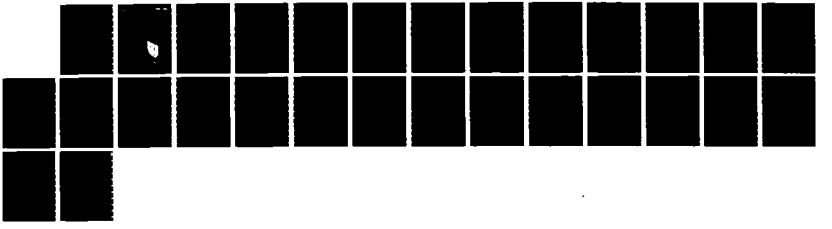
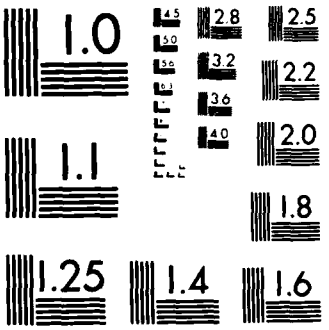
AD-A171 701 MAT: AN INTERACTIVE CALCULATOR FOR MATRIX OPERATIONS DESCRIPTION AND APPL..(U) CONSTRUCTION ENGINEERING RESEARCH LAB (ARMY) CHAMPAIGN IL K M BLOOMQUIST AUG 86 1/1

UNCLASSIFIED CERL-SR-N-86/16 F/G 5/3 NL





US Army Corps
of Engineers
Construction Engineering
Research Laboratory

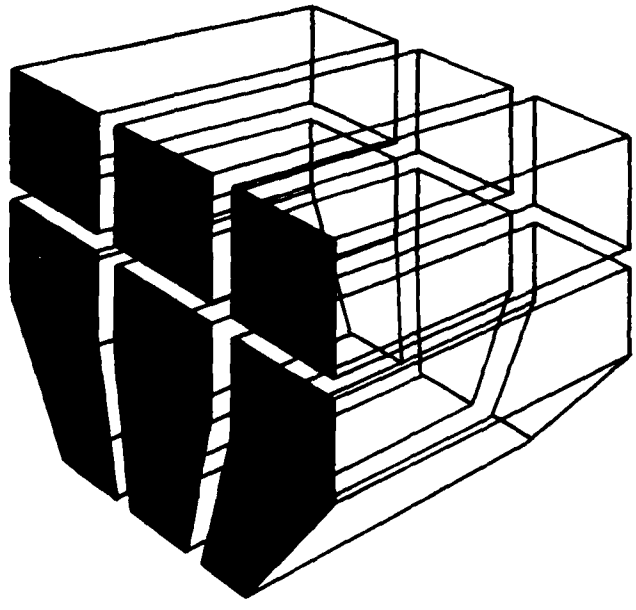
USA-CERL

SPECIAL REPORT N-86/16
August 1986

Mat -- An Interactive Calculator for Matrix Operations:

Description and Application Examples

by
Kim M. Bloomquist



This report describes the commands, operators, and functions of mat—an interactive interpreter and calculator for solving systems of equations expressed as matrices. Also provided are examples that show how the system is used to solve problems in matrix algebra, input-output analysis, and regression analysis. Mat features a full range of matrix and scalar operators and built-in functions for carrying out numeric and nonnumeric matrix transformations. It also provides checks for errors in conformability and other run-time problems.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CERL SR N-86/16	2. GOVT ACCESSION NO. ADA171701	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MAT--AN INTERACTIVE CALULATOR FOR MATRIX OPERATIONS: DESCRIPTION AND APPLICATION EXAMPLES		5. TYPE OF REPORT & PERIOD COVERED Final 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kim M. Bloomquist		8. CONTRACT OR GRANT NUMBER(s) MIPR F82-29 Work Unit N-92
9. PERFORMING ORGANIZATION NAME AND ADDRESS U. S. Army Construction Engr Research Laboratory P. O. Box 4005 Champaign, IL 61801-1305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFESC Tyndall AFB, FL 32403		12. REPORT DATE August 1986 13. NUMBER OF PAGES 26
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Copies are available from the National Technical Information Service
Springfield, VA 22161

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

mat
matrices (mathematics)
Comprehensive Economic Analysis System (CEAS)
computer program documentation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes the commands, operators, and functions of **mat**--an interactive interpreter and calculator for solving systems of equations expressed as matrices. Also provided are examples that show how the system is used to solve problems in matrix algebra, input-output analysis, and regression analysis. **Mat** features a full range of matrix and scalar operators and built-in functions for carrying out numeric and nonnumeric matrix transformations. It also provides checks for errors in conformability and other run-time problems.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

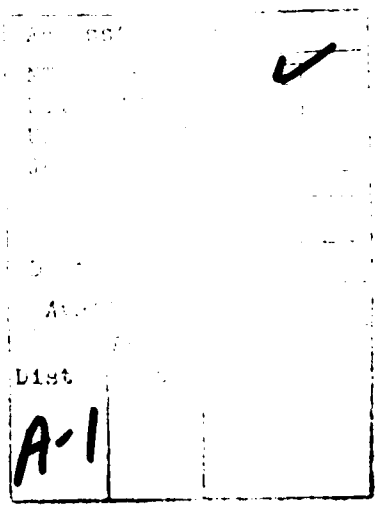
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

FOREWORD

This research was conducted for the U. S. Air Force Engineering and Services Center (AFESC), Tyndall AFB, FL, under MIPR F82-29, Work Unit N-92, "The Local Economic Consequence System." The work was performed by the Environmental Division (EN), U. S. Army Construction Engineering Research Laboratory (USA-CERL). Mr. W. Alan Nixon (AFESC/DEV) was the AFESC Technical Monitor.

