Chapter 17
Language Reference

Chapter Table of Contents

Chapter 17
Language Reference

Overview

This chapter describes all operators, statements, functions, and subroutines that can be used in SAS/IML software. All necessary details, such as arguments and operands, are included.

This chapter is divided into two sections. The first section contains operator descriptions. They are in alphabetic order according to the name of the operator. The second section contains descriptions of statements, functions, and subroutines also arranged alphabetically by name.

The following tables list all statements, functions, and subroutines available in SAS/IML software grouped by functionality.

Scalar Functions

ABS Function	takes the absolute value
EXP Function	calculates the exponential
INT Function	truncates a value
LOG Function	takes the natural logarithm
MOD Function	computes the modulo (remainder)
NORMAL Function	generates a pseudo-random normal deviate
SQRT Function	calculates the square root
UNIFORM Function	generates pseudo-random uniform deviates

Reduction Functions

MAX Function	finds the maximum value of a matrix
MIN Function	finds the smallest element of a matrix
SSQ Function	calculates the sum of squares of all elements
SUM Function	sums all elements

Matrix Inquiry Functions

ALL Function	checks for all nonzero elements
ANY Function	checks for any nonzero elements
LOC Function	finds nonzero elements of a matrix
NCOL Function	finds the number of columns of a matrix
NLENG Function	finds the size of an element
NROW Function	finds the number of rows of a matrix
TYPE Function	determines the type of a matrix

Matrix Reshaping Functions

BLOCK Function	forms block-diagonal matrices
BTRAN Function	computes block transpose
DIAG Function	creates a diagonal matrix
DO Function	produces an arithmetic series
I Function	creates an identity matrix
INSERT Function	inserts one matrix inside another
J Function	creates a matrix of identical values
REMOVE Function	discards elements from a matrix
REPEAT Function	creates a new matrix of repeated values
SHAPE Function	reshapes and repeats values
SQRSYM Function	converts a symmetric matrix to a square matrix
SYMSQR Function	converts a square matrix to a symmetric matrix
T Function	transposes a matrix
VECDIAG Function	creates a vector from a diagonal

Character Functionality

BYTE Function	translates numbers to ordinal characters
CHANGE Call	replaces text
CHAR Function	produces a character representation of a matrix
CHOOSE Function	conditionally chooses and changes elements
CONCAT Function	Concatenates elementwise strings
CONVMOD Function	converts modules to character matrices
CSHAPE Function	reshapes and repeats character values
LENGTH Call	finds the lengths of character matrix elements
NAME Function	lists the names of arguments
NUM Function	produces a numeric representation of a character matrix
ROWCAT Function	concatenates rows without using blank compression
ROWCATC Function	concatenates rows using blank compression
SUBSTR Function	takes substrings of matrix elements

Statistical Functionality

BRANKS Function	computes bivariate ranks
CUSUM Function	calculates cumulative sums
DESIGN Function	creates a design matrix
DESIGNF Function	creates a full rank design matrix
IPF Call	performs an iterative proportional fit
LAV Call	performs linear least absolute value regression by solving the L_1 norm minimization problem
LMS Call	performs robust regression
LTS Call	performs robust regression
MARG Call	evaluates marginal totals in a multiway contingency table
MAXQFORM Call	computes the subsets of a matrix system that maximize the quadratic form
MVE Call	finds the minimum volume ellipsoid estimator
OPSCAL Function	rescales qualitative data to be a least-squares fit to qualitative data

RANK Function	ranks elements of a matrix
RANKTIE Function	ranks matrix elements using tie-averaging
SEQSCALE Call	perform discrete sequential tests
SEQSHIFT Call	perform discrete sequential tests
SEQTESTS Calls	perform discrete sequential tests
SWEEP Function	sweeps a matrix

Time Series Functionality

ARMACOV Call	computes an autocovariance sequence for an ARMA model
ARMALIK Call	computes the log likelihood and residuals for an ARMA model
ARMASIM Function	simulates an ARMA series
CONVEXIT Function	calculates convexity of a non-contingent cash-flow
COVLAG Function	computes autocovariance estimates for a vector time series
DURATION Function	calculates modified duration of a non-contingent cash-flow
FORWARD Function	calculates forward rates
KALCVF Call	computes the one-step prediction $\mathbf{z}_{t+1\|t}$ and the filtered estimate $\mathbf{z}_{t\|t}$, as well as their covariance matrices. The call uses forward recursions, and you can also use it to obtain k-step estimates.
KALCVS Call	uses backward recursions to compute the smoothed estimate $\mathbf{z}_{t\|T}$ and its covariance matrix, $\mathbf{P}_{t\|T}$, where T is the number of observations in the complete data set.
KALDFF Call	computes the one-step forecast of state vectors in an SSM using the diffuse Kalman filter. The call estimates the conditional expectation of $\mathbf{z}_t$, and it also estimates the initial random vector, δ, and its covariance matrix.
KALDFS Call	computes the smoothed state vector and its mean square error matrix from the one-step forecast and mean square error matrix computed by KALDFF.
PV Function	calculates present value
RATES Function	converts interest rates from one base to another
SPOT Function	calculates spot rates
TSBAYSEA Call	performs Bayesian seasonal adjustment modeling
TSDECOMP Call	analyzes nonstationary time series by using smoothness priors modeling
TSMLOMAR Call	analyzes nonstationary or locally stationary multivariate time series by using the minimum AIC procedure
TSMULMAR Call	estimates VAR processes by using the minimum AIC procedure
TSPEARS Call	analyzes periodic AR models with the minimum AIC procedure
TSPRED Call	provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are input

TSROOT Call	calculates AR and MA coefficients from the characteristic roots of the model or calculates the characteristic roots of the model from the AR and MA coefficients
TSTVCAR Call	analyzes time series that are nonstationary in the covariance function
TSUNIMAR Call	determines the order of an AR process with the minimum AIC procedure and estimates the AR coefficients
VARMACOV Call	computes the theoretical auto-cross covariance matrices for the stationary VARMA(p, q) model
VARMASIM Call	generates VARMA(p, q) time series
VNORMAL Call	generates multivariate normal random series
VTSROOT Call	computes the characteristic roots for the VAR(p) or VMA(q) model
YIELD Function	calculates yield-to-maturity of a cash-flow stream

Numerical Analysis Functionality

FFT Function	performs the finite Fourier transform
IFFT Function	computes the inverse finite Fourier transform
JROOT Function	computes the first nonzero roots of a Bessel function of the first kind and the derivative of the Bessel function at each root
ODE Call	performs numerical integration of vector differential equations of the form
ORPOL Function	generates orthogonal polynomials
ORTVEC Call	provides columnwise orthogonalization by the Gram-Schmidt process and stepwise QR decomposition by the Gram-Schmidt process
POLYROOT Function	finds zeros of a real polynomial
PRODUCT Function	multiplies matrices of polynomials
QUAD Call	performs numerical integration of scalar functions in one dimension over infinite, connected semi-infinite, and connected finite intervals
RATIO Function	divides matrix polynomials
SPLINE Call	evaluates points on the spline
SPLINEC Call	evaluates points on the spline
SPLINEV Function	evaluates points on a spline
TPSPLINE Call	computes thin-plate smoothing splines
TPSPLNEV Call	evaluates the thin-plate smoothing spline at new data points

Linear Algebra Functionality

APPCORT CALL	complete orthogonal decomposition
COMPORT Call	complete orthogonal decomposition by Householder transformations
CVEXHULL Function	finds a convex hull of a set of planar points
DET Function	computes the determinant of a square matrix
ECHELON Function	reduces a matrix to row-echelon normal form

EIGEN Call	computes eigenvalues and eigenvectors of symmetric matrices
EIGVAL Function	computes eigenvalues
EIGVEC Function	computes eigenvectors
GENEIG Call	computes eigenvalues and eigenvectors of a generalized eigenproblem
GINV Function	computes the generalized inverse
GSORTH Call	computes the Gram-Schmidt orthonormalization
HALF Function	computes Cholesky decomposition
HANKEL Function	generates a Hankel matrix
HDIR Function	performs a horizontal direct product
HERMITE Function	reduces a matrix to Hermite normal form
HOMOGEN Function	solves homogeneous linear systems
INV Function	produces the inverse
INVUPDT Function	updates a matrix inverse
LUPDT Call	provides updating and downdating for rank deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses
QR Call	produces the QR decomposition of a matrix by Householder transformations
RDODT Call	downdate and update QR and Cholesky decompositions
ROOT Function	performs the Cholesky decomposition of a matrix
RUPDT Call	update QR and Cholesky decompositions
RZLIND Call	update QR and Cholesky decompositions
SOLVE Function	solves a system of linear equations
SVD Call	computes the singular value decomposition
TOEPLITZ Function	generates a Toeplitz or block-Toeplitz matrix
TRACE Function	sums diagonal elements
TRISOLV Function	solves linear systems with triangular matrices
XMULT Function	performs accurate matrix multiplication

Optimization Subroutines

LCP Call	solves the linear complementarity problem
LP Call	solves the linear programming problem
NLPCG Call	nonlinear optimization by conjugate gradient method
NLPDD Call	nonlinear optimization by double dogleg method
NLPFDD Call	approximates derivatives by finite differences method
NLPFEA Call	computes feasible points subject to constraints
NLPHQN Call	calculates hybrid quasi-Newton least squares
NLPLM Call	calculates Levenberg-Marquardt least squares
NLPNMS Call	nonlinear optimization by Nelder-Mead simplex method
NLPNRA Call	nonlinear optimization by Newton-Raphson method
NLPNRR Call	nonlinear optimization by Newton-Raphson ridge method
NLPQN Call	nonlinear optimization by quasi-Newton method
NLPQUA Call	nonlinear optimization by quadratic method
NLPTR Call	nonlinear optimization by trust region method

Set Functions

SETDIF Function	compares elements of two matrices
UNION Function	performs unions of sets
UNIQUE Function	sorts and removes duplicates
XSECT Function	intersects sets

Control Statements

ABORT Statement	ends IML
APPLY Function	applies an IML module
DO and END Statements	groups statements as a unit
DO, Iterative Statement	iteratively executes a Do group
DO and UNTIL Statement and Clause	conditionally executes statements iteratively
DO and WHILE Statement and Clause	conditionally executes statements iteratively
END Statement	ends a DO loop or DO statement
EXECUTE Call	executes SAS statements immediately
FINISH Statement	denotes the end of a module
FORCE Statement	see the description of the SAVE statement
FREE Statement	frees matrix storage space
GOTO Statement	jumps to a new statement
IFTHEN Statement	conditionally executes statement
LINK Statement	jump to another statement
MATTRIB Statement	associates printing attributes with matrices
PARSE Statement	parses matrix elements as statements
PAUSE Statement	interrupts module execution
PRINT Statement	prints matrix values
PURGE Statement	removes observations marked for deletion and renumbers records
PUSH Call	pushes SAS statements into the command input stream
QUEUE Call	queues SAS statements into the command input stream
QUIT Statement	exits from IML
REMOVE Statement	removes matrices from storage
RESET Statement	sets processing options
RESUME Statement	resumes execution
RETURN Statement	returns to caller
RUN Statement	executes statements in a module
SHOW Statement	prints system information
SOUND Call	produces a tone
START/FINISH Statements	define a module
STOP Statement	stops execution of statements
STORAGE Function	lists names of matrices and modules in storage
STORE Statement	stores matrices and modules in library storage
VALSET Call	performs indirect assignment
VALUE Function	assigns values by indirect reference

Dataset and File Functionality

APPEND Statement	adds observations to SAS dataset
CLOSE Statement	closes a SAS dataset
CLOSEFILE Statement	closes a file
CONTENTS Function	returns the variables in a SAS dataset
CREATE Statement	creates a new SAS dataset
DATASETS Function	obtains the names of SAS datasets
DELETE Call	deletes a SAS data set
DELETE Statement	marks observations for deletion
DO DATA Statement	repeats a loop until an end of file occurs
EDIT Statement	opens a SAS data set for editing
FILE Statement	opens or points to an external file
FIND Statement	finds observations
INDEX Statement	indexes a variable in a SAS data set
INFILE Statement	opens a file for input
INPUT Statement	inputs data
LIST Statement	displays observations of a data set
LOAD Statement	loads modules and matrices from library storage
PUT Statement	writes data to an external file
READ Statement	reads observations from a data set
RENAME Call	renames a SAS data set
REPLACE Statement	replaces values in observations and updates observations
SAVE Statement	saves data
SETIN Statement	makes a data set current for input
SETOUT Statement	makes a data set current for output
SORT Statement	sorts a SAS data set
SUMMARY Statement	computes summary statistics for SAS data sets
USE Statement	opens a SAS data set for reading

Graphics and Window Functions

DISPLAY Statement	displays fields in a display window
GBLKVP Call	defines a blanking viewport
GBLKVPD Call	deletes the blanking viewport
GCLOSE Call	closes the graphics segment
GDELETE Call	deletes a graphics segment
GDRAW Call	draws a polyline
GDRAWL Call	draws individual lines
GGRID Call	draws a grid
GINCLUDE Call	includes a graphics segment
GOPEN Call	opens a graphics segment
GPIE Call	draws pie slices
GPIEXY Call	converts from polar to world coordinates
GPOINT Call	plots points
GPOLY Call	draws and fills a polygon
GPORT Call	defines a viewport
GPORTPOP Call	pops the viewport
GPORTSTK Call	stacks the viewport

GSCALE Call	calculates round numbers for labeling axes
GSCRIPT Call	writes multiple text strings with special fonts
GSET Call	sets attributes for a graphics segment
GSHOW Call	shows a graph
GSTART Call	initializes the graphics system
GSTOP Call	deactivates the graphics system
GSTRLEN Call	finds the string length
GTEXT and GVTEXT Calls	place text horizontally or vertically on a graph
GWINDOW Call	defines the data window
GXAXIS and GYAXIS Calls	draw a horizontal or vertical axis
PGRAF Call	produces scatter plots
WINDOW Statement	opens a display window

Operators

All operators available in SAS/IML software are described in this section.

Addition Operator: +

adds corresponding matrix elements

matrix1 **+** *matrix2*
matrix **+** *scalar*

The addition infix operator (+) produces a new matrix containing elements that are the sums of the corresponding elements of *matrix1* and *matrix2*. The element in the first row, first column of the first matrix is added to the element in the first row, first column of the second matrix, with the sum becoming the element in the first row, first column of the new matrix, and so on.

For example, the statements

```
a={1 2,
   3 4};
b={1 1,
   1 1};
c=a+b;
```

produce the matrix **C**.

```
C              2 rows      2 cols    (numeric)

                       2         3
                       4         5
```

In addition to adding conformable matrices, you can also use the addition operator to add a matrix and a scalar or two scalars. When you use the *matrix + scalar* (or *scalar + matrix*) form, the scalar value is added to each element of the matrix to produce a new matrix.

For example, you can obtain the same result as you did in the previous example with the statement

```
c=a+1;
```

When a missing value occurs in an operand, IML assigns a missing value for the corresponding element in the result.

You can also use the addition operator on character operands. In this case, the operator does elementwise concatenation exactly as the CONCAT function.

Comparison Operators: < > = <= >= ^=

compare matrix elements

matrix1<*matrix2*
matrix1<=*matrix2*

matrix1>*matrix2*

matrix1>=*matrix2*

matrix1=*matrix2*

matrix1^=*matrix2*

The comparison operators compare two matrices element by element and produce a new matrix that contains only zeros and ones. If an element comparison is true, the corresponding element of the new matrix is 1. If the comparison is not true, the corresponding element is 0. Unlike in base SAS software or the MATRIX procedure, you cannot use the English equivalents GT and LT for the greater than and less than signs. Scalar values can be used instead of matrices in any of the forms shown above.

For example, let

```
a={1 7 3,
   6 2 4};
```

and

```
b={0 8 2,
   4 1 3};
```

Evaluation of the expression

```
c=a>b;
```

results in the matrix of values

```
C            2 rows      3 cols    (numeric)

               1         0         1
               1         1         1
```

In addition to comparing conformable matrices, you can apply the comparison operators to a matrix and a scalar. If either argument is a scalar, the comparison is between each element of the matrix and the scalar.

For example the expression

```
d=(a>=2);
```

yields the result

```
D            2 rows      3 cols    (numeric)

               0         1         1
               1         1         1
```

If the element lengths of two character operands are different, the shorter elements are padded on the right with blanks for the comparison.

If a numeric missing value occurs in an operand, IML treats it as lower than any valid number for the comparison.

When you are making conditional comparisons, all values of the result must be nonzero for the condition to be evaluated as true.

Consider the following statement:

```
if x>=y then goto loop1;
```

The GOTO statement is executed only if every element of **x** is greater than or equal to the corresponding element in **y**. See also the descriptions of the ALL and ANY functions.

Concatenation Operator, Horizontal: ||

concatenates matrices horizontally

matrix1||*matrix2*

The horizontal concatenation operator (||) produces a new matrix by horizontally joining *matrix1* and *matrix2*. *Matrix1* and *matrix2* must have the same number of rows, which is also the number of rows in the new matrix. The number of columns in the new matrix is the number of columns in *matrix1* plus the number of columns in *matrix2*.

For example, the statements

```
a={1 1 1,
   7 7 7};
b={0 0 0,
   8 8 8};
c=a||b;
```

result in

```
C             2 rows      6 cols    (numeric)

1         1         1         0         0         0
7         7         7         8         8         8
```

Also, if

```
b={A B C,
   D E F};
```

and

```
c={"GH" "IJ",
   "KL" "MN"};
```

then

```
a=b||c;
```

results in

```
A         2 rows      5 cols    (character, size 2)

          A  B  C  GH IJ
          D  E  F  KL MN
```

For character operands, the element size in the result matrix is the larger of the two operands. In the preceding example, **A** has element size 2.

You can use the horizontal concatenation operator when one of the arguments has no value. For example, if **A** has not been defined and **B** is a matrix, **A** ||**B** results in a new matrix equal to **B**.

Quotation marks (") are needed around matrix elements only if you want to embed blanks or maintain uppercase and lowercase distinctions.

Concatenation Operator, Vertical: //

concatenates matrices vertically

matrix1//*matrix2*

The vertical concatenation operator (//) produces a new matrix by vertically joining *matrix1* and *matrix2*. *Matrix1* and *matrix2* must have the same number of columns, which is also the number of columns in the new matrix. For example, if **A** has three rows and two columns and **B** has four rows and two columns, then **A**//**B** produces a matrix with seven rows and two columns. Rows 1 through 3 of the new matrix correspond to **A**; rows 4 through 7 correspond to **B**.

For example, the statements

```
a={1 1 1,
   7 7 7};
b={0 0 0,
   8 8 8};
c=a//b;
```

result in

```
            C            4 rows      3 cols    (numeric)

                         1         1         1
                         7         7         7
                         0         0         0
                         8         8         8
```

Also let

```
b={"AB" "CD",
   "EF" "GH"};
```

and

```
c={"I" "J",
   "K" "L",
   "M" "N"};
```

Then the statement

```
a=b//c;
```

produces the new matrix

```
A        5 rows      2 cols    (character, size 2)

                     AB CD
                     EF GH
                     I  J
                     K  L
                     M  N
```

For character matrices, the element size of the result matrix is the larger of the element sizes of the two operands.

You can use the vertical concatenation operator when one of the arguments has not been assigned a value. For example, if **A** has not been defined and **B** is a matrix, **A**//**B** results in a new matrix equal to **B**.

Quotation marks (") are needed around matrix elements only if you want to embed blanks or maintain uppercase and lowercase distinctions.

Direct Product Operator: @

takes the direct product of two matrices

matrix1@matrix2

The direct product operator (@) produces a new matrix that is the direct product (also called the *Kronecker product*) of *matrix1* and *matrix2*, usually denoted by $\mathbf{A} \otimes \mathbf{B}$. The number of rows in the new matrix equals the product of the number of rows in *matrix1* and the number of rows in *matrix2*; the number of columns in the new matrix equals the product of the number of columns in *matrix1* and the number of columns in *matrix2*.

