LINEAL: A system for numerical linear algebra

H. M. Khalil and D. L. Ulery

College of Arts and Science, Department of Statistics and Computer Science, University of Delaware, Newark, Delaware 19711, USA

This paper describes a computing system that enables linear algebra problems to be simply programmed and efficiently solved. The main features of this language, its translator, and the two modules for the automatic solution of the eigenproblem and systems of linear equations are described. The power of the system lies in the in-depth analysis techniques employed by these modules and in the quantitative error information accompanying the resultant solutions.

(Received June 1973)

Many of the day-to-day problems which confront numerical mathematicians, statisticians, engineers and econometricians involve solution of linear systems and the general eigenproblem. The solution of these problems requires frequent computation with matrices. In response to this need, many problem-oriented languages have been developed over the past decade which provide special facilities for the manipulation of matrices: APL (Iverson, 1971), ASP (Kalman and Englar, 1965), Burley (Burley, 1967), MARI (Brainin, et al., 1965), MAP (Kaplow and Brackett, 1966), MATLAN (System/360), MATRIX (MATRX), MM (Newbold and Agrawala, 1967), NAPSS (Rice, 1968), OMNITAB II (Agher, Pears and Varner, 1969), and POSE (Schlesinger and Sashkin, 1967). A survey of their capabilities can be found in Smith (1970) and Ulery and Khalil (1974).

It is our judgement that more than ease of matrix manipulation is needed in a special-purpose language for linear algebra. One feature we feel is of paramount importance is 'automatic' problem-solving, in which the system attempts to provide the method of solution for a stated problem. The problem of choosing an appropriate method of solution for problems in linear algebra is not at all simple. The types of matrices which appear in practice vary widely, and an algorithm perfectly suitable for solving one type of system may well be quite inappropriate for use with another. Much work has been done in this area of numerical computation, largely by Wilkinson (1965), and it has reached a very advanced state. The chance of an amateur selecting the algorithm best suited to his system from the large body of algorithms available to solve this class of problems is slim. An inappropriate choice can unfortunately lead to results with such a poor degree of accuracy as to make them totally without value, unbeknownst to the unsuspecting user.

Our aim is to relieve the user of this burden of selection by making the computer act as a professional trained in this area. The user can describe his problem quite simply in the LINEAL language, which resembles standard mathematical notation and is readily learned. The LINEAL system will then automatically select the algorithms, perform analysis, and output the results and statistics pertinent to that solution, thus providing the user with far better results than he would normally have been able to achieve. The system also serves the expert in this area by providing a convenient tool for the comparative study of algorithms, since it provides a measure of the accuracy of the solution.

A few of the existing systems listed above do provide facilities for automatic problem solving. In all cases, however, the number of algorithms and the logic employed to select them is quite limited. LINEAL represents an extension of these efforts. In designing the system our chief objectives were to provide
1. Simplicity and clearness of notation;
2. Concise and powerful operators for the computations of linear algebra;
3. 'Automatic' problem solving for systems of linear equations and the eigenproblem and a measure of goodness of the resulting solution;
4. Economisation of storage;
5. A formal description of the syntax and semantics of the language;
6. High machine independence.

The characteristics of the language, its formal syntax, a part of its formal semantic description, and a brief description of its translator and the two modules for the automatic solution of the eigenproblem and systems of equations are outlined here. APL was used for describing the semantics of LINEAL because of its operator richness and conciseness. It is the authors' belief that APL is the most natural way for describing a language whose data structures are scalars and arrays. Its syntax is described in terms of a modified BNF (Cocke and Schwartz, 1970). Repetitive concatenations of objects are indicated by the notation { . . . } where i is the minimum number of repetitions required and j is the maximum number of repetitions permitted. Where either index is represented by a variable, the domain of the variable is the metaexpression.

The LINEAL language

Some of the details of the language are formally described in Appendices 1 and 2. We will concentrate here instead on a more informal description of the main features of the language.

The two examples below illustrate the general format of LINEAL code.