The author is grateful to both D. Robinson and M. Shapiro of USA-CERL for their constructive comments and suggestions during development of the program.

Mr. Robert Riggins is Acting Chief of USA-CERL-EN. COL Norman C. Hintz is Commander and Director of USA-CERL, and Dr. L. R. Shaffer is Technical Director.



CONTENTS

MAT—AN INTERACTIVE CALCULATOR FOR MATRIX OPERATIONS: DESCRIPTION AND APPLICATION EXAMPLES

1 INTRODUCTION

Background

The military conducts many types of economic analyses for a variety of projects, and putting together these analyses often requires the use of matrices for solving systems of equations. Many computerized matrix packages are available, but most require the user to write subroutines, which makes operating these programs more complicated. Therefore, to make matrix programs easier for military economists to use, the U. S. Army and Air Force developed **mat**--an interactive interpreter and calculator for solving systems of equations expressed as matrices.

Mat is part of the Comprehensive Economic Analysis System (CEAS) maintained and operated by the U.S. Army Construction Engineering Research Laboratory (USA-CERL). CEAS is designed for regional economic modeling. It combines data, models, and tools for conducting various types of regional impact studies, including input-output analysis, economic-base analysis, and impact area definition. CEAS is available to users as a subsystem of the Environmental Technical Information System (ETIS)[1]--a comprehensive computer-aided methodology for gathering and storing environment-related information and helping environmental planners assess potential impacts of military projects and activities.

Mat was designed to complement many of the programs in CEAS by allowing users to conduct simulation and gaming exercises using scaled-down regional input-output tables prior to carrying out formal studies of regional economies. **Mat** is also a valuable teaching tool for introducing students of regional economics to the fundamental economic relationships often expressed in terms of matrix algebra. With this aim in mind, several specialized routines have also been developed for **mat**, including a function for calculating the Leontief Inverse $(I - A)^{-1}$, the Power Series Approximation to the Leontief Inverse, and a procedure for updating an input-output table using the RAS adjustment method. However, users not interested in doing economic studies should also find **mat** useful for other applications requiring the manipulation of data stored as matrices.

Mat allows a user to express matrix operations more naturally. For example, the command to multiply two matrices, a and b, is,

```
a * b
```

More complicated relationships are also easy to construct. For example, to derive the coefficients for a set of linear regression normal equations, the expression

[1] R. D. Webster, et al., *Modification and Extension of the Environmental Technical Information System (ETIS) for the Air Force*, Special Report N-81/ADA079441 (U. S. Army Construction Engineering Research Laboratory [USA-CERL], 1979).

$$b = (X'X)^{-1} X' Y$$

in **mat** is:

$$b = inv\ (x' * x) * x' * y$$

The main advantage of **mat** is its ease in expressing matrix relationships; however, it also does extensive error checking. Checks for conformability, syntax, and overflow are built-in features. All computations are done by double precision arithmetic.

Objective

The objective of this report is to provide the user of **mat** with a fundamental knowledge of the system's commands, operators, and functions, and to provide examples of practical problem solving using the system.

Approach

The commands used to operate **mat** are defined, and the various operators and functions available to carry out operations on matrices are described. Examples are provided that illustrate how **mat** can be used to solve problems in matrix algebra, input-output analysis, and regression analysis.

Mode of Technology Transfer

Mat may be accessed through the University of Illinois' ETIS Support Center. Users may obtain a login from the Center by calling 217-333-1369.

2 COMMANDS

Mat is implemented on a Pyramid computer running under OSx 2.5 (Berkeley 4.2 and System V UNIX*). The program, which is written in YACC,[2] reads input typed at a computer terminal and interprets these characters as a command to be executed by another program. A library of functions written in C carries out the actual matrix operations.[3]

This chapter describes what the user should know to begin using **mat.** Throughout the remainder of this report, input commands are represented in bold-face letters, while program output is shown as a user might see it on his/her computer terminal.

Variables

Twenty-six registers, represented by the lower-case letters **a...z**, are reserved for storing matrix variables, and 26 registers, represented by the upper-case letters **A...Z**, are reserved for storing scalar values. Once defined, these variables may be used freely in any valid expression.

Help

A help command **?** with the following options is provided.

Typing a single question mark on a line

```
1> ?
```

prints a document summarizing the various features and commands available in **mat** following the style of a UNIX user manual page. Typing a question mark followed by a matrix variable name prints the dimensions of the variable (if defined) in the format M rows x N columns.