For example, the statements

```
a={1 2,
   3 4};
b={0 2};
c=a@b;
```

result in

```
C              2 rows      4 cols    (numeric)

                0         2         0         4
                0         6         0         8
```

The statement

```
d=b@a;
```

results in

```
D              2 rows      4 cols    (numeric)

               0           0         2         4
               0           0         6         8
```

Division Operator: /

performs elementwise division

matrix1/*matrix2*
matrix/*scalar*

The division operator (/) divides each element of *matrix1* by the corresponding element of *matrix2*, producing a matrix of quotients.

In addition to dividing elements in conformable matrices, you can also use the division operator to divide a matrix by a scalar. If either operand is a scalar, the operation does the division for each element and the scalar value.

When a missing value occurs in an operand, the IML procedure assigns a missing value for the corresponding element in the result.

If a divisor is zero, the procedure prints a warning and assigns a missing value for the corresponding element in the result. An example of a valid statement using this operater follows:

```
c=a/b;
```

Element Maximum Operator: <>

selects the larger of two elements

matrix1<>*matrix2*

The element maximum operator (<>) compares each element of *matrix1* to the corresponding element of *matrix2*. The larger of the two values becomes the corresponding element of the new matrix that is produced.

When either argument is a scalar, the comparison is between each matrix element and the scalar.

The element maximum operator can take as operands two character matrices of the same dimensions or a character matrix and a character string. If the element lengths

of the operands are different, the shorter elements are padded on the right with blanks. The element length of the result is the longer of the two operand element lengths.

When a missing value occurs in an operand, IML treats it as smaller than any valid number.

For example, the statements

```
a={2 4 6, 10 11 12};
b={1 9 2, 20 10 40};
c=a<>b;
```

produce the result

```
              C             2 rows      3 cols    (numeric)

                              2          9          6
                             20         11         40
```

Element Minimum Operator: ><

selects the smaller of two elements

matrix1`><`*matrix2*

The element minimum operator (><) compares each element of *matrix1* with the corresponding element of *matrix2*. The smaller of the values becomes the corresponding element of the new matrix that is produced.

When either argument is a scalar, the comparison is between the scalar and each element of the matrix.

The element minimum operator can take as operands two character matrices of the same dimensions or a character matrix and a character string. If the element lengths of the operands are different, the shorter elements are padded on the right with blanks. The element length of the result is the longer of the two operand element lengths.

When a missing value occurs in an operand, IML treats it as smaller than any valid numeric value.

For example, the statements

```
a={2 4 6, 10 11 12};
b={1 9 2, 20 10 40};
c=a><b;
```

produce the result

```
              C             2 rows      3 cols    (numeric)

                              1          4          2
                             10         10         12
```

Index Creation Operator: :

creates an index vector

value1:*value2*

The index creation operator (:) creates a row vector with a first element that is *value1*. The second element is *value1*+1, and so on, as long as the elements are less than or equal to *value2*. For example, the statement

```
I=7:10;
```

results in

```
I             1 row       4 cols    (numeric)

              7         8         9        10
```

If *value1* is greater than *value2*, a reverse order index is created. For example, the statement

```
r=10:6;
```

results in the row vector

```
R             1 row       5 cols    (numeric)

10          9         8         7        6
```

The index creation operator also works on character arguments with a numeric suffix. For example, the statement

```
varlist='var1':'var5';
```

results in

```
VARLIST    1 row       5 cols    (character, size 4)

           var1  var2  var3  var4  var5
```

Use the DO function if you want an increment other than 1 or –1. See the description of the DO function later in this chapter.

Logical Operators: & | ^

perform elementwise logical comparisons

matrix1&*matrix2*
matrix&*scalar*

matrix1|*matrix2*

matrix|*scalar*

^ *matrix*

The AND logical operator (&) compares two matrices, element by element, to produce a new matrix. An element of the new matrix is 1 if the corresponding elements of *matrix1* and *matrix2* are both nonzero; otherwise, it is a zero.

An element of the new matrix produced by the OR operator (|) is 1 if either of the corresponding elements of *matrix1* and *matrix2* is nonzero. If both are zero, the element is zero.

The NOT prefix operator (^) examines each element of a matrix and produces a new matrix containing elements that are ones and zeros. If an element of *matrix* equals 0, the corresponding element in the new matrix is 1. If an element of *matrix* is nonzero, the corresponding element in the new matrix is 0.

The following statements illustrate the use of these logical operators:

```
z=x&r;
if a|b then print c;
if ^m then link x1;
```

Multiplication Operator, Elementwise:

performs elementwise multiplication

matrix1#*matrix2*
matrix#*scalar*

matrix#*vector*

The elementwise multiplication operator (#) produces a new matrix with elements that are the products of the corresponding elements of *matrix1* and *matrix2*.

For example, the statements

```
a={1 2,
   3 4};
b={4 8,
   0 5};
c=a#b;
```

result in the matrix

```
              C             2 rows      2 cols    (numeric)

                                4          16
                                0          20
```

In addition to multiplying conformable matrices, you can use the elementwise multiplication operator to multiply a matrix and a scalar. When either argument is a scalar, the scalar value is multiplied by each element in *matrix* to form the new matrix.

You can also multiply vectors by matrices. You can multiply matrices as long as they either conform in each dimension or one operand has dimension value 1. For example, a 2×3 matrix can be multiplied on either side by a 2×3, a 1×3, a 2×1, or a 1×1 matrix. Multiplying the 2×2 matrix $\mathbf{A}$ by the column vector $\mathbf{D}$, as in

```
d={10,100};
ad=a#d;
```

produces the matrix

```
              AD            2 rows      2 cols    (numeric)

                               10          20
                              300         400
```

whereas the statements

```
d={10 100};
ad=a#d;
```

produce the matrix

```
              AD            2 rows      2 cols    (numeric)

                               10         200
                               30         400
```

The result of elementwise multiplication is also known as the Schur or Hadamard product. Element multiplication (using the # operator) should not be confused with matrix multiplication (using the * operator).

When a missing value occurs in an operand, IML assigns a missing value in the result.

Multiplication Operator, Matrix: *

performs matrix multiplication

*matrix1***matrix2*

The matrix multiplication infix operator (*) produces a new matrix by performing matrix multiplication. The first matrix must have the same number of columns as the second matrix has rows. The new matrix has the same number of rows as the first matrix and the same number of columns as the second matrix. The matrix multiplication operator does not consistently propagate missing values.

For example, the statements

```
a={1 2,
   3 4};
b={1 2};
c=b*a;
```

result in

```
            C             1 row       2 cols    (numeric)

                               7          10
```

and the statement

```
d=a*b';
```

results in

```
             D             2 rows      1 col     (numeric)

                                5
                               11
```

Power Operator, Elementwise:

raises each element to a power

matrix1##*matrix2*
matrix##*scalar*

The elementwise power operator (##) creates a new matrix with elements that are the elements of *matrix1* raised to the power of the corresponding element of *matrix2*. If any value in *matrix1* is negative, the corresponding element in *matrix2* must be an integer.

In addition to handling conformable matrices, the elementwise power operator allows either operand to be a scalar. In this case, the operation takes the power for each element and the scalar value. Missing values are propagated if they occur.

For example, the statements

```
a={1 2 3};
b=a##3;
```

result in

```
B            1 row      2 cols     (numeric)

                  1          8         27
```

The statement

```
b=a##.5;
```

results in

```
 B            1 row       3 cols    (numeric)

1 1.4142136 1.7320508
```

Power Operator, Matrix: **

raises a matrix to a power

*matrix***scalar*

The matrix power operator (**) creates a new matrix that is *matrix* multiplied by itself *scalar* times. *Matrix* must be square; *scalar* must be an integer greater than or equal to −1. Large scalar values cause numerical precision problems. If the scalar is not an integer, it is truncated to an integer.

For example, the statements

```
a={1 2,
   1 1};
c=a**2;
```

result in

```
C           2 rows       2 cols    (numeric)

            3         4
            2         3
```

If the matrix is symmetric, it is preferable to power its eigenvalues rather than using the matrix power operator directly on the matrix (see the description of the EIGEN call). Note that the expression

```
A**(-1)
```

is permitted and is equivalent to **INV(A)**.

The matrix power operater does not support missing values.

Sign Reverse Operator: –

reverses the signs of elements

–matrix

The sign reverse prefix operator (–) produces a new matrix containing elements that are formed by reversing the sign of each element in *matrix*. A missing value is assigned if the element is missing.

For example, the statements

```
a={-1  7  6,
   2  0 -8};
b=-a;
```

result in the matrix

```
            B            2 rows      3 cols    (numeric)

                           1          -7         -6
                          -2           0          8
```

Subscripts: [ ]

select submatrices

matrix[*rows,columns*]
matrix[*elements*]

Subscripts are used with matrices to select submatrices, where *rows* and *columns* are expressions that evaluate to scalars or numeric vectors. These expressions contain valid subscript values of rows and columns in the argument matrix. A subscripted matrix can appear on the left side of the equal sign. The dimensions of the target submatrix must conform to the dimensions of the source matrix. See Chapter 4, "Working with Matrices," for further information.

For example, the statements

```
x={1 2 3,
   4 5 6,
   7 8 9};
a=3;
m=x[2,a];
```

select the element in the second row and third column of **X** and produce the matrix **M**:

```
M             1 row      1 col    (numeric)

                          6
```

The statements

```
a=1:3;
m=x[2,a];
```

select row 2, and columns 1 through 3 of **X**, producing the matrix **M**:

```
M             1 row      3 cols    (numeric)

                4         5         6
```

Subtraction Operator: –

subtracts corresponding matrix elements

matrix1–matrix2
matrix–scalar

The subtraction infix operator (–) produces a new matrix containing elements that are formed by subtracting the corresponding elements of *matrix2* from those of *matrix1*.

In addition to subtracting conformable matrices, you can also use the subtraction operator to subtract a matrix and a scalar. When either argument is a scalar, the operation is performed by using the scalar against each element of the matrix argument.

When a missing value occurs in an operand, IML assigns a missing value for the corresponding element in the result.

An example of a valid statement follows:

```
c=a-b;
```

Transpose Operator: ‘

transposes a matrix

matrix‘

The transpose operator (denoted by the backquote ‘ character) exchanges the rows and columns of *matrix*, producing the transpose of *matrix*. For example, if an element in *matrix* is in the first row and second column, it is in the second row and first column of the transpose; an element in the first row and third column of *matrix* is in the third row and first column of the transpose, and so on. If *matrix* contains three rows and two columns, its transpose has two rows and three columns.

For example, the statements

```
a={1 2,
   3 4,
   5 6};
b=a‘;
```

result in

```
B              2 rows      3 cols    (numeric)

                 1         3         5
                 2         4         6
```

If your keyboard does not have a backquote character, you can transpose a matrix with the T (transpose) function, documented later in this chapter.

Statements, Functions, and Subroutines

This section presents descriptions of all statements, functions, and subroutines available in IML.

ABORT Statement

stops execution and exits IML

ABORT;

The ABORT statement instructs IML to stop executing statements. It also stops IML from parsing any further statements, causing IML to close its files and exit. See also the description of the STOP statement.

ABS Function

takes the absolute value

ABS(*matrix***)**

where *matrix* is a numeric matrix or literal.

The ABS function is a scalar function that returns the absolute value of every element of the argument matrix. An example of how to use the ABS function follows.

```
c=abs(a);
```

ALL Function

checks for all elements nonzero

ALL(*matrix***)**

where *matrix* is a numeric matrix or literal.

The ALL function returns a value of 1 if all elements in *matrix* are nonzero. If any element of *matrix* is zero, the ALL function returns a value of 0. Missing values in *matrix* are treated as zeros.

You can use the ALL function to express the results of a comparison operator as a single 1 or 0. For example, the comparison operation $\mathbf{A} > \mathbf{B}$ yields a matrix containing elements that can be either ones or zeros. All the elements of the new matrix are ones only if each element of $\mathbf{A}$ is greater than the corresponding element of $\mathbf{B}$.

For example, consider the statement

```
if all(a>b) then goto loop;
```

IML executes the GOTO statement only if every element of **A** is greater than the corresponding element of **B**. The ALL function is implicitly applied to the evaluation of all conditional expressions. The statements

```
if (a>b) then goto loop;
```

and

```
if all(a>b) then goto loop;
```

have the same effect.

ANY Function

checks for any nonzero element

ANY(*matrix***)**

where *matrix* is a numeric matrix or literal.

The ANY function returns a value of 1 if any of the elements in *matrix* are nonzero. If all the elements of *matrix* are zeros, the ANY function returns a value of 0. Missing values in *matrix* are treated as zeros.

For example, consider the statement

```
if any(a=b) then print a b;
```

The matrices **A** and **B** are printed if at least one value in **A** is the same as the corresponding value in **B**. The following statements do not print the message:

```
a={-99 99};
b={-99 98};
if a^=b then print 'a^=b';
```

However, the following statement prints the message:

```
if any(a^=b) then print 'a^=b';
```

APPCORT Call

applies complete orthogonal decomposition by Householder transformations on the right-hand-side matrix, B for the solution of rank-deficient linear least-squares systems

CALL APPCORT(*prqb, lindep, a, b* **<,** *sing*>**);**

The inputs to the APPCORT subroutine are:

a is an $m \times n$ matrix $\mathbf{A}$, with $m \geq n$, which is to be decomposed into the product of the $m \times m$ orthogonal matrix $\mathbf{Q}$, the $n \times n$ upper triangular matrix $\mathbf{R}$, and the $n \times n$ orthogonal matrix $\mathbf{P}$,

$$\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \mathbf{\Pi}' \mathbf{P}' \mathbf{\Pi}$$

b is the $m \times p$ matrix $\mathbf{B}$ that is to be left multiplied by the transposed $m \times m$ matrix $\mathbf{Q}'$.

sing is an optional scalar specifying a singularity criterion.

The APPCORT subroutine returns the following values:

prqb is an $n \times p$ matrix product

$$\mathbf{P} \mathbf{\Pi} \begin{bmatrix} (\mathbf{L}')^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{Q}' \mathbf{B}$$

which is the minimum 2-norm solution of the (rank deficient) least-squares problem $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$. Refer to Golub and Van Loan (1989, pp. 241−242) for more details.

lindep is the number of linearly dependent columns in the matrix $\mathbf{A}$ detected by applying the r Householder transformations. That is, *lindep*$= n - r$, where $r = \text{rank}(\mathbf{A})$.

See "COMPORT Call" for information on complete orthogonal decomposition.

APPEND Statement

adds observations to the end of a SAS data set

APPEND < **VAR** *operand* > **;**
APPEND < **FROM** *from-name* < **[ROWNAME=***row-name***]** > > **;**

In the preceding statements,

operand can be specified as one of the following:

- a literal containing variable names
- a character matrix containing variable names
- an expression in parentheses yielding variable names
- one of the keywords described below:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

from-name is the name of a matrix containing data to append.

row-name is a character matrix or quoted literal containing descriptive row names.

Use the APPEND statement to add data to the end of the current output data set. The appended observations are from either the variables specified in the VAR clause or variables created from the columns of the FROM matrix. The FROM clause and the VAR clause should not be specified together.

You can specify a set of variables to use with the VAR clause.

Following are examples showing each possible way you can use the VAR clause.

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

If the VAR clause includes a matrix with more than one row and column, the APPEND statement adds one observation for each element in the matrix with the greatest number of elements. Elements are appended in row-major order. Variables in the VAR clause with fewer than the maximum number of elements contribute missing values to observations after all of their elements have been used.

The default variables for the APPEND statement are all matrices that match variables in the current data set with respect to name and type.

The ROWNAME= operand to the FROM clause specifies the name of a character matrix to contain row titles. The first *nrow* values of this matrix become values of a

variable with the same name in the output data set; *nrow* is the number of rows in the FROM matrix. The procedure uses the first *nrow* elements in row-major order.

Examples using the APPEND statement follow. The first example shows the use of the FROM clause when creating a new data set. See also the section "CREATE Statement" on page 500

```
x={1 2 3, 4 5 6};
create mydata from x[colname={x1 x2 x3}];
append from x;
show contents;
   /* shows 3 variables (x1 x2 x3) and 2 observations */
```

The next example shows the use of the VAR clause for selecting variables from which to append data.

```
names={'Jimmy' 'Sue' 'Ted'};
sex={m f m};
create folks var{names sex};
append;
show contents;
/* shows 2 variables (names,sex) and 3 observations in FOLKS */
```

You could achieve the same result with the statements

```
dsvar={names sex};
create folks var dsvar;
append;
```

APPLY Function

applies an IML module to its arguments

APPLY(*modname, argument1*<*, argument2,. . ., argument15*>**)**

In the preceding statement,

modname is the name of an existing module, supplied in quotes, as a matrix containing the module name, or an expression rendering the module name.

argument is an argument passed to the module. You must have at least one argument. You can specify up to 15 arguments.

The APPLY function applies a user-defined IML module to each element of the argument matrix or matrices and returns a matrix of results. The first argument to APPLY is the name of the module. The module must already be defined before the APPLY function is executed. The module must be a function module, capable of returning a result.

The subsequent arguments to the APPLY function are the arguments passed to the module. They all must have the same dimension. If the module takes n arguments, *argument1* through *argumentn* should be passed to APPLY where $1 \leq n \leq 15$. The APPLY function effectively calls the module. The result has the same dimension as the input arguments, and each element of the result corresponds to the module applied to the corresponding elements of the argument matrices. The APPLY function can work on numeric as well as character arguments. For example, the following statements define module ABC and then call the APPLY function, with matrix **A** as an argument:

```
start abc(x);
   r=x+100;
   return (r);
finish abc;

a={6 7 8,
   9 10 11};
r=apply("ABC",a);
```

The result is

```
            R             2 rows      3 cols    (numeric)

                          106       107       108
                          109       110       111
```

In the next example, the statements define the module SWAP and call the APPLY function:

```
start swap(a,b,c);
   r=a*b*c;
   a=b;
   if r<0 then return(0);
   return(r);
finish swap;

a={2  3, 4 5};
b={4  3, 5 6};
c={9 -1, 3 7};
mod={swap};
r=apply(mod,a,b,c);
print a r;
```

The results are

```
                    A                 R
                    4         3      72         0
                    5         6      60       210
```

ARMACOV Call

computes an autocovariance sequence for an ARMA model

CALL ARMACOV(*auto, cross, convol, phi, theta, num***);**

The inputs to the ARMACOV subroutine are as follows:

phi refers to a $1 \times (p+1)$ matrix containing the autoregressive parameters. The first element is assumed to have the value 1.

theta refers to a $1 \times (q+1)$ matrix containing the moving-average parameters. The first element is assumed to have the value 1.

num refers to a scalar containing n, the number of autocovariances to be computed, which must be a positive number.

The ARMACOV subroutine returns the following values:

auto specifies a variable to contain the returned $1 \times n$ matrix containing the autocovariances of the specified ARMA model, assuming unit variance for the innovation sequence.

cross specifies a variable to contain the returned $1 \times (q+1)$ matrix containing the covariances of the moving-average term with lagged values of the process.

convol specifies a variable to contain the returned $1 \times (q+1)$ matrix containing the autocovariance sequence of the moving-average term.

The ARMACOV subroutine computes the autocovariance sequence that corresponds to a given autoregressive moving-average (ARMA) time-series model. An arbitrary number of terms in the sequence can be requested. Two related covariance sequences are also returned.

The model notation for the ARMACOV and ARMALIK subroutines is the same. The ARMA(p, q) model is denoted

$$\sum_{j=0}^{p} \phi_j y_{t-j} = \sum_{i=0}^{q} \theta_i \epsilon_{t-i}$$

with $\theta_0 = \phi_0 = 1$. The notation is the same as that of Box and Jenkins (1976) except that the model parameters are opposite in sign. The innovations $\{\epsilon_t\}$ satisfy $E(\epsilon_t) = 0$ and $E(\epsilon_t \epsilon_{t-k}) = 1$ if k=0, and are zero otherwise. The formula for the kth element of the *convol* argument is

$$\sum_{i=k-1}^{q} \theta_i \theta_{i-k+1}$$

for $k = 1, 2, \ldots, q+1$. The formula for the kth element of the *cross* argument is

$$\sum_{i=k-1}^{q} \theta_i \psi_{i-k+1}$$

for $k = 1, 2, \ldots, q+1$, where ψ_i is the ith impulse response value. The ψ_i sequence, if desired, can be computed with the RATIO function. It can be shown that ψ_k is the same as $E(Y_{t-k}\epsilon_t^2)/\sigma$, which is used by Box and Jenkins (1976, p. 75) in their formulation of the autocovariances. The kth autocovariance, denoted γ_k and returned as the $k+1$ element of the *auto* argument $(k = 0, 1, \ldots, n-1)$, is defined implicitly for $k > 0$ by

$$\sum_{i=0}^{p} \gamma_{k-i}\phi_i = \eta_k$$

where η_k is the kth element of the *cross* argument. See Box and Jenkins (1976) or McLeod (1975) for more information.