**Example 1:**

```plaintext
COMMENT THIS PROGRAM GIVES A SOLUTION TO THE GENERAL EIGENPROBLEM ABX = \lambda X;
DECLARE M, N;
READ M, N;
DECLARE A(M, N), B(N, M);
READ A, B;
SOLVE Eigenproblem A * B;
END.
```

**Example 2:**

```plaintext
DECLARE N, I, J, S;
READ N;
DECLARE RHS(N, 1): SYMMETRIC HILB(N, N);
COMMENT GENERATE HILBERT MATRIX;
LOOP I = 1 TO N DO
S = 0;
LOOP J = 1 to N DO
HILB(I, J) = / ((I + J) - 1);
ENDLOOP J;
LOOP J = 1 TO N DO
S = S + HILB(I, J);
ENDLOOP J;
RHS(I, 1) = S;
```
The first example is a program to solve the general eigenproblem, given the coefficient matrices 'A' and 'B'. The second is a program to solve a system of linear equations whose coefficient matrix is an internally-generated Hilbert matrix of order 'N'.

The allowable data types in LINEAL are real, boolean, and alphanumeric. The latter type is used in composing strings. The permitted data structures are scalars and arrays which are treated as units of information and manipulated accordingly.

A LINEAL program is a sequence of statements which normally are executed sequentially. A feature of the language is the absence of labels and unconditional transfer statements, their place being taken by an iterative and a conditional statement. The former invokes an action using a succession of values of its looping index and terminating when the upper limit for the loop is exceeded. The latter invokes one of two alternate actions; the choice is based on the truth or falsity of its boolean expression.

The arithmetic expression permits direct manipulation of scalars and arrays with the basic operators (+, -, *, **) plus a large number of built-in numerical operators particularly suited to matrix computations. Included are operators for finding the absolute value, the trace, transpose, determinant, and inverse of a square array; element-wise reciprocals; the sum of elements of an array; a scaling operator; the maximum and Euclidean norms; and the spectral radius. In addition, built-in operators are included for the trigonometric functions sine, cosine, arctangent and logarithms.

An assignment statement is of the form

\[ T = \tau \]

where \( T \) is an identifier and \( \tau \) is an arithmetic expression. \( \tau \) is evaluated using the current values assigned to the variables appearing within it. The value(s) obtained is stored in \( T \). To avoid specifying precedence of operators, expressions have to be fully parenthesised.

For example,

\[ T = \text{NORME}(A * B) \]

defines the scalar \( T' \) as the value of the Euclidean norm of the matrix product \( A \cdot B \). (See Appendix 2 for the semantic description).

LINEAL distinguishes segments of the program called define declarations. Intuitively, a define declaration is equivalent to a procedure: it is a sequence of statements which form a semi-independent unit within the body of the main program. When the procedure is invoked, a branch from the main program to the first statement of the procedure occurs. Exit occurs only from the last statement in the procedure and returns control to the next statement in sequence in the main program. The header of a define declaration includes the name of the procedure followed by a list of formal parameters. A procedure is invoked by a call statement consisting of the word CALL followed by the name of the procedure and a parenthesised list of actual parameters. The procedure call represents a copy of the procedure body in which the actual parameters have been substituted for the dummy variables.

An example of the define declaration is given below:

```
COMMENT THE SAMPLE MEAN AND SAMPLE VARIANCE OF A RANDOM SAMPLE OF SIZE N ARE COMPUTED;
DEFINE STAT(X, N, MEAN, VAR);
MEAN = \sum X(I);
VAR = 0;
IF N GTR 1 BEGIN MEAN = (SUM X)/(N);
```

LINEAL permits the assignment of attributes to arrays for efficient storage management and algorithm selection. There is a special statement to perform this task, i.e. the declare statement. The allowable attributes are: vector, general, symmetric, band, band symmetric, triangular, and sparse. These attributes are used to provide control structures for selection of appropriate algorithms and means of efficient storage management. On translating the declare statement, the LINEAL translator associates with each declared array identifier a five-element attribute vector in which the type of the array, its dimensions, and other relevant information are stored for subsequent use. Finally, one of the most useful features of the system is the solve statement which has the form:

```
SOLVE TYPE VARS, (options);
```

where TYPE specifies the problem, i.e. linear equations, eigenvalues and/or eigenvectors; VARS are the equation labels; and options are either empty or represent directives to the system. On executing this statement, the system returns the results in reserved arrays (LAMBDa and/or VECtOr) and all the statistics regarding the solution in another reserved array STATUS. The user may exercise the options to:

1. Impose on the system the degree of accuracy required—if this option is not exercised the system sets the accuracy automatically.

2. Specify the algorithm to be used, i.e. control the method of solution. If no algorithm is specified, the system automatically selects the most appropriate algorithm for the particular problem.

