the case $m_1 \leq m_2$, execution proceeds as follows:

1) i is set equal to m_1
2) the range of the DO is executed
3) i is compared to m_2, if $i \geq m_2$ step 5 occurs, otherwise step 4
4) i is incremented by m_3 then step 2 occurs
5) control is transferred to the statement beyond the range of the DO, i.e. to the statement which immediately follows the statement numbered n.

Execution proceeds in the same manner for the case $m_1 > m_2$ except the sense of the test in step 3 is reversed.

Restrictions: A transfer into the range of DO from outside its range is forbidden. However, a transfer from inside the range to outside is permitted.

If the range of a DO contains another DO statement, the range of the inner DO must lie wholly within the range of the outer DO. The last statement in the range of a DO can <u>not</u> be one of the following:

```
DO
STOP
RETURN
END
ROUTINE
```

For example, the DO loop:

```
      DO 25 I = 1, J+1
      A (I) = I*C
   25 B''I'' = I**I
```

is equivalent to the following statements when the first element of J, i.e. j_{11} is such that $1 < j_{11} \leq 2$.

```
      A(1) = C
      B''1'' = 1
      A(2) = 2*C
      B''2'' = 4
      A(3) = 3*C
      B''3'' = 27
```

8.7 The CONTINUE Statement

General Form:

```
CONTINUE
```

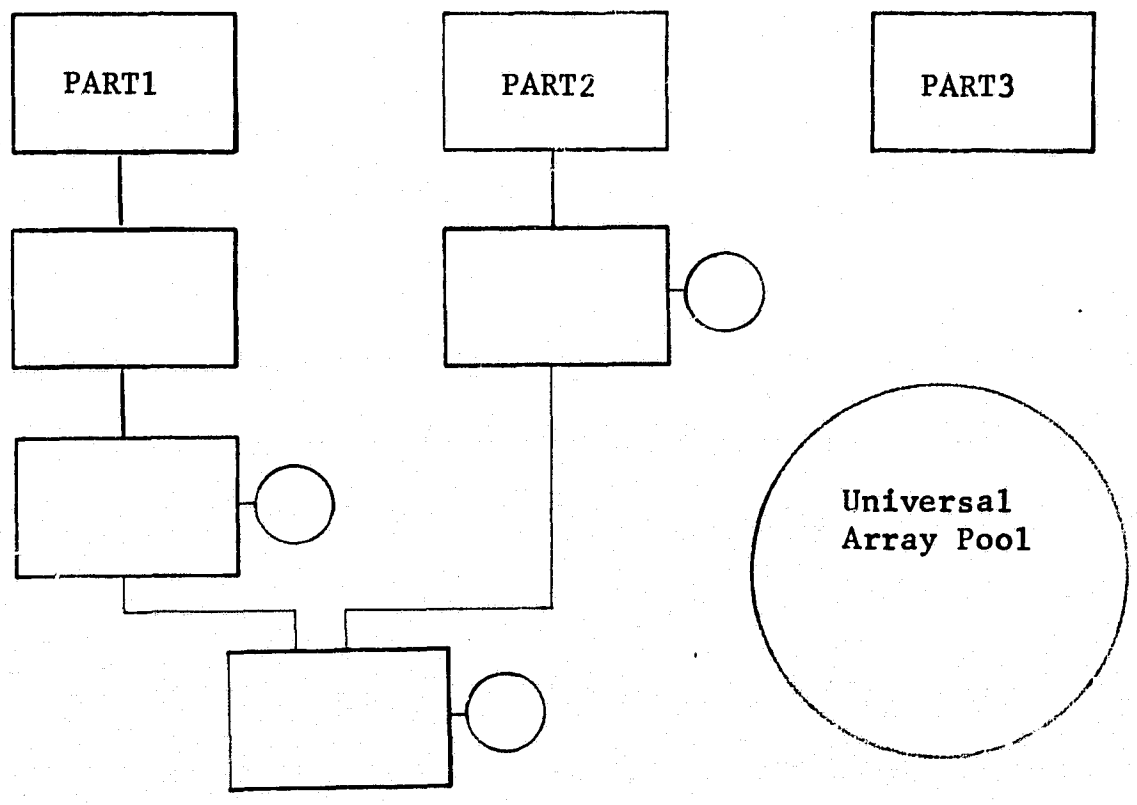
This statement does nothing. However, it can be useful as the last statement in a nested DO loop to comply with the nesting restrictions.

CHAPTER 9.0 ROUTINE REFERENCE STATEMENTS

MATE is a routine oriented language. By this, we mean that the user is strongly encouraged to modularize the coding of his instructions as much as possible; and in so doing, visualize the solution process as the successive execution of both special and general-purpose routines.

Some of the advantages of modularization are: that small routines are easier and less costly to check-out, correct, modify, and adopt to new applications than large ones; the use of routines often permits a sizable reduction in the coding effort required since a sequence of operations to be executed many times at various points in the solution process need only be coded once as a routine and not be rewritten every time it is required; and finally, some computer operating systems will allow routines which do not reference each other to share the same core space thus reducing the core storage necessary to contain the program.

Now, let us consider the structuring of routines in a moderately-sized problem such as shown in the chart below.



In this chart, we see that the solution has been broken-up into three major routines named PART1, PART2, and PART3. The remaining rectangles represent other MATE routines and functions which may have been written especially for the problem at hand or developed earlier for a previous solution and found to be applicable to the current problem.

The large circle represents the universal array pool. Every routine may use or define arrays in this pool by merely using the array name. That is, MATE assumes every array belongs in the universal array pool unless it is told otherwise. The small circles also represent array pools. These array pools are considered property of the routine to which they are attached. The arrays they contain are termed internal arrays. These internal arrays cannot be directly referenced by any other routine.

Finally, we turn our attention to the connecting lines in the chart. These lines symbolize routine or function references or in Fortran parlance "CALL's". The remainder of this chapter will be devoted to these routine references.

9.1 Routine Reference Statement

The simplest form of the routine reference statement occurs when there are no arguments to be transmitted to/from the called routine. In this case, the statement consists entirely of just the name of the routine to be executed.

In order to develop a meaningful example of such a routine reference, we will need a routine to reference; so we completely define a routine named SUMTRN by the statements:

```
ROUTINE SUMTRN
Q = A + 2 * B + C ** 2
P = Q...M * Q
END
```

We see that SUMTRN (re)defines the matrices P and Q using the numbers in the matrices A, B, C and M. The first and last statements, known as routine definition and END statements which will be discussed in the next chapter, may be ignored for now.

Where does the routine SUMTRN get the matrices A, B, C and M to perform its operations? From the universal array pool, of course. It also places its 'answers' P and Q in the universal array pool. Now since the routine which references SUMTRN is also using the universal array pool, it can define the matrices A, B, C and M for SUMTRN as required and can use the matrices P and Q directly. Therefore, the coding

```
      .
      .
      .
M = R + $SQRT(DELTA)
SUMTRN
S = Q + R*P
      .
      .
      .
```

is equivalent to the coding:

```
      .
      .
      .
M = R + $SQRT(DELTA)
Q = A + 2*B + C***2
P = Q ... M*Q
S = Q + R*P
      .
      .
      .
```

Although this particular example is rather trivial, it does demonstrate two important concepts. First, the automatic communication of data among routines allows routines which are thought too big to be readily broken up. Second, and most important, is the computing sequence. Note that whenever a routine is referenced

control passes to the first statement of the referenced routine; then marches thru that routine in the normal fashion; and upon completing that routine returns to the statement immediately following the statement which referenced the routine in the calling routine.

The automatic linkage of data thru the universal array pool does provide a convenience; however, it also has a major draw-back. Namely, that this form of data linkage does not lend itself to writing general-purpose routines. To elaborate on this point, let's reconsider our sample routine SUMTRN. Suppose we wished to execute SUMTRN in another application using the matrix D in place of C, or perhaps we don't want the 'answers' named P and Q since we have already defined a P or Q matrix and don't want it destroyed. Faced with these difficulties, we could either change SUMTRN or surround the reference to it with a flurry of arithmetic statements to save and restore the affected matrices under other names. Since neither of these solutions is desirable, we will introduce another method of data linkage between routines.

By using an argument string to transmit data to/from a routine, we can obtain a routine which is not dependent on the actual names used within it. Such a routine will usually find more applications than a routine which uses the universal array pool for communication, since only what it does is frozen, not what it does it to. The details of argument string data linkage will be deferred to the next chapter. The remainder of this chapter will be devoted to permissible forms of routine and function references.

A routine which requires, say, three arguments may be referenced by any of the following forms:

name $\not\beta$ a_1, a_2, a_3

or name, a_1, a_2, a_3

or name (a_1, a_2, a_3)

where 'name' is the name of the routine to be executed; 'a_1, a_2, a_3' are the arguments which we assume are matrix names for the time being; and '$\not b$' indicates at least one blank card column required. Some examples of routine reference statements are:

```
STIFGN K,GEOM,MAT,AREAS
A (B, C, A)
ASMBL, TA, TB
```

In these examples, STIFGN, A, and ASMBL are the names of the routines to be called. All other names, including the second occurrence of the name A, are array or matrix names. The different forms of this statement are interchangeable and which form is used is merely a matter of personal preference. Since the author prefers the first form, that form will be used in most subsequent examples.

The arguments, 'a_i' in the formal definition above, are not limited to bare matrix names. They may be any legal expression. For instance:

1. Any type of constant.
2. A matrix or array name with or without a qualifier field.
3. A submatrix or subarray, i.e. a matrix or array name which is given with limits. A qualifier field may also be appended.
4. A function reference.
5. Any arithmetic or logical expression built using the arithmetic, relational or logical operators to combine or complement any of the above.

In some instances, an argument may be considered optional and omitted from the argument string. We will call such omitted arguments <u>void</u> arguments. Void arguments are automatically implied at the end of an argument string. That is, in an argument string consisting of four expressions separated by commas, the fifth, sixth, etc. arguments are void arguments. Void arguments in the middle of an

argument string are denoted by consecutive commas. Note that if the first argument is to be void (a practice which is not recommended), two consecutive commas are required at the beginning of the argument string.

Some examples of routine reference statements with void arguments are given below.

Routine Reference Statement	Arguments Considered Void
ALPHA A,B,C	4,5,...
ALPHA ,A,B,C	4,5,...
ALPHA A,,C	2,4,5,...
ALPHA ,,B,C	1,4,5,...
ALPHA ,,,C	1,2,4,5,...
ALPHA	All

In addition to an expression or lack thereof, an argument may also be a routine name which the routine being referenced is to call during its execution. To distinguish between an array name and routine name argument, the routine name is enclosed in parenthesis and preceded by a dollar sign. For example:

$(EIG3)

where EIG3 is the name of a MATE routine or a Fortran subroutine.

Note that this discussion is directed toward what is possible to use as an argument, not what is correct to use for a particular application. As we shall see, the way the called routine uses a given argument will imply and apply restrictions on what is an acceptable form for the argument.

9.2 Function References

As we have seen in Chapters 4 and 5, a function is referenced by the general form:

$$\$ \text{ name } (a_1, a_2, \ldots, a_n)$$

where 'name' is the name of the function to be executed, 'a_i's are the arguments to the function, and both the preceding dollar sign and parentheses around the argument string are mandatory. The definition of possible arguments for functions are identical to those for routines just discussed.

CHAPTER 10.0 CREATION OF MATE ROUTINES

As we have said before, MATE is a routine oriented language. The only way to perform calculations in MATE is by executing routines, and conversely the only thing the user can code in this language is a MATE routine.

Let's return to the rough program organizational chart given in the last chapter. Along the top row of this chart are the major routines named PART1, PART2, and PART3. We call these routines major because of their position on the chart, not because they are necessarily large or do much. Ordinarily, these routines would direct the solution of a major step in the problem, and as such they are to be executed in a predetermined sequential fashion. Thus, to perform the problem's calculations, the MATE system calls PART1. Then after PART1 is completed, PART2 is executed, etc. The user may enter a list of the order in which he wishes the major routines to be executed. In the absence of such a list, the major routines are called in increasing order. At the present time, a job can contain a maximum of nine major routines which must have the names PART1, PART2, ..., PART9. The major routine names within a job need not be consecutive. That is, in our three part example, the major routines could have been named PART4, PART7, and PART9, respectively; and assuming no ordering list is given, the system will execute the major routine with the lowest number first, i.e. PART4, then PART7 and finally PART9. This discussion is somewhat academic since all problems can be, and a majority probably will be, coded using just one major routine.

10.1 Routine Definition Statement

The first statement is every segment of coding, i.e. routine, must be a routine definition statement. The simplest form of this statement is:

ROUTINE␣name

where "name" is the name given to the routine being defined and ␣ implies one or more blank card columns. The name must meet the standard specification for names as given in Section 4.1. Furthermore, the name cannot duplicate a name reserved for system use. The forbidden names are:

GO

GOTOnn where n is a blank or a digit

IF

CALL

STOP

RETURN

REAL

DOUBLE

END

This form of the routine definition statements is used to define a major routine, typically:

ROUTINE PART1

will be the first statement in a problem coding. This form is also used for all routines which use only the universal array pool for their data linkage.

To define a routine which uses an argument string for communication with the calling routine, we use the form:

ROUTINE␣name $(a_1,a_2,\ldots,a_n)$

where name defines the routine's entry name and is subject to the same restrictions given above and ␣ denotes a mandatory blank. The a_1, a_2, ..., a_n constitute the argument cross-reference list. Each of the a_i is an unmodified dummy array name. The name chosen for an a_i may be any legal array name that does not duplicate a

name chosen for another a_i in the argument cross-reference list. The name is unmodified in the sense that matrix name qualifiers may not be associated with the dummy array name in any statement throughout the routine. Moreover, the dummy array name may not appear with a limiter string in the ROUTINE statement itself, however, it may be used with a limiter string in the remainder of the routine. Finally, the name in the argument cross-reference list is referred to a dummy array name since the name is not an actual array name but rather symbolic of or a substitute of an actual array.

Routines defined with an argument list can not only directly reference arrays in the universal array pool, but also use indirect referencing via the argument cross-reference list. With the indirect referencing, each dummy array name, a_i, mentioned in the ROUTINE statement is replaced by the i^{th} argument given the routine reference statement which calls the routine.

Example 1:

Consider the routine reference statement

```
ALFGX (AQ, B ... TOP, 3*G,,27)
```

and the routine definition statement

```
ROUTINE ALFGX (B, C1, DOWN, M7, N)
```

When the routine reference statement is executed, control is transferred to the routine with entry name ALFGX. Within routine ALFGX, the following substitutions automatically take place:

Every occurrence of the name	is replaced by the matrix -
B	named AQ
C1	which contains the product (B ... TOP)
DOWN	which contains the product (3*G)

M7	which is void. (Note M7 may not be used in any arithmetic or logical statement for which execution will be attempted)
N	which contains the number 27 in its first and only element

Example 2:

Consider a routine which is to perform a similarity transformation involving non-unitary matrices, i.e. given a square A and non-singular B, form $B^{-1}AB$. Such a routine might be coded as:

```
ROUTINE SIMTRN (C,A,B)
C = $INVERS (B)*A*B
END
```

Then if in another routine the coding:

```
.
.
.
SIMTRN R,T,A
.
.
.
```

appears, it would be equivalent to coding:

```
.
.
.
R = $INVERS (A)*T*A
.
.
.
```

Now assume in the major or other routines of the job exist the following statements:

```
SIMTRN  DNEW, 32.2*DOLD,RX*RY*RZ
SIMTRN  A"3",A"3",BB"J"
SIMTRN  QNP7,TA+TB,S(1,1,$NOROWS(TA),$NOROWS(TA))
```

After the first of the above routine reference statements has been executed, the matrix DNEW has been defined as the transformed DOLD. More precisely, DNEW could be expressed in mathematical symbolism as $D_{new} = 32.2(R_x\ R_y\ R_z)^{-1} D_{old}\ R_x\ R_y\ R_z$. In similar fashion, execution of the second routine reference statement causes A_3 to be redefined as the result of transforming the original A_3 by the matrix BB_J. Finally, in the last example, the matrix QNP7 is set equal to the transformed sum (TA + TB) after it has been transformed by the upper left portion of the matrix S which is conformable with that sum.

10.2 Function Definition Statement

Any routine with arguments may be considered as a function as long as the routine produces only one "answer" and this "answer" array is the first argument of the routine. Thus, the routine SIMTRN given in the section above may be thought a function; and indeed, the routine reference statements in that example could be replaced by the following arithmetic statements:

```
R = $SIMTRN (T,A)
DNEW = $SIMTRN (32.2*DOLD, RX*RY*RZ)
A"3" = $SIMTRN (A"3",BB"J")
QNP7 = $SIMTRN (TA+TB, S(1,1,$NOROWS(TA),$NOROWS(TA)))
```

Note that when using the function form of routine referencing, the routine's first argument is <u>not</u> explicitly coded. The first argument is understood to be the value of the function. To emphasize this point consider the paltry routine:

```
ROUTINE PI (VALUE)
VALUE = 3.14159
END
```

which we could reference as a function by:

```
AREA = $PI*RADIUS**2
```

or as a routine by:

```
PI PI
AREA = PI*RADIUS**2
```

10.3 Summary of Restrictions on the Routine Definition Statement

Although the routine definition statement must be the first statement in every routine, it can and should be preceded by comment cards which describe what the routine does.

The routine definition statement is the only statement in MATE that cannot have a statement number. Card columns one thru six must be blank on the first card of a routine definition statement.

There can be only one routine definition statement within a routine.

Dummy argument names must be coded as simple array names without name qualifiers and/or limiter strings. Constants, expressions, etc. are forbidden.