To compute the autocovariance function at lags zero through four for the model

$$y_t = 0.5y_{t-1} + e_t + 0.8e_{t-1}$$

use the following statements:

```
proc iml;
      /* an arma(1,1) model */
   phi  ={1 -0.5};
   theta={1 0.8};
   call armacov(auto,cross,convol,phi,theta,5);
   print auto,,cross convol;
```

The result is

```
            AUTO
3.2533333 2.4266667 1.2133333 0.6066667 0.3033333

                  CROSS              CONVOL
                   2.04       0.8      1.64       0.8
```

ARMALIK Call

computes the log likelihood and residuals for an ARMA model

CALL ARMALIK(*lnl, resid, std, x, phi, theta***);**

The inputs to the ARMALIK subroutine are as follows:

x	is an $n \times 1$ or $1 \times n$ matrix containing values of the time series (assuming mean zero).
phi	is a $1 \times (p + 1)$ matrix containing the autoregressive parameter values. The first element is assumed to have the value 1.
theta	is a $1 \times (q + 1)$ matrix containing the moving-average parameter values. The first element is assumed to have the value 1.

The ARMALIK subroutine returns the following values:

lnl	specifies a 3×1 matrix containing the log likelihood concentrated with respect to the innovation variance; the estimate of the innovation variance (the unconditional sum of squares divided by n); and the log of the determinant of the variance matrix, which is standardized to unit variance for the innovations.
resid	specifies an $n \times 1$ matrix containing the standardized residuals. These values are uncorrelated with a constant variance if the specified ARMA model is the correct one.
std	specifies an $n \times 1$ matrix containing the scale factors used to standardize the residuals. The actual residuals from the one-step-ahead predictions using the past values can be computed as *std#resid*.

The ARMALIK subroutine computes the concentrated log likelihood function for an ARMA model. The unconditional sum of squares is readily available, as are the one-step-ahead prediction residuals. Factors that can be used to generate confidence limits associated with prediction from a finite past sample are also returned.

The notational conventions for the ARMALIK subroutine are the same as those used by the ARMACOV subroutine. See the description of the ARMACOV call for the model employed. In addition, the condition $\sum_{i=0}^{q} \theta_{iz}^{i} \neq 0$ for $|z| < 1$ should be satisfied to guard against floating-point overflow.

If the column vector $\mathbf{x}$ contains n values of a time series and the variance matrix is denoted $\Sigma = \sigma^2 \mathbf{V}$, where σ^2 is the variance of the innovations, then, up to additive constants, the log likelihood, concentrated with respect to σ^2, is

$$-\frac{n}{2} \log \left(\mathbf{x}' \mathbf{V}^{-1} \mathbf{x} \right) - \frac{1}{2} \log |\mathbf{V}| \ .$$

The matrix $\mathbf{V}$ is a function of the specified ARMA model parameters. If $\mathbf{L}$ is the lower Cholesky root of $\mathbf{V}$ (that is, $\mathbf{V} = \mathbf{L}\mathbf{L}'$), then the standardized residuals are computed as *resid*$= \mathbf{L}^{-1}\mathbf{x}$. The elements of *std* are the diagonal elements of $\mathbf{L}$. The variance estimate is $\mathbf{x}'\mathbf{V}^{-1}\mathbf{x}/n$, and the log determinant is $\log|\mathbf{V}|$. See Ansley (1979) for further details. To compute the log-likelihood for the model

$$y_t - y_t{-}1 + 0.25y_{t-2} = e_t + 0.5e_{t-1}$$

use the following IML code:

```
proc iml;
   phi={ 1 -1 0.25} ;
   theta={ 1 0.5} ;
   x={ 1 2 3 4 5} ;
   call armalik(lnl,resid,std,x,phi,theta);
   print lnl resid std;
```

The printed output is

```
                LNL     RESID       STD
         -0.822608 0.4057513 2.4645637
          0.8721154 0.9198158 1.2330147
          2.3293833 0.8417343 1.0419028
                    1.0854175 1.0098042
                    1.2096421 1.0024125
```

ARMASIM Function

simulates a univariate ARMA series

ARMASIM(*phi, theta, mu, sigma,* n*,* $<$*seed*$>$**)**

The inputs to the ARMASIM function are As follows:

phi	is a $1 \times (p+1)$ matrix containing the autoregressive parameters. The first element is assumed to have the value 1.
theta	is a $1 \times (q+1)$ matrix containing the moving-average parameters. The first element is assumed to have the value 1.
mu	is a scalar containing the overall mean of the series.
sigma	is a scalar containing the standard deviation of the innovation series.
n	is a scalar containing n, the length of the series. The value of n must be greater than 0.
seed	is a scalar containing the random number seed. If it is not supplied, the system clock is used to generate the seed. If it is negative, then the absolute value is used as the starting seed; otherwise, subsequent calls ignore the value of *seed* and use the last seed generated internally.

The ARMASIM function generates a series of length n from a given autoregressive moving-average (ARMA) time series model and returns the series in an $n \times 1$ matrix. The notational conventions for the ARMASIM function are the same as those used by the ARMACOV subroutine. See the description of the ARMACOV call for the model employed. The ARMASIM function uses an exact simulation algorithm as described in Woodfield (1988). A sequence $Y_0, Y_1, \ldots, Y_{p+q-1}$ of starting values is produced using an expanded covariance matrix, and then the remaining values are generated using the recursion form of the model, namely

$$Y_t = -\sum_{i=1}^{p} \phi_i Y_{t-i} + \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \quad t = p+q,\ p+q+1, \ldots,\ n-1\ .$$

The random number generator RANNOR is used to generate the noise component of the model. Note that the statement

armasim(1,1,0,1,*n*,*seed*);

returns n standard normal pseudo-random deviates.

For example, to generate a time series of length 10 from the model

$$y_t = 0.5 y_{t-1} + e_t + 0.8 e_{t-1}$$

use the following code to produce the result shown:

```
proc iml;
   phi={1 -0.5};
   theta={1 0.8};
   y=armasim(phi, theta, 0, 1, 10, -1234321);
   print y;
```

```
                                Y

                        2.3253578
                         0.975835
                        -0.376358
                        -0.878433
                        -2.515351
                        -3.083021
                        -1.996886
                        -1.839975
                        -0.214027
                        1.4786717
```

BLOCK Function

forms block-diagonal matrices

BLOCK(*matrix1*<, *matrix2*,..., *matrix15*>**)**

where *matrix* is a numeric matrix or literal.

The BLOCK function creates a new block-diagonal matrix from all the matrices specified in the argument matrices. Up to 15 matrices can be specified. The matrices are combined diagonally to form a new matrix. For example, the statement

```
block(a,b,c);
```

produces a matrix of the form

$$\begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{bmatrix}$$

The statements

```
a={2 2,
   4 4} ;
b={6 6,
   8 8} ;
c=block(a,b);
```

result in the matrix

```
C              4 rows      4 cols    (numeric)

               2           2         0         0
               4           4         0         0
               0           0         6         6
               0           0         8         8
```

BRANKS Function

computes bivariate ranks

BRANKS(*matrix***)**

where *matrix* is an $n \times 2$ numeric matrix.

The BRANKS function calculates the tied ranks and the bivariate ranks for an $n \times 2$ matrix and returns an $n \times 3$ matrix of these ranks. The tied ranks of the first column of *matrix* are contained in the first column of the result matrix; the tied ranks of the

second column of *matrix* are contained in the second column of the result matrix; and the bivariate ranks of *matrix* are contained in the third column of the result matrix.

The tied rank of an element x_j of a vector is defined as

$$\mathbf{R}_i = \frac{1}{2} + \sum_j u(x_i - x_j)$$

where

$$u(t) = \begin{cases} 1 & \text{if } t > 0 \\ \frac{1}{2} & \text{if } t = 0 \\ 0 & \text{if } t < 0\,. \end{cases}$$

The bivariate rank of a pair (x_j, y_j) is defined as

$$\mathbf{Q}_i = \frac{3}{4} + \sum_j u(x_i - x_j)u(y_i - y_j)\,.$$

For example, the following statements produce the result shown below:

```
x={1 0,
   4 2,
   3 4,
   5 3,
   6 3};
f=branks(x);
```

```
            F             5 rows      3 cols    (numeric)

                            1           1           1
                            3           2           2
                            2           5           2
                            4         3.5           3
                            5         3.5         3.5
```

BTRAN Function

computes the block transpose

BTRAN(x, n, m**)**

The inputs to the BTRAN function are as follows:

x is an $(inx) \times (jmx)$ numeric matrix.

n is a scalar with a value that specifies the row dimension of the submatrix blocks.

m is a scalar with a value that specifies the column dimension of the submatrix blocks.

The BTRAN function computes the block transpose of a partitioned matrix. The argument x is a partitioned matrix formed from submatrices of dimension $n \times n$. If the ith, jth submatrix of the argument x is denoted $\mathbf{A}_{ij}$, then the ith, jth submatrix of the result is $\mathbf{A}_{ji}$.

The value returned by the BTRAN function is a $(jn) \times (im)$ matrix, the block transpose of x, where the blocks are $n \times m$.

For example, the statements

```
proc iml;
   z=btran({1 2 3 4,
            5 6 7 8},2,2);
   print z;
```

produce the result

```
            Z             4 rows      2 cols    (numeric)

                                  1         2
                                  5         6
                                  3         4
                                  7         8
```

BYTE Function

translates numbers to ordinal characters

BYTE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The BYTE function returns a character matrix with the same shape as the numeric argument. Each element of the result is a single character with an ordinal position in the computer's character set that is specified by the corresponding numeric element in the argument. These numeric elements should generally be in the range 0 to 255.

For example, in the ASCII character set,

```
a=byte(47);
```

specifies that

```
a="/";       /* the slash character */
```

The lowercase alphabet can be generated with

```
y=byte(97:122);
```

which produces

```
Y            1 row      26 cols    (character, size 1)

a b c d e f g h i j k l m n o p q r s t u v w x y z
```

This function simplifies the use of special characters and control sequences that cannot be entered directly using the keyboard into IML source code. Consult the character set tables for the computer you are using to determine the printable and control characters that are available and their ordinal positions.

CALL Statement

calls a subroutine or function

CALL *name* <**(***arguments***)**> **;**

The inputs to the CALL statement are as follows:

name is the name of a user-defined module or an IML subroutine or function.

arguments are arguments to the module or subroutine.

The CALL statement executes a subroutine. The order of resolution for the CALL statement is

1. IML built-in subroutine
2. user-defined module

This resolution order needs to be considered only if you have defined a module with the same name as an IML built-in subroutine.

See also the section on the RUN statement.

CHANGE Call

search and replace text in an array

CALL CHANGE(*matrix, old, new*<, *numchange*>**);**

The inputs to the CHANGE call are as follows:

matrix	is a character matrix or quoted literal.
old	is the string to be changed.
new	is the string to replace the *old* string.
numchange	is the number of times to make the change.

The CHANGE subroutine changes the first *numchange* occurrences of the substring *old* in each element of the character array *matrix* to the form *new*. If *numchange* is not specified, the routine defaults to 1. If *numchange* is 0, the routine changes all occurrences of *old*. If no occurrences are found, the matrix is not changed. For example, the statements

```
a="It was a dark and stormy night.";
call change(a, "night","day");
```

produce

```
A="It was a dark and stormy day."
```

In the *old* operand, the following characters are reserved:

% $ [ ] { } < > − ? * # @ ’ ‘(backquote) ^

CHAR Function

produces a character representation of a numeric matrix

CHAR(*matrix*<, w <, d >>**)**

The inputs to the CHAR function are as follows:

matrix	is a numeric matrix or literal.
w	is the field width.
d	is the number of decimal positions.

The CHAR function takes a numeric matrix as an argument and, optionally, a field width w and a number of decimal positions d. The CHAR function produces a character matrix with dimensions that are the same as the dimensions of the argument

matrix and wtih elements that are character representations of the corresponding numeric elements.

The CHAR function can take one, two, or three arguments. The first argument is the name of a numeric matrix and must always be supplied. The second argument is the field width of the result. If the second argument is not supplied, the system default field width is used. The third argument is the number of decimal positions in the result. If no third argument is supplied, the best representation is used. See also the description of the NUM function, which does the reverse conversion.

For example, the statements

```
%\xxs NUM function\xe
   a={1 2 3 4};
   f=char(a,4,1);
```

produce the result

```
      F            1 row       4 cols    (character, size 4)

                          1.0  2.0  3.0  4.0
```

CHOOSE Function

conditionally chooses and changes elements

CHOOSE(*condition, result-for-true, result-for-false***)**

The inputs to the CHOOSE function are as follows:

condition	is checked for being true or false for each element.
result-for-true	is returned when *condition* is true.
result-for-false	is returned when *condition* is false.

The CHOOSE function examines each element of the first argument for being true (nonzero and not missing) or false (zero or missing). For each true element, it returns the corresponding element in the second argument. For each false element, it returns the corresponding element in the third argument. Each argument must be conformable with the others or be a single element to be propagated.

For example, suppose that you want to choose between x and y according to whether $x\#y$ is odd or even, respectively. The statements

```
x={1, 2, 3, 4, 5};
y={101, 205, 133, 806, 500};
r=choose(mod(x#y,2)=1,x,y);
print x y r;
```

result in

```
X          Y          R
1        101          1
2        205        205
3        133          3
4        806        806
5        500        500
```

Suppose you want all missing values in x to be changed to zeros. Submit the following statements to produce the result shown below:

```
x={1 2 ., 100 . -90, . 5 8};
print x;
```

```
             X             3 rows      3 cols    (numeric)

                             1          2           .
                           100          .         -90
                             .          5           8
```

The following statement replaces the missing values in **X** with zeros:

```
x=choose(x=.,0,x);
print x;
```

```
             X             3 rows      3 cols    (numeric)

                             1          2           0
                           100          0         -90
                             0          5           8
```

CLOSE Statement

closes a SAS data set

CLOSE <*SAS-data-set*>**;**

where *SAS-data-set* can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see Chapter 6, "Working with SAS Data Sets,". Also, refer to the chapter on SAS data sets in *SAS Language Reference: Concepts*. More than one SAS data set can be listed in a CLOSE statement.

The CLOSE statement is used to close one or more SAS data sets opened with the USE, EDIT, or CREATE statements. To find out which data sets are open, use the SHOW *datasets* statement; see also the section on the SAVE statement later in this chapter. IML automatically closes all open data sets when a QUIT statement is executed. See Chapter 6, "Working with SAS Data Sets," for more information. Examples of the CLOSE statement are as follows.

```
close mydata;
close mylib.mydata;
close;                /* closes the current data set */
```

CLOSEFILE Statement

closes an input or output file

CLOSEFILE *files*;

where *files* can be names (for defined filenames), literals, or expressions in parentheses (for filepaths).

The CLOSEFILE statement is used to close files opened by the INFILE or FILE statement. The file specification should be the same as when the file was opened. File specifications are either a name (for a defined filename), a literal, or an expression in parentheses (for a filepath). To find out what files are open, use the statement SHOW *files*. For further information, consult Chapter 7, "File Access." See also the description of the SAVE statement. IML automatically closes all files when a QUIT statement is executed.

Examples of the CLOSEFILE statement are shown below.

```
filename in1 'mylib.mydata';
closefile in1;
```

or

```
closefile 'mylib.mydata';
```

or

```
in='mylib/mydata';
closefile(in);
```

COMPORT Call

provides complete orthogonal decomposition by Householder transformations

CALL COMPORT(q, r, p, *piv, lindep,* a <, *b*><, *sing*>**);**

The COMPORT subroutine returns the following values:

q If b is not specified, q is the $m \times m$ orthogonal matrix $\mathbf{Q}$ that is the product of the $\min(m, n)$ separate Householder transformations. If b is specified, q is the $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$ that has the transposed Householder transformations $\mathbf{Q}'$ applied on the p columns of the argument matrix $\mathbf{B}$.

r is the $n \times n$ upper triangular matrix $\mathbf{R}$ that contains the $r \times r$ nonsingular upper triangular matrix $\mathbf{L}'$ of the complete orthogonal decomposition, where $r \leq n$ is the rank of $\mathbf{A}$. The full $m \times n$ upper triangular matrix $\mathbf{R}$ of the orthogonal decomposition of matrix $\mathbf{A}$ can be obtained by vertical concatenation (IML operator `//`) of the $(m - n) \times n$ zero matrix to the result r.

piv is an $n \times 1$ vector of permutations of the columns of $\mathbf{A}$. That is, the QR decomposition is computed, not of $\mathbf{A}$, but of the matrix with columns $[\mathbf{A}_{piv[1]} \cdots \mathbf{A}_{piv[n]}]$. The vector *piv* corresponds to an $n \times n$ permutation matrix, $\mathbf{\Pi}$, of the pivoted QR decomposition in the first step of orthogonal decomposition.

lindep specifies the number of linearly dependent columns in the matrix $\mathbf{A}$ detected by applying the r Householder transformation in the order specified by the argument *piv*. That is, *lindep*$= n - r$.

The inputs to the COMPORT subroutine are as follows:

a specifies the $m \times n$ matrix $\mathbf{A}$, with $m \geq n$, which is to be decomposed into the product of the $m \times m$ orthogonal matrix $\mathbf{Q}$, the $n \times n$ upper triangular matrix $\mathbf{R}$, and the $n \times n$ orthogonal matrix $\mathbf{P}$,

$$\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \mathbf{\Pi}' \mathbf{P}' \mathbf{\Pi}$$

b specifies an optional $m \times p$ matrix $\mathbf{B}$ that is to be left multiplied by the transposed $m \times m$ matrix $\mathbf{Q}'$.

sing is an optional scalar specifying a singularity criterion.

The *complete* orthogonal decomposition of the singular matrix $\mathbf{A}$ can be used to compute the Moore-Penrose inverse $\mathbf{A}^-$, $r = \mathrm{rank}(\mathbf{A}) < n$, or to compute the minimum 2-norm solution of the (rank deficient) least-squares problem $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$.

1. Use the QR decomposition of $\mathbf{A}$ with column pivoting,

$$\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \mathbf{\Pi}' = \begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{\Pi}'$$

   where $\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \end{bmatrix} \in \mathcal{R}^{r \times t}$ is upper trapezoidal, $\mathbf{R}_1 \in \mathcal{R}^{r \times r}$ is upper triangular and invertible, $\mathbf{R}_2 \in \mathcal{R}^{r \times s}$, $\mathbf{Q} = \begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix}$ is orthogonal, $\mathbf{Y} \in \mathcal{R}^{t \times r}$, $\mathbf{Z} \in \mathcal{R}^{t \times s}$, and $\mathbf{\Pi}$ permutes the columns of $\mathbf{A}$.