A question mark followed by a dollar sign "$" prints all the currently defined matrix variables and their dimensions. For example,

```
5> ?$
```

might produce the following output:

```
a: 10 x 10
i: 10 x 10
u: 1 x 10
v: 10 x 1
```

*UNIX is a trademark of Bell Laboratories.

S. J. Johnson, YACC: *Yet Another Compiler-Compiler* (Bell Laboratories, 1978). See also the *UNIX Programmer's Manual Supplementary Documents* (University of California at Berkeley, 1984).

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Prentice-Hall, Inc., 1978).

This says that four variables are currently defined: two matrices, **a** and **i**, having 10 rows and 10 columns each, a vector, **u**, with one row and 10 columns; and a vector, **v**, having 10 rows and one column. The symbol ">" is the program prompt, which indicates that **mat** is ready to accept input from the keyboard and the number preceding it (e. g., 5>) is the current command line number.

Comments

Any line beginning with the "#" character is interpreted as a comment. For example: **# This is a comment.** Such lines are ignored when **mat** is used in interactive mode. However, when commands are read from a file, comments are printed to the standard output and can be used to label output. Noninteractive use of **mat** is covered in the tutorial introduction provided in Chapter 4.

Input and Output

Matrix variables are initialized or defined either by reading the contents of a file in which a matrix has been stored previously or by assigning a variable to an array typed on the terminal. Exceptions are the identity matrix and unity vector which may be created using the **id()** and **uv()** functions, respectively (explained on p 14).

The **read()** function is used to read a matrix from a file.

```
1> read(a,nrows,ncols, "myfile" )
```

This command defines a variable **a** to be a matrix having the dimensions **nrows** rows and **ncols** columns; the matrix is stored in file "myfile."

File names must be surrounded by double quotes " " and may contain letters, numbers, dots ".", underscores " ", or slashes "/". The length of file names is restricted by the UNIX system to 14 or fewer characters. A file name containing one or more "/" characters is interpreted as a full pathname (e. g., "/usr/smith/data") and may be up to 99 characters long.

Matrices may also be assigned values typed directly at a user's terminal. The format for this procedure is shown in the following example.

```
1> a = [.25 .5 .15 .75] 2 2
```

This defines "**a**" to be a 2 x 2 matrix containing the elements between the square brackets. There are four important points to remember when defining a matrix in this way:

1. Elements within the square brackets must be separated by space or tab characters.

2. Matrix elements are assigned row by row. That is, in an M x N matrix, the first N elements within the brackets make up the first row. If the matrix dimensions following the right bracket exceed the number of elements, the remaining elements are assigned the value zero. If, instead, the following had been typed,

1> **a** = [.25 .5 .15 .75] 3 2

the first two rows of "**a**" would contain the elements within the brackets and the third row would contain all zeros. As a special case,

1> **a** = [] **M N**

defines an M x N null matrix.

3. A maximum of 100 elements can be assigned to a matrix when input is read from a terminal. Matrices containing more than 100 elements must be read from a file.

4. A newline character (linefeed or carriage return) typed before the rightmost dimension is a syntax error.

Output is automatically displayed on the user's terminal following execution of any valid expression. To see the contents of "**a**" just type its name alone on a line followed by a carriage return.

```
2> a
0.250000        0.500000
0.150000        0.750000
```

The contents of a variable may be written to a file using the **put()** function.

2> **put(a, "myfile")**

This writes the contents of "**a**" to file "myfile." The same restrictions on file names mentioned on p 8 also apply here.

Output from an expression may also be redirected to a file rather than being displayed on the terminal. This may be useful when working with larger matrices that would be difficult to read if printed on an 80-character width screen or if one is using a terminal with a slow baud rate (< 1200 baud). The command to redirect output to a file is

2> **a > "myfile"**

In this case, the contents of variable "**a**" are saved in file "myfile." The ">" symbol is used to indicate redirected output, which is familiar to UNIX users. When redirecting output from a more complicated expression, the left-hand side must be surrounded by parentheses, such as

2> **(a * b) > "myfile"**

A final way to print output is to use the **putf()** (formatted put) function. An example is

2> **putf(a,15,8, "myfile")**

This will print the contents of "**a**" to "myfile" with a minimum of 15 positions used to display each array element and eight digits displayed to the right of the decimal point. As a special case, the file name "term," when used with **putf()**, will print the matrix elements on the user's terminal.

Size Limitations

The current implementation of **mat** requires matrices to be no larger than 100 rows by 100 columns. In an interactive environment, operations on very large matrices are wasteful of computer resources. Programs written for specific tasks can handle these applications much more efficiently.

Getting out of Mat, Interrupts, and Suspending Output

To exit **mat**, type "**quit**". This will cause an immediate exit from the program, erasing any variables the user has defined (except those previously saved in files). To suspend output on the terminal (i. e., to inspect results), type control-S; to restart output, type control-Q. Hitting the interrupt key will cause **mat** to stop what it is doing (e. g., printing out a large matrix) and provide a new prompt (>). On some terminals, this may be the "rubout" key; on others, control-C will cause an interrupt signal to be sent.