3. Suppress output—as mentioned before, the system dynamically allocates arrays for the solution of the problem at hand. Once the problem is solved, the system outputs the results and the relevant statistics, and frees these dynamically allocated arrays. If the suppress option is exercised, the user must save his results by an assignment statement immediately following the solve statement.

For example, to have the system solve the eigenproblem

\[ AX = \lambda BX \]

for all eigenvalues and eigenvectors, the user would simply have to write

```
SOLVE EIGENPROBLEM A/B;
```

and the system would automatically select an appropriate algorithm, find a solution with the best accuracy possible for the particular machine and matrices, and output the results. Other features include format-free I/O and the option to free storage space previously reserved for identifiers specified in a declare statement.

The polynomials

Computations performed during program execution are handled by a set of polynomials. Polynomials are handled for arithmetic operations and element manipulation are straightforward. Of more interest are the two polynomials invoked by the solve statement: the LEQ module and the EIGEN module. The design objectives of a module for the automatic solution of numerical analysis problems have been enumerated by Rice (1968) and will not be elaborated upon here. Instead, we will confine our attention to the strategy used.

Each of the modules comprises a number of algorithms together with the logic necessary to apply one or more of these to a given problem. The main reference sources of these algorithms are Wilkinson and Reinsch (1971) and Dekker (1970).

The following information, where applicable, is returned to the
user by each module:
1. Errors incurred in attempting to solve the problem;
2. The determinant and rank of the coefficient matrix;
3. An estimate of the condition number of the coefficient matrix;
4. The number of iterative improvements required to achieve
   desired accuracy;
5. An estimate of the number of correct digits in the solution;
6. The method(s) employed.

Each of the component algorithms may contain one or more of the
following features: scaling, partial or complete pivoting and
iterative improvement.

The strategy employed is based on system size, attributes of the
system, and accuracy specified (Khalil and Ulery, 1973;
Ulery, 1972). In addition, the EIGEN module takes into
account whether both the latent roots and vectors are required
or only one of them, as well as the number of required roots
and/or vectors.

The LEQ module comprises four decomposition algorithms
(Gauss, Crout, Cholesky and Band), two iterative algorithms
(with or without a relaxation factor), and Golub and Reinsch’s
Singular Value Decomposition (SVD), (Golub and Reinsch,
1965). For small, square systems one of the decomposition
methods is chosen initially; for large and/or non-dense systems,
an iterative method is selected. If the initial method should fail,
another approach, based upon the type of failure incurred, is
tried. For example, if failure occurs due to rank deficiency, the
SVD algorithm is applied. This method of recovery is con-
tinued until all applicable paths have been exhausted. The user
is then provided with the best possible solution and relevant
data concerning the methods tried and the results achieved.

The EIGEN module consists of three submodules: symmetric,
general and reduction. The first of these is used to solve
$Ax = \lambda x$ where $A$ is symmetric. If $A$ is of small order and the
accuracy specified is not high, the Jacobi algorithm is applied.
Otherwise $A$ is reduced to tridiagonal form using Householder’s

---

Fig. 1a

Fig. 1b

Fig. 1c
transformation. This is followed by one of the QR versions depending upon whether or not the latent vectors are required. On the other hand, if only few latent values and/or vectors are needed, a bisection algorithm followed by inverse iteration is applied. The general submodule comprises the following algorithms: a QR-algorithm for finding the latent roots, an algorithm based upon the norm-reducing Jacobi for computing both values and vectors (Ebelein, 1970), and an iterative algo-

---

**Fig. 2a**

ENTRY REDUCE

\begin{itemize}
\item A \in \mathbb{R}^{m\times n}
\item a, \text{Actual: A, A}^T, A\text{I, A}^{-1}
\item b, \text{Global: LAMBDA, VECTOR, FLAG}
\end{itemize}

REPLACE A = \begin{bmatrix}
1 & 1 \\
0 & 1
\end{bmatrix}

\text{GENERIC}

\begin{itemize}
\item No. of values or vectors needed in small.
\item A of band type.
\item A Tridiagonal.
\item (Reduce to symmetric tridiagonal).
\item Reduction successful.
\end{itemize}