2. Use the transpose $\mathbf{L}_{12}$ of the upper trapezoidal matrix $\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \end{bmatrix}$,

$$\mathbf{L}_{12} = \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{bmatrix} = \mathbf{R}' \in \mathcal{R}^{t \times r}$$

with $\mathrm{rank}(\mathbf{L}_{12}) = \mathrm{rank}(\mathbf{L}_1) = r$, $\mathbf{L}_1 \in \mathcal{R}^{r\times r}$ lower triangular, $\mathbf{L}_2 \in \mathcal{R}^{s\times r}$. The lower trapezoidal matrix $\mathbf{L}_{12} \in \mathcal{R}^{t\times r}$ is premultiplied with r Householder transformations $\mathbf{P}_1, \ldots, \mathbf{P}_r$:

$$\mathbf{P}_r \cdots \mathbf{P}_1 \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{L} \\ \mathbf{0} \end{bmatrix}$$

each zeroing out one of the r columns of $\mathbf{L}_2$ and producing the nonsingular lower triangular matrix $\mathbf{L} \in \mathcal{R}^{r\times r}$. Therefore, you obtain

$$\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{L}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{\Pi}'\mathbf{P}' = \mathbf{Y} \begin{bmatrix} \mathbf{L}' & \mathbf{0} \end{bmatrix} \mathbf{\Pi}'\mathbf{P}'$$

with $\mathbf{P} = \mathbf{\Pi}\mathbf{P}_r \cdots \mathbf{P}_1 \in \mathcal{R}^{t\times t}$ and upper triangular $\mathbf{L}'$. This second step is described in Golub and Van Loan (1989, p. 220 and p. 236).

3. Compute the Moore-Penrose Inverse $\mathbf{A}^-$ explicitly.

$$\mathbf{A}^- = \mathbf{P}\mathbf{\Pi} \begin{bmatrix} (\mathbf{L}')^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{Q}' = \mathbf{P}\mathbf{\Pi} \begin{bmatrix} (\mathbf{L}')^{-1} \\ \mathbf{0} \end{bmatrix} \mathbf{Y}'$$

   (a) Obtain $\mathbf{Y}$ in $\mathbf{Q} = \begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix}$ explicitly by applying the r Householder transformations obtained in the first step to $\begin{bmatrix} \mathbf{I}_r \\ \mathbf{0} \end{bmatrix}$.

   (b) Solve the $r \times r$ lower triangular system $(\mathbf{L}')^{-1}\mathbf{Y}'$ with t right hand sides using backward substitution, which yields an $r \times t$ intermediate matrix.

   (c) Left-apply the r Householder transformations in $\mathbf{P}$ on the $r \times t$ intermediate matrix $\begin{bmatrix} (\mathbf{L}')^{-1}\mathbf{Y}' \\ \mathbf{0} \end{bmatrix}$, which results in the symmetric matrix $\mathbf{A}^- \in \mathcal{R}^{t\times t}$.

The GINV function computes the Moore-Penrose inverse $\mathbf{A}^-$ using the singular value decomposition of $\mathbf{A}$. Using complete orthogonal decomposition to compute $\mathbf{A}^-$ usually needs far fewer floating point operations. However, it may be slightly more sensitive to rounding errors, which can disturb the detection of the true rank of $\mathbf{A}$, than singular value decomposition.

CONCAT Function

performs elementwise string concatenation

CONCAT(*argument1, argument2*<, . . ., *argument15*>**)**

where *arguments* are character matrices or quoted literals.

The CONCAT function produces a character matrix containing elements that are the concatenations of corresponding element strings from each argument. The CONCAT function accepts up to 15 arguments, where each argument is a character matrix or

a scalar. All nonscalar arguments must conform. Any scalar arguments are used repeatedly to concatenate to all elements of the other arguments. The element length of the result equals the sum of the element lengths of the arguments. Trailing blanks of one matrix argument appear before elements of the next matrix argument in the result matrix. For example, if you specify

```
b={"AB" "C ",
   "DE" "FG"};
```

and

```
c={"H " "IJ",
   " K" "LM"};
```

then the statement

```
a=concat(b,c);
```

produces the new 2×2 matrix

```
        A         2 rows      2 cols    (character, size 4)

                              ABH  C IJ
                              DE K FGLM
```

Quotation marks (") are needed only if you want to embed blanks or maintain uppercase and lowercase distinctions. You can also use the ADD infix operator to concatenate character operands. See the description of the addition operator.

CONTENTS Function

obtains the variables in a SAS data set

CONTENTS(<*libref*><, *SAS-data-set*>**)**

where *SAS-data-set* can be specified with a one-word name or with a libref and a SAS-data-set name. For more information on specifying SAS data sets, see Chapter 6, "Working with SAS Data Sets." Also, refer to the chapter on SAS data sets in *SAS Language Reference: Concepts.*

The CONTENTS function returns a character matrix containing the variable names for *SAS-data-set*. The result is a character matrix with n rows, one column, and 8 characters per element, where n is the number of variables in the data set. The variable list is returned in the order in which the variables occur in the data set. If a one-word name is provided, IML uses the default SAS data library (as specified in the DEFLIB= option). If no arguments are specified, the current open input data set is used. Some examples follow.

```
x=contents();              /* current open input data set */

x=contents('work','a');    /* contents of data set A in   */
                           /* WORK library                */
```

See also the description of the SHOW *contents* statement.

CONVEXIT Function

calculates and returns a scalar containing the convexity of a non-contingent cash-flow

CONVEXIT($times, flows, ytm$)

The CONVEXIT function calculates and returns a scalar containing the convexity of a non-contingent cash-flow.

times is an n-dimensional column vector of times. Elements should be non-negative.

flows is an n-dimensional column vector of cash-flows.

ytm is the per-period yield-to-maturity of the cash-flow stream. This is a scalar and should be positive.

Convexity is essentially a measure of how duration, the sensitivity of price to yield, changes as interest rates change:

$$C = \frac{1}{P}\frac{d^2P}{dy^2}$$

With cash-flows that are not yield sensitive, and the assumption of parallel shifts to a flat term-structure, convexity is given by

$$C = \frac{\sum_{k=1}^{K} t_k(t_k+1)\frac{c(k)}{(1+y)^{t_k}}}{P(1+y)^2}$$

where P is the present value, y is the effective per period yield-to-maturity, K is the number of cash-flows, the k-th cash-flow being $c(k)$ t_k periods from the present.

Example

```
proc iml;
   timesn=do(1,100,1);
   timesn=T(timesn);
   flows=repeat(10,100);
   ytm={.1};
   convexit=convexit(timesn,flows,ytm);
print convexit ;
quit;
```

```
CONVEXIT
199.26229
```

CONVMOD Function

converts modules to character matrices

CONVMOD(*module-name***)**

where *module-name* is a character matrix or quoted literal containing the name of an IML module.

The CONVMOD function returns a character matrix with n rows and 1 column, where n is the number of statements in the module. The element length is determined from the longest statement in the module. The CONVMOD function is supported to maintain compatibility with Version 5 SAS/IML software. It should be used only in conjunction with the STORE and PARSE statements. For example, consider the statements

```
start abc;
   \ob statements \obe
finish;
r=convmod('abc');   /* convert module ABC to matrix R   */
store r;            /* store module as character matrix */
```

Note that this can also be done in just one step with the statement

```
store module='abc';
```

This statement stores module ABC in the storage library. You should use the STORE MODULE= command instead to store modules. See Chapter 14, "Storage Features," for details concerning storage of modules.

COVLAG Function

computes autocovariance estimates for a vector time series

COVLAG(x, k**)**

The inputs to the COVLAG function are as follows:

x is an $n \times nv$ matrix of time series values; n is the number of observations, and nv is the dimension of the random vector.

k is a scalar, the absolute value of which specifies the number of lags desired. If k is positive, a mean correction is made. If k is negative, no mean correction is made.

The COVLAG function computes a sequence of lagged crossproduct matrices. This function is useful for computing sample autocovariance sequences for scalar or vector time series.

The value returned by the COVLAG function is an $nv \times (k * nv)$ matrix. The ith $nv \times nv$ block of the matrix is the sum

$$\frac{1}{n}\sum_{j=i}^{n} x_j' x_{j-i+1} \quad \text{if } k < 0$$

where x_j is the jth row of x. If k>0, then the ith $nv \times nv$ block of the matrix is

$$\frac{1}{n}\sum_{j=i}^{n} (x_j - \bar{x})'(x_{j-i+1} - \bar{x})$$

where $\bar{x}$ is a row vector of the column means of x. For example, the statements

```
x={-9,-7,-5,-3,-1,1,3,5,7,9};
cov=covlag(x,4);
```

produce the matrix

```
COV          1 row       4 cols    (numeric)

          33      23.1      13.6       4.9
```

CREATE Statement

creates a new SAS data set

CREATE *SAS-data-set* <**VAR** *operand*>**;**
CREATE *SAS-data-set* **FROM** *matrix-name*

<**[COLNAME=***column-name* **ROWNAME=***row-name***]**>**;**

The inputs to the CREATE statement are as follows:

SAS-data-set can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see Chapter 6, "Working with SAS Data Sets." Also, refer to the chapter on SAS data sets in *SAS Language Reference: Concepts*.

operand gives a set of existing IML variables to become data set variables.

matrix-name names a matrix containing the data.

column-name is a character matrix or quoted literal containing descriptive names to associate with data set variables.

row-name is a character matrix or quoted literal containing descriptive names to associate with observations on the data set.

The CREATE statement creates a new SAS data set and makes it both the current input and output data sets. The variables in the new SAS data set are either the variables listed in the VAR clause or variables created from the columns of the FROM matrix. The FROM clause and the VAR clause should not be specified together.

You can specify a set of variables to use with the VAR clause, where *operand* can be specified as one of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of the keywords described below:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

Following are examples showing each possible way you can use the VAR clause.

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

You can specify a COLNAME= and a ROWNAME= matrix in the FROM clause. The COLNAME= matrix gives names to variables in the SAS data set being created. The COLNAME= operand specifies the name of a character matrix. The first *ncol* values from this matrix provide the variable names in the data set being created, where *ncol* is the number of columns in the FROM matrix. The CREATE statement uses the first *ncol* elements of the COLNAME= matrix in row-major order.

The ROWNAME= operand adds a variable to the data set to contain row titles. The operand must be a character matrix that exists and has values. The length of the data set variable added is the length of a matrix element of the operand. The same ROWNAME= matrix should be used on any subsequent APPEND statements for this data set.

The variable types and lengths are the current attributes of the matrices specified in the VAR clause or the matrix in the FROM clause. The default type is numeric when the name is undefined and unvalued. The default, when no variables are specified, is all active variables. To add observations to your data set, you must use the APPEND statement.

For example, the following statements create a new SAS data set CLASS having variables NAME, SEX, AGE, HEIGHT, and WEIGHT. The data come from IML matrices with the same names. You must initialize the character variables (NAME and SEX) and set the length prior to invoking the CREATE statement. NAME and SEX are character variables of lengths 12 and 1, respectively. AGE, HEIGHT, and WEIGHT are, by default, numeric.

```
name="123456789012";
sex="M";
create class var {name sex age height weight};
append;
```

In the next example, you use the FROM clause with the COLNAME= operand to create a SAS data set named MYDATA. The new data set has variables named with the COLNAME= operand. The data are in the FROM matrix **X**, and there are two observations because **X** has two rows of data. The COLNAME= operand gives descriptive names to the data set variables.

```
x={1 2 3, 4 5 6};
varnames='x1':'x3';
   /* creates data set MYDATA with variables X1, X2, X3  */
create mydata from x [colname=varnames];
append;
```

CSHAPE Function

reshapes and repeats character values

CSHAPE(*matrix, nrow, ncol, size*<*, padchar*>**)**

The inputs to the CSHAPE function are as follows:

matrix	is a character matrix or quoted literal.
nrow	is the number of rows.
ncol	is the number of columns.
size	is the element length.
padchar	is a padding character.

The CSHAPE function shapes character matrices. See also the description of the SHAPE function, which is used with numeric data. The dimension of the matrix created by the CSHAPE function is specified by *nrow* (the number of rows), *ncol* (the number of columns), and *size* (the element length). A padding character is specified by *padchar*.

The CSHAPE function works by looking at the source matrix as if the characters of the source elements had been concatenated in row-major order. The source characters are then regrouped into elements of length *size*. These elements are assigned to the result matrix, once again in row-major order. If there are not enough characters for the result matrix, the source of the remaining characters depends on whether padding was specified with *padchar*. If no padding was specified, the source matrix's characters are cycled through again. If a padding character was specified, the remaining characters are all the padding character.

If one of the dimension arguments (*nrow*, *ncol*), or *size*) is zero, the function computes the dimension of the output matrix by dividing the number of elements of the input matrix by the product of the nonzero arguments.

Some examples follow. The statement

```
r=cshape('abcd',2,2,1);
```

results in

```
      R        2 rows      2 cols    (character, size 1)

                           a b
                           c d
```

The statement

```
r=cshape('a',1,2,3);
```

results in

```
          R        1 row       2 cols    (character, size 3)

                              aaa aaa
```

The statement

```
r=cshape({'ab' 'cd',
          'ef' 'gh',
          'ij' 'kl'}, 2, 2, 3);
```

results in

```
          R         2 rows      2 cols    (character, size 3)

                              abc def
                              ghi jkl
```

The statement

```
r=cshape('XO',3,3,1);
```

results in

```
          R          3 rows      3 cols    (character, size 1)

                               X O X
                               O X O
                               X O X
```

And finally, the statement

```
r=cshape('abcd',2,2,3,'*');
```

results in

```
          R           2 rows      2 cols    (character, size 3)

                                 abc d**
                                 *** ***
```

CUSUM Function

calculates cumulative sums

CUSUM(*matrix***)**

where *matrix* is a numeric matrix or literal.

The CUSUM function returns a matrix of the same dimension as the argument matrix. The result contains the cumulative sums obtained by scanning the argument and summing in row-major order.

For example, the statements

```
a=cusum({1 2 4 5});
b=cusum({5 6, 3 4});
```

produce the result

```
A             1 row       4 cols    (numeric)

              1         3         7        12


B             2 rows      2 cols    (numeric)

                        5        11
                       14        18
```

CVEXHULL Function

finds a convex hull of a set of planar points

CVEXHULL(*matrix***)**

where *matrix* is an $n \times 2$ matrix of (x, y) points.

The argument for the CVEXHULL function is an $n \times 2$ matrix of (x, y) points. The result matrix is an $n \times 1$ matrix of indices. The indices of points in the convex hull

in counter-clockwise order are returned as the first part of the result matrix, and the negative of the indices of the internal points are returned as the remaining elements of the result matrix. Any points that lie on the convex hull but lie on a line segment joining two other points on the convex hull are not included as part of the convex hull. The result matrix can be split into positive and negative parts using the LOC function. For example, the statements

```
z=cvexhull(x);
c=z[loc(z>0),];
```

yield the index vector for the convex hull.

DATASETS Function

obtains the names of SAS data sets in a SAS data library

DATASETS(<*libref*>**)**

where *libref* is the name of a SAS data library. For more information on specifying a SAS data library, see Chapter 6, "Working with SAS Data Sets."

The DATASETS function returns a character matrix containing the names of the SAS data sets in the specified SAS data library. The result is a character matrix with n rows and one column, where n is the number of data sets in the library. If no argument is specified, IML uses the default libname. (See the DEFLIB= option in the description of the RESET statement.)

For example, suppose you have several data sets in the SAS data library SASUSER. You can list the names of the data sets in SASUSER by using the DATASETS function as follows.

```
lib={sasuser};
a=datasets(lib);
```

```
     A          6 rows      1 col     (character, size 8)

                           CLASS
                           FITNESS
                           GROWTH
                           HOUSES
                           SASPARM
                           TOBACCO
```

DELETE Call

deletes a SAS data set

CALL DELETE(<*libname*,> *member-name***);**

The inputs to the DELETE subroutine are as follows:

libname is a character matrix or quoted literal containing the name of a SAS data library.

member-name is a character matrix or quoted literal containing the name of a data set.

The DELETE subroutine deletes a SAS data set in the specified library. If a one word name is specified, the default SAS data library is used. (See the DEFLIB= option in the description of the RESET statement.)

Some examples follow.

```
call delete(work,a);  /* deletes WORK.A  */

reset deflib=work;    /* sets default libname to WORK */
call delete(a);       /* also deletes WORK.A */

d=datasets('work');   /* returns all data sets in WORK */
call delete(work,d[1]);
                      /* deletes data set whose name is */
                      /* first element of matrix D */
```

DELETE Statement

marks observations for deletion

DELETE <*range*> <**WHERE(***expression***)**>**;**

The inputs to the DELETE statement are as follows:

range specifies a range of observations.

expression is an expression that is evaluated for being true or false.

Use the DELETE statement to mark records for deletion in the current output data set. To delete records and renumber the remaining observations, use the PURGE statement.

You can specify *range* by using a keyword or by record number using the POINT operand. The following keywords are valid values for *range*:

ALL specifies all observations.

CURRENT specifies the current observation.

NEXT <*number*> specifies the next observation or the next *number* of observations.

AFTER specifies all observations after the current one.

POINT *operand* specifies observations by number, where *operand* is one of the following:

Operand	Example
a single record number	`point 5`
a literal giving several record numbers	`point {2 5 10}`
the name of a matrix containing record numbers	`point p`
an expression in parentheses	`point (p+1)`

CURRENT is the default value for *range*. If the current data set has an index in use, the POINT option is invalid.

The WHERE clause conditionally selects observations that are contained within the *range* specification. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable is a variable in the SAS data set.

comparison-op is one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the conditon:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an AND clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables and the expression on the right-hand side refers to matrix values.

Here are several examples of DELETE statements:

```
delete;                       /* deletes the current obs  */
delete point 34;              /* deletes obs 34           */
delete all where(age<21);     /* deletes obs where age<21 */
```

You can use the SETOUT statement with the DELETE statement as follows:

```
setout class point 34;     /* makes CLASS current output */
delete;                    /* deletes ob 34              */
```

Observations deleted using the DELETE statement are not physically removed from the data set until a PURGE statement is issued.

DESIGN Function

creates a design matrix

DESIGN(*column-vector***)**

where *column-vector* is a numeric column vector or literal.

The DESIGN function creates a design matrix of 0s and 1s from *column-vector*. Each unique value of the vector generates a column of the design matrix. This column contains ones in elements with corresponding elements in the vector that are the current value; it contains zeros elsewhere. The columns are arranged in the sort order of the original values.

For example, the statements

```
a={1,1,2,2,3,1};
a=design(a);
```

produce the design matrix

```
A             6 rows      3 cols     (numeric)

                1         0          0
                1         0          0
                0         1          0
                0         1          0
                0         0          1
                1         0          0
```

DESIGNF Function

creates a full-rank design matrix

DESIGNF(*column-vector***)**

where *column-vector* is a numeric column vector or literal.

The DESIGNF function works similar to the DESIGN function; however, the result matrix is one column smaller and can be used to produce full-rank design matrices. The result of the DESIGNF function is the same as if you took the last column off the DESIGN function result and subtracted it from the other columns of the result.

For example, the statements

```
a={1,1,2,2,3,3};
b=designf(a);
```

produce the following design matrix.

```
B              6 rows      2 cols    (numeric)

                    1         0
                    1         0
                    0         1
                    0         1
                   -1        -1
                   -1        -1
```

DET Function

computes the determinant of a square matrix

DET(*square-matrix***)**

where *square-matrix* is a numeric matrix or literal.

The DET function computes the determinant of *square-matrix*, which must be square. The determinant, the product of the eigenvalues, is a single numeric value. If the determinant of a matrix is zero, then that matrix is singular; that is, it does not have an inverse.

The method performs an LU decomposition and collects the product of the diagonals (Forsythe, Malcolm, and Moler 1967). For example, the statements

```
a={1 1 1,1 2 4,1 3 9};
c=det(a);
```

produce the matrix $\mathbf{C}$ containing the determinant:

```
C              1 row       1 col     (numeric)

                           2
```

The DET function (as well as the INV and SOLVE functions) uses the following criterion to decide whether the input matrix, $\mathbf{A} = [a_{ij}]_{i,j=1,\ldots,n}$, is singular:

$$sing = 100 \times MACHEPS \times \max_{1 \le i,j \le n} |a_{ij}|$$

where *MACHEPS* is the relative machine precision.

All matrix elements less than or equal to *sing* are now considered rounding errors of the largest matrix elements, so they are taken to be zero. For example, if a diagonal or triangular coefficient matrix has a diagonal value less than or equal to *sing*, the matrix is considered singular by the DET, INV, and SOLVE functions.