UNIX System Interface

The **mat** user can communicate easily with the operating system. Any line that begins with the character "!" is passed to the operating system for execution. This facility can be used to list files, enter a text editor, etc. UNIX system commands (e. g., text editors) that interact with the user return to **mat** when the interaction is complete.

```
1> !ed myfile
...edit file myfile with text editor...
q
2> # ready for next mat expression
```

3 OPERATORS AND FUNCTIONS

This chapter describes the various operators (Table 1) and functions available in **mat** to carry out operations on matrices. A brief description and an example of each is given. Chapter 4 provides a tutorial introduction that demonstrates how these operators and functions may be combined to solve real problems in matrix algebra, input-output analysis, and regression analysis.

In the following discussion, "**a**" and "**b**" are assumed to be matrices that have been defined previously and are compatible for use in the specified operations.

Assignments

The = sign assigns the contents of one matrix to another or sets a variable equal to the results returned by an expression. This assigns to the left-hand side the result contained by the right-hand side. For example, **b = a**, and **b = expr.** (**expr** is any valid combination of variables, operators, and/or functions.) Also, because the assignment operator itself returns an expression, multiple assignments are possible (e. g., **c = b = a**). The result of an assignment is not printed on the user's terminal.

Matrix Operators

Addition and Subtraction

Matrix addition and subtraction are denoted by the + and - operators (for example, **a + b** and **a - b**). As previously mentioned, **mat** checks for conformability before attempting to carry out the operation.

Multiplication

The operator for matrix multiplication is *. An example is: **a * b.** Again, conformability is checked.

Division by Column and Row Sums

Division by column sums and division by row sums are two related operations that are often useful in input-output analysis. The operator for division by column sums is represented by /, while that for division by row sums is \. For example, to divide the

Table 1

Operators, in Decreasing Order of Precedence

'	transpose
^	exponentiation (FORTRAN **), right associative
-	(unary) minus
* / \	multiplication, division, division by column and row sums
+ -	addition and subtraction
=	assignment, right associative

elements of "**a**" by its column sums "**b**", type **a / b**. Similarly, for division by row sums, type **a \ b**. The section describing matrix functions (pp 13-14) mentions two functions - **csum()** and **rsum()**--that return the column and row sums of their arguments. Thus, instead of the above expression for division by column sums, one could substitute the results of a function call; e. g., **a / csum(a)**.

Transpose

Mat uses the ' sign to indicate the need to transpose a matrix or a vector. Although a function is actually called to produce a transposed matrix, the ' operator is a true operator in the sense that the transpose operation has a higher order of precedence than any of the operators for +, -, *, /, or \. To take the transpose of "**a**" and assign it to "**b**", the command is: **b = a'**.

Parentheses

Precedence can be explicitly determined by surrounding an expression with parentheses. Normally, **mat** will interpret an expression like **a - b - c** as **(a - b) - c**. If, instead, the result should be interpreted as **a - (b - c)**, the user must put parentheses around **(b - c)**.

Scalar Operators

Mat also supports combined operations with matrices and scalars. (Note: Scalars are defined as numbers containing a decimal point.) Operations with whole numbers that lack a trailing decimal point are treated as a syntax error. Scalar values may be assigned to any single upper-case letter. In the following discussion, the symbol **K** is assumed to be a constant scalar (e. g., **K** = 3.14159).

Addition and Subtraction

The operations of adding a scalar to a matrix and subtracting a scalar from a matrix are handled as expected: **a + K** and **a - K**. Both pre- and post-scalar operations are recognized, so that **K + a** is equal to **a + K**.

Multiplication and Division

Scalar multiplication and division are **a * K** and **a / K**, respectively.

Exponentiation

The exponentiation operator is ^. For example, to raise each element of "**a**" to the power of 2, type **a ^ 2.0**.

Unary Minus

Mat also recognizes the unary operator, -, so that an expression such as **a - -1.0** (add 1.0 to each element of **a**) is interpreted correctly.

Functions

Inverse

The **inv()** function is used to produce the inverse of matrix "**a**". For example, to take the inverse of "**a**" and assign it to "**b**, type **b = inv(a)**. This function has several error-checking mechanisms, including checks for conformability and singularity.

Determinants

The **det()** function returns the determinant of a matrix in double precision form. An example of its use is **det(a)**. The argument to **det()** must be a square matrix.

Diagonalization

Diagonalizing a vector implies the creation of a square matrix having the same number of rows and columns as elements in the original vector, and placing the elements of the vector on the main diagonal with zero in all the off-diagonal elements. The function that carries out this operation is **diag()**; e. g., **diag(a)**.

Trace

The trace of a square matrix is the sum of its main diagonal elements. The command to calculate the trace of a matrix "**a**" is **tr(a)**.