\text{GENERAL}

\begin{itemize}
\item Solves Ax = b; A is real and nonsymmetric.
\item GIVE (Iterative approach).
\item A of band type.
\item A Tridiagonal.
\item (Reduce to symmetric tridiagonal).
\item Reduction successful.
\item Accuracy achieved.
\item Vectors required.
\item GIMPROVE.
\item Sort \sigma_i and \mu_i in descending order.
\item Return.
\end{itemize}

\begin{itemize}
\item QUR (GBQR): QR-algorithm for \lambda's of general (band) matrix.
\item GRN4 (BNRJ): Norm Reducing Jacobi algorithm for \lambda's and \mu's of general (band) matrix.
\end{itemize}

---

**Fig. 2c**

ENTRY SYMDC

\begin{itemize}
\item A of band type.
\item Band width = 3.
\item \text{JACOBI (Jacobi reduction)}
\item Only Values Needed.
\end{itemize}

\begin{itemize}
\item \text{SVGL (Sturm sequence)}.
\item \text{SVLC (Sturm sequence)}.
\item Only Values Needed.
\end{itemize}

\begin{itemize}
\item \text{REDUCE} (Estimate accuracy of solution).
\item \text{THRESH} (Iterative improvement).
\item Accuracy achieved.
\item Return.
\end{itemize}

---

**Fig. 2b**

**Fig. 2d**
6. Inclusion of polyalgorithms
The user has the option of either using a compiler version of a polyalgorithm at running time (on the B6700) or of copying the source version of it into his ALGOL program.

7. Error checking and recovery
The generator checks for syntactic, semantic and logic errors during both the recognition and generation phases. Once an error is detected, code generation ceases; however, scanning of the remaining source program continues to completion. At run-time numerous checks are made as well. Error messages regarding abnormal job termination indicate probable causes of difficulty. The statement number in the LINEAL source program attributed to the error is specified also.

8. Options
The user has available to him options to:
8.1 Display the LINEAL source and/or ALGOL source.
8.2 Compile and/or execute the generated program.
8.3 Copy the generated program onto disc and/or cards.
8.4 Copy the object program produced by the ALGOL compiler onto disc.
8.5 Include in the generated program the source code for required polyalgorithms or code to bind the generated program to a compiled polyalgorithm at run-time.
8.6 Specify maximum size of identifiers.
8.7 Specify maximum amount of processor time to be used for execution of the generated program.

The above characteristics are illustrated by examples in Appendix 3.

Thirty test programs were run on the B6700 to obtain a measure of machine requirements. The average size of each LINEAL program was twelve statements; corresponding to each source statement, the generator produced 5-3 ALGOL statements (excluding the polyalgorithms). The average processor time required for the translation was 1-8 seconds. (It is expected that the average time/statement will be reduced as the size of the source program increases.) The generated ALGOL program required an average of 1-9 seconds for compilation.

Summary
A specialised, high-level, language-oriented system for automatically providing optimal solutions to the common computational problems of linear algebra is presented. One important characteristic of the system is the assignment of array attributes to provide efficient storage management and appropriate algorithm selection. Another is the use of the LEQ module and the EIGEN module to automatically solve systems of linear equations and the eigenproblem and provide quantitative error information with respect to these solutions. The system is well-adapted for use by both the low-level user whose knowledge of programming and numerical mathematics is limited, and the sophisticated user interested in the comparative study of different algorithms as well as the accuracy of the solution.