Previously, a much smaller singularity criterion was used, which caused algebraic operations to be performed on values that were essentially floating point error. This occasionally yielded numerically unstable results. The new criterion is much more

conservative, and it generates far fewer erroneous results. In some cases, you may need to scale the data to avoid singular matrices. If you think the new criterion is too strong,

- try the GINV function to compute the generalized inverse
- examine the size of the singular values returned by the SVD fall. The SVD fall can be used to compute a generalized inverse with a user-specified singularity criterion.

DIAG Function

creates a diagonal matrix

DIAG(*argument***)**

where *argument* can be either a numeric square matrix or a vector.

If *argument* is a square matrix, the DIAG function creates a matrix with diagonal elements equal to the corresponding diagonal elements. All off-diagonal elements in the new matrix are zeros.

If *argument* is a vector, the DIAG function creates a matrix with diagonal elements that are the values in the vector. All off-diagonal elements are zeros.

For example, the statements

```
a={4 3,
   2 1};
c=diag(a);
```

result in

```
C             2 rows      2 cols    (numeric)

                    4         0
                    0         1
```

The statements

```
b={1 2 3};
d=diag(b);
```

result in

```
D             3 rows      3 cols    (numeric)

                  1         0         0
                  0         2         0
                  0         0         3
```

DISPLAY Statement

displays fields in display windows

DISPLAY <*group-spec group-options*<, . . ., *group-spec group-options*>>**;**

The inputs to the DISPLAY statement are as follows:

group-spec specifies a group. It can be specified as either a compound name of the form *windowname.groupname* or a window name followed by a group of the form *window-name* (*field-specs*), where *field-specs* is as defined for the WINDOW statement.

group-options can be any of the following:

NOINPUT displays the group with all fields protected so that no data can be entered in the fields.

REPEAT repeats the group for each element of the matrices specified as field operands.

BELL rings the bell, sounds the alarm, or beeps the speaker on your workstation when the window is displayed.

The DISPLAY statement directs IML to gather data into fields defined on the screen for purposes of display, data entry, or menu selection. The DISPLAY statement always refers to a window that has been previously opened by a WINDOW statement. The statement is described completely in Chapter 13, "Window and Display Features."

Following are several examples of using the DISPLAY statement:

```
display;
display w(i);
display w ("BELL") bell;
display w.g1 noinput;
display w (i protect=yes
           color="blue"
           j color="yellow");
```

DO Function

produces an arithmetic series

DO(*start, stop, increment***)**

The inputs to the DO function are as follows:

start	is the starting value for the series.
stop	is the stopping value for the series.
increment	is an increment value.

The DO function creates a row vector containing a sequence of numbers starting with *start* and incrementing by *increment* as long as the elements are less than or equal to *stop* (greater than or equal to *stop* for a negative increment). This function is a generalization of the index creation operator (:).

For example, the statement

```
i=do(3,18,3);
```

yields the result

```
        I             1 row       6 cols    (numeric)

        3         6         9        12        15        18
```

The statement

```
j=do(3,-1,-1);
```

yields the result

```
         J             1 row       5 cols    (numeric)

         3         2         1         0        -1
```

DO and END Statements

groups statements as a unit

DO;
statements

END;

The DO statement specifies that the statements following the DO statement are executed as a group until a matching END statement appears. DO statements often appear in IF-THEN/ELSE statements, where they designate groups of statements to be performed when the IF condition is true or false.

For example, consider the following statements:

```
if x=y then
   do;
      i=i+l;
      print x;
   end;
print y;
```

The statements between the DO and END statements (called the DO group) are performed only if $\mathbf{X} = \mathbf{Y}$; that is, only if all elements of $\mathbf{X}$ are equal to the corresponding elements of $\mathbf{Y}$. If any element of $\mathbf{X}$ is not equal to the corresponding element of $\mathbf{Y}$, the statements in the DO group are skipped and the next statement is executed, in this case

```
print y;
```

DO groups can be nested. Any number of nested DO groups is allowed. Here is an example of nested DO groups:

```
if y>z then
   do;
      if z=0 then
         do;
            z=b*c;
            x=2#y;
         end;
   end;
```

It is good practice to indent the statements in a DO group as shown above so that their positions indicate their levels of nesting.

DO Statement, Iterative

iteratively executes a DO group

DO *variable*=*start* **TO** *stop* <**BY** *increment*>**;**

The inputs to the DO statement are as follows:

variable	is the name of a variable indexing the loop.
start	is the starting value for the looping variable.
stop	is the stopping value for the looping variable.
increment	is an increment value.

When the DO group has this form, the statements between the DO and END statements are executed repetitively. The number of times the statements are executed depends on the evaluation of the expressions given in the DO statement.

The *start*, *stop*, and *increment* values should be scalars or expressions with evaluations that yield scalars. The *variable* is given a new value for each repetition of the group. The index variable starts with the *start* value, then is incremented by the *increment* value each time. The iterations continue as long as the index variable is less than or equal to the *stop* value. If a negative increment is used, then the rules reverse so that the index variable decrements to a lower bound. Note that the *start*, *stop*, and *increment* expressions are evaluated only once before the looping starts.

For example, the statements

```
do i=1 to 5 by 2;
   print 'THE VALUE OF I IS:' i;
end;
```

produce the output

```
                                    I
                THE VALUE OF I IS:  1


                                    I
                THE VALUE OF I IS:  3


                                    I
                THE VALUE OF I IS:  5
```

DO DATA Statement

repeats a loop until an end of file occurs

DO DATA <*variable*=*start* **TO** *stop*>**;**

The inputs to the DO DATA statement are as follows:

variable	is the name of a variable indexing the loop.
start	is the starting value for the looping variable.
stop	is the stopping value for the looping variable.

The DO DATA statement is used for repetitive DO loops that need to be exited upon the occurrence of an end of file for an INPUT, READ, or other I/O statement. This form is common for loops that read data from either a sequential file or a SAS data set. When an end of file is reached inside the DO DATA group, IML immediately jumps from the group and starts executing the statement following the END statement. DO DATA groups can be nested, where each end of file causes a jump from the most local DO DATA group. The DO DATA loop simulates the end-of-file behavior of the SAS DATA step. You should avoid using GOTO and LINK statements to jump out of a DO DATA group.

Examples of valid statements follow. The first example inputs the variable NAME from an external file for the first 100 lines or until the end of file, whichever occurs first.

```
do data i=1 to 100;
   input name $8.;
end;
```

Or, if reading from a SAS data set, then the code can be

```
do data;             /* read next obs until eof is reached */
   read next var{x}; /* read only variable X               */
end;
```

DO Statement with an UNTIL Clause

conditionally executes statements iteratively

DO UNTIL(*expression***);**
DO *variable*=*start* **TO** *stop* <**BY** *increment*> **UNTIL(***expression***);**

The inputs to the DO UNTIL statement are as follows:

expression	is an expression that is evaluated at the bottom of the loop for being true or false.
variable	is the name of a variable indexing the loop.
start	is the starting value for the looping variable.
stop	is the stopping value for the looping variable.
increment	is an increment value.

Using an UNTIL expression makes possible the conditional execution of a set of statements iteratively. The UNTIL expression is evaluated at the bottom of the loop, and the statements inside the loop are executed repeatedly as long as the expression yields a zero or missing value. In the example that follows, the body of the loop executes until the value of X exceeds 100:

```
x=1;
do until (x>100);
   x+1;
end;
print x;                  /* x=101 */
```

DO Statement with a WHILE Clause

conditionally executes statements iteratively

DO WHILE(*expression***);**
DO *variable*=*start* **TO** *stop* <**BY** *increment*> **WHILE(***expression***);**

The inputs to the DO WHILE statement are as follows:

expression	is an expression that is evaluated at the top of the loop for being true or false.
variable	is the name of a variable indexing the loop.
start	is the starting value for the looping variable.
stop	is the stopping value for the looping variable.
increment	is an increment value.

Using a WHILE expression makes possible the conditional execution of a set of statements iteratively. The WHILE expression is evaluated at the top of the loop, and the statements inside the loop are executed repeatedly as long as the expression yields a nonzero or nonmissing value.

Note that the incrementing is done before the WHILE expression is tested. The following example demonstrates the incremeting:

```
x=1;
do while(x<100);
   x+1;
end;
print x;          /* x=100                              */
```

The next example increments the starting value by 2:

```
y=1;
do x=1 to 100 by 2 while(y<200);
   y=y#x;
end;              /* at end of loop, x=11 and y=945  */
```

DURATION Function

calculates and returns a scalar containing the modified duration of a non-contingent cash-flow.

DURATION($times, flows, ytm$)

The Duration function returns the modified duration of a non-contingent cash-flow as a scalar.

times is an n-dimensional column vector of times. Elements should be non-negative.

flows is an n-dimensional column vector of cash-flows.

ytm is the per-period yield-to-maturity of the cash-flow stream. This is a scalar and should be positive.

Duration of a security is generally defined as

$$D = -\frac{\frac{dP}{P}}{dy}$$

In other words, it is the relative change in price for a unit change in yield. Since prices move in the opposite direction to yields, the sign change preserves positivity for convenience. With cash-flows that are not yield-sensitive and the assumption of parallel shifts to a flat term-structure, duration is given by:

$$D_{\text{mod}} = \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{(1+y)^{t_k}}}{P(1+y)}$$

where P is the present value, y is the per period effective yield-to-maturity, K is the number of cash-flows, the k-th cash flow being $c(k)$, t_k periods from the present. This measure is referred to as *modified duration* to differentiate it from the first duration measure ever proposed, *Macaulay duration*:

$$D_{\text{Mac}} = \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{(1+y)^{t_k}}}{P}$$

This expression also reveals the reason for the name duration, since it is a present-value-weighted average of the duration (i.e. timing) of all the cash-flows and is hence an "average time-to-maturity" of the bond.

Example

```
proc iml;
   times={1};
   ytm={.1};
   flow={10};
   duration=duration(times,flow,ytm);
print duration;
quit;
```

DURATION
0.9090909

ECHELON Function

reduces a matrix to row-echelon normal form

ECHELON(*matrix***)**

where *matrix* is a numeric matrix or literal.

The ECHELON function uses elementary row operations to reduce a matrix to row-echelon normal form as in the following example (Graybill 1969, p. 286):

```
a={3  6  9,
   1  2  5,
   2  4 10};
e=echelon(a);
```

The resulting matrix is

E	3 rows	3 cols	(numeric)
	1	2	0
	0	0	1
	0	0	0

If the argument is a square matrix, then the row-echelon normal form can be obtained from the Hermite normal form by rearranging rows that are all zeros.

EDIT Statement

opens a SAS data set for editing

EDIT *SAS-data-set* <**VAR** *operand*> <**WHERE(***expression***)**>
<**NOBS** *name*>**;**

The inputs to the EDIT statement are as follows:

SAS-data-set can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, refer to the chapter on SAS data sets in *SAS Language Reference: Concepts*.

operand selects a set of variables.

expression selects observations conditionally.

name names a variable to contain the number of observations.

The EDIT statement opens a SAS data set for reading and updating. If the data set has already been opened, the EDIT statement makes it the current input and output data sets.

You can specify a set of variables to use with the VAR clause, where *operand* can be specified as one of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of the keywords described below:

  ALL for all variables
  CHAR for all character variables
  NUM for all numeric variables

Following are examples showing each possible way you can use the VAR clause.

```
var {time1 time5 time9}; /* a literal giving the variables  */
var time;                /* a matrix containing the names   */
var('time1':'time9');    /* an expression                   */
var _all_;               /* a keyword                       */
```

The WHERE clause conditionally selects observations, within the range specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable is a variable in the SAS data set.

comparison-op is any one of the following comparison operators:

< less than
<= less than or equal to
= equal to
> greater than
>= greater than or equal to
^= not equal to
? contains a given string

^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause*	(for an AND clause)
clause|*clause*	(for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables and the expression on the right-hand side refers to matrix values.

The EDIT statement can define a set of variables and the selection criteria that are used to control access to data set observations. The NOBS clause returns the total number of observations in the data set in the variable *name*.

The VAR and WHERE clauses are optional and can be specified in any order. The NOBS clause is also optional.

See Chapter 6, "Working with SAS Data Sets," for more information on editing SAS data sets.

To edit the data set DAT, or WORK.DAT, use the statements

```
edit dat;
edit work.dat;
```

To control the variables you want to edit and conditionally select observations for editing, use the VAR and WHERE clauses. For example, to read and update observations for variable I where I is greater than 9, use the statement

```
edit work.dat var{i} where (i>9);
```

Following is an example using the NOBS option.

```
/* if MYDATA has 10 observations,                    */
/* then ct is a numeric matrix with value 10        */
edit mydata nobs ct;
```

EIGEN Call

computes eigenvalues and eigenvectors

CALL EIGEN(*eigenvalues, eigenvectors,* $\mathbf{A}$**)** $<$*VECL="vl"*$>$**;**

where $\mathbf{A}$ is an arbitrary square numeric matrix for which eigenvalues and eigenvectors are to be calculated.

The EIGEN call returns the following values:

eigenvalues	a matrix to contain the eigenvalues of the input matrix.
eigenvectors	names a matrix to contain the right eigenvectors of the input matrix.
vl	is an optional $n \times n$ matrix containing the left eigenvectors of $\mathbf{A}$ in the same manner that *eigenvectors* contains the right eigenvectors.

The EIGEN subroutine computes *eigenvalues*, a matrix containing the eigenvalues of $\mathbf{A}$ arranged in descending order. If $\mathbf{A}$ is symmetric, *eigenvalues* in the $n \times 1$ vector containing the n real eigenvalues of $\mathbf{A}$. If $\mathbf{A}$ is not symmetric (as determined by the criterion described below) *eigenvalues* is an $n \times 2$ matrix containing the eigenvalues of the $n \times n$ matrix $\mathbf{A}$. The first column of $\mathbf{A}$ contains the real parts, Re(λ), and the second column contains the imaginary parts Im(λ). Each row represents one eigenvalue, Re(λ) + iIm(λ). Complex conjugate eigenvalues, Re(λ) $\pm$ iIm(λ), are stored in standard order; that is, the eigenvalue of the pair with a positive imaginary part is followed by the eigenvalue of the pair with the negative imaginary part.

The EIGEN subroutine also computes *eigenvectors*, a matrix. If $\mathbf{A}$ is symmetric, then *eigenvectors* has orthonormal column eigenvectors of $\mathbf{A}$ arranged so that the matrices correspond; that is, the first column of *eigenvectors* is the eigenvector corresponding to the largest eigenvalue, and so forth. If $\mathbf{A}$ is not symmetric, then *eigenvectors* is an $n \times n$ matrix containing the right eigenvectors of $\mathbf{A}$. If the eigenvalue in row i of *eigenvalues* is real, then column i of *eigenvectors* contains the corresponding real eigenvector. If rows i and $i+1$ of *eigenvalues* contain complex conjugate eigenvalues Re(λ) $\pm$ iIm(λ), then columns i and $i+1$ of *eigenvectors* contain the real, $\mathbf{v}$, and imaginary, $\mathbf{u}$, parts, respectively, of the two corresponding eigenvectors $\mathbf{v} \pm i\mathbf{u}$.

The eigenvalues of a matrix $\mathbf{A}$ are the roots of the characteristic polynomial, which is defined as $p(z) = \det(z\mathbf{I} - \mathbf{A})$. The spectrum, denoted by $\lambda(A)$, is the set of eigenvalues of the matrix A. If $\lambda(\mathbf{A}) = \{\lambda_1, \ldots, \lambda_n\}$, then $\det(\mathbf{A}) = \lambda_1\lambda_2 \cdots \lambda_n$.

The trace of $\mathbf{A}$ is defined by

$$\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

and $\mathrm{tr}(\mathbf{A}) = \lambda_1 + \cdots + \lambda_n$.

An eigenvector is a nonzero vector, $\mathbf{x}$, that satisfies $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ for $\lambda \in \lambda(\mathbf{A})$. Right eigenvectors satisfy $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, and left eigenvectors satisfy $\mathbf{x}'\mathbf{A} = \lambda\mathbf{x}'$.

The following are properties of the unsymmetric *real* eigenvalue problem, in which the real matrix $\mathbf{A}$ is square but not necessarily symmetric:

- The eigenvalues of an unsymmetric matrix $\mathbf{A}$ can be complex. If $\mathbf{A}$ has a complex eigenvalue $\mathrm{Re}(\lambda)+i\mathrm{Im}(\lambda)$, then the conjugate complex value $\mathrm{Re}(\lambda)-i\mathrm{Im}(\lambda)$ is also an eigenvalue of $\mathbf{A}$.
- The right and left eigenvectors corresponding to a real eigenvalue of $\mathbf{A}$ are real. The right and left eigenvectors corresponding to conjugate complex eigenvalues of $\mathbf{A}$ are also conjugate complex.
- The left eigenvectors of $\mathbf{A}$ are the same as the complex conjugate right eigenvectors of $\mathbf{A}'$.

The three routines, EIGEN, EIGVAL, and EIGVEC, use the following test of symmetry for a square argument matrix $\mathbf{A}$:

1. Select the entry of $\mathbf{A}$ with the largest magnitude:

$$a_{max} = \max_{i,j=1,\ldots,n} |a_{i,j}|$$

2. Multiply the value of a_{max} with the square root of the machine precision ϵ. (The value of ϵ is the largest value stored in double precision that, when added to 1 in double precision, still results in 1.)
3. The matrix $\mathbf{A}$ is considered *unsymmetric* if there exists at least one pair of symmetric entries that differs in more than $a_{max}\sqrt{\epsilon}$,

$$|a_{i,j} - a_{j,i}| > a_{max}\sqrt{\epsilon}$$

If $\mathbf{A}$ is *symmetric*, the result of the statement

```
call eigen(m,e,a);
```

has the properties

$$\begin{aligned} \mathbf{A} * \mathbf{E} &= \mathbf{E} * \mathrm{diag}(\mathbf{M}) \\ \mathbf{E}' * \mathbf{E} &= \mathbf{I}(\mathbf{N}) \end{aligned}$$

that is,

$$\mathbf{E}' = \mathrm{inv}(\mathbf{E})$$

so that

$$\mathbf{A} = \mathbf{E} * \mathrm{diag}(\mathbf{M}) * \mathbf{E}' \ .$$

The QL method is used to compute the eigenvalues (Wilkinson and Reinsch 1971).

In statistical applications, nonsymmetric matrices for which eigenvalues are desired are usually of the form $\mathbf{E}^{-1}\mathbf{H}$, where $\mathbf{E}$ and $\mathbf{H}$ are symmetric. The eigenvalues $\mathbf{L}$ and eigenvectors $\mathbf{V}$ of $\mathbf{E}^{-1}\mathbf{H}$ can be obtained by using the GENEIG subroutine or as follows:

```
f=root(einv);
a=f*h*f';
call eigen(l,w,a);
v=f'*w;
```

The computation can be checked by forming the residuals:

```
r=einv*h*v-v*diag(l);
```

The values in $\mathbf{R}$ should be of the order of round-off error.

EIGVAL Function

computes eigenvalues

EIGVAL(A)

where $\mathbf{A}$ is a square numeric matrix.

The EIGVAL function returns a column vector of the eigenvalues of $\mathbf{A}$. See the description of the EIGEN subroutine for more details.

The following code computes Example 7.1.1 from Golub and Van Loan (1989):

```
proc iml;

   a = {  67.00  177.60  -63.20 ,
         -20.40   95.88  -87.16 ,
          22.80   67.84   12.12 };

   val = EIGVAL(a);
   print val;
```

The matrix produced containing the eigenvalues is

```
          VAL

          75        100
          75       -100
          25          0
```

Notice that since a is not symmetric the eigenvalues are complex. The first column of the *VAL* matrix is the real part and the second column is the complex part of the three eigenvalues.

A symmetric example follows:

```
x={1 1,1 2,1 3,1 4};
xpx=t(x)*x;
a=eigval(xpx);         /* xpx is a symmetric matrix */
```

The matrix produced containing the eigenvalues is

```
        A             2 rows      1 col     (numeric)

                     33.401219
                      0.5987805
```

EIGVEC Function

computes right eigenvectors

EIGVEC(A)

where **A** is a square numeric matrix.