Norm

The norm of a matrix is defined as the largest of the column sums of the absolute values of the elements in a matrix. The **norm()** function returns the norm of its matrix argument in double precision form; e. g.,

$$\mathbf{norm(a)} = \max_{k} \sum_{i=1}^{n} |a_{ij}|$$

Minimum

The **min()** function returns the smallest element of a matrix in double precision form; e. g., **min(a)**.

Maximum

The **max()** function returns the largest element of a matrix in double precision form; e. g., **max(a)**.

Element

The **el()** function returns an element of a matrix in double precision form. For example, **el(a,3,1)** returns element a_{31}.

Column Sums

The **csum()** function returns a 1 x N vector, each element of which is the sum of the jth column elements of its matrix argument; e. g., **csum(a)**.

Row Sums

The **rsum()** function returns an N x 1 vector, each element of which is the sum of the ith row elements of its matrix argument; e. g., **rsum(a)**.

Identity Matrix

An identity matrix has each cell of the main diagonal equal to 1 and zero in all the off-diagonal cells. Such a matrix is useful in transformation operations and is integral in solving systems of equations in input-output analysis. An identity matrix may be produced using the **id()** function. For example, **id(3)** generates a 3 x 3 identity matrix.

Unity Vector

A unity vector's elements are all equal to 1. Unity vectors are also useful in certain transformations. The command **uv(3)** creates a 3 x 1 unity vector.

Test for Symmetry

To determine if a matrix is symmetric, the **isym()** function is used; e. g., **isym(a)**. If the matrix is symmetric, this command returns silently, and the next line prompt is given. If the matrix is not symmetric, a message is printed identifying the first a_{ij} ,a_{ji} pair found not to be equal.

Transformation Functions

Transformation operations are useful for rescaling raw data into a more appropriate form. This is often the case in regression analysis. Several types of transformation functions are available in **mat.**

Absolute Value

The **abs()** function takes a single matrix argument and returns a matrix whose elements are the absolute values of the elements in the original matrix.

Truncation

The **int()** function takes a single matrix argument and returns a matrix whose elements are equal to the truncated integer component of the values in the original matrix.

Exponentiation

The **exp()** function takes a single matrix argument and returns a matrix whose elements are equal to $e^{a}ij$.

Natural Logarithm

The **log()** function takes a single matrix argument and returns a matrix whose elements are equal to log $e^{a}ij$.

Common Logarithm

The **log**10() function takes a single matrix argument and returns a matrix whose elements are equal to $\log_{10} a_{ij}$.

Square Root

The **sqrt**() function takes a single matrix argument and returns a matrix whose elements are equal to $(a_{ij})^{1/2}$.

Reciprocal

The **recip**() function takes a single matrix argument and returns a matrix whose elements are equal to $\frac{1}{a_{ij}}$.

Special Applications Functions

The following specialized functions are provided in **mat** but their use is limited to certain types of applications, such as input-output analysis, or to use only with symmetric matrices.

Leontief Inverse

The **linv**() function computes the Leontief inverse of its matrix argument. This command is shorthand for the equivalent expression **inv(id**(3) - **a**), assuming "**a**" is a 3 x 3 matrix. However, **linv**() should run faster for most applications since the command interpreter has to do less work, and related function calls are internalized.

Power Series Approximation of the Leontief Inverse

An alternative approach for estimating the Leontief inverse is the method of power series expansion.[4] While the power series approximation method is usually mentioned in the context of being less computationally intense than directly inverting a matrix (a debatable point), its real virtue is in providing an economic interpretation for the Leontief inverse.

The **psa**() function is used to compute the power series approximation of the Leontief inverse. This function takes two arguments. The first is the matrix for which the Leontief inverse is to be estimated, and the second is the number of approximations (rounds) that the user wishes to make. For example, **psa**(**a**,5) will return an approximation of the Leontief inverse after five rounds. (Note: A single round is defined to include both the direct (I) and first-round (A) effects.)

RAS Adjustment Method

The RAS adjustment method is used to update an input-output direct coefficients matrix using partial information.[5] This operation is carried out using the **ras**() function,

---

[4] Ronald E. Miller and Peter D. Blair, *Input-Output Analysis: Foundations and Extensions* (Prentice-Hall, Inc., 1985).

[5] Miller and Blair, pp 277-296.

which takes five arguments. The first argument is the current direct coefficients matrix; the second is an N x 1 vector containing estimates of total industry outputs; the third is an N x 1 vector of total industry sales; the fourth is a 1 x N vector of estimated total industry purchases; and the fifth is the level of accuracy desired. For example, **ras(a,x,u,v,.001)** adjusts "**a**" using **x**, **u**, and **v** to an accuracy of 0.001.

The operation terminates when the desired level of accuracy is achieved. If convergence has not occurred before 10 iterations have been completed, the user is asked if he/she wishes to continue. If the answer is "yes," the RAS method is carried out until convergence or another 10 iterations when the user is again asked to continue, and so on. If the user decides not to continue, a copy of the results is saved in a file called RAS dump (overwritten, not appended). This file may be used to continue the RAS procedure later.

At each iteration the **ras()** function prints the maximum values of the R and S vectors so the user may better determine if continuing the procedure seems worthwhile or whether cycling has begun.