Acknowledgement
This work was partially supported by two University of Delaware Research Foundation grants.
Appendix 1 Syntax of LINEAL

```
<program> ::= {<sentence>;}<0;n> END.
<sentence> ::= <declaration> | <stm>.
<declaration> ::= <(sd decl)> | <def decl>.
<stm> ::= <(sd st)> | <if st>.
<sd st> ::= <(assign st)> | <(comment st)> | <call st> | <(control st)> | <(erase st)> | <(IO st)> | <(solve st)>.
<d-sentence> ::= <(sd decl)> | <(stm)>.
<letter> ::= A | B | ... | Y | Z.
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
<bi-op> ::= + | - | * | **.
<uni-op> ::= = | > | < | >= | <= | <--> | Q.
<relation> ::= == | <= | >= | < | > | <= | >= | <--> | Q.
<attribute> ::= SYMMETRIC | TRIANGULAR | IDENTITY | BAND | SYMBOLE | SPARSE | GENERAL.
<keyword> ::= END | DECLARE | READ | WRITE | LOOP | ENDOOP | FREE | SOLVE | TOLERANCE | USE | IF | BEGIN | ENDCOBD | DEFINE | CALL | EIGENPROBLEM | EIGENVALUES | EIGENVECTORS | LINEQ | Suppress | Comment.
<special> ::= LAMBDA | VECTOR | STATUS | FLAG.
<integer> ::= {<digit>}"_"<11>.
<number> ::= {<digit>}_{0}<11} . {<digit>}_{11-1}.
<string> ::= {<symbol excluding; and>}.
<scalar id> ::= <(letter)> | <(digit)> | <(symbol)> | <(list)>.
<array id> ::= <(letter)> | <(digit)> | <(symbol)> | <(list)>.
<def id> ::= <(letter)> | <(digit)> | <(symbol)>.
{id} ::= <(scalar id)> | <(array id)>.
{id list} ::= <(id)> | <(id)>_{0}^n.
<subscripted id> ::= <(array id)> <(subscript list)>.
<subscript list> ::= <(sexpr)> | <(sexpr)>_{0}.
<sexpr> ::= <(digit)> | <(scalar id)>.
<simple id> ::= <(elementary id)> | <(number)>.
<elementary id> ::= <(scalar id)> | <(subscripted id)>.
<variable> ::= <(elementary id)> | <(array id)>.
<variable list> ::= <(variable)> | <(variable)>_{0}^n.
<scalar expr> ::= {<simple id> | <(bi-op)> | <(uni-op)> | <(symbol)>}_{0} <(simple id)>.
<arithmetic expr> ::= {<primary> | <(bi-op)> | <(uni-op)> | <(symbol)>}_{0} <(primary)>.
<primary> ::= <(number)> | <(variable)> | <(arithmetic expr)>.
<boolean expr> ::= <(scalar expr)> | <(relation)> <(scalar expr)>.
<comment st> ::= COMMENT | <(string)>.
<IO st> ::= {READ | WRITE}_1 | <(paren)> | <(variable list)> | <(string)>._1.
<paren> ::= <(subscript list)>.
<assign st> ::= LOOP | <(scalar id)> = <(sexpr) | TO <(sexpr) | BY <(sexpr)>_0.
<erase st> ::= FREE | <(id list)>.
<call st> ::= CALL | <(def id)> <(primary)> | <primary>.
<condition st> ::= SOVLE | <(paren)> | <(solve word) | <(id expr) | <(error expr) | <(use expr)> | <(supress expr)>.
<supress expr> ::= TOLERANCE <(paren)> | <(symbol)>_1.
<error expr> ::= USE | <(leq algorthim)> | <(symbol algorthim)>_1 | e.
<use expr> ::= SUPPRESS | e.
<if st> ::= IF <(boolean expr) | BEGIN <(sd st)> | ENDCOND.
<sd decl> ::= DECLARE <(att spec)> | <(att spec)>_0;
<att spec> ::= {<attrib> | <(id)> <(paren)> | <(id)> <(paren)>.
<attrib> ::= SYMMETRIC | TRIANGULAR | IDENTITY | GENERAL | SPARSE (<(integer)> | <(integer)>, <(integer)>, <(integer)> | <(integer)>).
<def decl> ::= DEFINE <(def id) | <(id list)> | <(def id)> | <(sd-sentence)>_1 | END | <(def id)>.
```

Appendix 2 Semantic description

(a) \( \{ \langle \text{assign st} \rangle \} \colon = \{ \langle \text{primary} \rangle \cup \langle \text{bi-op} \rangle \cup \langle \text{uni-op} \rangle \cup \langle \text{symbol} \rangle \} \langle \text{primary} \rangle \)

Let \( t : \text{LINEAL} \to \text{APL} \), then \( t(\langle \text{var} \rangle) \to T : t(\langle \text{primary} \rangle) \to A ; t(\langle \text{primary} \rangle) \to B \), and the semantics of the form \( T = A \langle \text{bi-op} \rangle B \) is given by:

\[
\begin{array}{cccc}
\text{LINEAL Statement} & \rho_A & \rho_B & \rho_T \quad \text{Conditions for which 't' is defined} \\
\hline
T = A + B & T \leftarrow A + B \\
T = A - B & T \leftarrow A - B \\
T = A \times B & T \leftarrow A \times B \\
T = A + B & \text{M} & \text{M} & (\rho_T) \geq \text{M} \\
\end{array}
\]

The Computer Journal
Appendix 3  LINEAL to ALGOL translator

B6700 LINEAL TO ALGOL TRANSLATOR LEVEL II.03.00

COMMENT EXAMPLE 1

PROBLEM = SOLVE A SYSTEM OF LINEAR EQUATIONS WHERE COEFFICIENT
MATRIX M IS A 3 BY 3 HILBERT MATRIX.