As with the routine reference statement, several punctuation options exist for the routine definition statement. All options are listed below again with 'name' being the routine name, a_i a dummy array name, and "$\not b$" symbolizing a required blank.

ROUTINE$\not b$name

ROUTINE$\not b$name $(a_1,a_2,\ldots,a_n)$

ROUTINE,name

ROUTINE,name $(a_1,a_2,\ldots,a_n)$

ROUTINE(name)

ROUTINE(name,$a_1,a_2,\ldots,a_n$)

ROUTINE␣name,a_1,a_2,a_3,...,a_n

ROUTINE,name,a_1,a_2,a_3,...a_n

10.4 Individual Array Pool Definition

The INTERNAL statement specifies arrays which are to be placed in the routine's individual array pool.

The most common form of this statement is:

INTERNAL␣'ALL'

which specifies that all arrays used in the routine are to be stored in the routine's own data pool.

If only some of the arrays in the routine are to be in the local data pool, one of the following forms may be used. To specify which arrays are internal:

INTERNAL␣a_1,a_2,...a_n

or to specify which arrays are in the universal array pool:

INTERNAL␣'ALL BUT' a_1,a_2,...a_n

where a blank between the words ALL and BUT is optional.

In both of the previous examples, the a_i are unmodified array names. The name string can be enclosed in parenthesis if desired.

Since array name qualifiers cannot be used in an INTERNAL statement, MATE assumes that if the alphabetic (fixed) part of an array name appears in an INTERNAL statement, then all possible names with that alphabetic part are also mentioned in the INTERNAL statement.

If a dummy array name appears in an INTERNAL statement, it is an error since the dummy name is only symbolizing another array name which is from the calling routine and hence cannot be internal.

The INTERNAL statement may be placed anywhere within the routine.

Consider yet another sample routine:

```
ROUTINE SAMPLE (A,B,Y,Z)
INTERNAL 'ALL'
SUM = A+B
PROD = A*B
Y = PROD/SUM
Z = $ABS (SUM)
END
```

Since this routine contains an INTERNAL 'ALL' statement, all communication with it must be thru the argument string, furthermore, any array which is not a dummy array name (SUM and PROD in this case) is an internal array. Thus, if the universal array pool contains arrays named SUM and PROD, they are undisturbed by this routine (unless, of course, they are given as the third or fourth argument in a call to SAMPLE). The important point of all this is that a routine which contains an INTERNAL 'ALL' statement can be placed in any MATE job without concern about array name conflicts. Therefore, whenever writing a routine which you think may have future applications as well as the present one, it is well to consider transmitting the arrays thru an argument string and including an INTERNAL'ALL' statement.

Since INTERNAL arrays are by their nature temporary arrays, they are automatically deleted from memory upon completion of the routine. An exception to this are the qualified INTERNAL arrays which are retained unless and until they are specifically mentioned in a DELETE statement. The DELETE statement is defined in Chapter 12.

Data may be placed into INTERNAL arrays before the routine is executed by defining the INTERNAL array in a standard data block. Remember that an INTERNAL array name appears hyphenated on the array header card in the data block with the second half of the name being the routine name as defined by the ROUTINE statement. Also note that the array will be forgotten after the first use of the routine, unless it is defined with a qualified name with non-zero qualifiers.

10.5 Argument Restrictions

In the previous chapter, we listed the types of quantities that could be given as arguments to a routine when referencing it, but pointed out the routine being called would place restrictions on the arguments and determine what types were acceptable. To demonstrate this process, return to routine SAMPLE above. First, from the arithmetic operators we see that the arrays which A,B,Y and Z stand for must be matrices. Next, since A and B must be conformable for both addition and multiplication, we can conclude they are either square matrices of the same size or at least one of them is a scalar. Since A and B are used only on the right side of an equal sign, they can be any matrix expressions which meet the size criteria. On the other hand, since Y and Z are used on the left of an equal sign, the quantities they represent can be at most a qualified matrix name with a limiter string.

Now suppose it is desired to modify SAMPLE so that the last argument is optional, i.e. may become a void argument. Since a void argument cannot be used in an arithmetic statement, it will be necessary to test for the existence of Z before it is set. To perform this test, MATE contains the logical function GOTARG which requires as arguments the number of the argument in the string and the dummy array name associated with it. Thus, to allow the fourth argument to be void in routine SAMPLE, we could replace the card before the END statement with:

```
IF ($GOTARG(4,Z)) Z = $ABS(SUM)
```

With this change, routine SAMPLE can now be called with either four or three arguments depending on whether the absolute value of A+B is or is not desired, respectively.

Up to this point, we have implied that a routine definition statement can specify any number of dummy arguments, and indeed this is true. However, there is a catch. Suppose a void argument was inadvertently given to a routine which did not test for a void argument before using the dummy variable in an expression or statement. If the void argument was one of the first ten arguments, the MATE executor would recognize the error, comment on it, then if so directed, make an assumption and continue execution. But, if the void argument is not one of the first ten, the results are completely unpredictable. Therefore, when writing a routine which can accept more than ten arguments, the user should always test all arguments beyond the tenth before using them.

The dummy names in the routine definition statement can also be considered as routine names. The only proper argument to correspond to a dummy variable used this way is a routine name enclosed in parentheses and proceded by a dollar sign. An example of such a routine is:

```
ROUTINE COMPNT (LEG, RADIUS, TRIG, THETA)
LEG = RADIUS*$TRIG(THETA)
END
```

If this routine is referenced by the statement:

```
COMPNT X,R,$(COS),PHI
```

it would be equivalent to coding:

```
X = R*$COS(PHI)
```

while the reference:

```
COMPNT Y,R,$(SIN),PHI
```

would simulate:

```
Y = R*$SIN(PHI)
```

10.6 The RETURN Statement

The form of the RETURN statement is:

```
RETURN
```

The RETURN statement marks an end (usually premature) of processing within a routine. When a RETURN is encountered, the execution of the routine is considered complete; the arrays in the routines individual array pool (if any) are deleted; and control is transferred to the statement which immediately follows the routine reference statement which called the routine. In the case the routine was called via a function reference, control is transferred to a point provided by the system where evaluation of the expression continues. As many RETURN statements as required may be used in a routine.

10.7 The END Statement

The last card of each and every routine must contain an END statement. The form of the END statement is:

```
END
```

The primary purpose of the END card is to signal the end of the user's instruction list to the translator so that it may commence final processing. When the END statement is encountered during execution, it is treated as a RETURN statement. Conversely, a RETURN statement can be thought of as a GO TOn statement where n would be the statement number of the END statement.

CHAPTER 11.0 CALLING FORTRAN SUBROUTINES

MATE has the ability to directly reference most general-purpose Fortran subroutines. The MATE statement which accomplishes this has the general form:

CALL␣name (a_1, a_2, ..., a_n)

where

'name' is the name of the Fortran subroutine, ␣ denotes a required blank, and the a_i are arguments to the subroutine. If there are any arguments, they must be enclosed in parentheses. No punctuation options are defined for this statement.

The arguments a_i may consist of:

1. Any type of constant.
2. Any legal MATE expression which contains at least one arithmetic or logical operator.
3. A previously defined array name. This name may have array name qualifiers. However, at the present time, it may not have a limiter string. That is, a subarray cannot be used as an argument to a Fortran subroutine.
4. A name of a Fortran subroutine or function. This name must be enclosed in parentheses which are preceeded by a dollar sign, i.e. $(name).
5. A statement number for a nonstandard return. The statement number must be preceded by a dollar sign to differentiate from an integer constant, i.e. $123.

Note again that an array in a Fortran argument list must be previously defined since Fortran cannot set-up a MATE array but can only change the data in an existing array.

There are several other restrictions on both the Fortran subroutine and the arguments to it which we will now describe.

First, the subroutine must transfer all data thru the argument string. It cannot use Fortran COMMON to communicate with the calling routine. Since arguments to a Fortran subroutine must be in core, all array and expressions used in an argument string must fit into the limited core room available for data in MATE. The user must also be concerned with the number type of the data in a matrix or expression. It may be necessary to precede the call to the subroutine by a number of type definition statements to convert matrix number types to the type required by the subroutine. The type definition statements are described in the next chapter. Finally, the subroutine may not use a tape or other external storage device which has been assigned to the MATE system.

The Fortran DIMENSION information required by some subroutines is identical to the row size of the argument array.

To illustrate the use of a Fortran subroutine in MATE, suppose we have a small symmetric matrix named DYN and we desire the eigenvalues and eigenvectors of this matrix. To get them, we could use the GE-SD subroutine EIG1 whose (rather cryptic) documentation is presented in Figure 1.

From the write-up of EIG1, we see that it requires four matrix arguments: the matrix to be diagonalized (DYN), a matrix of the same size to hold the eigenvectors which we'll name PHI, and two vectors for temporary storage. Since all these matrices must be defined before we can CALL EIG1, we code the following.

```
C     GIVEN THE REAL SYMMETRIC MATRIX DYN

C     DEFINE PHI TO BE THE SAME SIZE AS DYN
      PHI = DYN

C     SET-UP TA AND TB AS VECTORS WITH ROW SIZE SAME AS DYN'S
      TA ($NOROWS(DYN)) = 0
      TB = TA

C     NOW CALL EIG1
      CALL EIG1 (DYN, PHI, $NOROWS(DYN), 0.0, TA, TB, $NOROWS(DYN),
     1$NOROWS(DYN))

C     NOW DIAGONAL TERMS OF DYN
C     CONTAIN THE EIGENVALUES AND
C     PHI CONTAINS THE EIGENVECTORS.
      .
      .
      .
```

Note that zero given for EPS had to be coded with a decimal point because EIG1 expected a REAL value.

FIGURE 1

AUXILIARY LIBRARY

EIG1 (605, 635, 7094)

Eigenvalues/Eigenvectors

PURPOSE

To find the eigenvalues and eigenvectors of a real symmetric matrix by the JACOBI-CORBATO method.

USAGE

The calling sequence is -

CALL EIG1(A, B, N, EPS, TEMP1, TEMP2, IDIMA, IDIMB)

1. A is the name of the array containing the elements of the matrix. There are two options -

   IDIMA .EQ. 1. This signifies that A is a single dimensioned variable in which the elements of the upper triangular portion of the matrix are stored contiguously and row-wise.
   The eigenvalues are stored in the first N cells of A.

   IDIMA .GT. 1. This signifies that A is a double dimensioned variable in which the elements of the upper triangular portion of the matrix are stored, i.e. A(IDIMA, J).
   The eigenvalues are stored in the principal diagonal elements of A.

2. B is the name of the array in which the eigenvectors are stored column-wise.

3. N is the order of the matrix.

4. EPS is the convergence criterion. The sum of the squares of the off diagonal elements will be less than EPS. If EPS = 0., 1.0E-12 will be used as EPS.

5. TEMP1 is the name of an array containing at least N cells that is used for internal storage.

6. TEMP2 is the name of an array containing at least N cells that is used for internal storage.

7. IDIMA - see above, 1.

8. IDIMB is the first dimension of the B array, i.e. B(IDIMB, K).

METHOD

See P.I.R. 5540-12, The JACOBI Method For Hermitian Matrices, by F. E. LILLEY.

CHAPTER 12.0 MISCELLANEOUS MATE STATEMENTS

12.1 Data Type Specification Statements

The internal computer representation of matrix data may be specified by the following statements.

REAL␣a_1, a_2, ..., a_n

INTEGER␣a_1, a_2, ..., a_n

DOUBLE␣PRECISION␣a_1, a_2, ..., a_n

or alternately

DOUBLE␣a_1, a_2, ..., a_n

where a_i is a matrix name which may be qualified; ␣ denotes a required blank; and n is at most ten.

If a matrix mentioned in a type specification statement exists when the statement is encountered, it is converted to the specified type if its type differs. If the matrix is not currently defined a flag is set so that when it is generated, its data will be of the proper type. These specifications will override any numeric type indicator given on an array header card. Any conversion to integer type data will utilize rounding from the nearest half.

12.2 The DELETE Statement

A statement of the form:

DELETE␣a_1, a_2, ..., a_n

where the nomenclature is as above, will cause the matrix or array named a_i to be deleted from memory. The space it occupied then becomes available for other data.

12.3 The READ Statement

A statement of the form:

READ

causes the next standard data block in the input file to be processed. If an unprocessed standard data block does not exist in the input file at the time this statement is executed, execution stops.

12.4 The REPORT Statement

A routine reference statement of the form:

REPORT$\not\beta$a, b_1, b_2, b_3

where a is an expression and the b_i are arrays or alphanumeric data (b_1 maybe a constant) and $\not\beta$ denotes a required blank. The arguments to this routine may involve an array name which can be qualified and/or limited.

This routine causes the expression a to be printed preceded by the title information in b_1. If b_2 and b_3 are alphanumeric vectors with twelve or less characters per element, their elements are used to label the rows and columns of the expression, respectively. Any of the b_i may be void arguments, i.e. omitted. An example of the REPORT reference statement is:

```
REPORT KFIX, 'STIFFNESS MATRIX WITH RESTRAINTS', PTIDS, PTIDS
```

CHAPTER 13.0 A SAMPLE PROBLEM

In this chapter, we will develop the coding for a sample problem, elaborate on it, and hopefully clear-up any remaining questions about the MATE language.

The sample problem chosen for discussion consists of an arbitrary system of point masses. We wish to find the total mass of the system, the location of the center of mass, and the principal mass moments of inertia and their orientation.

The coding of this problem utilizes the standard MATE functions $NOROWS, $NOCOLS, and $IDENT which are explained in Appendix A. It also uses the "advanced" material presented in the later sections of Chapter 4 and in Chapter 11. However, most of this presentation will probably be decipherable without prior knowledge of that material.

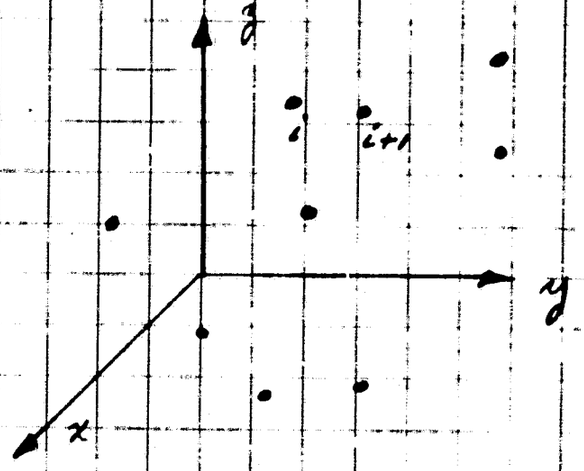
BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

SAMPLE PROBLEM TO DEMONSTRATE MATE CODING

GIVEN:

A MATRIX OF THE COORDINATES OF m POINTS NAMED "COORD", e.g.

$$COORD = \begin{bmatrix} x_1 & x_2 & \cdots & x_i & \cdots & x_m \\ y_1 & y_2 & \cdots & y_i & \cdots & y_m \\ z_1 & z_2 & \cdots & z_i & \cdots & z_m \end{bmatrix}$$

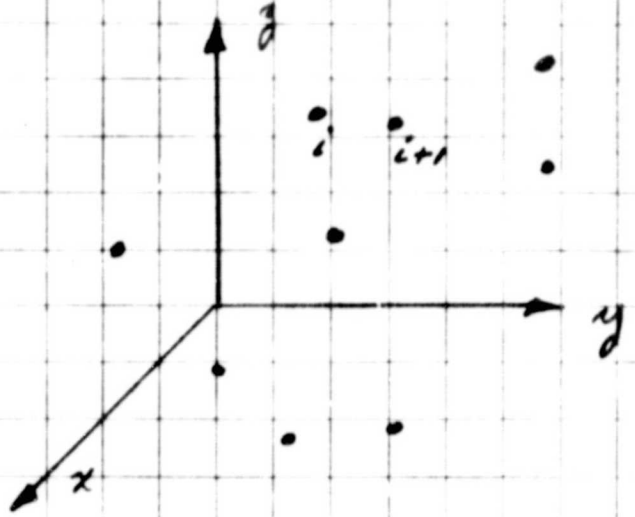


AND A VECTOR CONTAINING THE MASS ASSOCIATED WITH EACH POINT,

$$MASS = \begin{Bmatrix} m_1 \\ m_2 \\ \vdots \\ m_i \\ \vdots \\ m_m \end{Bmatrix}$$

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

SAMPLE PROBLEM TO DEMONSTRATE MATE CODING



GIVEN:

A MATRIX OF THE COORDINATES OF m POINTS NAMED "COORD", e.g.

$$COORD = \begin{bmatrix} x_1 & x_2 & \cdots & x_i & \cdots & x_m \\ y_1 & y_2 & \cdots & y_i & \cdots & y_m \\ z_1 & z_2 & \cdots & z_i & \cdots & z_m \end{bmatrix}$$

AND A VECTOR CONTAINING THE MASS ASSOCIATED WITH EACH POINT,

$$MASS = \begin{Bmatrix} m_1 \\ m_2 \\ \vdots \\ m_i \\ \vdots \\ m_m \end{Bmatrix}$$

FORM 1-8136 B (9-58)

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

TASK 1. FIND TOTAL MASS OF THE SYSTEM

ANALYSIS: WE WANT THE SUM OF THE ELEMENTS OF THE VECTOR MASS. WE COULD GET THIS BY DIRECT SUMMING OR BY EVALUATING THE MATRIX PRODUCT

$$\begin{bmatrix} 1 & 1 & \cdots & 1 & \cdots & 1 \end{bmatrix} \begin{Bmatrix} m_1 \\ m_2 \\ \vdots \\ m_i \\ \vdots \\ m_n \end{Bmatrix}$$

CODING: APPROACH 1 - DIRECT SUMMING

```
      MTOTAL = 0
      DO 10  I = 1, $NOROWS (MASS)
10    MTOTAL = MTOTAL + MASS (I)
```

CODING: APPROACH 2 - DOT PRODUCT

```
      MTOTAL = (MASS/MASS) ... MASS
```

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

TASK 2. FIND THE CENTER OF MASS

ANALYSIS: THE MATRIX PRODUCT OF THE "COORD" AND "MASS" MATRICES GIVES THE UNBALANCE OF THE SYSTEM ABOUT THE x, y, z AXES.