Characteristic Roots and Vectors (Eigenvalues and Eigenvectors)

The following two functions are limited in use to symmetric matrices only. Their application is intended primarily for Principal Components Analysis, although there may be other applications.

The **cr()** function is used to extract the characteristic roots of a square symmetric matrix. This function takes three arguments. The first is the matrix itself. The second is an integer number less than or equal to the dimension (number of rows or columns) of the matrix. This number indicates the number of roots to extract. The order in which roots are extracted is in descending order, beginning with the dominant root, the second largest root, the third largest, and so on. The third argument is the level of accuracy desired.

As an example, the command **cr(a,3,.001)** will extract the largest three characteristic roots of "a" with the level of accuracy specified by the third argument. The result returned is an N x 1 vector with the roots in descending order.

Characteristic vectors associated with the characteristic roots are computed using the **cv()** function, which has the same arguments as **cr()**. The result returned is a matrix having the same number of rows as the original matrix, with each column representing a characteristic vector, again in descending order from left to right.

Editing

The following functions perform nonnumeric transformations of matrices. For example, one may wish to read in a large matrix "**a**", using **read()**, and select a submatrix of "**a**" to carry out an operation. Alternatively, one may want to append two or more matrices before saving them in a file with **put()** or **putf()**. Another possibility is to selectively change a matrix element to perform a sensitivity analysis (e. g., change an element in the final demand vector of an input-output table).

Extracting Columns and Rows

The functions **xc()** (extract by columns) and **xr()** (extract by rows) are used to extract a submatrix from a matrix. Both functions take three arguments. The first is the original matrix, the second is the position of the first column or row to be extracted, and the third is the total number of columns or rows to be extracted. To illustrate the use of **xc()**, the command **xc(a,3,5)** will extract five columns from "**a**", beginning with column 3. Note: The expression **a** = **xc(a,3,5)** is valid.

Replacing Columns and Rows

A submatrix of a matrix may be replaced by another matrix using the **rc()** (replace by columns) and **rr()** (replace by rows) commands. Each function has four arguments. The first argument is the matrix on which the operation is to be carried out, and the second is the position of the first column or row to be replaced. The third argument is the total number of columns or rows (including the first) being replaced, and the fourth is the replacement matrix. An example should make this clear. **rc(a,2,3,b)** is the command to replace columns 2 through 4 of matrix "**a**" by matrix "**b**". Note: the number of columns (rows) in "**b**" need not be the same as the number of columns (rows) being replaced in "**a**"; for example, if "**a**" is a 3 x 4 matrix and "**b**" is a 4 x 4 matrix, then the command **rr(a,1,2,b)** prints a 5 x 4 matrix.

Appending Columns and Rows

Matrices may be appended to each other using the **ac()** (append by columns) and **ar()** (append by rows) functions. Each function has three arguments. The first is the matrix being operated on, the second is the matrix being appended, and the third is the position (column or row) in the first matrix after which the second matrix is to be appended. If "**a**" and "**b**" are both 3 x 3 matrices, then **ac(a,b,3)** appends "**b**" to "**a**", resulting in a 3 x 6 matrix. Alternatively, **ar(a,b,3)** appends "**b**" to the bottom of "**a**," giving a 6 x 3 matrix. A matrix may also be appended to itself and the result assigned to the same variable; for example, **a** = **ac(a,a,0)**. Here, position 0 implies to begin appending to the first column (row).

Swapping Columns and Rows

The **sc()** (swap columns) and **sr()** (swap rows) functions are used to change the position of any two columns or rows in a matrix. For example, **sr(a,1,3)** produces a matrix in which the positions of rows 1 and 3 are interchanged.

Modify an Entry Value

The function **mod()** is used to selectively change a single element in a matrix or vector. The command **mod(a,3,1,3.5)** changes the current value of $\mathbf{a}_{31}$ to the value 3.5.

4 TUTORIAL INTRODUCTION

This chapter shows how **mat** can be used to solve real problems in matrix algebra, input-output analysis, and regression analysis. Four examples illustrate different types of commands. The first demonstrates the basic use of various operators and functions; it also shows how to solve systems of equations of the form

$$AX = b$$

The second example features the functions used to perform nonnumeric transformations of matrices. The third example demonstrates functions relating to input-output analysis. It also illustrates how **mat** commands may be read from a file and how in-line comments are used to label output when input is taken from a file. Finally, the fourth example demonstrates the use of **mat** for solving a problem in multiple regression analysis and shows how to compute the characteristic roots and vectors of a symmetric matrix.