DECLARE RHS(3,1),RES(3,1),T(3,1); GENERAL HILB(3,3),INVERSE HILB(3,3);

>>WARNING 01>>  *IDENTIFIER LONGER THAN 5 CHARACTERS <<

COMMENT GENERATE MATRIX:

LOOP 1 = 1 TO 3 DO
  HILB(1,1)=/(I=1-1); 1
  RHS(1,1)=0; 1
  I=I-1;
  LOOP J=1 TO 3 DO
    HILB(I,1)=/(J=1-1); 2
    HILB(I,J)=HILB(I,J); 2
    ENDLOOP J;
  ENDLOOP I;

RES(1,1)=1;

SOLVE LINEQ HILB/RHS, USE CHOLESKY, SUPPRESS;

>>WARNING 02>>  *SHOULD ONLY BE USED WITH SYMMETRIC ARRAYS <<

WRITE"SOLUTION OF LINEAR EQUATIONS.
WRITE"COEFFICIENT MATRIX IS:
WRITE(T(1,1));
WRITE(T(2,1));
WRITE(T(3,1));
WRITE(RHS(1,1));
WRITE(SOLUTION IS:);
WRITE VECTOR:
INVERSE HILB="INVERSE HILB";

>>WARNING 03>>  *IDENTIFIER LONGER THAN 5 CHARACTERS <<

WRITE"INVERSE MATRIX IS:
WRITE RES;
WRITE INVERSE HILB;

>>WARNING 04>>  *IDENTIFIER LONGER THAN 5 CHARACTERS <<

ASSIGN RES=TRANSPOSE(INVERSE HILB);RES;

>>WARNING 05>>  *IDENTIFIER LONGER THAN 5 CHARACTERS <<

ASSIGN RES=TRANSPOSE(INVERSE HILB);RES;

NUMBER OF ERRORS DETECTED = 0000.
NUMBER OF WARNINGS = 0005.
TRANSLATION TIME = 000.96 SECONDS PROCESSING.
PROGRAM SIZE = 0034 CLOCKS.
TRANSLATOR COMPILED 04/30/73 09:38 PM.

SOLUTION OF LINEAR EQUATIONS.

COEFFICIENT MATRIX IS:

ROW 1 OF HILB 1.00000000000 0.50000000000 0.33333333333 0.25000000000 0.20000000000
ROW 2 OF HILB 0.33333333333 0.33333333333 0.33333333333 0.33333333333 0.33333333333
ROW 3 OF HILB 0.33333333333 0.33333333333 0.33333333333 0.33333333333 0.33333333333

RIGHT HAND SIDE IS:

ROW 1 OF RHS 1.00000000000 0.00000000000 0.00000000000 0.00000000000 0.00000000000

ROW 2 OF RHS 0.00000000000 0.00000000000 0.00000000000 0.00000000000 0.00000000000

ROW 3 OF RHS 0.00000000000 0.00000000000 0.00000000000 0.00000000000 0.00000000000

SOLUTION IS:

ROW 1 OF VECTOR 9.00000000000E 00 3.60000000000E 01 3.00000000000E 01
ROW 2 OF VECTOR -3.60000000000E 01 1.92000000000E 02 -1.80000000000E 02
ROW 3 OF VECTOR 3.00000000000E 01 -1.80000000000E 02 1.80000000000E 02

INVERSE MATRIX IS:

ROW 1 OF INVERSE 9.00000000000E 00 -3.60000000000E 01 3.00000000000E 01
ROW 2 OF INVERSE -3.60000000000E 01 1.92000000000E 02 -1.80000000000E 02
ROW 3 OF INVERSE 3.00000000000E 01 -1.80000000000E 02 1.80000000000E 02

INVERSE TIMES RIGHT HAND SIDE IS:

ROW 1 OF RES 9.00000000000E 00 -3.60000000000E 01 3.00000000000E 01
ROW 2 OF RES -3.60000000000E 01 1.92000000000E 02 -1.80000000000E 02
ROW 3 OF RES 3.00000000000E 01 -1.80000000000E 02 1.80000000000E 02
**REFERENCES**


SYSTEM/360 Matrix Language Application Description, New York: IBM, H20-0479-0.