DIVIDING THE UNBALANCE BY THE TOTAL MASS GIVES THE ARMS.

CODING:

```
OFFSET = (COORD * MASS)/MTOTAL
```

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

TASK 3. GET THE COORDINATES OF THE POINTS WITH RESPECT TO THE CENTER OF MASS

ANALYSIS : SUBTRACT "OFFSET" FROM EACH COLUMN OF "COORD". AGAIN WE COULD DO THIS DIRECTLY OR BY

$$[TCOORD] = [COORD] - \{OFFSET\}[1\,1\,1\cdots 1]$$

CODING : APPROACH 1 - DIRECT SUBTRACTION

```
      DO  20  I=1, $NOCOLS(COORD)
20    TCOORD(,I) = COORD(,I) - OFFSET
```

CODING : APPROACH 2 - MATRIX SUBTRACTION

```
      TCOORD = COORD - OFFSET * COORD(1)**0
```

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

TASK 4. FIND THE MASS MOMENTS OF INERTIA ABOUT THE CENTER OF MASS WITH RESPECT TO THE TRANSLATED x, y, z AXES.

ANALYSIS: THE INERTIA MATRIX DUE TO THE ith MASS IS

$$INERTA_i = m_i \begin{bmatrix} y_i^2 + z_i^2 & -x_i y_i & -x_i z_i \\ -x_i y_i & x_i^2 + z_i^2 & -y_i z_i \\ -x_i z_i & -y_i z_i & x_i^2 + y_i^2 \end{bmatrix}$$

WHERE m_i = MASS OF ith POINT

x_i, y_i, z_i = COORDINATES OF ith POINT WITH RESPECT TO CG

WE COULD FORM THE ABOVE ELEMENTS DIRECTLY BUT LETS LOOK FOR AN EASIER WAY.

THE INERTIA MATRIX IS DEFINED AS THAT WHICH RELATES ANGULAR VELOCITY TO ANGULAR MOMENTUM. i.e.

$$\{\vec{L}\} = [I]\{\vec{\omega}\}$$

WHERE $\vec{L}$ = ANGULAR MOMENTUM VECTOR

$\vec{\omega}$ = ANGULAR VELOCITY VECTOR

I = INERTIA MATRIX

SO SET THE ith ELEMENT OF $\vec{\omega}$ TO ONE AND COMPUTE ANGULAR MOMENTUM TO GET ith COL. OF $[I]$

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

NOW THE ANGULAR MOMENTUM OF THE ith PARTICLE IS

$$\vec{L}_i = \vec{r}_i \times m_i \vec{v}_i$$

$$= \vec{r}_i \times m_i (\vec{\omega} \times \vec{r}_i)$$

$$= m_i (\vec{r}_i \times (\vec{\omega} \times \vec{r}_i))$$

WHERE $\vec{r}_i = \begin{Bmatrix} x_i \\ y_i \\ z_i \end{Bmatrix}$

$$\omega = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix}$$

CODING:

```
      INERTA(3,3) = 0
      DO  30   I = 1,$NOCOLS(COORD)
30    INERTA = INERTA + TCOORD(,I) .X.
     1                 ($IDENT(3) .X. TCOORD(,I)) * MASS(I)
```

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

TASK 5. FIND PRINCIPLE MASS MOMENTS OF INERTIA AND THE PRINCIPLE AXES

ANALYSIS: WE RECOGNIZE THIS AS AN EIGENVALUE PROBLEM WHERE WE REDUCE THE "INERTA" MATRIX TO DIAGONAL FORM. i.e. FIND "PRNCI" AND "DC" SUCH THAT

$$[INERTA]\,[DC] = [DC]\,[PRNCI]$$

WHERE:

PRNCI = PRINCIPAL INERTIAS

DC = A MATRIX OF DIRECTION COSINES WHICH RELATES THE SYSTEMS BY

$$\{r_o\} = [DC]\{r_p\}$$

CODING: USE GE-3D LIBRARY FORTRAN SUBROUTINE 'EIGI'

```
PRINCI = INERTA
DC(3,3) = 0
TEMP1(3) = 0
TEMP2(3) = 0
CALL EIGI (PRINCI, DC, 3, 0., TEMP1, TEMP2,
1                  3,3)
```

DOCUMENTION OF SUBROUTINE EIGI IS GIVEN IN CHAPTER 11.

BY CK. DATE REV.	GENERAL ELECTRIC	PAGE MODEL REPORT

TASK 5. (CONT.)

ALTHO THE DIAGONAL ELEMENTS OF "PRINCI" HAVE BEEN CORRECTLY SET, THE OFF-DIAGONAL TERMS CONTAIN SPURIOUS NUMBERS WHICH WE NOW SET TO ZERO BY:

```
PRINCI = PRINCI .*. $IDENT(3)
```

Now that the necessary equations have been developed, we have little more to do to obtain a working MATE program. The main subject we have not yet covered is input/output. The input consists of the matrices "COORD" and "MASS" which will constitute the initial data block. No instructions are required to process this data block. As for output, MATE will automatically print every array which appears on the left of an equal sign as it is (re)defined, under an Executor option known as the "check-out mode". The current Executor options are presented in Appendix C. We will use this option since every matrix we might wish to see appears on the left of an equal sign, except matrix "DC". We could use the REPORT statement of Section 12.4 to print "DC" or the arithmetic statement DC = DC.

Then to compose the MATE program we: code a ROUTINE PART1 statement, copy the equations we've developed (with some minor changes for aesthetic purposes), and code an END statement. The finished program appears on the following page. It is to be emphasized that this represents the <u>complete</u> coding of the problem.

IBM

FORTRAN Coding Form

X28-7327-6 U/M 025
Printed in U.S.A.

PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC: 0 (LETTER O) Ø | PAGE 1 OF 2
PROGRAMMER | DATE | PUNCH: Ø (ZERO) | CARD ELECTRO NUMBER*

COMM	STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
C			SAMPLE PROBLEM CODING - PROPERTIES OF POINT MASS SYSTEM	SAMPLØØØØ
C			PROGRAM REQUIRES THE FOLLOWING MATRICES AT THE START -	ØØ5
C			COORD - A 3 BY N WHOSE ROWS ARE THE X, Y, Z COORDINATES OF	ØØ6
C			THE POINTS.	ØØ7
C			MASS - AN N ROW VECTOR CONTAINING THE MASSES ASSIGNED TO THE	Ø1Ø
C			POINTS DESCRIBED IN COORD.	Ø11
C				Ø3Ø
			ROUTINE PART1	Ø5Ø
C				1ØØ
C	**		FIND THE TOTAL MASS AND THE CENTER OF MASS	11Ø
			MTOTAL = (MASS/MASS)...MASS	12Ø
			OFFSET = COORD * MASS/MTOTAL	13Ø
C				15Ø
C	**		SHIFT ORIGIN TO CENTER OF MASS	16Ø
			TCOORD=COORD-OFFSET*COORD(1)**Ø	17Ø
C				2ØØ
C	**		DEFINE W AS 3X3 IDENTITY MATRIX. RATHER THAN W=$IDENT(3) LETS USE	21Ø
			IDENT W,3	22Ø
C				25Ø
C	**		DEVELOP THE SYSTEM INERTIA MATRIX	26Ø
			INERTA=Ø*W	27Ø
C				3ØØ
			DO 3Ø I=1, $NOCOLS(COORD)	32Ø
	3Ø		INERTA=INERTA + MASS(I)*TCOORD(,I).X.(W.X.TCOORD(,I))	SAMPL34Ø

*A standard card form, IBM electro 888157, is available for punching statements from this form

IBM FORTRAN Coding Form

K28-7327-1 U/M 025
Printed in U.S.A.

PROGRAM | PUNCHING INSTRUCTIONS | GRAPHIC | PAGE 2 OF 2
PROGRAMMER | DATE | PUNCH | CARD ELECTRO NUMBER*

COM.	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
C				SAMPL400
C	**		GET PRINCIPAL MASS MOMENTS AND THEIR ORIENTATION	410
C			FIRST DEFINE REQUIRED ARRAYS	420
			PRINCI=INERTA	450
			DC=W	460
			TEMP1(3)=0	470
			TEMP2(3)=0	480
C			THEN CALL FORTRAN SUBROUTINE EIG1	500
			CALL EIG1 (PRINCI, DC, 3, 0.0, TEMP1, TEMP2, 3, 3)	520
C			ZERO OFF-DIAGONAL TERMS OF PRINCI	550
			PRINCI= W.*.PRINCI	560
C				600
C	**		USE DC ON LEFT OF EQUAL SIGN SO IT CAN BE AUTOMATICALLY PRINTED	610
			DC=DC	620
C				700
C	**		THAT'S ALL	710
			END	SAMPL720

*A standard card form, IBM electro 888157, is available for punching statements from this form

If the model we wished to run contained five points located such that:

$$\text{COORD} = \begin{bmatrix} 0. & 10.1 & 27.3 & 207.4 & 207.4 \\ 12.5 & -2.7 & -9.7 & 96.3 & 96.3 \\ 17.3 & 18.6 & 6.5 & 207.4 & -207.4 \end{bmatrix}$$

and having masses of:

$$\text{MASS} = \begin{Bmatrix} 0.031347 \\ 0.018912 \\ 0.011917 \\ 0.00090674 \\ 0.00090674 \end{Bmatrix}$$

the card images for the initial data block might be:

```
MASS 5,1  /  MASS VECTOR
3.1347-2, 1.8912-2, 1.1917-2,
2*9.0674-4
   $ END OF MASSES
COORD 3,5  / COORDINATES - INCHES
0.  12.5  17.3 / PT.1
10.1  -2.7  18.6 / PT.2
27.3  -9.7  6.5 / PT.3
207.4  96.3   207.4 / PT.4
207.4  96.3  207.4 $ PT.5
COMPUTE
```

13.1 The Sample Problem as a General-Purpose Routine

The previous coding has solved the sample problem as it was defined. Suppose however, that we thought the equations should be written as a general-purpose routine so that they could be readily incorporated into future programs. To accomplish this, we introduce an argument string to transfer the data to/from the routine, mark any temporary arrays as INTERNAL and choose the name MASINR as the routine name to obtain the coding shown on the next page.

IBM

FORTRAN Coding Form

X25-7327-6 U M 025
Printed in U.S.A.

PROGRAM | DATE | PUNCHING INSTRUCTIONS | GRAPHIC PUNCH | CARD ELECTRO NUMBER | PAGE 1 OF 2

PROGRAMMER

COMMENT	STATEMENT NUMBER	CONTINUATION	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
C			ROUTINE TO COMPUTE MASS PROPERTIES OF POINT MASS SYSTEM	MASIN000
			ROUTINE MASINR (COORD,MASS,MTOTAL,OFFSET,PRINCI,DC)	010
C			WHERE COORD - A 3 BY N WHOSE ROWS ARE THE X,Y,Z COORDINATES	020
C			OF THE POINTS - GIVEN	021
C			MASS - AN N ROW VECTOR OF THE POINT MASSES - GIVEN	025
C			MTOTAL- SCALAR EQUAL TO TOTAL SYSTEM MASS - RETURNED	030
C			OFFSET- A VECTOR FROM THE ORIGIN TO CENTER OF MASS	035
C			- RETURNED	036
C			PRINCI- 3 BY 3 DIAGONAL MATRIX OF PRINCIPAL INERTIAS	040
C			- RETURNED	041
C			DC - 3 BY 3 DIRECTION COSINE MATRIX WHICH RELATES	045
C			PRINCIPAL AXES TO ORIGINAL AXES - RETURNED	046
C				100
C	**		FIND THE TOTAL MASS AND THE CENTER OF MASS	110
			MTOTAL = MASS/MASS...MASS	120
			OFFSET = COORD*MASS/MTOTAL	130
			TCOORD = COORD-OFFSET*COORD(1)**0	150
C	**		DEFINE W AS 3X3 IDENTITY MATRIX	160
			W=$IDENT(3)	170
C				200
C	**		DEVELOP THE SYSTEM INERTIA MATRIX	210
			INERTA=0*W	220
			DO 30 J=1,$NOCOLS(COORD)	250
	30		INERTA=INERTA+MASS(I)*TCOORD(,J) .X. (W .X. TCOORD(,J))	MASIN260

*A standard card form, IBM electro 888157, is available for punching statements from this form

IBM

FORTRAN Coding Form

X28-7327-6 U/M 025
Printed in U.S.A.

PROGRAM | PROGRAMMER | DATE | PUNCHING INSTRUCTIONS | GRAPHIC | PUNCH | PAGE 2 OF 2 | CARD ELECTRO NUMBER*

COMM.	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
C				
C	**		GET PRINCIPAL MASS MOMENTS AND THEIR ORIENTATION	MASIN 300
C			FIRST DEFINE REQUIRED ARRAYS	310
			PRINCI=INERTA	320
			DC=W	350
			TEMP1(3)=0	360
			TEMP2=TEMP1	370
C			THEN CALL EIG1	380
			CALL EIG1 (PRINCI,DC,3,0.,TEMP1,TEMP2,3,3)	400
C			ZERO OFF-DIAGONAL TERMS	410
			PRINCI=PRINCI.*.W	420
C				430
C	**		PRINT DC	500
			REPORT DC	510
C				520
C	**		LET W,INERTA,TEMP1,TEMP2 BE TEMPORARY ARRAYS	600
			INTERNAL 'ALL'	610
C				620
			END	700
				MASIN 720

*A standard card form, IBM electro 888157, is available for punching statements from this form

With routine MASINR available, the coding for the original sample problem reduces to:

```
ROUTINE PART01
MASINR COORD, WEIGHT/G, MTOT, CGVEC, INERTA, DIRCOS
END
```

where we have decided to input weights and the acceleration due to gravity rather than the masses. The initial data block then becomes:

```
COORD 3,5
0.,  12.5,  17.3  10.1,  -2.7,  18.6
27.3,  -9.7,  6.5  207.4, 96.3, 207.4
207.4,  96.3  -207.4 $
G 1, 1
386 $ IN/SEC**2
WEIGHT 5,1 / LBS.
12.1  7.3  4.6  2*.35 $
COMPUTE
```

APPENDIX A

MATE SYSTEM FUNCTIONS

Listed below are some standard MATE functions. To indicate the form of acceptable arguments to these routines, we introduce the following symbols:

A - An array or matrix name. The name may be given with array name qualifiers.

E - Any expression.

and to indicate data types:

N - Numeric, i.e. REAL, INTEGER, or DOUBLE PRECISION numbers.

C - Complex numbers.

NC - Implies either type N or type C.

FUNCTION	ACCEPTABLE ARGUMENT FORM	ARGUMENT DATA TYPE	FUNCTION VALUE DATA TYPE	DESCRIPTION
$NOROWS (X)	A	Any	Integer Scalar	Value is number of rows in argument array. Zero if array not defined.
$NOCOLS (X)	A	Any	Integer Scalar	Value equals number of columns in argument array. Zero if array not defined.
$EXIST (X)	A	Any	Logical Scalar	.TRUE. if the argument array is currently defined (i.e., has data), .FALSE. otherwise.
$GOTARG (X,Y)	E,A	N, Any	Logical Scalar	.TRUE. if the Xth argument (dummy named Y) was given by the calling routine, .FALSE. otherwise. See Note 1
$SIN (X)	E	NC	NC	i,j term of value = sin x_{ij} x_{ij} in radians
$COS (X)	E	NC	NC	i,j term of value = cos x_{ij} x_{ij} in radians
$SQRT (X)	E	NC	NC	i,j term of value = $\sqrt{x_{ij}}$ where x_{ij} positive or complex
$CSQRT (X)	E	NC	C	i,j term of value = $\sqrt{x_{ij}}$
$LOG10 (X)	E	N	N	i,j term of value = $\log_{10} x_{ij}$ See Note 4
$LOG (X)	E	NC	NC	i,j term of value = $\log_e x_{ij} = \ln x_{ij}$
$EXP (X)	E	NC	NC	i,j term of value = $e^{x_{ij}}$ where e base of natural logarithm

FUNCTION	ACCEPTABLE ARGUMENT FORM	ARGUMENT DATA TYPE	FUNCTION VALUE DATA TYPE	DESCRIPTION
$ABS (X)	E	NC	NC	i,j term of value = $\lvert x_{ij} \rvert$
$IMAG (X)	E	C	N	i,j term of value = imaginary part of x_{ij}
$REAL (X)	E	NC	REAL	i,j term of value = x_{ij} converted to REAL type i,j term of value = REAL part of complex x_{ij}
$CONJG (X)	E	C	C	i,j term of value = complex conjugate of x_{ij}
$LOP (X)	E	NC	INTEGER	i,j term of value = truncated x_{ij}. Sign of x_{ij} times largest integer $\leq$ x_{ij} (e.g. 1.9→1) See Note 2
$INT (X)	E	NC	INTEGER	i,j term of value = x_{ij} rounded from nearest half to form whole number (e.g. 1.9→2) See Note 2
$DBLE (X)	E	NC	DOUBLE PRECISION	i,j term of value = x_{ij} expressed as a DOUBLE PRECISION number See Note 2
$CMPLX (X,Y)	E,E	N,N	C	i,j term of value = complex number with x_{ij} as is real part and y_{ij} as its imaginary part See Note 3
$MAX (X,Y)	E,E	N,N	N	i,j term of value = maximum of (x_{ij} , y_{ij}) See Note 3

FUNCTION	ACCEPTABLE ARGUMENT FORM	ARGUMENT DATA TYPE	FUNCTION VALUE DATA TYPE	DESCRIPTION
$MIN (X,Y)	E,E	N,N	N	i,j term of value = minimum of (x_{ij}, y_{ij}) See Note 3
$ATAN (X)	E	N	N	i,j term of value = arctangent of x_{ij} $-\pi/2 < \text{value} < \pi/2$
$ATAN (X,Y)	E,E	N,N	N	i,j term of value = arctangent of (x_{ij}/y_{ij}) $-\pi < \text{value} \leq \pi$ See Note 3
$MOD (X,Y)	E,E	N,N	N	i,j term of value = x_{ij} (modulo y_{ij}) = $x_{ij} - Z\, y_{ij}$ where Z is the largest integer which does not exceed the magnitude of x_{ij}/y_{ij} See Note 3
$SIGN (X,Y)	E,E	N,N	N	i,j term of value = sign of y_{ij} times $\lvert x_{ij} \rvert$ See Note 3
$DIM (X,Y)	E,E	N,N	N	i,j term of value = x_{ij} - minimum of (x_{ij}, y_{ij}) See Note 3
$IDENT (X)	E	N	REAL	Value is an identity matrix of order the maximum of (1, $\lvert x_{11} \rvert$ rounded to the nearest integer).
$TRANSP (X)	E	NC	NC	Value of the function is an array which is the transpose of X (the transpose conjugate if X if complex)

NOTES

1. See Section 10.5 for description of use.

2. If the argument is complex, only its real part is considered.

3. Conformability restrictions are the same as those for the operator + ; i.e. either both matrices are the same size or at least one of them is a scalar.