Example 1: Matrix Algebra

This example demonstrates the basic **mat** operators and functions. All text following a sharp sign "#" is a comment.

```
% mat
type ? for help, quit to exit

1> a = [2 1 3 4 6 12] 2 3          # define a to be a 2 x 3 matrix

2> a                               # print a

2.000000        1.000000        3.000000
4.000000        6.000000       12.000000

3> b = [1 0 7 3 6 9] 2 3           # define b to be a 2 x 3 matrix

4> a + b                           # sum a and b

3.000000        1.000000       10.000000
7.000000       12.000000       21.000000

5> c = [0 0 6 0 5 1] 2 3           # define c

6> a + b + c                       # sum a + b + c

3.000000        1.000000       16.000000
7.000000       17.000000       22.000000

7> e = a + b + c                   # set e equal to a + b + c

8> e                               # print e

3.000000        1.000000       16.000000
7.000000       17.000000       22.000000
```

```
9> d = a - b                        # subtract b from a and set equal to d

10> d                               # print d

1.000000        1.000000       -4.000000
1.000000        0.000000        3.000000

11> 2. * a                          # multiply a by 2 (a scalar)

4.000000        2.000000        6.000000
8.000000       12.000000       24.000000

12> f = [2 1 3 4 0 7 1 1 6] 3 3     # define f to be a 3 x 3 matrix

13> a * f                           # multiply a and f

11.000000       5.000000       31.000000
44.000000      16.000000      126.000000

14> a / csum (a)                    # divide a by its column sums

 0.333333       0.142857        0.200000
 0.666667       0.857143        0.800000

15> a \ rsum(a)                     # divide a by its row sums

 0.333333       0.166667        0.500000
 0.181818       0.272727        0.545455

16> a'                              # transpose of a

 2.000000       4.000000
 1.000000       6.000000
 3.000000      12.000000

17> a * a'                          # crossproduct of a

14.000000      50.000000
50.000000     196.000000

18> ?a                              # print dimensions of a

2 x 3

19> ?$                              # list all defined variables and their dimensions

a: 2 x 3
b: 2 x 3
c: 2 x 3
d: 2 x 3
e: 2 x 3
f: 3 x 3
```

```
20> a - b - c                    # evaluate (a - b) - c

1.000000        1.000000      -10.000000
1.000000       -5.000000        2.000000

21> a - (b - c)

1.000000        1.000000        2.000000
1.000000        5.000000        4.000000

22> tr(f)                        # trace of f

8

23> inv(a)                       # inverse of a (an error)

inv: matrix not square

24> inv(f)                       # inverse of f

 0.368421       0.157895       -0.368421
 0.894737      -0.473684        0.105263
-0.210526       0.052632        0.210526

25> f * inv(f)                   # check definition of inv(f)

 1.000000       0.000000        0.000000
 0.000000       1.000000       -0.000000
 0.000000       0.000000        1.000000

26> id(3) + id(3)                # add two identity matrices

 2.000000       0.000000        0.000000
 0.000000       2.000000        0.000000
 0.000000       0.000000        2.000000

27> (uv(3) + uv(3))'             # add two unity vectors and transpose result

 2.000000       2.000000        2.000000

28> min(f)                       # smallest value of f

0

29> max(f)                       # largest value of f

7

30> norm(a - b - c)              # norm of result of (a - b) - c

12
```

```
31> recip(a)                    # compute reciprocal of a

0.500000        1.000000        0.333333
0.250000        0.166667        0.083333

32> a = [1 2 2 -2] 2 2          # define a to be a 2 x 2 matrix

33> b = [10 8] 2 1              # define b to be a 2 x 1 vector

34> det(a)                      # check if a is nonsingular

-6                              # a is nonsingular

35> inv(a) * b                  # solve AX = b

6.000000
2.000000

36> quit                        # end this session

%
```

Example 2: Nonnumeric Transformations

This example illustrates the use of editing functions in **mat.**

```
% mat
type ? for help, quit to exit

1> a = [2 1 3 4 0 7 1 1 6] 3 3   # define a to be a 3 x 3 matrix

2> a

2.000000        1.000000        3.000000
4.000000        0.000000        7.000000
1.000000        1.000000        6.000000

3> b = xc(a,2,1)                # extract column 2 of a and assign to b

4> b = b * 2.5                  # multiply b by 2.5

5> a = rc(a,2,1,b)              # replace column 2 of a by the new b

6> a

2.000000        2.500000        3.000000
4.000000        0.000000        7.000000
1.000000        2.500000        6.000000

7> u = uv(3)                    # define u to be a 3 x 1 unity vector

8> a = ac(a,u,0)                # append u to the first column of a
```

```
9> a

1.000000        2.000000        2.500000        3.000000
1.000000        4.000000        0.000000        7.000000
1.000000        1.000000        2.500000        6.000000

10> a = sr(a,2,3)               # swap rows 2 and 3

11> a

1.000000        2.000000        2.500000        3.000000
1.000000        1.000000        2.500000        6.000000
1.000000        4.000000        0.000000        7.000000

12> a = mod(a,3,3,-4.5)         # change a(3,3) to -4.5

13> a

1.000000        2.000000        2.500000        3.000000
1.000000        1.000000        2.500000        6.000000
1.000000        4.000000       -4.500000        7.000000

14> quit                        # end this session

%
```