The EIGVEC function creates a matrix containing the right eigenvectors of **A**. See the description of the EIGEN subroutine for more details.

You can obtain the left eigenvectors by first transposing **A**.

An example calculating the eigenvectors of a symmetric matrix follows:

```
x={1 1,1 2,1 3,1 4};
xpx=t(x)*x;
a=eigvec(xpx);      /* xpx is a symmetric matrix   */
```

The matrix produced containing the eigenvectors is

```
                A             2 rows      2 cols    (numeric)

                             0.3220062 0.9467376
                             0.9467376 -0.322006
```

END Statement

ends a DO loop or DO statement

END:

See the description of the DO and END statements.

EXECUTE Call

executes SAS statements immediately

CALL EXECUTE(*operands***);**

where *operands* are character matrices or quoted literals containing valid SAS statements.

The EXECUTE subroutine pushes character arguments to the input command stream, executes them, and then returns to the calling module. You can specify up to 15 arguments. The subroutine should be called from a module rather than from the immediate environment (because it uses the *resume* mechanism that works only from modules). The strings you push do not appear on the log.

Following are examples of valid EXECUTE subroutines:

```
call execute("x={1 2 3, 4 5 6};");
call execute(" x 'ls';");
call execute(" dm 'log; color source red';");
call execute(concat(" title '",string,"';"));
```

For more details on the EXECUTE subroutine, see Chapter 15, "Using SAS/IML Software to Generate IML Statements."

EXP Function

calculates the exponential

EXP(*matrix***)**

where *matrix* is a numeric matrix or literal.

The EXP function is a scalar function that takes the exponential function of every element of the argument matrix. The exponential is the natural number e raised to the indicated power. An example of a valid statement follows:

```
b={2 3 4};
a=exp(b);
```

```
             A            1 row       3 cols    (numeric)

                    7.3890561 20.085537  54.59815
```

FFT Function

performs the finite Fourier transform

FFT(x)

where x is a $1 \times n$ or $n \times 1$ numeric vector.

The FFT function returns the cosine and sine coefficients for the expansion of a vector into a sum of cosine and sine functions.

The argument of the FFT function, x, is a $1 \times n$ or $n \times 1$ vector. The value returned is the resulting transform, an $np \times 2$ matrix, where

$$np = \text{floor}\left(\frac{n}{2} + 1\right)$$

The elements of the first column of the returned matrix are the cosine coefficients; that is, the ith element of the first column is

$$\sum_{j=1}^{n} x_j \cos\left(\frac{2\pi}{n}(i-1)(j-1)\right)$$

for $i = 1, \ldots, np$, where the elements of x are denoted as x_j. The elements of the second column of the returned matrix are the sine coefficients; that is, the ith element of the second column is

$$\sum_{j=1}^{n} x_j \sin\left(\frac{2\pi}{n}(i-1)(j-1)\right)$$

for $i = 1, \ldots, np$.

Note: For most efficient use of the FFT function, n should be a power of 2. If n is a power of 2, a fast Fourier transform is used (Singleton 1969); otherwise, a Chirp-Z algorithm is used (Monro and Branch 1976).

The FFT function can be used to compute the periodogram of a time series. In conjunction with the inverse finite Fourier transform routine IFFT, the FFT function can be used to efficiently compute convolutions of large vectors (Gentleman and Sande 1966; Nussbaumer 1982). An example of a valid statement follows:

```
a=fft(c);
```

FILE Statement

opens or points to an external file

FILE *file-name* <**RECFM=N**> <**LRECL=***operand*>**;**

The inputs to the FILE statement are as follows:

file-name	is a name (for defined filenames), a quoted literal, or an expression in parentheses (for filepaths).
RECFM=N	specifies that the file is to be written as a pure binary file without record-separator characters.
LRECL=*operand*	specifies the record length of the output file. The default record length is 512.

You can use the FILE statement to open a file for output, or if the file is already open, to make it the current output file so that subsequent PUT statements write to it. The FILE statement is similar in syntax and operation to the INFILE statement. The FILE statement is described in detail in Chapter 7, "File Access."

The *file-name* is either a predefined filename or a quoted string or character expression in parentheses referring to the filepath. There are two ways to refer to an input or output file: by a filepath and by a filename. The filepath is the name as known to the operating system. The filename is a SAS reference to the file established directly through a connection made with the FILENAME statement. You can specify a file in either way in the FILE and INFILE statements. To specify a filename as the operand, just give the name. The name must be one already connected to a filepath by a previously issued FILENAME statement. There are, however, two special filenames that are recognized by IML: LOG and PRINT. These refer to the standard output streams for all SAS sessions. To specify a filepath, put it in quotes or specify an expression yielding the filepath in parentheses.

When the filepath is specified, there is a limit of 64 characters to the operand.

Following are several valid uses of FILE statement.

```
file "student.dat";           /* by literal filepath   */

filename out "student.dat";   /* specify filename OUT  */
file out;                     /* refer to by filename  */

file print;                   /* standard print output */
file log;                     /* output to log         */

file "student.dat" recfm=n;   /* for a binary file     */
```

FIND Statement

finds observations

FIND <*range*> <**WHERE(***expression***)**> **INTO** *matrix-name***;**

The inputs to the FIND statement are as follows:

range	specifies a range of observations.
expression	is an expression that is evaluated for being true or false.
matrix-name	names a matrix to contain the observation numbers.

The FIND statement finds the observation numbers of records in *range* that satisfy the conditions of the WHERE clause. The FIND statement places these observation numbers in the numeric matrix whose name follows the INTO keyword.

You can specify a *range* of observations with a keyword or by record number using the POINT option. You can use any of the following keywords to specify *range:*

ALL	all observations
CURRENT	the current observation
NEXT <*number*>	the next observation or the next *number* of observations
AFTER	all observations after the current one
POINT *operand*	observations specified by number, where *operand* is one of the following.

Operand	Example
a single record number	`point 5`
a literal giving several record numbers	`point {2 5 10}`
the name of a matrix containing record numbers	`point p`
an expression in parentheses	`point (p+1)`

If the current data set has an index in use, the POINT option is invalid.

The WHERE clause conditionally selects observations, within the range specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the preceding statement,

variable is a variable in the SAS data set.

comparison-op is one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the conditon:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

Following are some valid examples of the FIND statement:

```
find all  where(name=:"Smith") into p;
find next where(age>30) into p2;
```

P and **P2** are column vectors containing the observation numbers that satisfy the WHERE clause in the given range. The default range is all observations.

FINISH Statement

denotes the end of a module

FINISH <*module-name*>**;**

where *module-name* is the name of a user-defined module.

The FINISH statement signals the end of a module and the end of module definition mode. Optionally, the FINISH statement can take the module name as its argument. See the description of the START statement and consult Chapter 5, "Programming Statements," for further information on defining modules. Some examples follow.

```
finish;
finish mod1;
```

FORCE Statement

see the description of the SAVE statement

FORWARD Function

calculates a column vector of forward rates given vectors of spot rates and times

FORWARD($times$, $spot_$rates)

The FORWARD function returns an n x 1 vector of forward rates.

times is an n x 1 column vector of times in consistent units. Elements should be non-negative.

spot_*rates* is an n x 1 column vector of corresponding per-period spot rates. Elements should be positive.

The FORWARD function transforms the given spot rates as.

$$f_1 = s_1$$

$$f_i = [\frac{(1+s_i)_i^t}{(1+s_{i-1})_{i-1}^t}]^{\frac{t_i}{t_{i-1}}} - 1.0; \quad i = 2, ...n$$

Example

```
proc iml;
   spt={.75};
   times={1};
   forward=forward(times,spt);
print forward;
quit;
```

FORWARD
0.75

FREE Statement

frees matrix storage space

FREE *matrices*;
FREE / <*matrices*>;

where *matrices* are names of matrices.

The FREE statement causes the specified matrices to lose their values; the memory is then freed for other uses. After execution of the FREE statement, the matrix does not have a value, and it returns 0 for the NROW and NCOL functions. Any printing attributes (assigned by the MATTRIB statement) are not released.

The FREE statement is used mostly in large applications or under tight memory constraints to make room for more data (matrices) in the workspace.

For example, to free the matrices **a**, **b**, and **c**, use the statement

```
free a b c;
```

If you want to free all matrices, specify a slash (/) after the keyword FREE. If you want to free all matrices except a few, then list the ones you do not want to free after the slash. For example, to free all matrices except **d** and **e**, use the statement

```
free / d e;
```

For more information, see the discussion of workspace storage in Chapter 16, "Further Notes."

GBLKVP Call

defines a blanking viewport

CALL GBLKVP(*viewport* <, *inside*>**);**

The inputs to the GBLKVP subroutine are as follows:

viewport is a numeric matrix or literal defining a viewport. This rectangular area's boundary is specified in normalized coordinates, where you specify the coordinates of the lower left corner and the upper right corner of the rectangular area in the form

{*minimum-x minimum-y maximum-x maximum-y*}

inside is a numeric argument that specifies whether graphics output is to be clipped inside or outside the blanking area. The default is to clip outside the blanking area.

The GBLKVP subroutine defines an area, called the blanking area, in which nothing is drawn until the area is released. This routine is useful for clipping areas outside the graph or for blanking out inner portions of the graph. If *inside* is set to 0 (the default), no graphics output appears outside the blanking area. Setting *inside* to 1 clips inside the blanking areas.

Note that the blanking area (as specified by the viewport argument) is defined on the current viewport, and it is released when the viewport is changed or popped. At most one blanking area is in effect at any time. The blanking area can also be released by the GBLKVPD subroutine or another GBLKVP call. The coordinates in use for this graphics command are given in normalized coordinates because it is defined relative to the current viewport.

For example, to blank out a rectangular area with corners at the coordinates (20,20) and (80,80), relative to the currently defined viewport, use the statement

```
call gblkvp({20 20, 80 80});
```

No graphics or text can be written outside this area until the blanking viewport is ended.

Alternatively, if you want to clip inside of the rectangular area as above, use the *inside* parameter:

```
call gblkvp({20 20, 80 80},1);
```

See also the description of the CLIP option in the RESET statement.

GBLKVPD Call

deletes the blanking viewport

CALL GBLKVPD;

The GBLKVPD subroutine releases the current blanking area. It allows graphics output to be drawn in the area previously blanked out by a call to the GBLKVP subroutine.

To release an area previously blanked out, as in the example for the GBLKVP subroutine, use the following statement.

```
/* define blanking viewport                          */
call gblkvp({20 20,80 80});
```
more graphics statements
```
/*  now release the blanked out area                 */
call gblkvpd;
/*  graphics or text can now be written to the area */
```
continue graphics statements

See also the description of the CLIP option in the RESET statement.

GCLOSE Call

closes the graphics segment

CALL GCLOSE;

The GCLOSE subroutine closes the current graphics segment. Once a segment is closed, no other primitives can be added to it. The next call to a graph-generating function begins building a new graphics segment. However, the GCLOSE subroutine does not have to be called explicitly to terminate a segment; the GOPEN subroutine causes GCLOSE to be called.

GDELETE Call

deletes a graphics segment

CALL GDELETE(*segment-name***);**

where *segment-name* is a character matrix or quoted literal containing the name of the segment.

The GDELETE subroutine searches the current catalog and deletes the first segment found with the name *segment-name*.

An example of a valid statement follows.

```
   /* SEG_A is defined as a character matrix        */
   /* that contains the name of the segment to delete */
call gdelete(seg_a);
```

The segment can also be specified as a quoted literal:

```
call delete("plot_13");
```

GDRAW Call

draws a polyline

CALL GDRAW(x, y <, *style*><, *color*><, *window*><, *viewport*>**);**

The inputs to the GDRAW subroutine are as follows:

x	is a vector containing the x coordinates of points used to draw a sequence of lines.
y	is a vector containing the y coordinates of points used to draw a sequence of lines.
style	is a numeric matrix or literal that specifies an index corresponding to a valid line style.
color	is a valid SAS color, where *color* can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}
viewport	is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

The GDRAW subroutine draws a sequence of connected lines from points represented by values in x and y, which must be vectors of the same length. If x and y have n points, there will be $n - 1$ lines. The first line will be from the point $(x(1), y(1))$ to $(x(2), y(2))$. The lines are drawn in the same color and line style. The coordinates in use for this graphics command are world coordinates. An example using the GDRAW subroutine follows:

```
   /* line from (50,50) to (75,75) - x and y take */
   /* default window range of 0 to 100            */
call gdraw({50 75},{50 75});
call gshow;
```

GDRAWL Call

draws individual lines

CALL GDRAWL($xy1$, $xy2$ <, *style*><, *color*><, *window*><, *viewport*>**);**

The inputs to the GDRAWL subroutine are as follows:

$xy1$	is a matrix of points used to draw a sequence of lines.
$xy2$	is a matrix of points used to draw a sequence of lines.
style	is a numeric matrix or literal that specifies an index corresponding to a valid line style.
color	is a valid SAS color, where *color* can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}
viewport	is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

The GDRAWL subroutine draws a sequence of lines specified by their beginning and ending points. The matrices $xy1$ and $xy2$ must have the same number of rows and columns. The first two columns (other columns are ignored) of $xy1$ give the x, y coordinates of the beginning points of the line segment, and the first two columns of $xy2$ have x, y coordinates of the corresponding end points. If $xy1$ and $xy2$ have n rows, n lines are drawn. The first line is from $(xy1(1,1), xy1(1,2))$ to $(xy2(1,1), xy2(1,2))$. The lines are drawn in the same color and line style. The coordinates in use for this graphics command are world coordinates. An example using the GDRAWL call follows:

```
   /* line from (25,25) to (50,50) - x and y take */
   /* default window range of 0 to 100    */
call gdrawl({25 25},{50 50});
call gshow;
```

GENEIG Call

computes eigenvalues and eigenvectors of a generalized eigenproblem

CALL GENEIG(*eigenvalues, eigenvectors, symmetric-matrix1, symmetric-matrix2***);**

The inputs to the GENEIG subroutine are as follows:

eigenvalues	is a returned vector containing the eigenvalues.
eigenvectors	is a returned matrix containing the corresponding eigenvectors.
symmetric-matrix1	is a symmetric numeric matrix.
symmetric-matrix2	is a positive definite symmetric matrix.

The GENEIG subroutine computes eigenvalues and eigenvectors of the generalized eigenproblem. The statement

```
call geneig (m,e,a,b);
```

computes eigenvalues $\mathbf{M}$ and eigenvectors $\mathbf{E}$ of the generalized eigenproblem $\mathbf{A} * \mathbf{E} = \mathbf{B} * \mathbf{E} * \mathrm{diag}(\mathbf{M})$, where $\mathbf{A}$ and $\mathbf{B}$ are symmetric and $\mathbf{B}$ is positive definite. The vector $\mathbf{M}$ contains the eigenvalues arranged in descending order, and the matrix $\mathbf{E}$ contains the corresponding eigenvectors in the columns.

The following example is from Wilkinson and Reinsch (1971, p. 311).

```
a={10   2   3   1   1,
    2  12   1   2   1,
    3   1  11   1  -1,
    1   2   1   9   1,
    1   1  -1   1  15};

b={12   1  -1   2    1,
    1  14   1  -1    1,
   -1   1  16  -1    1,
    2  -1  -1  12   -1,
    1   1   1  -1   11};

call geneig(m,e,a,b);
```

The matrices produced are as follows.

```
M
1.49235
1.10928
0.94385
0.66366
0.43278

E
-0.07638   0.14201   0.19171  -0.08292  -0.13459
 0.01709   0.14242  -0.15899  -0.15314   0.06129
-0.06666   0.12099   0.07483   0.11860   0.15790
 0.08604   0.12553  -0.13746   0.18281  -0.10946
 0.28943   0.00769   0.08897  -0.00356   0.04147
```

GGRID Call

draws a grid

CALL GGRID(x, y <, *style*><, *color*><, *window*><, *viewport*>**);**

The inputs to the GGRID subroutine are as follows:

x and y are vectors of points used to draw sequences of lines.

style is a numeric matrix or literal that specifies an index corresponding to a valid line style.

color is a valid SAS color, where *color* can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.

window is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

viewport is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GGRID subroutine draws a sequence of vertical and horizontal lines specified by the x and y vectors, respectively. The start and end of the vertical lines are implicitly defined by the minimum and maximum of the y vector. Likewise, the start and end of the horizontal lines are defined by the minimum and maximum of the x vector. The grid lines are drawn in the same color and line style. The coordinates in use for this graphics command are world coordinates.

For example, use the following statements to place a grid in the lower left corner of the screen.

```
x={10,20,30,40,50};
y=x;

   /* The following GGRID command will place a GRID    */
   /* in lower left   corner of the screen             */
   /* assuming the default window and viewport         */
call ggrid(x,y);
call gshow;
```

GINCLUDE Call

includes a graphics segment

CALL GINCLUDE(*segment-name***);**

where *segment-name* is a character matrix or quoted literal specifying a graphics segment.

The GINCLUDE subroutine includes into the current graph a previously defined graph named *segment-name* from the same catalog. The included segment is defined in the current viewport but not the current window.

The implementation of the GINCLUDE subroutine makes it possible to include other segments to the current segment and reposition them in different viewports. Furthermore, a segment can be included by different graphs, thus effectively reducing storage space. Examples of valid statements follow:

```
   /* segment1 is a character variable        */
   /*containing the segment name              */
segment1={myplot};
call ginclude(segment1);

   /* specify the segment with quoted literal */
call ginclude("myseg");
```

GINV Function

computes the generalized inverse

GINV(*matrix***)**

where *matrix* is a numeric matrix or literal.

The GINV function creates the Moore-Penrose generalized inverse of *matrix*. This inverse, known as the four-condition inverse, has these properties:

If $\mathbf{G} = \text{GINV}(\mathbf{A})$ then

$$\mathbf{AGA} = \mathbf{A} \quad \mathbf{GAG} = \mathbf{G} \quad (\mathbf{AG})' = \mathbf{AG} \quad (\mathbf{GA})' = \mathbf{GA}$$

The generalized inverse is also known as the *pseudoinverse*, usually denoted by $\mathbf{A}^{-}$. It is computed using the singular value decomposition (Wilkinson and Reinsch 1971).

Least-squares regression for the model

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon$$

can be performed by using

```
b=ginv(x)*y;
```

as the estimate of β. This solution has minimum $\mathbf{b}'\mathbf{b}$ among all solutions minimizing $\epsilon'\epsilon$, where $\epsilon = \mathbf{Y} - \mathbf{X}\mathbf{b}$.

Projection matrices can be formed by specifying GINV$(\mathbf{X}) * \mathbf{X}$ (*row space*) or $\mathbf{X}*$GINV$(\mathbf{X})$ (*column space*).

See Rao and Mitra (1971) for a discussion of properties of this function.

GOPEN Call

opens a graphics segment

CALL GOPEN(<*segment-name*><, *replace*><, *description*>**);**

The inputs to the GOPEN subroutine are as follows:

segment-name	is a character matrix or quoted literal specifying the name of a graphics segment.
replace	is a numeric argument.
description	is a character matrix or quoted text string with a maximum length of 40 characters.

The GOPEN subroutine starts a new graphics segment. The window and viewport are reset to the default values ($\{0\ 0\ 100\ 100\}$) in both cases. Any attribute modified using a GSET call is reset to its default value, which is set by the attribute's corresponding GOPTIONS value.

A nonzero value for *replace* indicates that the new segment should replace the first found segment with the same name, and zero indicates otherwise. If you do not specify the *replace* flag, the flag set by a previous GSTART call is used. By default, the GSTART subroutine sets the flag to NOREPLACE.

The *description* is a text string of up to 40 characters that you want to store with the segment to describe the graph.

Two graphs cannot have the same name. If you try to create a segment, say PLOT_A, twice, the second segment is named using a name generated by IML.

To open a new segment named COSINE, set *replace* to replace a like-named segment, and attach a description to the segment, use the statement

```
call gopen('cosine',1,'Graph of Cosine Curve');
```

GOTO Statement

jumps to a new statement

GOTO *label*;

where *label* is a labeled statement. Execution jumps to this statement. A label is a name followed by a colon (:).