4. Note the last character in this name is a zero.

APPENDIX B

TRANSLATOR ERROR MESSAGES

Upon encountering an error in a MATE statement, the translator prints the number associated with the error and a succinct message immediately following the offending statement. In addition, it also re-prints at least a portion of the statement with the letter V (to serve as a pointer) above the region where the error resides if applicable.

The translator proceeds directly to the next statement after commenting on the error. (Thus, if a statement contains multiple errors, only the first one found is mentioned.) All remaining statements in the routine are thoroughly checked in the usual manner. However, when the end of the routine is encountered, the final translation steps, which produce the Fortran output, are skipped. Any routines which follow the erroneous one are then translated in the normal fashion.

This appendix contains two tables. The first table gives the error number and the number of the corresponding message in the second table. The letter P after the message number indicates that the region of the error will be pointed to. Also the first table contains an expanded error comment.

Error No./ Message No.	Comments
1, 1P	Illegal character in statement.
2, 2	Statement ends abnormally.
3, 3P	The exponent field on this number is bad, or an operator or a dot which should precede the 'D' or 'E' is missing.
4, 4P	An operator or punctuation is missing between numeric constant and name or illegal form of alphanumeric field-count constant.
5, 5P	The operator word enclosed in dots is not in the table of acceptable words. Check for misspelling.
6, 5P	The leading dot has caused the translator to expect an operator word; but the format of an OP word is violated by the absence of a terminal dot or by the appearance of a non-alphabetic character or by too many letters between the dots.
7, 6P	Either a dot is missing here or the leading dot is extraneous.
8, 7P	Bad entry name reference argument format or illegal $.
9, 7P	$ can't precede this item.
10, 8	Can't have multiple equal signs in a statement.
11, 9	Parentheses don't balance.
12, 10	Qualifier delimiters '' don't balance.
13, 10P	Mate to this qualifier mark not within the same parentheses level or void qualifier field.
14, 4P	An operator or comma which should precede this item is missing.
15, 11P	Too many limiters - 4 max.
16, 12P	Too many name qualifiers - 2 max.
17, 13P	This parentheses encloses a bad expression or the operator in front of it or behind it should not be there.
18, 14P	Can't use qualifier string with dummy argument names.
19, 15P	The name, constant, or expression which should precede this operator is missing.
20, 16P	The name, constant, or expression which should follow this operator is missing.

21, 17P — First item not a name.

22, 2 — Right side of equal sign vacant.

23, 18P — This item can't appear on right-hand side of the statement.

24, 19P — The name, constant, or expression which should precede this comma is missing.

25, 20P — This item can't precede equal sign.

26, 21P — Illegal form of GO TO statement.

27, 22P — Bad statement no. on GO TO statement.

28, 23P — Too many characters in routine reference name or misspelt system control name.

29, 24P — This parenthesis must be immediately preceded by the word IF for this statement to be legal. The word IF is either missing or misspelt.

30, 25P — This word should be the 1st item in the statement.
What's with the garbage before it?

31, 26 — Upper limit on DO loop missing.

32, 27P — Index on DO loop is missing or was given as a constant or an expression.

33, 28P — Multiple names on left of equal sign.

34, 29 — Too many items on right of equal sign.

35, 22P — Statement no. to define end of DO range missing or illegal.

36, 53P — This item is neither a legal statement no. nor a legal way to start a statement. These are the only things that can follow an IF clause.

37, 22P — This field must consist of statement numbers separated by commas, as such this item is illegal.

38, 30P — The word IF must be followed by parentheses which enclose the expression to be tested.

39, 31 — Can't have two logical IF statements back-to-back.

40, 32P — This list may contain only unmodified array names.

41, 33P — A dummy argument to a routine may not be INTERNAL.

42, 34P — Arithmetic IF statement must have exactly three statement numbers given or implied.

43, 7P	Statement number arguments not legal here.
44, 7P	Entry name arguments not legal here.
45, 35P	Fortran subroutine name missing.
46, 36	Void arguments not permitted in this statement.
47, 32P	Constants and expressions not allowed in this statement.
48, 34P	Inner level parentheses not allowed within statement number list.
49, 34P	No statement numbers given in the list.
50, 37P	Exactly one expression required here.
51, 38	Equal sign missing from DO loop statement.
52, 39P	This statement can't be preceded by conditional phrase.
53, 40	Routine statement not at beginning of deck.
54, 22	Non-numeric character in external statement number field.
55, 41	Card columns 1-5 not blank on a continuation card.
56, 22	Statement numbers exceeds 9999 in external statement number field.
57, 42	Too many continuation cards - 5 max.
58, 41	Continuation card does not follow a statement.
59, 23P	Too many characters in this name.
60, 43	Name table overflow. Routine too big. Limit of 1200 different names and statement numbers per routine.
61, 22P	This statement number is illegal.
62, 44P	End of DO range preceeds DO statement.
63, 45	Duplicate external statement number.
64, 46	Routine can't reference itself.
65, 47	Too many nested DO's. Limit is 50 in a nest.
66, 48P	Unrecognizable option word, on INTERNAL statement.
67, 49	Entry name missing from ROUTINE statement.
68, 40	ROUTINE statement missing.
69, 36P	Void argument in ROUTINE statement.

70, 50	Too many expressions in statement. Limit 150.
71, 51	Too many constants in the routine. Limit 999.
72, 47	Too many DO loops in routine. Limit 249.
73, 39	END statement can't be conditional.
74, 52	DO ends range of previous DO.
75, 54	The external statement numbers listed were referenced by some control statements but were never defined.

TRANSLATOR ERROR MESSAGES

Message No.	Message
1	ILLEGAL CHARACTER
2	ABNORMAL STMT. ENDING
3	BAD EXPNT. / MISSING OP.
4	MISSING OP. OR COMMA
5	ILLEGAL OP. WORD
6	MISSING OR EXTRA DOT
7	MISUSE OF $ SIGN
8	MULTIPLE EQUAL SIGNS
9	PARENTHESES UNBALANCED
10	QUOTE MARKS UNBALANCED
11	TOO MANY LIMITERS
12	TOO MANY QUALIFIERS
13	BAD EXPRESSION IN ( )
14	CAN'T QUALIFY ARG. NAMES
15	ITEM BEFORE OP. MISSING
16	ITEM BEHIND OP. MISSING
17	FIRST ITEM NOT A NAME
18	CAN'T BE RIGHT OF = SIGN
19	NOTHING BEFORE COMMA
20	CAN'T BE LEFT OF = SIGN
21	ILLEGAL FORM OF GO TO
22	BAD EXT. STMT. NUMBER
23	TOO MANY CHARACTERS
24	WORD IF IS MISSING
25	SHOULD BE 1ST STMT. ITEM
26	DO LOOP LIMIT MISSING
27	BAD DO LOOP INDEX
28	TOO MANY NAMES LEFT OF =
29	TOO MANY NAMES RT. OF =
30	( ) REQD. AFTER WORD IF
31	MULTIPLE IF STMTS.
32	BAD LIST OF NAMES
33	CAN'T BE INTERNAL
34	BAD EXT. STMT. NO. LIST
35	FORT. ENTRY NAME MISSING
36	VOID ARGUMENTS ILLEGAL
37	ONE EXPRESSION REQUIRED
38	DO LOOP = SIGN MISSING
39	CAN'T BE CONDITIONAL
40	ROUTINE STMT. NOT FIRST
41	BAD CONTINUATION CARD
42	TOO MANY CONTINUATIONS
43	PROGRAM TOO BIG
44	END OF DO LOOP BEFORE DO
45	DUPLICATE EXT. STMT. NO.
46	ROUTINE CALLS ITSELF
47	TOO MANY DO LOOPS
48	BAD OPTION WORD
49	ROUTINE NAME MISSING
50	TOO MANY EXPRESSIONS
51	TOO MANY CONSTANTS
52	DO ENDS RANGE OF DO LOOP
53	BAD ITEM AFTER IF CLAUSE
54	MISSING EXT. STMT. NOS.

APPENDIX C

MATE EXECUTOR CONTROL MATRICES

The MATE executor scans the initial data block for two special vectors which serve as controls for the MATE executor itself. These vectors are named PARTZZ and OPTNZZ.

The elements of PARTZZ specify the order in which the major MATE routines are to be executed (reference Section 10.0). For instance if

$$\text{PARTZZ} = \begin{Bmatrix} 2 \\ 9 \\ 7 \\ 2 \end{Bmatrix}$$

then the MATE executor will start problem execution by referencing routine PART2, then PART9, then PART7, and finally call PART2 again to complete the job. An element of PARTZZ which is not between one and nine inclusive will be ignored. If the vector PARTZZ is not given, the major routines are called in the standard increasing order.

The vector OPTNZZ gives the user's options for the MATE executor. At the present time meaning is attached only to the first, second, and fifth elements of this vector.

Element one: "check-out mode" indicator

= 0 ⇒ print every array or subarray which appears on the left of an equal sign as soon as it is defined or changed.

> 0 ⇒ don't print.

Element two: This element equals the maximum allowable executor eror level that can occur and have execution continue. If given negative, execution will stop if any executor error is sensed.

Element five: This element is used as a safety to prevent "unlimited" execution of DO loops. It is interpreted as follows:

= n where n > 0. States that no DO loop should be executed more than n times.

= 0 implies the executor should use a DO loop limit of 200.

= -1 or any negative number implies DO loop limits are not to be checked.

The role of the second and fifth items in OPTNZZ is further explained in the next appendix.

APPENDIX D

EXECUTOR ERROR MESSAGES

Unlike the translator error messages, the errors discovered during execution are graded as to their severity. Ideally, level zero errors should reflect trivial mistakes while level ten errors imply blunders with tragic consequences. However this is not the case since we are dealing with an idiot machine which is incapable of making such subtle distinctions.

Although intermediate error levels will probably be introduced later, at this time there are but two error levels used in the MATE executor - zero and ten. Level zero errors are those for which, if the user desires, an assumption can be made and execution continued. Execution stops immediately if a level ten error is encountered.

For any error a trace-back table is given in reverse chronological order. This table gives the routine name and the MATE statement number of the statement in that routine which contains the error as well as show how execution arrived at that point. The complaint message is then printed; and if execution is to be continued, the assumption that will be made is described.

Only two level ten errors are currently defined in the MATE system and both involve storage problems. The first of these is:

> INSUFFICIENT CORE STORAGE AVAILABLE
>
> Barring an error in the MATE executor itself this message can occur only in a CALL to a Fortran subroutine whose arguments are too large to fit into the in-core data region of MATE.

The other level ten error message is:

MATE ARRAY STORAGE UNITS FILLED

This means the tapes and/or disk storage devices assigned to the MATE executor for out-of-core array storage have been filled. To bypass this roadblock either delete any arrays no longer required or have more units assigned to the MATE executor.

The level zero errors in the MATE executor proper are tabulated below.

Executor errors arising from an expression of the form A op B where op is an arithmetic, relational or logical operation symbol:

1. UNDEFINED ARRAY USED IN OPERATION
   ARRAY name
   TO CONTINUE ASSUME THAT THE ARRAY IS A SCALAR WITH A VALUE OF value

   where value is either zero, one, or .FALSE. if the operator is: (+, -, **, or relational), (*, 1, .*., ..., .X., .D.), or logical respectively. Note this defines the undefined array just as A = 0.0 would.

2. VOID ARGUMENT USED IN OPERATION
   TO CONTINUE ASSUME THAT THE ANSWER IS A SCALAR WITH A VALUE OF value

   where value is either zero or .FALSE.. Note since we can't assign a value to a void argument, we guess the value of the expression.

3. CROSS PRODUCT REQUIRES 2-ROW OR 3-ROW VECTORS
   ARRAY name i, j, k, l
   ARRAY name i, j, k, l

   where i, j and k, l are the starting and ending element subscript pairs of the matrix or submatrix used in the expression. The problem here is that (k - i +1) is not 2 or 3 for one or both arrays.

   TO CONTINUE ASSUME THAT THE MATRICES CONSIST OF 3-ROW VECTORS

   either by pretending additional rows of zeros exist or by ignoring superfluous rows.

4. NEITHER ARGUMENT A VECTOR FOR .D. OPERATION
   ARRAY name i, j, k, l
   ARRAY name i, j, k, l

   TO CONTINUE ASSUME THAT THE OPERATION IS A STANDARD MATRIX MULTIPLICATION

5. NO. OF ROWS/COLUMNS NON-CONFORMABLE FOR OPERATION
   ARRAY name i, j, k, l
   ARRAY name i, j, k, l

   TO CONTINUE ASSUME THAT ADDITIONAL ROWS/COLUMNS EXIST AS REQUIRED

   The additional rows and/or columns of zeros are not actually appended but only simulated.

6. NON-NUMERIC ARRAY USED IN OPERATION
   ARRAY name i, j, k, l

   The continuation assumption is the same as that used for number 2 above.

   The array named is not a matrix and the operator is either arithmetic or relational.

7. OPERATION REQUIRES ARRAY WITH DATA OF THE LOGICAL TYPE
   ARRAY name i, j, k, l

   Continuation assumption same as number 2. above.

   Logical operator, non-logical data.

8. OPERATION NOT DEFINED FOR ARRAY OF COMPLEX NUMBERS
   ARRAY name i, j, k, l

   Continuation assumption same as number 2. above.

9. DIVISION BY ZERO ATTEMPTED

   TO CONTINUE ASSUME THAT THE ANSWER IS ZERO

Executor errors arising from the basic MATE functions:

1. REAL SQUARE ROOT OF NEGATIVE NUMBER ATTEMPTED

   TO CONTINUE ASSUME THAT THE ABSOLUTE VALUE OF THE ARGUMENT TO BE USED.

Executor errors in the use of Equivalent Integer Expressions (EIE's) or in expressions to be tested or as DO loop limits: These errors share the same continuation assumption.

1. NON-LOGICAL ARRAY USED IN LOGICAL TEST

   The item to be tested in a logical IF statement isn't a logical scalar.

2. NON-NUMERIC DATA USED AS SUBSCRIPT OR IN TEST

   The item is used as an EIE or in an arithmetic IF or DO loop statement but it isn't numeric.

3. UNDEFINED ARRAY USED AS SUBSCRIPT OR IN TEST
4. VOID ARGUMENT USED AS SUBSCRIPT OR IN TEST

The continuation assumption used by all of the above is:

TO CONTINUE ASSUME THAT THE VALUE TO BE USED IS value

where value is either zero or .FALSE. as required.

Another error which can occur to DO loops involves the limit on the number of times a loop may be executed. This limit is intended to prevent "runaway" DO loops.

DO LOOP PARAMETERS EXCEED LOOPING LIMIT. PARAMETERS i, j, k

where i, j, k are the lower limit, upper limit, and increment of the DO loop respectively.

If execution is to be continued and the number of times the loop is to be executed exceeds the limit by less than 50%, the following message is printed.

TO CONTINUE ASSUME THAT LIMIT TO BE IGNORED. PARAMETERS USED AS SHOWN.

Otherwise to continue execution we have

TO CONTINUE ASSUME THAT LOOP TO BE EXECUTED ONLY n TIMES

where n is the looping limit.

Executor errors which can occur in the processing of st ement of the form A=B are discussed here. Again, all these errors make the same assumption to continue.

1. UNDEFINED ARRAY USED IN STATEMENT
   ARRAY name

   The name on the right of the equal sign hasn't been defined.

2. VOID ARGUMENT USED IN STATEMENT
   ARRAY name

   Continuation clause as in number 1. above. Either the A or B in A=B is a dummy argument name which refers to an argument given as void.