Example 3: Input-Output Analysis/Noninteractive Use

This example shows how a file containing **mat** commands (in this instance, the I-O related commands **linv()** and **psa()**) can be run in noninteractive mode. The results may be printed on the terminal, as in this example, or redirected to a file (e. g., **mat < ex3 > out.ex3**). This is followed by a problem demonstrating an application of the **ras()** function.

```
% cat ex3                      # type file ex3
read (a, 3, 3, "A")           # Read a 3 x 3 matrix "a"
a
linv(a)                       # Leontief inverse: linv(a)
inv(id(3) - a)                # I - a inverse: inv(id(3) - a)
psa(a,10)                     # Power Series Approx. after 10 iterations: psa(a,10)
quit

% mat < ex3                   # read mat input from file ex3 and print results to stdout

Read a 3 x 3 matrix "a"

0.120000        0.100000        0.049000
0.210000        0.247000        0.265000
0.026000        0.249000        0.145000

Leontief inverse: linv(a)

1.188156        0.200897        0.130359
0.383365        1.544494        0.500674
0.147778        0.455909        1.319365
```

```
I - a inverse: inv(id(3) - a)

1.188156        0.200897        0.130359
0.383365        1.544494        0.500674
0.147778        0.455909        1.319365

Power Series Approx. after 10 iterations: psa(a,10)

1.188019        0.200665        0.130172
0.382969        1.543828        0.500134
0.147496        0.455436        1.318982

% mat                           # example of RAS adjustment

type ? for help, quit to exit

1> a = [.12 .1 .049 .21 .247 .265 .026 .249 .145] 3 3 # direct coefficients

2> x = [421 284 283] 3 1        # total outputs

3> u = [245 136 159] 3 1        # total interindustry sales

4> v = [251 107 182] 1 3        # total interindustry purchases

5> b = ras(a,x,u,v,.001)        # RAS adjustment

1: Max R = 2.64045610, Max S = 1.36368708  # max R and max S at each iteration
2: Max R = 1.15500540, Max S = 1.03809930
3: Max R = 1.02262738, Max S = 1.00746016
4: Max R = 1.00449947, Max S = 1.00151540
5: Max R = 1.00091821, Max S = 1.00031001  # procedure took 5 iterations

6> b                            # RAS adjusted coefficients matrix

0.392452        0.121906        0.159669
0.150887        0.066153        0.189714
0.052860        0.188702        0.293727

7> quit

%
```

Example 4: Regression Analysis/Eigenvalues and Eigenvectors

This example demonstrates a problem in multiple regression analysis solved using **mat**. An example illustrating the use of the **cr()** and **cv()** functions is also included.

```
% mat

type ? for help, quit to exit

1> read(x,15,3,"X")             # read file containing dependent variables
```

```
2> x

1.000000        1.000000        1.000000
1.000000        2.000000        4.000000
1.000000        3.000000        9.000000
1.000000        4.000000       16.000000
1.000000        5.000000       25.000000
1.000000        6.000000       36.000000
1.000000        7.000000       49.000000
1.000000        8.000000       64.000000
1.000000        9.000000       81.000000
1.000000       10.000000      100.000000
1.000000       11.000000      121.000000
1.000000       12.000000      144.000000
1.000000       13.000000      169.000000
1.000000       14.000000      196.000000
1.000000       15.000000      225.000000

3> read(y,15,1,"Y")             # read file containing independent variables

4> y

0.580000
1.100000
1.200000
1.300000
1.950000
2.550000
2.600000
2.900000
3.450000
3.500000
3.600000
4.100000
4.350000
4.400000
4.500000

5> b = inv(x' * x) * x' * y     # solve for coefficients

6> b

 0.100527
 0.421320
-0.008053

7> ((y - x * b)' * (y - x * b)) / (15. - 3.)  # compute estimated variance

0.028376

8> S = 0.028376                 # set S equal to estimated variance

9> c = S * inv(x' * x)          # compute variance covariance matrix
```

```
10> c

 0.022514       -0.005800        0.000312
-0.005800        0.001862       -0.000110
 0.000312       -0.000110        0.000007

11> cr(c,3,.00001)              # characteristic roots (eigenvalues)

0.024036
0.000347
0.000000

12> cv(c,3,.00001)              # characteristic vectors (eigenvectors)

 0.967344        0.253368        0.013503
-0.253095        0.964101        0.078606
 0.013719       -0.079459        0.996814

13> quit

75.5u 2.9s 14:28 9% 10+453k 6+8io 3pf+0w

%
```

5 SUMMARY

This report has described the commands, operators, and functions of **mat** an interactive interpreter and calculator for solving systems of equations expressed as matrices. **Mat** provides error checking, and checks for conformability, syntax, and overflow. It can also be useful for applications requiring manipulation of data stored as matrices. Examples have been provided that show how the system is used to solve problems in matrix algebra, input-output analysis, and regression analysis.

USA-CERL DISTRIBUTION

AFESC
Tyndall AFB, FL 32403
ATTN: AFESC/DEV (W. Alan Nixon)
ATTN: Linda Merritt

Defense Technical Information Center
ATTN: DDA (2)

Institute for Water Resources
Fort Belvoir, VA
ATTN: Dennis Robinson

LEEV
Boling AFB
ATTN: MAJ K. Small

END

10-86

DTIC