The GOTO (or GO TO) statement directs IML to jump immediately to the statement with the given *label* and begin executing statements from that point. Any IML statement can have a label, which is a name followed by a colon preceding any executable statement.

GOTO statements are usually clauses of IF statements, for example,

```
if x>y then goto skip;
y=log(y-x);
yy=y-20;
skip: if y<0 then
   do;
      more statements
   end;
```

The function of GOTO statements is usually better performed by DO groups. For example, the statements above could be better written

```
if x<=y then
   do;
      y=log(y-x);
      yy=y-20;
   end;
    more statements
```

CAUTION: You can only use the GOTO statement inside a module or a DO group. As good programming practice, you should avoid GOTO statements when they refer to a label above the GOTO statement; otherwise, an infinite loop is possible.

GPIE Call

draws pie slices

CALL GPIE(*x*, *y*, *r* <, *angle1*><, *angle2*><, *color*><, *outline*>
<, *pattern*><, *window*><, *viewport*>**);**

The inputs to the GPIE subroutine are AS FOLLOWS:

x and y	are numeric scalars (or possibly vectors) defining the center (or centers) of the pie (or pies).
r	is a scalar or vector giving the radii of the pie slices.
angle1	is a scalar or vector giving the start angles. It defaults to 0.
angle2	is a scalar or vector giving the terminal angles. It defaults to 360.
color	is a valid SAS color, where *color* can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
outline	is an index indicating the side of the slice to draw. The default is 3.
pattern	is a character matrix or quoted literal that specifies the pattern with which to fill the interior of a closed curve.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form **{\ob minimum-x minimum-y maximum-x maximum-y\obe}**
viewport	is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

The GPIE subroutine draws one or more pie slices. The number of pie slices is the maximum dimension of the first five vectors. The angle arguments are specifed in degrees. The start angle (*angle1*) defaults to 0, and the terminal angle (*angle2*) defaults to 360. *Outline* is an index that indicates the side of the slice to draw. The *outline* specification can be one of the following:

<0	uses absolute value as the line style and draws no line segment from center to arc.
0	draws no line segment from center to arc.
1	draws an arc and line segment from the center to the starting angle point.
2	draws an arc and line segment from the center to the ending angle point.
3	draws all sides of the slice. This is the default.

Color, *outline*, and *pattern* can have more than one element. The coordinates in use for this graphics command are world coordinates. An example using the GPIE subroutine follows:

```
/* draws a pie with 4 slices of equal size */
call gpie(50,50,30,{0 90 180 270},{90 180 270 0});
```

GPIEXY Call

converts from polar to world coordinates

CALL GPIEXY(x, y, *fract-radii, angles*<, *center*><, *radius*><, *window*>**);**

The inputs to the GPIEXY subroutine are as follows:

x and y	are vectors of coordinates returned by GPIEXY.
fract-radii	is a vector of fractions of the radius of the reference circle.
angles	is the vector of angle coordinates in degrees.
center	defines the reference circle.
radius	defines the reference circle.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

The GPIEXY subroutine computes the world coordinates of a sequence of points relative to a circle. The x and y arguments are vectors of new coordinates returned by the GPIEXY subroutine. Together, the vectors *fract-radii* and *angles* define the points in polar coordinates. Each pair from the *fract-radii* and *angles* vectors yields a corresponding pair in the x and y vectors. For example, suppose *fract-radii* has two elements, 0.5 and 0.3, and the corresponding two elements of *angles* are 90 and 30. The GPIEXY subroutine returns two elements in the x vector and two elements in the y vector. The first (x, y) pair locates a point half way from the center to the reference circle on the vertical line through the center, and the second (x, y) pair locates a point one-third of the way on the line segment from the center to the reference circle, where the line segment slants 30 degrees from the horizontal. The reference circle can be defined by an earlier GPIE call or another GPIEXY call, or it can be defined by specifying *center* and *radius*.

Graphics devices can have diverse aspect ratios; thus, a circle may appear distorted when drawn on some devices. The SAS graphics system adjusts computations to compensate for this distortion. Thus, for any given point, the transformation from polar coordinates to world coordinates may need an equivalent adjustment. The GPIEXY subroutine ensures that the same adjustment applied in the GPIE subroutine is applied to the conversion. An example using the GPIEXY call follows.

```
   /* add labels to a pie with 4 slices of equal size */
call gpie(50,50,30,{0 90 180 270},{90 180 270 0});
call gpiexy(x,y,1.2,{45 135 225 315},{50 50},30,{0 0 100 100});

   /* adjust for label size: */
x [4,]=x[4,]-3;
x [1,]=x[1,]-4;
x [2,]=x[2,]+1;
call gscript(x,y,{'QTR1' 'QTR2' 'QTR3' 'QTR4'});
call gshow;
```

GPOINT Call

plots points

CALL GPOINT(*x*, *y* <, *symbol*><, *color*><, *height*><, *window*> <, *viewport*>**);**

The inputs to the GPOINT subroutine are as follows:

x	is a vector containing the x coordinates of points.
y	is a vector containing the y coordinates of points.
symbol	is a character vector or quoted literal that specifies a valid plotting symbol or symbols.
color	is a valid SAS color, where *color* can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
height	is a numeric matrix or literal specifying the character height.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}
viewport	is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

The GPOINT subroutine marks one or more points with symbols. The x and y vectors define the points where the markers are to be placed. The *symbol* and *color* arguments can have from one up to as many elements as there are well-defined points. The coordinates in use for this graphics command are world coordinates.

In the example that follows, points on the line $Y = X$ are generated for $30 \leq X \leq 80$ and then plotted with the GPOINT call:

```
x=30:80;
y=x;
```

```
call gpoint(x,y);
call gshow;
```

As another example, you can plot symbols at specific locations on the screen using the GPOINT subroutine. To print **i** in the lower left corner and **j** in the upper right corner, use the statements

```
call gpoint({10 80},{5 95},{i j});
call gshow;
```

See Chapter 12, "Graphics Examples," for examples using the GPOINT subroutine.

GPOLY Call

draws and fills a polygon

CALL GPOLY(*x*, *y* <, *style*><, *ocolor*><, *pattern*><, *color*>
<, *window*><, *viewport*>**);**

The inputs to the GPOLY subroutine are as follows.

x is a vector defining the x coordinates of the corners of the polygon.

y is a vector defining the y coordinates of the corners of the polygon.

style is a numeric matrix or literal that specifies an index corresponding to a valid line style.

ocolor is a matrix or literal specifying a valid outline color. The *ocolor* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.

pattern is a character matrix or quoted literal that specifies the pattern to fill the interior of a closed curve.

color is a valid SAS color used in filling the polygon. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.

window is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

viewport is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GPOLY subroutine fills an area enclosed by a polygon. The polygon is defined by the set of points given in the vectors x and y. The *color* argument is the color used in shading the polygon, and *ocolor* is the outline color. By default, the shading color and the outline color are the same, and the interior pattern is empty. The coordinates in use for this graphics command are world coordinates. An example using the GPOLY subroutine follows:

```
xd={20 20 80 80};
yd={35 85 85 35};
call gpoly (xd,yd, , ,'X','red');
```

GPORT Call

defines a viewport

CALL GPORT(*viewport***);**

where *viewport* is a numeric matrix or literal defining the viewport. The rectangular area's boundary is specified in normalized coordinates, where you specify the coordinates of the lower left corner and the upper right corner of the rectangular area in the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GPORT subroutine changes the current viewport. The *viewport* argument defines the new viewport using device coordinates (always 0 to 100). Changing the viewport may affect the height of the character fonts; if so, you may want to modify the HEIGHT parameter. An example of a valid statement follows:

```
call gport({20 20 80 80});
```

The default values for viewport are 0 0 100 100.

GPORTPOP Call

pops the viewport

CALL GPORTPOP;

The GPORTPOP subroutine deletes the top viewport from the stack.

GPORTSTK Call

stacks the viewport

CALL GPORTSTK(*viewport***);**

where *viewport* is a numeric matrix or literal defined in normalized coordinates in the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GPORTSTK subroutine stacks the viewport defined by the matrix *viewport* onto the current viewport; that is, the new viewport is defined relative to the current viewport. The coordinates in use for this graphics command are world coordinates. An example of a valid statement follows:

```
call gportstk({5 5 95 95});
```

GSCALE Call

calculates round numbers for labeling axes

CALL GSCALE(*scale, x, nincr*<*, nicenum*><*, fixed-end*>**);**

The inputs to the GSCALE subroutine are as follows:

scale	is a returned vector containing the scaled minimum data value, the scaled maximum data value, and a grid increment.
x	is a numeric matrix or literal.
nincr	is the number of intervals desired.
nicenum	is numeric and provides up to ten numbers to use for scaling. By default, *nicenum* is (1,2,2.5,5).
fixed-end	is a character argument and specifies which end of the scale is held fixed. The default is **X**.

The GSCALE subroutine obtains simple (round) numbers with uniform grid interval sizes to use in scaling a linear axis. The GSCALE subroutine implements algorithm 463 of Collected Algorithms from CACM. The scale values are integer multiples of the interval size. They are returned in the first argument, a vector with three elements. The first element is the scaled minimum data value. The second element is the scaled maximum data value. The third element is the grid increment.

The required input parameters are *x*, a matrix of data values, and *nincr*, the number of intervals desired. If *nincr* is positive, the scaled range includes approximately *nincr* intervals. If *nincr* is negative, the scaled range includes exactly ABS(*nincr*) intervals. The *nincr* parameter cannot be zero.

The *nicenum* and *fixed-end* arguments are optional. The *nicenum* argument provides up to ten numbers, all between 1 and 10 (inclusive of the end points), to be used for scaling. The default for *nicenum* is 1, 2, 2.5, and 5. The linear scale with this set of numbers is a scale with an interval size that is the product of an integer power of 10 and 1, 2, 2.5, or 5. Changing these numbers alters the rounding of the scaled values.

For *fixed-end*, **U** fixes the upper end; **L** fixes the lower end; **X** allows both ends to vary from the data values. The default is **X**. An example using the GSCALE subroutine follows:

```
   /* scalemat is set to {0,1000,100} */
call gscale(scalmat, {1 1000}, 10);
```

GSCRIPT Call

writes multiple text strings with special fonts

CALL GSCRIPT(*x, y, text*<*, angle*><*, rotate*><*, height*><*, font*><*, color*><*, window*><*, viewport*>**);**

The inputs to the GSCRIPT subroutine are as follows:

x is a scalar or vector containing the x coordinates of the lower left starting position of the text string's first character.

y is a scalar or vector containing the y coordinates of the lower left starting position of the text string's first character.

text is a character vector of text strings.

angle is the slant of each text string.

rotate is the rotation of individual characters.

height is a real number specifying the character height.

font is a character matrix or quoted literal that specifies a valid font name.

color is a valid SAS color. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.

window is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

viewport is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GSCRIPT subroutine writes multiple text strings with special character fonts. The x and y vectors describe the coordinates of the lower left starting position of the text string's first character. The *color* argument can have more than one element.

Note: Hardware characters cannot always be obtained if you change the HEIGHT or ASPECT parameters or if you use a viewport.

The coordinates in use for this graphics command are world coordinates. Examples of valid statements follow:

```
call gscript(7,y,names);
call gscript(50,50,"plot of height vs weight");
call gscript(10,90,"yaxis",-90,90);
```

GSET Call

sets attributes for a graphics segment

CALL GSET(*attribute*<, *value*>**);**

The inputs to the GSET subroutine are as follows:

attribute is a graphics attribute. The *attribute* argument can be a character matrix or quoted literal.

value is the value to which the attribute is set. The *value* argument is specified as a matrix or quoted literal.

The GSET subroutine enables you to change the following attributes for the current graphics segment.

aspect a numeric matrix or literal that specifies the aspect ratio (width relative to height) for characters.

color a valid SAS color. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the colorlist.

font a character matrix or quoted literal that specifies a valid font name.

height a numeric matrix or literal that specifies the character height.

pattern a character matrix or quoted literal that specifies the pattern to use to fill the interior of a closed curve.

style a numeric matrix or literal that specifies an index corresponding to a valid line style.

thick an integer specifying line thickness.

To reset the IML default value for any one of the attributes, omit the second argument. Attributes are reset back to the default with a call to the GOPEN or GSTART subroutines. Single or double quotes can be used around this argument. For more information on the attributes, see Chapter 12, "Graphics Examples."

Examples of valid statements follow:

```
call gset('pattern','m1n45');
call gset('font','simplex');

f='font';
s='simplex';
call gset(f,s);
```

For example, the statement

```
call gset("color");
```

resets *color* to its default.

GSHOW Call

shows a graph

CALL GSHOW <**(***segment-name***)**>**;**

where *segment-name* is a character matrix or literal specifying a graphics segment.

If you do not specify *segment-name*, the GSHOW subroutine displays the current graph. If the current graph is active at the time that the GSHOW subroutine is called, it remains active after the call; that is, graphics primitives can still be added to the segment. On the other hand, if you specify *segment-name*, the GSHOW subroutine closes any active graphics segment, searches the current catalog for a segment with the given name, and then displays that graph. Examples of valid statements follow.

```
call gshow;
call gshow("plot_a5");

seg={myplot};
call gshow(seg);
```

GSORTH Call

computes the Gram-Schmidt orthonormalization

CALL GSORTH(p, t, *lindep*, a**);**

The inputs to the GSORTH subroutine are as follows:

p	is an $m \times n$ column-orthonormal output matrix.
t	is an upper triangular $n \times n$ output matrix.
lindep	is a flag with a value of 0 if columns of a are independent and a value of 1 if they are dependent. The *lindep* argument is an output scalar.
a	is an input $m \times n$ matrix.

The GSORTH subroutine computes the Gram-Schmidt orthonormal factorization of the $m \times n$ matrix $\mathbf{A}$, where m is greater than or equal to n; that is, the GSORTH subroutine computes the column-orthonormal $m \times n$ matrix $\mathbf{P}$ and the upper triangular $n \times n$ matrix $\mathbf{T}$ such that

$$\mathbf{A} = \mathbf{P} * \mathbf{T} .$$

If the columns of $\mathbf{A}$ are linearly independent (that is, rank$(\mathbf{A}) = n$), then $\mathbf{P}$ is full-rank column-orthonormal: $\mathbf{P}'\mathbf{P} = \mathbf{I}_w$, $\mathbf{T}$ is nonsingular, and the value of *lindep* (a scalar) is set to 0. If the columns of $\mathbf{A}$ are linearly dependent (say rank$(\mathbf{A}) = k < n$) then $n - k$ columns of $\mathbf{P}$ are set to 0, the corresponding rows of $\mathbf{T}$ are set to 0 ($\mathbf{T}$ is singular), and *lindep* is set to 1. The pattern of zero columns in $\mathbf{P}$ corresponds to the pattern of linear dependencies of the columns of $\mathbf{A}$ when columns are considered in left-to-right order.

The GSORTH subroutine is not recommended for the construction of matrices of values of orthogonal polynomials; the ORPOL functionhould be used for that purpose.

If *lindep* is 1, you can rearrange the columns of $\mathbf{P}$ and rows of $\mathbf{T}$ so that the zero columns of $\mathbf{P}$ are right-most, that is, $\mathbf{P} = (\mathbf{P}(,1), \mathbf{P}(,k), 0, \ldots, 0)$, where k is the column rank of $\mathbf{A}$ and $\mathbf{A} = \mathbf{P} * \mathbf{T}$ is preserved. The following statements make this rearrangement:

```
d=rank((ncol(t)-(1:ncol(t))')#(vecdiag(t)=0));
temp=p;
p[,d]=temp;
temp=t;
t[,d]=temp;
```

An example of a valid GSORTH call follows:

```
x={1 1 1, 1 2 4, 1 3 9};
xpx=x'*x;
call gsorth(p, t, l, xpx);
```

These statements produce the output matrices

```
P             3 rows      3 cols    (numeric)

          0.193247 -0.753259 0.6286946
          0.386494 -0.530521 -0.754434
         0.9018193 0.3887787 0.1886084


T             3 rows      3 cols    (numeric)

         15.524175 39.035892 104.99753
                 0 2.0491877 8.4559365
                 0         0 0.1257389


L             1 row       1 col     (numeric)

                          0
```

See "Acknowledgments" in the front of this book for authorship of the GSORTH subroutine.

GSTART Call

initializes the graphics system

CALL GSTART(<*catalog*><, *replace*>**);**

The inputs to the GSTART subroutine are as follows:

catalog is a character matrix or quoted literal specifying the SAS catalog for saving the graphics segments.

replace is a numeric argument.

The GSTART subroutine activates the graphics system the first time it is called. A catalog is opened to capture the graphics segments to be generated in the session. If you do not specify a catalog, IML uses the temporary catalog WORK.GSEG.

The *replace* argument is a flag; a nonzero value indicates that the new segment should replace the first found segment with the same name. The *replace* flag set by the GSTART subroutine is a global flag, as opposed to the *replace* flag set by the GOPEN subroutine. When set by GSTART, this flag is applied to all subsequent segments created for this catalog, whereas with GOPEN, the *replace* flag is applied only to the segment that is being created. The GSTART subroutine sets the *replace* flag to 0 when the *replace* argument is omitted. The *replace* option can be very inefficient for a catalog with many segments. In this case, it is better to create segments with different names (if necessary) than to use the *replace* option.

The GSTART subroutine must be called at least once to load the graphics subsystem. Any subsequent GSTART calls are generally to change graphics catalogs or reset the global *replace* flag.

The GSTART subroutine resets the defaults for all graphics attributes that can be changed by the GSET subroutine. It does not reset GOPTIONS back to their defaults unless the GOPTION corresponds to a GSET parameter. The GOPEN subroutine also resets GSET parameters.

An example of a valid statement follows:

```
call gstart;
```

GSTOP Call

deactivates the graphics system

CALL GSTOP;

The GSTOP subroutine deactivates the graphics system. The graphics subsystem is disabled until the GSTART subroutine is called again.

GSTRLEN Call

finds the string length

CALL GSTRLEN(*length, text*<, *height*><, *font*><, *window*>**);**

The inputs to the GSTRLEN subroutine are as follows:

length	is a matrix of lengths specified in world coordinates.
text	is a matrix of text strings.
height	is a numeric matrix or literal specifying the character height.
font	is a character matrix or quoted literal that specifies a valid font name.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

The GSTRLEN subroutine returns in world coordinates the graphics text lengths in a given font and for a given character height. The *length* argument is the returned matrix. It has the same shape as the matrix *text*. Thus, if *text* is an $n \times m$ matrix of text strings, then *length* will be an $n \times m$ matrix of lengths in world coordinates. If you do not specify *font*, the default font is assumed. If you do not specify *height*, the default height is assumed. An example using the GSTRLEN subroutine follows.

```
   /* centers text strings about coordinate      */
   /* points (50, 90) assume font=simplex        */
ht=2;
x=30;
y=90;
str='Nonparametric Cluster Analysis';
call gstrlen(len, str, ht, 'simplex');
call gscript(x-(len/2), y, str, ,,ht,'simplex');
```

GTEXT and GVTEXT Calls

place text horizontally or vertically on a graph

CALL GTEXT(*x*, *y*, *text*<, *color*><, *window*><, *viewport*>**);**
CALL GVTEXT(*x*, *y*, *text*<, *color*><, *window*><, *viewport*>**);**

The inputs to the GTEXT and GVTEXT subroutines are as follows:

x is a scalar or vector containing the x coordinates of the lower left starting position of the text string's first character.

y is a scalar or vector containing the y coordinates of the lower left starting position of the text string's first character.

text is a vector of text strings

color is a valid SAS color. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.

window is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

viewport is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GTEXT subroutine places text horizontally across a graph; the GVTEXT subroutine places text vertically on a graph. Both subroutines use hardware characters when possible. The number of text strings drawn is the maximum dimension of the first three vectors. The *color* argument can have more than one element. Hardware characters cannot always be obtained if you change the HEIGHT or ASPECT parameters (using GSET or GOPTIONS) or if you use a viewport. The coordinates in use for this graphics command are world coordinates.