3. CONSTANT ON LEFT OF EQUAL SIGN

   Continuation clause as in number 1 above. In this case, A is a dummy argument name which refers to a constant.

4. INCOMPATIBLE DATA TYPES ACROSS EQUAL SIGN
   ARRAY name
        TYPE xxxx
   ARRAY name
        TYPE xxxx

   Continuation clause same as number 1 above.

The first array mentioned is the array on the left of the equal sign. A portion of this array is to be updated with the data in the second array mentioned; however the data in the second array cannot be converted to the type already existing in the other locations of the first array.

If problem execution is allowed to continue after any of the above errors are encountered the following message is printed:

TO CONTINUE ASSUME THAT THE STATEMENT IS TO BE IGNORED

The MATE executor keeps its table of array names in core. Since this table is of finite size it could be overfilled. If this happens, the following messages are printed:

ARRAY NAME TABLE FULL. MAXIMUM NUMBER OF ACTIVE ARRAY NAMES PERMITTED = n

TO CONTINUE ASSUME THAT THE FOLLOWING ARRAY IS TO BE DELETED
ARRAY name

where the array named will be the oldest array (non-scalar if possible) in the problem which is not required in the current statement.

In some instances, the array name printed in the above error comments may not be one of user's array names. For instance, if the array involved in the error is the result of an expression it will have a name of the form BxxxZZ where the x's are digits. Should it become necessary to find out what arrays were used to form the expression, these can be determined by tracing the Fortran listing produced by the MATE translator. In this listing, the third argument to subroutine OPR1ZZ and OPR2ZZ is the expression being formed by operating the next one or two arguments, respectively. The author doubts such investigation would ever be required.

APPENDIX E

RESTRICTIONS ON THE IBM 7090/94 VERSION OF MATE

1. Matrices of complex numbers are not allowed. Therefore complex arithmetic and functions which use complex arguments or yield complex results are not defined. Most of the executor routines required to process complex arrays are available but the severe core shortage problem on the IBM 7090/94 prohibits their use.

2. A numeric constant used in the coding of a MATE expression may not contain more than nine digits unless it is followed by a D exponent.

3. No EIE can exceed 32,767. Therefore numbers used as array name qualifiers and limiters may not exceed this limit. Also the number of rows (as well as the number of columns) in any array may not exceed 32,767.

4. The maximum number of active arrays at any time cannot exceed 100.

5. The maximum number of arguments that can be passed to a Fortran subroutine is eight.

6. The arguments to a Fortran subroutine can occupy no more than 1744 cells.

7. An alphanumeric constant (ref. Chapter 6) argument to a Fortran subroutine can contain at most 18 characters.

8. The basic functions utilize the basic Fortran library supplied by IBM. MATE does not check the arguments to these library subroutines and thus is subject to the error commenting and handling provisions of these routines.

Restrictions numbered 4 and 6 above are also the result of core storage problems and the counts may be juggled at the expense of each other or some other feature of the language. Restriction number 5 is arbitrary and the number of arguments allowable may be readily increased.

APPENDIX F

SAMPLE PROBLEM-PROPERTIES OF POINT MASS SYSTEM

The sample program to obtain the properties of a point-mass system which is described in Chapter 11 was key-punched and run. The results of that run are presented here. The run was made on an IBM 360/44G computer (the G configuration has a 32K word core memory).

The enclosures of this appendix are:

1. The listing of the MATE instructions (input to the translator) for the sample problem. The coding of these cards is presented in Section 13.0.

2. The cross-reference list for this routine as generated by the MATE translator.

3. The listing of the MATE library routine IDENT. This listing was included because it serves as a good demonstration of the material in Chapter 4 and it helps explain the executor output.

4. The cross-reference list for the IDENT routine.

5. The Fortran source listing for the sample problem as output by the MATE translator and given to the Fortran compilier.

6. The compilation map produced by the IBM 360/44 compilier.

7. The Fortran source listing for the routine IDENT.

8. The compilation map for IDENT.

9. Our answers, i.e. the output of the MATE executor.

EXTERNAL STATEMENT NUMBER	MATE STATEMENT		MATE REFERENCE NUMBER
	C SAMPLE PROBLEM CODING - PROPERTIES OF POINT MASS SYSTEM	SAMPL000	
	C PROGRAM REQUIRES THE FOLLOWING MATRICES AT THE START -	SAMPL005	
	C COORD - A 3 BY N WHOSE ROWS ARE THE X,Y,Z COORDINATES OF	SAMPL006	
	C THE POINTS.	SAMPL007	
	C MASS - AN N ROW VECTOR CONTAINING THE MASSES ASSIGNED TO THE	SAMPL010	
	C POINTS DESCRIBED IN COORD.	SAMPL011	
	C	SAMPL030	
	ROUTINE PART1	SAMPL050	0
	C	SAMPL100	
	C ** FIND THE TOTAL MASS AND THE CENTER OF MASS	SAMPL110	
	MTOTAL = (MASS/MASS)...MASS	SAMPL120	1
	OFFSET = COORD * MASS/MTOTAL	SAMPL130	2
	C	SAMPL150	
	C ** SHIFT ORIGIN TO CENTER OF MASS	SAMPL160	
	TCOORD=COORD-OFFSET*COORD(1)**0	SAMPL170	3
	C	SAMPL200	
	C ** DEFINE W AS 3X3 IDENTITY MATRIX. RATHER THAN W=$IDENT(3) LETS USE	SAMPL210	
	IDENT W,3	SAMPL220	4
	C	SAMPL250	
	C ** DEVELOP THE SYSTEM INERTIA MATRIX	SAMPL260	
	INERTA=0*W	SAMPL270	5
	C	SAMPL300	
	DO 30 I=1, $NOCOLS(COORD)	SAMPL320	6
30	INERTA=INERTA + MASS(I)*TCOORD(,I).X.(W.X.TCOORD(,I))	SAMPL340	7
	C	SAMPL400	
	C ** GET PRINCIPAL MASS MOMENTS AND THEIR ORIENTATION	SAMPL410	
	C FIRST DEFINE REQUIRED ARRAYS	SAMPL420	
	PRINCI=INERTA	SAMPL450	8
	DC=W	SAMPL460	9
	TEMP1(3)=0	SAMPL470	10
	TEMP2(3)=0	SAMPL480	11
	C THEN CALL FORTRAN SUBROUTINE EIG1	SAMPL500	
	CALL EIG1 (PRINCI, DC, 3, 0.0, TEMP1,TEMP2, 3,3)	SAMPL520	12
	C ZERO OFF-DIAGONAL TERMS OF PRINCI	SAMPL550	
	PRINCI= W.*.PRINCI	SAMPL560	13
	C	SAMPL600	
	C ** USE DC ON LEFT OF EQUAL SIGN SO IT CAN BE AUTOMATICALLY PRINTED	SAMPL610	
	DC=DC	SAMPL620	14
	C	SAMPL700	
	C ** THAT'S ALL	SAMPL710	
	END	SAMPL720	15

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED

PART1	0

ROUTINE NAMES REFERENCED

EIG1	12
IDENT	4
NOCOLS	6

UNIVERSAL ARRAYS

COORD	2	3	6		
DC	9	12	14		
I	6	7			
INERTA	5	7	8		
MASS	1	2	7		
MTOTAL	1	2			
OFFSET	2	3			
PRINCI	8	12	13		
TCOORD	3	7			
TEMP1	10	12			
TEMP2	11	12			
W	4	5	7	9	13

EXTERNAL STATEMENT NUMBERS

30	6	7

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C ROUTINE TO FORM IDENTITY MATRIX OF A GIVEN ORDER	
	ROUTINE IDENT (I,ORDER)	0
	INTERNAL U	1
	U(ORDER)=0.	2
	U=U**0	3
	I=U.D.U	4
	END	5

ROUTINE IDENT

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
IDENT 0

DUMMY NAMES USED FOR ARGUMENTS
I 0 4

ORDER 0 2

INTERNAL ARRAYS
U 1 2 3 4

```
C       SAMPLE PROBLEM CODING - PROPERTIES OF POINT MASS SYSTEM                  10
C     PROGRAM REQUIRES THE FOLLOWING MATRICES AT THE START -                     11
C         COORD - A 3 BY N WHOSE ROWS ARE THE X,Y,Z COORDINATES OF               12
C                 THE POINTS.                                                    13
C         MASS  - AN N ROW VECTOR CONTAINING THE MASSES ASSIGNED TO THE          14
C                 POINTS DESCRIBED IN COORD.                                     15
C                                                                                16
C      ROUTINE PART1                                                             17
0001            SUBROUTINE PART1  (N050ZZ,N051ZZ)                                18
0002            INTEGER MASS(5),MTOTAL(5),COORD(5),OFFSET(5),TCOORD(5),W(5),     5000
               1INERTA(5),I(5),PRINCI(5),DC(5),TEMP1(5),TEMP2(5),B001ZZ(5),      5001
               1B002ZZ(5),B003ZZ(5),A001ZZ(7),A002ZZ(7),A003ZZ(7),X001ZZ(7)      5002
0003            INTEGER C001ZZ(2),C002ZZ(2),C003ZZ(2),C004ZZ(2),C005ZZ(2),       5003
               1C006ZZ(2),C007ZZ(2),C008ZZ(2),C009ZZ(2),C010ZZ(2),C011ZZ(2),     5004
               1C012ZZ(2),C013ZZ(2)                                              5005
0004            REAL E011ZZ(2)                                                   5006
0005            INTEGER N000ZZ(3),N050ZZ(3),K000ZZ(2),K001ZZ(2)                  5007
0006            COMMON R001ZZ(8)                                                 5008
0007            EQUIVALENCE (E011ZZ,C011ZZ)                                      5009
0008            DATA MASS(1),MTOTAL(1),COORD(1),OFFSET(1),TCOORD(1),W(1),        5010
               1INERTA(1),I(1),PRINCI(1),DC(1),TEMP1(1),TEMP2(1),B001ZZ(1),      5011
               1B002ZZ(1),B003ZZ(1) / 15* 1/                                     5012
0009            DATA MASS(2),MASS(3),MASS(4)/4HMASS,4H    ,4H    /,MTOTAL(2)     5013
               1,MTOTAL(3),MTOTAL(4)/4HMTOT,4HAL  ,4H    /,COORD(2)             5014
               1,COORD(3),COORD(4)/4HCOOR,4HD   ,4H    /,OFFSET(2)              5015
               1,OFFSET(3),OFFSET(4)/4HOFFS,4HET  ,4H    /,TCOORD(2)            5016
               1,TCOORD(3),TCOORD(4)/4HTCOO,4HRD  ,4H    /,W(2),W(3)            5017
               1,W(4)/4HW   ,4H    ,4H    /,INERTA(2),INERTA(3),INERTA(4)       5018
               1/4HINER,4HTA  ,4H    /,I(2),I(3),I(4)/4HI   ,4H    ,4H    /,    5019
               1PRINCI(2),PRINCI(3),PRINCI(4)/4HPRIN,4HCI  ,4H    /,DC(2)       5020
               1,DC(3),DC(4)/4HDC  ,4H    ,4H    /,TEMP1(2),TEMP1(3)           5021
               1,TEMP1(4)/4HTEMP,4H1   ,4H    /,TEMP2(2),TEMP2(3)              5022
               1,TEMP2(4)/4HTEMP,4H2   ,4H    /,B001ZZ(2),B001ZZ(3)            5023
               1,B001ZZ(4)/4HB001,4HZZPA,4HRT1 /                                5024
0010            DATA B002ZZ(2),B002ZZ(3),B002ZZ(4)/4HB002,4HZZPA,4HRT1 /,B003ZZ(2)  5025
               1,B003ZZ(3),B003ZZ(4)/4HB003,4HZZPA,4HRT1 /                       5026
0011            DATA MASS(5),MTOTAL(5),COORD(5),OFFSET(5),TCOORD(5),W(5),        5027
               1INERTA(5),I(5),PRINCI(5),DC(5),TEMP1(5),TEMP2(5),B001ZZ(5),      5028
               1B002ZZ(5),B003ZZ(5) / 15* 0/                                     5029
0012            DATA C001ZZ(1)/   3/,C001ZZ(2)/1/,C002ZZ(1)/   3/,C002ZZ(2)/0/,  5030
               1C003ZZ(1)/   3/,C003ZZ(2)/3/,C004ZZ(1)/   3/,C004ZZ(2)/0/,      5031
               1C005ZZ(1)/   3/,C005ZZ(2)/1/,C006ZZ(1)/   3/,C006ZZ(2)/0/,      5032
               1C007ZZ(1)/   3/,C007ZZ(2)/3/,C008ZZ(1)/   3/,C008ZZ(2)/0/,      5033
               1C009ZZ(1)/   3/,C009ZZ(2)/3/,C010ZZ(1)/   3/,C010ZZ(2)/3/,      5034
               1C011ZZ(1)/   4/,E011ZZ(2)/0.E0/                                  5035
0013            DATA C012ZZ(1)/   3/,C012ZZ(2)/3/,C013ZZ(1)/   3/,C013ZZ(2)/3/   5036
0014            DATA  K000ZZ /3,0/, K001ZZ /3,-1/                                5037
0015            DATA N000ZZ(2),N000ZZ(3)/4HPART,2H1 /                            5038
0016            CALL BEGNZZ (N050ZZ,N051ZZ)                                      5039
```

```
                C                                                                10001
                C  ** FIND THE TOTAL MASS AND THE CENTER OF MASS                 10002
                C     MTOTAL = (MASS/MASS)...MASS                                10003
0017                  N000ZZ(1)=0001                                             10004
0018                  CALL LINKZZ (6HOPR2ZZ)                                     15000
0019                  CALL OPR2ZZ (N000ZZ,B001ZZ,MASS,MASS, 71)                  15000
0020                  CALL LINKZZ (6HOPR2ZZ)                                     15001
0021                  CALL OPR2ZZ (N000ZZ,B002ZZ,B001ZZ,MASS, 90)                15001
0022                  CALL LINKZZ (6HCOPXZZ)                                     15002
0023                  CALL COPXZZ (N000ZZ,MTOTAL,B002ZZ)                         15002
                C     OFFSET = COORD * MASS/MTOTAL                               20001
0024                  N000ZZ(1)=0002                                             20002
0025                  CALL LINKZZ (6HOPR2ZZ)                                     25000
0026                  CALL OPR2ZZ (N000ZZ,B001ZZ,COORD,MASS, 70)                 25000
0027                  CALL LINKZZ (6HOPR2ZZ)                                     25001
0028                  CALL OPR2ZZ (N000ZZ,B002ZZ,B001ZZ,MTOTAL, 71)              25001
0029                  CALL LINKZZ (6HCOPXZZ)                                     25002
0030                  CALL COPXZZ (N000ZZ,OFFSET,B002ZZ)                         25002
                C                                                                30001
                C  ** SHIFT ORIGIN TO CENTER OF MASS                             30002
                C     TCOORD=COORD-OFFSET*COORD(1)**0                            30003
0031                  N000ZZ(1)=0003                                             30004
0032                  CALL LINKZZ (6HENCDZZ)                                     35000
0033                  CALL ENCDZZ (A001ZZ,COORD,2,K000ZZ,K000ZZ,                 35000
                     1C001ZZ,K001ZZ,K001ZZ,K001ZZ)                               35001
0034                  CALL LINKZZ (6HOPR2ZZ)                                     35002
0035                  CALL OPR2ZZ (N000ZZ,B001ZZ,A001ZZ,C002ZZ, 50)              35002
0036                  CALL LINKZZ (6HOPR2ZZ)                                     35003
0037                  CALL OPR2ZZ (N000ZZ,B002ZZ,OFFSET,B001ZZ, 70)              35003
0038                  CALL LINKZZ (6HOPR2ZZ)                                     35004
0039                  CALL OPR2ZZ (N000ZZ,B001ZZ,COORD,B002ZZ,101)               35004
0040                  CALL LINKZZ (6HCOPXZZ)                                     35005
0041                  CALL COPXZZ (N000ZZ,TCOORD,B001ZZ)                         35005
                C                                                                40001
                C  ** DEFINE W AS 3X3 IDENTITY MATRIX.  RATHER THAN W=$IDENT(3) LETS USE  40002
                C     IDENT W,3                                                  40003
0042                  N000ZZ(1)=0004                                             40004
0043                  CALL LINKZZ (6HIDENT )                                     45000
0044                  CALL IDENT  (N000ZZ,  2,W,C003ZZ,0,0,0,0,0,0,0,0)          45000
                C                                                                50001
                C  ** DEVELOP THE SYSTEM INERTIA MATRIX                          50002
                C     INERTA=0*W                                                 50003
0045                  N000ZZ(1)=0005                                             50004
0046                  CALL LINKZZ (6HOPR2ZZ)                                     55000
0047                  CALL OPR2ZZ (N000ZZ,B001ZZ,C004ZZ,W, 70)                   55000
0048                  CALL LINKZZ (6HCOPXZZ)                                     55001
0049                  CALL COPXZZ (N000ZZ,INERTA,B001ZZ)                         55001
                C                                                                60001
                C     DO 30  I=1, $NOCOLS(COORD)                                 60002
```

```
0050            N000ZZ(1)=0006                                                  60003
0051            CALL LINKZZ (6HNOCOLS)                                          65000
0052            CALL NOCOLS (N000ZZ,  2,B001ZZ,COORD,0,0,0,0,0,0,0,0)           65000
0053            CALL LINKZZ (6HLOOPZZ)                                          65001
0054            CALL LOOPZZ (N000ZZ,C005ZZ,B001ZZ,0,I001ZZ,I002ZZ,I003ZZ,       65001
               1Q001ZZ,Q002ZZ,Q003ZZ,I,X001ZZ)                                  65002
0055            DO 10030 I004ZZ=I001ZZ,I002ZZ,I003ZZ                            65003
0056            CALL LINKZZ (6HINDXZZ)                                          65004
0057            CALL INDXZZ (X001ZZ,I004ZZ,Q001ZZ,Q002ZZ,Q003ZZ)                65004
          C  30 INERTA=INERTA + MASS(I)*TCOORD(,I).X.(W.X.TCOORD(,I))           70001
0058            30 CONTINUE                                                     70002
0059            N000ZZ(1)=0007                                                  70003
0060            CALL LINKZZ (6HENCDZZ)                                          75000
0061            CALL ENCDZZ (A001ZZ,TCOORD,2,K000ZZ,K000ZZ,                     75000
               1K000ZZ,I,K001ZZ,K001ZZ)                                         75001
0062            CALL LINKZZ (6HOPR2ZZ)                                          75002
0063            CALL OPR2ZZ (N000ZZ,B001ZZ,W,A001ZZ, 80)                        75002
0064            CALL LINKZZ (6HENCDZZ)                                          75003
0065            CALL ENCDZZ (A002ZZ,MASS,2,K000ZZ,K000ZZ,                       75003
               1I,K001ZZ,K001ZZ,K001ZZ)                                         75004
0066            CALL LINKZZ (6HENCDZZ)                                          75005
0067            CALL ENCDZZ (A003ZZ,TCOORD,2,K000ZZ,K000ZZ,                     75005
               1K000ZZ,I,K001ZZ,K001ZZ)                                         75006
0068            CALL LINKZZ (6HOPR2ZZ)                                          75007
0069            CALL OPR2ZZ (N000ZZ,B002ZZ,A002ZZ,A003ZZ, 70)                   75007
0070            CALL LINKZZ (6HOPR2ZZ)                                          75008
0071            CALL OPR2ZZ (N000ZZ,B003ZZ,B002ZZ,B001ZZ, 80)                   75008
0072            CALL LINKZZ (6HOPR2ZZ)                                          75009
0073            CALL OPR2ZZ (N000ZZ,B001ZZ,INERTA,B003ZZ,100)                   75009
0074            CALL LINKZZ (6HCOPXZZ)                                          75010
0075            CALL COPXZZ (N000ZZ,INERTA,B001ZZ)                              75010
0076      10030 CONTINUE                                                        75011
          C                                                                     80001
          C  ** GET PRINCIPAL MASS MOMENTS AND THEIR ORIENTATION                80002
          C       FIRST DEFINE REQUIRED ARRAYS                                  80003
          C       PRINCI=INERTA                                                 80004
0077            N000ZZ(1)=0008                                                  80005
0078            CALL LINKZZ (6HCOPYZZ)                                          85000
0079            CALL COPYZZ (N000ZZ,PRINCI,INERTA)                              85000
          C       DC=W                                                          90001
0080            N000ZZ(1)=0009                                                  90002
0081            CALL LINKZZ (6HCOPYZZ)                                          95000
0082            CALL COPYZZ (N000ZZ,DC,W)                                       95000
          C       TEMP1(3)=0                                                    100001
0083            N000ZZ(1)=0010                                                  100002
0084            CALL LINKZZ (6HENCDZZ)                                          105000
0085            CALL ENCDZZ (A001ZZ,TEMP1,2,K000ZZ,K000ZZ,                      105000
               1C007ZZ,K001ZZ,K001ZZ,K001ZZ)                                    105001
0086            CALL LINKZZ (6HCOPYZZ)                                          105002
```

```
0087              CALL COPYZZ (N000ZZ,A001ZZ,C006ZZ)                                  105002
                C     TEMP2(3)=0                                                      110001
0088              N000ZZ(1)=0011                                                      110002
0089              CALL LINKZZ (6HENCDZZ)                                              115000
0090              CALL ENCDZZ (A001ZZ,TEMP2,2,K000ZZ,K000ZZ,                          115000
                 1C009ZZ,K001ZZ,K001ZZ,K001ZZ)                                        115001
0091              CALL LINKZZ (6HCOPYZZ)                                              115002
0092              CALL COPYZZ (N000ZZ,A001ZZ,C008ZZ)                                  115002
                C     THEN CALL FORTRAN SUBROUTINE EIG1                             120001
0093              CALL LINKZZ (6HEIG1  )                                              120002
                C     CALL EIG1 (PRINCI, DC, 3, 0.0, TEMP1,TEMP2, 3,3)              120002
0094              N000ZZ(1)=0012                                                      120003
0095              CALL LINKZZ (6HLOADZZ)                                              125000
0096              CALL LOADZZ (N000ZZ,  8,0,PRINCI,J001ZZ,DC,J002ZZ,C010ZZ,J003ZZ,    125000
                 1C011ZZ,J004ZZ,TEMP1,J005ZZ,TEMP2,J006ZZ,C012ZZ,J007ZZ,C013ZZ,      125001
                 1J008ZZ)                                                           125002
0097              CALL LINKZZ (6HEIG1  )                                              125003
0098              CALL EIG1 (R001ZZ(J001ZZ),R001ZZ(J002ZZ),R001ZZ(J003ZZ),R001ZZ(     125003
                 1J004ZZ),R001ZZ(J005ZZ),R001ZZ(J006ZZ),R001ZZ(J007ZZ),R001ZZ(       125004
                 1J008ZZ))                                                          125005
0099              CALL LINKZZ (6HUNLDZZ)                                              125006
0100              CALL UNLDZZ (N000ZZ,  8,0,PRINCI,J001ZZ,DC,J002ZZ,C010ZZ,J003ZZ,    125006
                 1C011ZZ,J004ZZ,TEMP1,J005ZZ,TEMP2,J006ZZ,C012ZZ,J007ZZ,C013ZZ,      125007
                 1J008ZZ)                                                           125008
                C     ZERO OFF-DIAGONAL TERMS OF PRINCI                             130001
                C       PRINCI= W.*.PRINCI                                          130002
0101              N000ZZ(1)=0013                                                      130003
0102              CALL LINKZZ (6HOPR2ZZ)                                              135000
0103              CALL OPR2ZZ (N000ZZ,B001ZZ,W,PRINCI, 72)                            135000
0104              CALL LINKZZ (6HCOPXZZ)                                              135001
0105              CALL COPXZZ (N000ZZ,PRINCI,B001ZZ)                                  135001
                C                                                                     140001
                C  ** USE DC ON LEFT OF EQUAL SIGN SO IT CAN BE AUTOMATICALLY PRINTED 140002
                C     DC=DC                                                         140003
0106              N000ZZ(1)=0014                                                      140004
0107              CALL LINKZZ (6HCOPYZZ)                                              145000
0108              CALL COPYZZ (N000ZZ,DC,DC)                                          145000
                C                                                                     150001
                C  ** THAT'S ALL                                                      150002
                C     END                                                           150003
0109            30000 CONTINUE                                                      155000
0110              CALL LINKZZ (6HDONEZZ)                                              155001
0111              CALL DONEZZ                                                         155001
0112              CALL LINKZZ (6HDELTZZ)                                              155002
0113              CALL DELTZZ ( 3,B001ZZ,B002ZZ,B003ZZ)                               155002
0114              RETURN                                                              155003
0115              END                                                                 155004
```

COMMON BLOCK / / MAP SIZE 000020

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
R001ZZ	000000								

EQUIVALENCE DATA MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
E011ZZ	0000DC	C011ZZ	0000DC						

SCALAR MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
N051ZZ	0000E4	I001ZZ	0000E8	I002ZZ	0000EC	I003ZZ	0000F0	Q001ZZ	0000F4
Q002ZZ	0000F8	Q003ZZ	0000FC	I004ZZ	000100	J001ZZ	000104	J002ZZ	000108
J003ZZ	00010C	J004ZZ	000110	J005ZZ	000114	J006ZZ	000118	J007ZZ	00011C
J008ZZ	000120								

ARRAY MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
MASS	000124	MTOTAL	000138	COORD	00014C	OFFSET	000160	TCOORD	000174
W	000188	INERTA	00019C	I	000180	PRINCI	0001C4	DC	0001D8
TEMP1	0001EC	TEMP2	000200	B001ZZ	000214	B002ZZ	000228	B003ZZ	00023C
A001ZZ	000250	A002ZZ	00026C	A003ZZ	000288	X001ZZ	0002A4	C001ZZ	0002C0
C002ZZ	0002C8	C003ZZ	0002D0	C004ZZ	0002D8	C005ZZ	0002E0	C006ZZ	0002E8
C007ZZ	0002F0	C008ZZ	0002F8	C009ZZ	000300	C010ZZ	000308	C012ZZ	000310
C013ZZ	000318	N000ZZ	000320	N050ZZ	00032C	K000ZZ	000330	K001ZZ	000338

SUBPROGRAMS CALLED

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
BEGNZZ	000340	LINKZZ	000344	OPR2ZZ	000348	COPXZZ	00034C	ENCDZZ	000350
IDENT	000354	NOCOLS	000358	LOOPZZ	00035C	INDXZZ	000360	COPYZZ	000364
LOADZZ	000368	EIG1	00036C	UNLDZZ	000370	DONEZZ	000374	DELTZZ	000378

LABEL MAP

LABEL	LOCATION	LABEL	LOCATION	LABEL	LOCATION	LABEL	LOCATION	LABEL	LOCATION
30	000844	10030	0008F0	30000	000DDC				

TOTAL MEMORY REQUIREMENTS 000E8C BYTES

COMPILER HIGHEST SEVERITY CODE WAS 0
// RESET 155004
/&

IA58I	CUU	RW	RR	RN	PW	PR	PN
IA59I	083	3	0	0	0	0	0

```
                C     ROUTINE TO FORM IDENTITY MATRIX OF A GIVEN ORDER                    10
                C     ROUTINE IDENT (I,ORDER)                                             11
0001                  SUBROUTINE IDENT  (N050ZZ,N051ZZ,I,ORDER)                           12
0002                  INTEGER I(5),ORDER(5),U(5),B001ZZ(5),A001ZZ(7)                    5000
0003                  INTEGER C001ZZ(2),C002ZZ(2)                                       5001
0004                  REAL E001ZZ(2)                                                    5002
0005                  INTEGER N000ZZ(3),N050ZZ(3),K000ZZ(2),K001ZZ(2)                   5003
0006                  EQUIVALENCE (E001ZZ,C001ZZ)                                       5004
0007                  DATA U(1),B001ZZ(1) /  2* 1/                                      5005
0008                  DATA U(2),U(3),U(4)/4HU   ,4H  ID,4HENT /,B001ZZ(2)               5006
                     1,B001ZZ(3),B001ZZ(4)/4HB001,4HZZID,4HENT /                        5007
0009                  DATA U(5),B001ZZ(5) /  2* 0/                                      5008
0010                  DATA C001ZZ(1)/   4/,E001ZZ(2)/0.E0/,C002ZZ(1)/   3/,C002ZZ(2)/0/  5009
0011                  DATA  K000ZZ /3,0/, K001ZZ /3,-1/                                 5010
0012                  DATA N000ZZ(2),N000ZZ(3)/4HIDEN,2HT /                             5011
0013                  CALL BEGNZZ (N050ZZ,N051ZZ)                                       5012
                C     INTERNAL U                                                       10001
                C     U(ORDER)=0.                                                      20001
0014                  N000ZZ(1)=0002                                                   20002
0015                  CALL LINKZZ (6HENCDZZ)                                           25000
0016                  CALL ENCDZZ (A001ZZ,U,2,K000ZZ,K000ZZ,                           25000
                     1ORDER,K001ZZ,K001ZZ,K001ZZ)                                     25001
0017                  CALL LINKZZ (6HCOPYZZ)                                           25002
0018                  CALL COPYZZ (N000ZZ,A001ZZ,C001ZZ)                               25002
                C     U=U**0                                                           30001
0019                  N000ZZ(1)=0003                                                   30002
0020                  CALL LINKZZ (6HOPR2ZZ)                                           35000
0021                  CALL OPR2ZZ (N000ZZ,B001ZZ,U,C002ZZ, 50)                         35000
0022                  CALL LINKZZ (6HCOPXZZ)                                           35001
0023                  CALL COPXZZ (N000ZZ,U,B001ZZ)                                    35001
                C     I=U.D.U                                                          40001
0024                  N000ZZ(1)=0004                                                   40002
0025                  CALL LINKZZ (6HOPR2ZZ)                                           45000
0026                  CALL OPR2ZZ (N000ZZ,B001ZZ,U,U, 60)                              45000
0027                  CALL LINKZZ (6HCOPXZZ)                                           45001
0028                  CALL COPXZZ (N000ZZ,I,B001ZZ)                                    45001
                C     END                                                              50001
0029            30000 CONTINUE                                                         55000
0030                  CALL LINKZZ (6HDONEZZ)                                           55001
0031                  CALL DONEZZ                                                      55001
0032                  CALL LINKZZ (6HDELTZZ)                                           55002
0033                  CALL DELTZZ ( 1,B001ZZ)                                          55002
0034                  CALL LINKZZ (6HDELTZZ)                                           55003
0035                  CALL DELTZZ ( 1,U)                                               55003
0036                  RETURN                                                           55004
0037                  END                                                              55005
```

EQUIVALENCE DATA MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
E001ZZ	0000D4	C001ZZ	C000D4						

SCALAR MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
N051ZZ	0000DC								

ARRAY MAP

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
I	0000E0	ORDER	0000E4	U	0000E8	B001ZZ	0000FC	A001ZZ	000110
C002ZZ	00012C	N000ZZ	C00134	N050ZZ	000140	K000ZZ	000144	K001ZZ	00014C

SUBPROGRAMS CALLED

SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION	SYMBOL	LOCATION
BEGNZZ	000154	LINKZZ	000158	ENCDZZ	00015C	COPYZZ	000160	OPR2ZZ	000164
COPXZZ	000168	DONE7Z	00016C	DELTZZ	000170				

LABEL MAP

LABEL	LOCATION	LABEL	LOCATION	LABEL	LOCATION	LABEL	LOCATION	LABEL	LOCATION
30000	0003A8								

TOTAL MEMORY REQUIREMENTS 000428 BYTES

COMPILER HIGHEST SEVERITY CODE WAS 0

```
IA50I  ABN EOJ
IA58I   CUU  RW  RR  RN  PW  PR  PN
IA59I   082   2   0   0   0   1   0
IA52A  STOP
```

ARRAY COORD

DEFINED SIZE 3 ROWS BY 5 COLUMNS

ROW	COLUMN 1	2	3	4	5
1	0.0	1.01000E 01	2.73000E 01	2.07400E 02	2.07400E 02
2	1.25000E 01	-2.70000E 00	-9.70000E 00	9.63000E 01	9.63000E 01
3	1.73000E 01	1.86000E 01	6.50000E 00	2.07400E 02	-2.07400E 02

~ FROM THE INITIAL DATA BLOCK

ARRAY MASS

DEFINED SIZE 5 ROWS BY 1 COLUMNS

ROW	
1	3.13470E-02
2	1.89120E-02
3	1.19170E-02
4	9.06740E-04
5	9.06740E-04

ARRAY MTOTAL

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	6.39895E-02

ARRAY OFFSET

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.39470E 01
2	6.24819E 00
3	1.51826E 01

MTOTAL, OFFSET, & TCOORD AS DEFINED BY STATEMENTS 1, 2, & 3, RESPECTIVELY

ARRAY TCOORD

DEFINED SIZE 3 ROWS BY 5 COLUMNS

ROW	COLUMN 1	2	3	4	5
1	-1.39470E 01	-3.84699E 00	1.33530E 01	1.93453E 02	1.93453E 02
2	6.25181E 00	-8.94819E 00	-1.59482E 01	9.00518E 01	9.00518E 01
3	2.11740E 00	3.41741E 00	-8.68258E 00	1.92217E 02	-2.22583E 02

```
  ARRAY  U      IDENT                                  (    3,    1,    3,    1)

                DEFINED SIZE       3 ROWS BY    1 COLUMNS
ROW
  3    0.0
```

FROM ROUTINE IDENT

1ST STATEMENT DEFINES U AS 3×1 ZERO VECTOR

```
  ARRAY  U      IDENT

                DEFINED SIZE       3 ROWS BY    1 COLUMNS
ROW
  1    1.00000E 00
  2    1.00000E 00
  3    1.00000E 00
```

2ND STATEMENT SETS

$$U = \begin{Bmatrix} 1 \\ 1 \\ 1 \end{Bmatrix}$$

```
  ARRAY  W

                DEFINED SIZE       3 ROWS BY    3 COLUMNS

     COLUMN      1              2              3
ROW
  1    1.00000E 00    0.0            0.0
  2    0.0            1.00000E 00    0.0
  3    0.0            0.0            1.00000E 00
```

3RD STATEMENT SETS 'W' (DUMMY NAME 'I') TO 3×3 IDENTITY MATRIX

```
  ARRAY  INERTA

                DEFINED SIZE       3 ROWS BY    3 COLUMNS

     COLUMN      1              2              3
ROW
  1    0.0            0.0            0.0
  2    0.0            0.0            0.0
  3    0.0            0.0            0.0
```

FROM THE 4TH STATEMENT WHICH INITIALIZES INERTA

```
  ARRAY  BOOLZZPART1

                DEFINED SIZE       1 ROWS BY    1 COLUMNS
ROW
  1              5
```

RESULT OF $NOCOLS(COORD) NOTE STRANGE NAME ASSIGNED BY MATE TRANSLATOR

```
  ARRAY  I

                DEFINED SIZE       1 ROWS BY    1 COLUMNS
ROW
  1    1.00000E 00
```

THE NEXT 5 PAIRS PRINTED GIVE THE VALUE OF THE DO LOOP INDEX 'I' AND THE

ARRAY INERTA

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	1.36574E 00	2.73327E 00	9.25721E-01
2	2.73327E 00	6.23811E 00	-4.14959E-01
3	9.25722E-01	-4.14959E-01	7.32277E 00

ARRAY I

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	2.000(c 00

ARRAY INERTA

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	3.10089E 00	2.08225E 00	1.17435E 00
2	2.08225E 00	6.73887E 00	1.63362E-01
3	1.17435E 00	1.63362E-01	9.11694E 00

ARRAY I

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	3.00000E 00

ARRAY INERTA

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	7.03031E 00	4.62004E 00	2.55599E 00
2	4.62004E 00	9.76208E 00	-1.48680E 00
3	2.55599E 00	-1.48680E 00	1.42728E 01

ARRAY I

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	4.00000E 00

ARRAY INERTA

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	4.78851E 01	-1.11761E 01	-3.11611E 01
2	-1.11761E 01	7.71977E 01	-1.71820E 01
3	-3.11611E 01	-1.71820E 01	5.55597E 01

ARRAY I

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	5.00000E 00

ARRAY INERTA

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	1.00161E 02	-2.69722E 01	7.88240E 00
2	-2.69722E 01	1.56054E 02	9.92615E-01
3	7.88239E 00	9.92630E-01	9.68466E 01

THIS IS OUR FINAL INERTIA MATRIX

ARRAY PRINCI

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	1.00161E 02	-2.69722E 01	7.88240E 00
2	-2.69722E 01	1.56054E 02	9.92615E-01
3	7.88239E 00	9.92630E-01	9.68466E 01

FROM STATEMENT NO. 8

ARRAY DC

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW	COLUMN 1	2	3
1	1.00000E 00	0.0	0.0
2	0.0	1.00000E 00	0.0
3	0.0	0.0	1.00000E 00

FROM STATEMENT NO. 9

ARRAY TEMP1 (3, 1, 3, 1)

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
3	0

ARRAY TEMP2 (3, 1, 3, 1)

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
3	0

RESULTS OF STATEMENTS 10 & 11 WHICH DEFINE TEMP1 = TEMP2 = $\begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}$

ARRAY PRINCI

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW / COLUMN	1	2	3
1	8.44791E 01	0.0	0.0
2	0.0	1.67007E 02	0.0
3	0.0	0.0	1.01577E 02

FROM STATEMENT NO. 13 THE PRINCIPAL INERTIAS

ARRAY DC

DEFINED SIZE 3 ROWS BY 3 COLUMNS

ROW / COLUMN	1	2	3
1	7.91629E-01	-3.76991E-01	4.80838E-01
2	3.05651E-01	9.25756E-01	2.22611E-01
3	-5.29061E-01	-2.92568E-02	8.48082E-01

FROM STATEMENT NO. 14 THE DIRECTION COSINE MATRIX

JAOAI-JOB CNCLED

APPENDIX G

SAMPLE PROBLEM - ELASTIC TWO-BODY ATM SIMULATION

This problem was furnished by MSFC to be run on the MATE system. The equations given in the reference document (see next page) were condensed from their scalar form into equivalent matrix and vector equations. These equations were then coded as MATE routines. Then a main routine to perform numerical integration was written. For simplicity, Euler's integration formula was used here.

The listings of the MATE routines for this job and their cross-reference lists are presented on the following pages. These are followed by selected portions of the executor output. Due to the large volume of output from this run (using "check-out mode") only the quantities generated in the integration routine for a few time steps are shown.

EXTERNAL STATEMENT NUMBER | MATE STATEMENT | MATE REFERENCE NUMBER

```
C
C
C        *****          M     M      A      TTTTTTT  EEEEE         *****
C                       MM   MM     A A        T     E
C        *****          M M M M    A   A       T     EEE           *****
C                       M  M  M   AAAAAAA      T     E
C        *****          M     M  A       A     T     EEEEEE        *****
C
C
C
C              *****          SAMPLE  PROBLEM          *****
C
C
C     ALL EQUATIONS, ASSUMPTIONS, AND CONSTANTS USED IN THIS SAMPLE
C        PROBLEM ARE PRESENTED IN REF. 1.
C
C              REFERENCE 1 -
C                            DAUGHTREY, T. W. AND LOWE, J. A.,
C                            ELASTIC TWO-BODY ATM ANALOG/HYBRID
C                            SIMULATION, VOLUME 1, CSC REPORT NO.
C                            68-31-34, FEBRUARY 1969.
C
C
C                       NOTE - THE VERNIER CONTROL SYSTEM MODULE WILL NOT
C                              BE INCLUDED IN THIS SIMULATION.
C
C
C     THE PROBLEM EQUATIONS ARE SET UP IN TWO MAJOR ROUTINES -
C        1. AN INTEGRATION ROUTINE WHERE ALL DERIVATIVES ARE INTEGRATED
C           AND WHICH CALLS THE -
C
C        2. DERIVATIVE ROUTINE WHICH SOLVES EQUATIONS FOR THE HIGHEST
C           ORDER DERIVATIVES. THIS ROUTINE IS FURTHER DIVIDED INTO
C           THREE ROUTINES --  HINGED-BODY DYNAMICS, DGCMG CONTROL
C           SYSTEM, AND ELASTIC-BODY DYNAMICS.
C
C
C     THE FOLLOWING CONSTANTS FORM A PART OF THE UNIVERSAL ARRAY POOL
C        AND MAY CALLED BY ANY ROUTINE. THE VALUES ARE DEFINED AT
C        EXECUTION.  DEFINITIONS ARE INCLUDED.
C
C
C           RVX    3.4686                    RADIUS VECTOR FROM
C     RV  = RVY =-0.1615                     CLUSTER C.M. TO HINGE
C           RVZ    5.9000                    POINT
C
C
C
C           RTX    0.006096                  RADIUS VECTOR FROM
C     RT  = RTY  = 0.006096                  TELESCOPE C.M. TO
C           RTZ    0.006096                  HINGE POINT
C
```

ROUTINE

```
C
C            PHI2X PHI9X   -2.0994E-4  1.2881E-1   MODAL DISPLACEMENT
C      PHI = PHI2Y PHI9Y =  2.7777E-2  1.2937E-4   COEFFICIENTS -
C            PHI2Z PHI9Z   7.0281E-4  3.8484E-3   MODE/COL.
C
C
C           K0X   2.0E5                            ATTITUDE GAIN
C      K0 = K0Y = 4.0E5                            CMG CONTROL SYSTEM
C           K0Z   1.0E6
C
C
C           K1X   4.0E5                            ATTITUDE GAIN RATE
C      K1 = K1Y = 2.0E6                            CMG CONTROL SYSTEM
C           K1Z   3.33E6
C
C
C          H1   2712.0                             NOMINAL MAGNITUDE OF
C      H = H2 = 2712.0                             CMG ANGULAR MOMENTUM
C           H3   2712.0
C
C
C      HCMAX = 8472.0                              MAX. COMAND. ANG. MOM.
C
C      HCMAXD = 270.0                              MAX COMAND.RATE ANG M
C
C      HEMAX = 45.0                                MAX ANG MOM ERROR
C
C      KSL = 2.582E-3                              H-VECTOR PROP. CONST.
C
C      DCMAXD = 0.0611                             MAX CMG GIMBAL RATE CM
C
C      DIMAX = 1.396                               MAX CMG INNER GIM. ANG
C
C
C           D3P = 3.84                             MAX POS CMG OUT GIM AN
C           D3N = 2.27                             MAX NEG CMG OUT GIM AN
C
C
C      TAUM = 0.02                                 CMG GIM MOTOR TIME CON
C
C
C            PHIP2X PHIP9X   -4.8985E-3 -1.1740E-5 MODAL SLOPE COEF.
C      PHIP = PHIP2Y PHIP9Y = -2.9762E-5  3.1097E-2 MODE/COL.
C            PHIP2Z PHIP9Z    1.8997E-4 -2.0258E-5
C
C
C      MB = MB2 = 175.126                          MODAL MASS
C           MB9   175.126
C
C
C      ZETAB = ZETAB2 = 0.005                      MODAL DAMPING
C              ZETAB9   0.005
C
C
```

```
ROUTINE

C      OMGB = OMGB2 = 0.283                          UNDAMPED NATURAL
C             OMGB9   3.110                          FREQUENCY
C
C
C
C                 1 0 0
C      IDNTA =    0 1 0                             OPERATIONAL MATRICIES
C                 0 0 1
C
C
C                 0 0 1
C      IDNTB =    1 0 0
C                 0 1 0
C
C
C                 0 1 0
C      IDNTC =    0 0 1
C                 1 0 0
C
C
C      M = 2118.8 = PRODUCT OVER SUM OF CLUSTER AND TELESCOPE MASSES
C
C
C      INERTA = 6X6 INVERSE OF EFFECTIVE INERTIA MATRIX
C
C
C
C
C      ALL INTEGRATION VARIABLES ARE INITIALLY SET TO ZERO (0.0)
C
C
C              WVT  = 0.0                            ANG. VEL VECTOR
C              PHVT = 0.0                            ATTITUDE ERROR VECTOR
C              HC   = 0.0                            COM ANG MOM - CLUSTER
C              D1D  = 0.0                            INNER GIMBAL ANG RATE
C              D3D  = 0.0                            OUTER GIMBAL ANG RATE
C              D1   = 0.0                            INNER GIMBAL ANGLE
C              D3   = 0.0                            OUTER GIMBAL ANGLE
C              ETAD = 0.0                            GEN. VELOCITY
C              ETA  = 0.0                            GEN. DISPLACEMENT
C              XID  = 0.0                            TRANS RATE - BENDING
C              XI   = 0.0                            TRANSLATION - BENDING
C
C
```

ROUTINE

```
C
C      SYMBOL DEFINITIONS
C
C
C
C           TGX                          DGCMG SYSTEM OUTPUT
C      TG = TGY                          TORQUE
C           TGZ
C
C
C      WVTD = WVD                        ANGULAR ACCEL VECTOR
C             WTD                        OF HINGED BODY SYSTEM
C
C
C             HX                         CMG ANG MOMENTUM ALONG
C      HXYZ = HY                         CLUSTER AXES
C             HZ
C
C
C           HEX                          LIMITER OUTPUT OF ANG
C      HE = HEY                          MOM ERROR ALONG CLUST.
C           HEZ                          AXES
C
C
C            HCXD                        COMMANDED RATE OF
C      HCD = HCYD                        CHANGE OF ANG MOM
C            HCZD                        ALONG CLUSTER AXES
C
C
C      GAMV                              ROT CONT DUE TO BEND
C
C      GAMVD                             ROT RATE CONT - BEND
C
C
C
C           D1                           INNER GIMBAL ANGLE
C           D3                           OUTER GIMBAL ANGLE
C
C
C           D1D                          INNER GIM  ANG RATE
C           D3D                          OUTER GIM  ANG RATE
C
C
C            D1MD                        INT. VAR INNER GIMBAL
C            D3MD                        INT. VAR OUTER GIMBAL
C
C
C
C              S31   0   C33             TRIGNOMETRIC
C      TRNA  = C31  S32   0               TRANSFORMATIONS
C               0   C32  S33              - USING REF.1 SYMBOLS
C
C
C             C31   1   S33
```

```
ROUTINE

C      TRNAC = S31 C32  1                                                    
C                1  S32 C33                                                  
C                                                                            
C                                                                            
C                S11  C12 -S13                                               
C      TRNB  =-S11  S12  C13                                                 
C                C11 -S12  S13                                               
C                                                                            
C                                                                            
C                C11 -S12 -C13                                               
C      TRNBC =-C11  C12 -S13                                                 
C               -S11 -C12  C13                                               
C                                                                            
C                                                                            
C                                                                            
C                                                                            
C      ROUTINE TO INTEGRATE VARIABLES USING EULER'S METHOD                   
C                                                                            
       ROUTINE PART1                                                      0  
C                                                                            
C      COMPUTE DERIVATIVES OF THE SYSTEM                                     
  10   DERIVE                                                             1  
C                                                                            
C      APPLY EULER'S FORMULA                                                 
       TIME = TIME+DT                                                     2  
       PHVT=PHVT+DT*WVI                                                   3  
       WVT=WVT+DT*WVTD                                                    4  
       HC=HC+DT*HCD                                                       5  
       D1=D1+DT*D1D                                                       6  
       D3=D3+DT*D3D                                                       7  
       D1D=D1D+DT*D1MD                                                    8  
       D3D=D3D+DT*D3MD                                                    9  
       ETA=ETA+DT*ETAD                                                   10  
       ETAD=ETAD+DT*ETADD                                                11  
       XI=XI+DT*XID                                                      12  
       XID=XID+DT*XIDD                                                   13  
C                                                                            
       GO TO 10                                                          14  
       END                                                               15  
```

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED

PART1	0

ROUTINE NAMES REFERENCED

DERIVE	1

UNIVERSAL ARRAYS

DT	2 3 4 5 6 7 8 9 10 11 12 13
D1	6
D1D	6 8
D1MD	8
D3	7
D3D	7 9
D3MD	9
ETA	10
ETADD	11
ETAD	10 11
HC	5
HCD	5
PHVT	3
TIME	2
WVT	3 4
WVTD	4
XI	12
XID	12 13
XIDD	13

EXTERNAL STATEMENT NUMBERS

ROUTINE PART1

10 1 14

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C THE FOLLOWING IS THE DERIVATIVE ROUTINE WHICH CALLS THREE ROUTINES	
	C IN ORDER TO EVALUATE ELASTIC BODY DYNAMICS, HINGED BODY	
	C DYNAMICS, AND THE DGCMG CONTROL SYSTEM.	
	C	
	C	
	ROUTINE DERIVE	0
	C	
	C	
	C **CALL THE DGCMG CONTROL SYSTEM ROUTINE	
	C	
	DGCMG	1
	C	
	C **CALL THE ELASTIC BODY DYNAMICS ROUTINE	
	C	
	EBDYN	2
	C	
	C **CALL THE HINGED-BODY DYNAMICS ROUTINE	
	C	
	HBDYN	3
	C	
	END	4

ROUTINE DERIVE

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
DERIVE 0

ROUTINE NAMES REFERENCED
DGCMG 1

EBDYN 2

HBDYN 3

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C DGCMG CONTROL SYSTEM EQUATIONS	
	ROUTINE DGCMG	0
	C	
	C	
	C FORM THE DIRECTION COSINE MATRICES	
	S1=$SIN(D1)	1
	C1=$COS(D1)	2
	S3=$SIN(D3)	3
	C3=$COS(D3)	4
	C THE BASIC 3X3 TRANSFORMATIONS	
	TRNA =IDNTA.D.S3+IDNTB.D.C3	5
	TRNAC=IDNTA.D.C3+IDNTB.D.S3+IDNTC	6
	C	
	TRNB =IDNTA.D.S1-IDNTB.D.S1+IDNTC.D.C1	7
	TRNBC=IDNTA.D.C1-IDNTB.D.C1-IDNTC.D.S1	8
	C	
	C FORM ROTATIONS AND RATES DUE TO BENDING	
	GAMV =PHIP*ETA	9
	GAMVD=PHIP*ETAD	10
	C	
	C FORM PHI AND OMEGA SUB R	
	PHV=PHVT(1,1,3,1)	11
	PHT=PHVT(4,1,6,1)	12
	WV=WVT(1,1,3,1)	13
	WT=WVT(4,1,6,1)	14
	PHR=PHV+GAMV	15
	WR =WV +GAMVD	16
	C	
	C RESOLVE ANGULAR MOMENTUM COMPONENTS TO CLUSTER AXES	
	HXYZ=(TRNAC.*.TRNBC)*H	17
	C	
	C GET TOTAL CMG CONTROL TORQUE ABOUT CLUSTER AXES	
	TG=(TRNAC.*.TRNB)*(H.*.D1D)+(TRNA.D.C1)*(H.*.D3D)+HXYZ.X.WR	18
	C	
	C FORM LIMITER INPUT COMPONENTS OF COMMAND RATE OF CHANGE OF H	
	HCD=K0.*.PHR+K1.*.WR	19
	C	
	C APPLY MAXIMUM LIMITS	
	LIMIT HCD,HCMAXD,-HCMAXD	20
	C	
	C TURN OFF IF HC TOO BIG	
	STOPIT HCD,HC.HCMAX,-HCMAX	21
	C	
	C GET MOMENTUM ERRORS	
	HE=HC-HXYZ	22
	C	
	C APPLY LIMITS	
	LIMIT HE,HEMAX,-HEMAX	23
	C	
	C GET DERIVATIVE OF GIMBAL ANGLES	
	D1DC= -KSL*(TRNAC.*.TRNB)...HE	24
	D3DC= -KSL*TRNA...HE	25

```
ROUTINE  DGCMG

  C
  C       APPLY LIMITS AND RATE STOPS                                    26
          LIMIT D1DC,DCMAXD,-DCMAXD                                      27
          LIMIT D3DC,DCMAXD,-DCMAXD
  C                                                                      28
          STOPIT D1DC,D1,D1MAX,-D1MAX                                    29
          STOPIT D3DC,D3,D3P,D3N
  C
  C       FORM INTERMEDIATE ANGULAR RATES                                30
          FACTOR=$EXP(-TIME/TAUM)/TAUM                                   31
          D1MD=FACTOR*D1DC                                               32
          D3MD=FACTOR*D3DC
  C                                                                      33
          END
```

ROUTINE DGCMG

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
DGCMG 0

ROUTINE NAMES REFERENCED
COS 2 4

EXP 30

LIMIT 20 23 26 27

SIN 1 3

STOPIT 21 28 29

UNIVERSAL ARRAYS
C1 2 7 8 18

C3 4 5 6

DCMAXD 26 27

D1 1 2 28

D1D 18

D1DC 24 26 28 31

D1MAX 28

D1MD 31

D3 3 4 29

D3D 18

D3DC 25 27 29 32

D3MD 32

D3N 29

D3P 29

ETA 9

ETAD 10

ROUTINE	DGCMG			
FACTOR	30	31	32	
GAMVD	10	16		
GAMV	9	15		
H	17	18		
HC	21	22		
HCD	19	20	21	
HCMAX	21			
HCMAXD	20			
HE	22	23	24	25
HEMAX	23			
HXYZ	17	18	22	
IDNTA	5	6	7	8
IDNTB	5	6	7	8
IDNTC	6	7	8	
KSL	24	25		
KO	19			
K1	19			
PHIP	9	10		
PHR	15	19		
PHT	12			
PHV	11	15		
PHVT	11	12		
S1	1	7	8	
S3	3	5	6	
TAUM	30			
TG	18			
TIME	30			

ROUTINE	DGCMG			
TRNAC	6	17	18	24
TRNA	5	18	25	
TRNBC	8	17		
TRNB	7	18	24	
WR	16	18	19	
WT	14			
WV	13	16		
WVT	13	14		

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C ELASTIC BODY DYNAMICS	
	ROUTINE EBDYN	0
	C	
	C OBTAIN ETADD AND XIDD	
	C	
	ETADD = (PHIP ...TG) / MB - 2 * ZETAB .*. OMGB .*. ETAD	1
	1 - OMGB ** 2 .*. ETA	
	XIDD = PHI * ETADD	2
	C	
	C DONE	
	C	
	END	3

ROUTINE EBDYN

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
EBDYN 0

UNIVERSAL ARRAYS
ETA 1

ETADD 1 2

ETAD 1

MB 1

OMGB 1

PHI 2

PHIP 1

TG 1

XIDD 2

ZETAB 1

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C HINGED-BODY DYNAMICS	
	ROUTINE HBDYN	0
	C	
	SKEW RVBAR,RV	1
	SKEW RTBAR,RT	2
	SKEW WVBAR,WV	3
	SKEW WTBAR,WT	4
	C	
	C	
	TXI(1,,3,) = -M * RVBAR * (PHI * ETADD + 2 * WVBAR * PHI * ETAD)	5
	TXI(4,,6,) = M * RTBAR * (PHI * ETADD + 2 * WTBAR * PHI * ETAD)	6
	WVTD=INERTA*(TG(1,1,6,1)+TXI)	7
	C	
	C DONE	
	C	
	END	8

ROUTINE HBDYN

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
HBDYN 0

ROUTINE NAMES REFERENCED
SKEW 1 2 3 4

UNIVERSAL ARRAYS
ETADD 5 6
ETAD 5 6
INERTA 7
M 5 6
PHI 5 6
RT 2
RTBAR 2 6
RV 1
RVBAR 1 5
TG 7
TXI 5 6 7
:T 4
WTBAR 4 6
WV 3
WVBAR 3 5
WVTD 7

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C ROUTINE TO LIMIT EXCURSION OF A VARIABLE	
	ROUTINE LIMIT (A,UPRL,LWRL)	0
	DO 20 I=1,3	1
	IF (A(I).GT.UPRL) A(I)=UPRL	2
20	IF (A(I).LT.LWRL) A(I)=LWRL	3
	END	4

ROUTINE LIMIT

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
LIMIT 0

DUMMY NAMES USED FOR ARGUMENTS
A 0 2 3

LWRL 0 3

UPRL 0 2

UNIVERSAL ARRAYS
I 1 2 3

EXTERNAL STATEMENT NUMBERS
20 1 3

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
	C ROUTINE TO LIMIT DERIVATIVE OF A VARIABLE WHICH IS TOO LARGE	
	ROUTINE STOPIT (AD,A,UPRL,LWRL)	0
	DO 50 I=1,3	1
	IF (AD(I)) 30,50,	2
	IF (A(I).LE.UPRL) GO TO 50	3
	GO TO 40	4
30	IF (A(I).GE.LWRL) GO TO 50	5
40	AD(I)=0	6
50	CONTINUE	7
	END	8

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
STOPIT 0

DUMMY NAMES USED FOR ARGUMENTS
A 0 3 5
AD 0 2 6
LWRL 0 5
UPRL 0 3

UNIVERSAL ARRAYS
I 1 2 3 5 6

EXTERNAL STATEMENT NUMBERS
50 1 2 3 5 7
40 4 6
30 2 5

EXTERNAL STATEMENT NUMBER	MATE STATEMENT	MATE REFERENCE NUMBER
C	ROUTINE TO GENERATE 3X3 SKEW-SYMMETRIC MATRIX	
	ROUTINE SKEW (SS,V)	0
	SS=IDNTA.X.V	1
	END	2

ROUTINE SKEW

CROSS REFERENCE LIST - ROUTINE NAMES, ARRAY NAMES, AND EXTERNAL STATEMENT NUMBERS VS
A LIST OF MATE REFERENCE NUMBERS WHICH MARK THE STATEMENTS IN WHICH THEY OCCUR

ROUTINE NAMES DEFINED
SKEW 0

DUMMY NAMES USED FOR ARGUMENTS
SS 0 1

V 0 1

UNIVERSAL ARRAYS
IDNTA 1

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	1.00000E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY DI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY D1D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.05500E-02
2	3.05500E-02
3	3.05500E-02

ARRAY D3D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.05500E-02
2	3.05500E-02
3	3.05500E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY XID

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	2.00000E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.0

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	2.42453E-06
2	3.41901E-07
3	2.05840E-07
4	5.02757E-08
5	-1.70473E-07
6	2.05840E-07

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY D1

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.05500E-04
2	3.05500E-04
3	3.05500E-04

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.05500E-04
2	3.05500E-04
3	3.05500E-04

ARRAY D10

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	4.90795E-02
2	4.90795E-02
3	4.90795E-02

ARRAY D30

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	4.90795E-02
2	4.90795E-02
3	4.90795E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-4.48334E-05
2	2.93935E-04

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	0.0
2	0.0
3	0.0

ARRAY XID

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.78712E-05
2	-1.20731E-06
3	1.02967E-06

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	3.00000E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	2.42453E-08
2	3.41901E-09
3	2.05840E-09
4	5.02757E-10
5	-1.70473E-09
6	2.05840E-09

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	6.32037E-06
2	8.91345E-07
3	5.36693E-07
4	1.31058E-07
5	-4.44375E-07
6	5.36693E-07

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.05628E-02
2	1.89674E-01
3	6.37257E-03

ARRAY DI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.96294E-04
2	7.96294E-04
3	7.96294E-04

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.96294E-04
2	7.96294E-04
3	7.96294E-04

ARRAY D1D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	6.03182E-02
2	6.03182E-02
3	6.03182E-02

ARRAY D3D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	6.03182E-02
2	6.03182E-02
3	6.03182E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-4.48334E-07
2	2.93935E-06

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-1.16873E-04
2	7.66214E-04

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.78711E-07
2	-1.20731E-08
3	1.09967E-08

ARRAY XID

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	9.87206E-05
2	-3.14725E-06
3	2.86656E-06

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	4.00000E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	8.74489E-08
2	1.23325E-08
3	7.42532E-09
4	1.81334E-09
5	-6.14847E-09
6	7.42532E-09

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	1.11101E-05
2	1.56685E-06
3	9.43593E-07
4	2.30380E-07
5	-7.80914E-07
6	9.43593E-07

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.81521E-02
2	6.84489E-01
3	2.30074E-02

ARRAY D1

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.39948E-03
2	1.39948E-03
3	1.39948E-03

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.39948E-03
2	1.39948E-03
3	1.39948E-03

ARRAY D10

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	6.71348E-02
2	6.71348E-02
3	6.71348E-02

ARRAY D30

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	6.71348E-02
2	6.71348E-02
3	6.71348E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-1.61706E-06
2	1.06015E-05

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-2.05440E-04
2	1.34653E-03

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.36592E-06
2	-4.35456E-08
3	3.96623E-08

ARRAY XID

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.73490E-04
2	-5.53231E-06
3	5.03761E-06

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW
1 5.00000E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW
1 1.98550E-07
2 2.80009E-08
3 1.68613E-08
4 4.11713E-09
5 -1.39576E-08
6 1.68613E-08

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW
1 1.64437E-05
2 2.31892E-06
3 1.39685E-06
4 3.40970E-07
5 -1.15518E-06
6 1.39685E-06

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW
1 8.67441E-02
2 1.55478E 00
3 5.22900E-02

ARRAY D1

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW
1 2.07082E-03
2 2.07082E-03
3 2.07082E-03

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	2.07082E-03
2	2.07082E-03
3	2.07082E-03

ARRAY D1D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.12693E-02
2	7.12693E-02
3	7.12693E-02

ARRAY D3D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.12693E-02
2	7.12693E-02
3	7.12693E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-3.67146E-06
2	2.40668E-05

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-3.04061E-04
2	1.99198E-03

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.10081E-06
2	-9.88687E-08
3	9.00383E-08

ARRAY XTD

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	2.56650E-04
2	-8.18818E-06
3	7.45222E-06

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	6.00000E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	3.62987E-07
2	5.11901E-08
3	3.08297E-08
4	7.52703E-09
5	-2.55094E-08
6	3.08297E-08

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	2.21088E-05
2	3.11745E-06
3	1.87844E-06
4	4.58498E-07
5	-1.55189E-06
6	1.87844E-06

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	1.58816E-01
2	2.84333E 00
3	9.56945E-02

ARRAY D1

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	2.78352E-03
2	2.78352E-03
3	2.78352E-03

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	2.78352E-03
2	2.78352E-03
3	2.78352E-03

ARRAY D1D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.37770E-02
2	7.37770E-02
3	7.37770E-02

ARRAY D3D

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.37770E-02
2	7.37770E-02
3	7.37770E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-6.71207E-06
2	4.39865E-05

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-4.08807E-04
2	2.67624E-03

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	5.66731E-06
2	-1.80750E-07
3	1.64561E-07

ARRAY XID

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.44811E-04
2	-1.10092E-05
3	1.00119E-05

ARRAY TIME

DEFINED SIZE 1 ROWS BY 1 COLUMNS

ROW	
1	6.99999E-02

ARRAY PHVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	5.84075E-07
2	8.23646E-08
3	4.96142E-08
4	1.21120E-08
5	-4.10283E-08
6	4.96142E-08

ARRAY WVT

DEFINED SIZE 6 ROWS BY 1 COLUMNS

ROW	
1	2.77767E-05
2	3.94408E-06
3	2.37745E-06
4	5.80247E-07
5	-1.96159E-06
6	2.37745E-06

ARRAY HC

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	2.55926E-01
2	4.57606E 00
3	1.54142E-01

ARRAY D1

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.52129E-03
2	3.52129E-03
3	3.52129E-03

ARRAY D3

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	3.52129E-03
2	3.52129E-03
3	3.52129E-03

ARRAY D10

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.52980E-02
2	7.52980E-02
3	7.52980E-02

ARRAY D30

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	7.52980E-02
2	7.52980E-02
3	7.52980E-02

ARRAY ETA

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-1.08091E-05
2	7.07489E-05

ARRAY ETAD

DEFINED SIZE 2 ROWS BY 1 COLUMNS

ROW	
1	-5.17295E-04
2	3.38306E-03

ARRAY XI

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	9.11542E-06
2	-2.90842E-07
3	2.64680E-07

ARRAY XID

DEFINED SIZE 3 ROWS BY 1 COLUMNS

ROW	
1	4.35880E-04
2	-1.39312E-05
3	1.26558E-05

APPENDIX H

DECK SET-UP FOR IBM 7090/94 VERSION OF MATE

The deck set-up required to run MATE as a "batch" job under control of the IBM 7090/7094 IBSYS Operating System Version 13 is discussed here. To reduce the number of cards that must be handled by the user, this set-up will utilize an Alternate Input tape read under control of the IBJOB Processor. This Alternate Input unit contains two files. The MATE translator object decks reside in the first file while the second file contains object decks for the MATE executor. We assigned this tape as SYSBL3 on say, B8.

Required Tape Units

Other tapes required by the translator are:

Fortran File Number	System Unit Name
5	SYSIN1
6	SYSOU1
2	SYSUT1
3	SYSUT5
4	SYSUT6
7	SYSCK1
8	SYSUT4

In addition to the above assignments of Fortran File numbers made by the MATE translator, we use the following units for Overlay link storage:

SYSUT2

SYSUT3

The output of the translator (the generated Fortran source decks) are written on SYSCK1.

The tapes used by the executor are:

Fortran File Numbers	System Unit Name
5	SYSIN1
6	SYSOU1
7	SYSCK1

In addition, the executor requires from three to twenty-five units for storage of arrays that won't fit in-core. At the present time, four units are assigned:

Fortran File Numbers	System Unit Name
2	SYSUT1
3	SYSUT5
4	SYSUT6
8	SYSUT4

Again, we have a unit assigned to hold Overlay links, namely:

SYSUT2

Now of the above System Unit names, all but perhaps SYSUT5, SYSUT6, and SYSCK1 will have standard unit assignments. Should these three not have standard assignments $ATTACH, $AS cards may be included in the deck to assign them. As an example of this in the deck set-ups we shall presume that the following system names do not have standard unit assignments and that the corresponding units are available:

System Unit Name	Channel/Unit Designation
SYSLB3	B8
SYSCK1	A5
SYSUT5	B5
SYSUT6	B6

Control Card Decks

Five decks of system control cards are furnished to run MATE. We shall label these decks T1, T2, TE, E1 and E2. For the sake of complete documentation the contents of the leading card columns are described below.

DECK T1

```
$IBSYS
$ATTACH          B8
$AS              SYSLB3
$ATTACH          A5
$AS              SYSCK1
$ATTACH          B5
$AS              SYSUT5
$ATTACH          B6
$AS              SYSUT6
$EXECUTE         IBJOB
$IBJOB           GO, NOLOGIC, NOMAP, FLOW
$IEDIT           SYSLB3, SCHF1
$IBLDR
  .
  .
  .
$IBLDR
$IEDIT
$DATA
```

DECK T2

```
        END
7 over 8                    END-OF-FILE CARD
$EXECUTE                    IBJOB
$IBJOB                      NOGO, NOLOGIC, NOMAP
```

DECK TE

```
        END
7 over 8                    END-OF-FILE CARD
$EXECUTE                    IBJOB
$IBJOB                      GO, NOLOGIC, MAP, FLOW
```

NOTE that the only difference between the T2 and TE decks is the option field on the $IBJOB card. Deck TE may be substituted for deck T2 at the expense of a few extra seconds of machine time and a page of loader error messages which can be ignored.

DECK E1

```
$IBSYS
$ATTACH                     B8
$AS                         SYSLB3
$ATTACH                     A5
$AS                         SYSCK1
$ATTACH                     B5
$AS                         SYSUT5
$ATTACH                     B6
$AS                         SYSUT6
$EXECUTE                    IBJOB
$IBJOB                      GO, NOLOGIC, MAP, FLOW
```

DECK E2

```
$IEDIT                    SYSLB3, SCHF2
$IBLDR
  .
  .
  .
$IBLDR
$IEDIT
$DATA
```

Deck Set-Up

Finally we present the deck set-up for three types of runs: translation only, execution only, and translation and execution.

Translation only job

1. DECK T1
2. User's MATE source decks to be translated
3. DECK T2
4. Call deck for user's routines

Execution only job

1. DECK E1
2. Object (binary) decks of previously translated MATE routines used in this job plus any special Fortran subroutines required.
3. DECK E2
4. The initial standard data block (reference Chapter 3) followed by any other data blocks required in the run.

Translate and Execute Job

1. DECK T1
2. User's MATE source decks to be translated.
3. DECK TE
4. Call deck for user's routines.
5. Object (binary) decks of any previously translated MATE routines used in this job plus any special Fortran subroutines required.
6. DECK E2
7. The initial standard data block followed by any other data blocks required in the run.

The decks other than T1, T2, TE, E1, and E2 are now defined.

The User MATE source decks are the MATE routines as defined in this report. The first statement of each routine must be a ROUTINE definition statement which names the routine. This statement may be preceded by comment cards. The last card in each routine must be an END statement. There are no special cards between routines. The end of the translator input is marked by two back-to-back END cards. The first of these is the last card of last routine. The second END card is the leading card in deck T2 or TE.

The MATE translator writes its output on SYSCK1. To get this output into the Fortran IV compiler a deck known as the call deck for user's routines is required. The user must furnish this deck. it takes the form:

```
1                   8                   16          card column
|                   |                   |
$IEDIT                                  SYSCK1, SRCH

$IBFTC              name 1              DECK

$IBFTC              name 2              DECK

$IBFTC              name 3              DECK
   :
   :
$IBFTC              name N              DECK

$IEDIT
```

Where "name 1" etc., are the names assigned to the routines by the ROUTINE statements. There must be one $IBFTC card for each routine translated. The routine name must start in card column 8. And the $IBFTC cards must be sandwiched between $IEDIT cards as shown.

After the routines have been fed thru the Fortran compiler, binary object decks are obtained. These decks may be saved for future execution of the routines to eliminate the need for retranslating and recompiling.

The data blocks are as described in Chapter 3.