Examples of the GTEXT and GVTEXT subroutines follow:

```
call gopen;
call gport({0 0 50 50});
call gset('height',4); /* shrink to a 4th of the screen */
call gtext(50,50,'Testing GTEXT: This will start in the
                  center of the viewportp ');
call gshow;
call gopen;
call gvtext(.35,4.6,'VERTICAL STRING BY GVTEXT',
           'white',{0.2 -1,1.5 6.5},{0 0,100 100});
call gshow;
```

GWINDOW Call

defines the data window

CALL GWINDOW(*window***);**

where *window* is a numeric matrix or literal specifying a window. The rectangular area's boundary is given in world coordinates, where you specify the lower left and upper right corners in the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GWINDOW subroutine sets up the window for scaling data values in subsequent graphics primitives. It is in effect until the next GWINDOW call or until the segment is closed. The coordinates in use for this graphics command are world coordinates. An example using the GWINDOW subroutine follows:

```
ydata={2.358,0.606,3.669,1.000,0.981,1.192,0.926,1.590,
       1.806,1.962,4.028,3.148,1.836,2.845,1.013,0.414};
xdata={1.215,0.930,1.152,1.138,0.061,0.696,0.686,1.072,
       1.074,0.934,0.808,1.071,1.009,1.142,1.229,0.595};

   /* WD shows the actual range of the data */
wd=(min(xdata)||min(ydata))//(max(xdata)||max(ydata));
call gwindow(wd);
```

GXAXIS and GYAXIS Calls

draw a horizontal or vertical axis

CALL GXAXIS(*starting-point, length, nincr* <, *nminor*>< , *noticklab*>
<, *format*>< , *height*>< , *font*>< , *color*>< , *fixed-end*>

<, *window*>< , *viewport*>**);**

CALL GYAXIS(*starting-point, length, nincr* <, *nminor*>< , *noticklab*>

<, *format*>< , *height*>< , *font*>< , *color*>< , *fixed-end*>

<, *window*>< , *viewport*>**);**

The inputs to the GXAXIS and GYAXIS subroutines are as follows:

starting-point is the (x, y) starting point of the axis, specified in world coordinates.

length is a numeric scalar giving the length of the axis.

nincr is a numeric scalar giving the number of major tick marks on the axis.

nminor is an integer specifying the number of minor tick marks between major tick marks.

noticklab is a flag that is nonzero if the tick marks are not labeled. The default is to label tick marks.

format is a character scalar that gives a valid SAS numeric format used in formatting the tick-mark labels. The default format is 8.2.

height is a numeric matrix or literal that specifies the character height. This is used for the tick-mark labels.

font is a character matrix or quoted literal that specifies a valid font name. This is used for the tick-mark labels.

color is a valid color. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.

fixed-end allows one end of the scale to be held fixed. **U** fixes the upper end; **L** fixes the lower end; **X** and allows both ends to vary from the data values. In addition, you may specify **N**, which causes the axis routines to bypass the scaling routine. The interval between tick marks is *length* divided by (*nincr*-1). The default is **X**.

window is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

viewport is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form

{*minimum-x minimum-y maximum-x maximum-y*}

The GXAXIS subroutine draws a horizontal axis; the GYAXIS subroutine draws a vertical axis. The first three arguments are required.

The *starting-point* argument is a matrix of two numbers given in world coordinates. The matrix is the (x, y) starting point of the axis.

The *length* argument is a scalar value giving the length of the x axis or y axis in world coordinates along the x or y direction.

The *nincr* argument is a scalar value giving the number of major tick marks shown on the axis. The first tick mark will be on the starting point as specified.

The axis routines use the same scaling algorithm as the GSCALE subroutine. For example, if the x starting point is 10 and the length of the axis is 44, and if you call the GSCALE subroutine with the x vector containing the two elements, 10 and 44, the scale obtained should be the same as that obtained by the GXAXIS subroutine. Sometimes, it may be helpful to use the GSCALE subroutine in conjunction with the axis routines to get more precise scaling and labeling.

For example, suppose you want to draw the axis for $-2 \leq X \leq 2$ and $-2 \leq Y \leq 2$. The code below draws these axes. Each axis is 4 units long. Note that the x axis begins at the point $(-2, 0)$ and the y axis begins at the point $(0, -2)$. The tick marks can be set at each integer value, with minor tick marks in between the major tick marks. The *noticklab* option is turned off, so that the tick marks are not labeled.

```
call gport({20 20 80 80});
call gwindow({-2 -2 2 2});
call gxaxis({-2,0},4,5,2,1);
call gyaxis({0,-2},4,5,2,1);
```

HALF Function

computes Cholesky decomposition

HALF(*matrix***)**

where *matrix* is a numeric matrix or literal.

The HALF function is the same as the ROOT function. See the description of the ROOT function for Cholesky decomposition.

HANKEL Function

generates a Hankel matrix

HANKEL(*matrix*)

where *matrix* is a numeric matrix or literal.

The HANKEL function generates a Hankel matrix from a vector, or a block Hankel matrix from a matrix. A block Hankel matrix has the property that all matrices on the reverse diagonals are the same. The argument matrix is an $(np) \times p$ or $p \times (np)$ matrix; the value returned is the $(np) \times (np)$ result.

The Hankel function uses the first $p \times p$ submatrix $\mathbf{A}_1$ of the argument matrix as the blocks of the first reverse diagonal. The second $p \times p$ submatrix $\mathbf{A}_2$ of the argument matrix forms the second reverse diagonal. The remaining reverse diagonals are formed accordingly. After the values in the argument matrix have all been placed, the rest of the matrix is filled in with 0. If $\mathbf{A}$ is $(np) \times p$, then the first p columns of the returned matrix, $\mathbf{R}$, will be the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, then the first p rows of $\mathbf{R}$will be the same as $\mathbf{A}$. The HANKEL function is especially useful in time-series applications, where the covariance matrix of a set of variables representing the present and past and a set of variables representing the present and future is often assumed to be a block Hankel matrix. If

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2|\mathbf{A}_3|\cdots|\mathbf{A}_n]$$

and if $\mathbf{R}$ is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \left[\begin{array}{c|c|c|c|c} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 & \cdots & \mathbf{0} \\ \mathbf{A}_3 & \mathbf{A}_4 & \mathbf{A}_5 & \cdots & \mathbf{0} \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{array}\right]$$

If

$$\mathbf{A} = \left[\begin{array}{c} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{array}\right]$$

and if $\mathbf{R}$ is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \left[\begin{array}{c|c|c|c|c} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 & \cdots & \mathbf{0} \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{array}\right]$$

For example, the IML code

```
r=hankel({1 2 3 4 5});
```

results in

```
R            5 rows      5 cols    (numeric)

        1         2         3         4         5
        2         3         4         5         0
        3         4         5         0         0
        4         5         0         0         0
        5         0         0         0         0
```

The statement

```
r=hankel({1 2 ,
          3 4 ,
          5 6 ,
          7 8});
```

returns the matrix

```
R             4 rows      4 cols    (numeric)

          1         2         5         6
          3         4         7         8
          5         6         0         0
          7         8         0         0
```

And the statement

```
r=hankel({1 2 3 4 ,
          5 6 7 8});
```

returns the result

```
R             4 rows      4 cols    (numeric)

          1         2         3         4
          5         6         7         8
          3         4         0         0
          7         8         0         0
```

HDIR Function

performs a horizontal direct product

HDIR(*matrix1, matrix2***)**

where *matrix1* and *matrix2* are numeric matrices or literals.

The HDIR function performs a direct product on all rows of *matrix1* and *matrix2* and creates a new matrix by stacking these row vectors into a matrix. This operation is useful in constructing design matrices of interaction effects. The *matrix1* and *matrix2* arguments must have the same number of rows. The result has the same number of rows as *matrix1* and *matrix2*. The number of columns is equal to the product of the number of columns in *matrix1* and *matrix2*.

For example, the statements

```
a={1 2,
   2 4,
   3 6};
b={0  2,
   1  1,
   0 -1};
c=hdir(a,b);
```

produce a matrix containing the values

```
C             3 rows      4 cols    (numeric)

                0           2         0          4
                2           2         4          4
                0          -3         0         -6
```

The HDIR function is useful for constructing crossed and nested effects from main effect design matrices in ANOVA models.

HERMITE Function

reduces a matrix to Hermite normal form

HERMITE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The HERMITE function uses elementary row operations to reduce a matrix to Hermite normal form. For square matrices this normal form is upper-triangular and idempotent.

If the argument is square and nonsingular, the result will be the identity matrix. In general the result satisfies the following four conditions (Graybill 1969, p. 120):

- It is upper-triangular.
- It has only values of 0 and 1 on the diagonal.
- If a row has a 0 on the diagonal, then every element in that row is 0.
- If a row has a 1 on the diagonal, then every off-diagonal element is 0 in the column in which the 1 appears.

Consider the following example (Graybill 1969, p. 288):

```
a={3  6  9,
   1  2  5,
   2  4 10};
h=hermite(a);
```

These statements produce

```
H             3 rows      3 cols    (numeric)

                1         2         0
                0         0         0
                0         0         1
```

If the argument is a square matrix, then the Hermite normal form can be transformed into the row echelon form by rearranging rows in which all values are 0.

HOMOGEN Function

solves homogeneous linear systems

HOMOGEN(*matrix***)**

where *matrix* is a numeric matrix or literal.

The HOMOGEN function solves the homogeneous system of linear equations $\mathbf{A} * \mathbf{X} = \mathbf{0}$ for $\mathbf{X}$. For at least one solution vector $\mathbf{X}$ to exist, the $m \times n$ matrix $\mathbf{A}$, $m \geq n$, has to be of rank $r < n$. The HOMOGEN function computes an $n \times (n - r)$ column orthonormal matrix $\mathbf{X}$ with the property $\mathbf{A} * \mathbf{X} = \mathbf{0}$, $\mathbf{X}'\mathbf{X} = \mathbf{I}$. If $\mathbf{A}'\mathbf{A}$ is ill conditioned, rounding-error problems can occur in determining the correct rank of $\mathbf{A}$ and in determining the correct number of solutions $\mathbf{X}$. Consider the following example (Wilkinson and Reinsch 1971, p. 149):

```
a={22  10   2   3   7,
   14   7  10   0   8,
   -1  13  -1 -11   3,
   -3  -2  13  -2   4,
    9   8   1  -2   4,
    9   1  -7   5  -1,
    2  -6   6   5   1,
    4   5   0  -2   2};
x=homogen(a);
```

These statements produce the solution

```
        X             5 rows      2 cols    (numeric)

                   -0.419095           0
                   0.4405091   0.4185481
                   -0.052005   0.3487901
                   0.6760591    0.244153
                   0.4129773   -0.802217
```

In addition, this function could be used to determine the rank of an $m \times n$ matrix $\mathbf{A}$, $m \geq n$.

I Function

creates an identity matrix

I(*dimension***)**

where *dimension* specifies the size of the identity matrix.

The I function creates an identity matrix with *dimension* rows and columns. The diagonal elements of an identity matrix are 1s; all other elements are 0s. The value of *dimension* must be an integer greater than or equal to 1. Noninteger operands are truncated to their integer part.

For example, the statement

```
a=I(3);
```

yields the result

```
A
1  0  0
0  1  0
0  0  1
```

IF-THEN/ELSE Statement

conditionally executes statements

IF *expression* **THEN** *statement1*;
ELSE *statement2*;

The inputs to the IF-THEN/ELSE statements are

expression is an expression that is evaluated for being true or false.

statement1 is a statement executed when *expression* is true.

statement2 is a statement executed when *expression* is false.

The IF statement contains an expression to be evaluated, the keyword THEN, and an action to be taken when the result of the evaluation is true.

The ELSE statement optionally follows the IF statement and gives an action to be taken when the IF expression is false. The expression to be evaluated is often a comparison, for example,

```
if max(a)<20 then p=0;
else p=1;
```

The IF statement results in the evaluation of the condition MAX(A)<20. If the largest value found in matrix **A** is less than 20, **P** is set to 0. Otherwise, **P** is set to 1. See the description of the MAX function for details.

When the condition to be evaluated is a matrix expression, the result of the evaluation is another matrix. If all values of the result matrix are nonzero and nonmissing, the condition is true; if any element in the result matrix is 0 or missing, the condition is false. This evaluation is equivalent to using the ALL function.

For example, writing

```
if x<y then
   do;
```

produces the same result as writing

```
if all(x<y) then
   do;
```

IF statements can be nested within the clauses of other IF or ELSE statements. Any number of nesting levels is allowed. Below is an example.

```
if x=y then if abs(y)=z then
   do;
```

CAUTION: Execution of THEN clauses occurs as if you were using the ALL function.

The statements

```
if a^=b then do;
```

and

```
if ^(a=b) then do;
```

are both valid, but the THEN clause in each case is only executed when all corresponding elements of **A** and **B** are unequal; that is, when none of the corresponding elements are equal.

Evaluation of the statement

```
if any(a^=b) then do;
```

requires only one element of **A** and **B** to be unequal for the expression to be true.

IFFT Function

computes the inverse finite Fourier transform

IFFT(*f*)

where f is an $np \times 2$ numeric matrix.

The IFFT function expands a set of sine and cosine coefficients into a sequence equal to the sum of the coefficients times the sine and cosine functions. The argument f is an $np \times 2$ matrix; the value returned is an $n \times 1$ vector.

Note: If the element in the last row and second column of f is exactly 0, then n is $2np - 2$; otherwise, n is $2np - 1$.

The inverse finite Fourier transform of a two column matrix $\mathbf{F}$, denoted by the vector $\mathbf{x}$ is

$$x_i = F_{1,1} + 2\sum_{j=2}^{n}\left(F_{j,1}\cos\left(\frac{2\pi}{n}(j-1)(i-1)\right) + F_{j,2}\sin\left(\frac{2\pi}{n}(j-1)(i-1)\right)\right) + q_i$$

for $i = 1, \ldots, n$, where $q_i = (-1)^i \mathbf{F}_{np,1}$ if n is even, or $q = 0$ if n is odd.

Note: For most efficient use of the IFFT function, n should be a power of 2. If n is a power of 2, a fast Fourier transform is used (Singleton 1969); otherwise, a Chirp-Z algorithm is used (Monro and Branch 1976).

IFFT(FFT(X)) returns n times $\mathbf{x}$, where n is the dimension of $\mathbf{x}$. If f is not the Fourier transform of a real sequence, then the vector generated by the IFFT function is not a true inverse Fourier transform. However, applications exist where the FFT and IFFT functions may be used for operations on multidimensional or complex data (Gentleman and Sande 1966; Nussbaumer 1982).

The convolution of two vectors $\mathbf{x}$ ($n \times 1$) and $\mathbf{y}$ ($m \times 1$) can be accomplished using the following statements:

```
a=fft(x//j(nice-nrow(x),1,0));
b=fft(y//j(nice-nrow(y),1,0));
z=(a#b)],+[;
b],2[=-b],2[;
z=ifft(z||((a#(b],2 1[))],+[));
```

where NICE is a number chosen to allow efficient use of the FFT and IFFT functions and also is greater than $n + m$.

Windowed spectral estimates and inverse autocorrelation function estimates can also be readily obtained.

INDEX Statement

indexes a variable in a SAS data set

INDEX *variables*|**NONE**

where *variables* are the names of variables for which indexes are to be built.

You can use the INDEX statement to create an index for the named variables in the current input SAS data set. An index is created for each variable listed if it does not already have an index. Current retrieval is set to the last variable indexed. Subsequent I/O operations such as LIST, READ, FIND, and DELETE may use this index to retrieve observations from the data set if IML determines that indexed retrieval will be faster. The indices are automatically updated when a data set is edited with the APPEND, DELETE, or REPLACE statements. Only one index is in effect at any given time. The SHOW *contents* command indicates which index is in use.

For example, the following statement creates indexes for the SAS data set CLASS in the order of NAME and the order of SEX:

```
index name sex;
```

Current retrieval is set to use SEX. A LIST*all* statementould list females before males.

An INDEX *none* statement can be used to set retrieval back to physical order.

When a WHERE clause is being processed, IML automatically determines which index to use, if any. The decision is based on the variables and operators involved in the WHERE clause, and the decision criterion is based on the efficiency of retrieval.

INFILE Statement

opens a file for input

INFILE *operand* <*options*>**;**

The inputs to the INFILE statement are as follows:

operand is either a predefined filename or a quoted string containing the filename or character expression in parentheses referring to the filepath.

options are explained below.

You can use the INFILE statement to open an external file for input or, if the file is already open, to make it the current input file so that subsequent INPUT statements read from it.

The options available for the INFILE statement are described as follows.

LENGTH=variable
specifies a variable in which the length of a record will be stored as IML reads it in.

RECFM=N
specifies that the file is to be read in as a pure binary file rather than as a file with record separator characters. To do this, you must use the byte operand (<) on the INPUT statement to get new records rather than using separate input statements or the new line (/) operator.

The following options control how IML behaves when an INPUT statement tries to read past the end of a record. The default is STOPOVER.

FLOWOVER
allows the INPUT statement to go to the next record to obtain values for the variables.

MISSOVER
tolerates attempted reading past the end of the record by assigning missing values to variables read past the end of the record.

STOPOVER
treats going past the end of a record as an error condition, which triggers an end-of-file condition.

Several examples of INFILE statements are given below:

```
filename in1 'student.dat';  /* specify filename IN1   */
infile in1;                  /* infile filepath       */

infile 'student.dat';        /* path by quoted literal */

infile 'student.dat' missover;       /* using options  */
```

See Chapter 7, "File Access," for further information.

INPUT Statement

inputs data

INPUT <*variables*> <*informats*> <*record-directives*> <*positionals*>**;**

where the clauses and options are explained below.

You can use the INPUT statement to input records from the current input file, placing the values into IML variables. The INFILE statement sets up the current input file. See Chapter 7, "File Access,", for details.

The INPUT statement contains a sequence of arguments that include the following:

variables specify the variable or variables you want to read from the current position in the record. Each variable can be followed immediately by an input format specification.

informats specify an input format. These are of the form $w.d$ or $\$w.$ for standard numeric and character informats, respectively, where w is the width of the field and d is the decimal parameter, if any. You can also use a SAS format of the form *NAMEw.d*, where *NAME* is the name of the format. Also, you can use a single $ or & for list input applications. If the width is unspecified, the informat uses list-input rules to determine the length by searching for a blank (or comma) delimiter. The special format $RECORD. is used for reading the rest of the record into one variable. For more information on formats, refer to *SAS Language Reference: Dictionary*.

Record holding is always implied for RECFM=N binary files, as if the INPUT statement has a trailing @ sign. For more information, see Chapter 7, "File Access."

Examples of valid INPUT statements are shown below:

```
input x y;
input @1 name $ @20 sex $ @(20+2) age 3.;

eight=8;
input >9 <eight  number2 ib8.;
```

Below is an example using binary input:

```
proc iml;
   file 'out2.dat' recfm=n ;
   number=499; at=1;
   do i = 1 to 5;
      number=number+1;
      put >at number ib8.; at=at+8;
   end;
   closefile 'out2.dat';

   infile 'out2.dat' recfm=n;
   size=8;     /* 8 bytes */
   do pos=1 to 33 by size;
      input >pos number ib8.;
      print number;
   end;
```

record-directives are used to advance to a new record. *Record-directives* are the following:

holding @ sign — is used at the end of an INPUT statement to instruct IML to hold the current record so that you can continue to read from the record with later INPUT statements. Otherwise, IML automatically goes to the next record for the next INPUT statement.

/ — advances to the next record.

> *operand* — specifies that the next record to be read starts at the indicated byte position in the file (for

RECFM= N files only). The *operand* is a literal number, a variable name, or an expression in parentheses.

< *operand* instructs IML to read the indicated number of bytes as the next record. The record directive must be specified for binary files (RECFM=N). The *operand* is a literal number, a variable name, or an expression in parentheses.

positionals instruct PROC IML to go to a specific column on the record. The *positionals* are the following:

@ *operand* instructs IML to go to the indicated column, where *operand* is a literal number, a variable name, or an expression in parentheses. For example, @30 means to go to column 30. The operand can also be a character operand when pattern searching is needed. For more information, see Chapter 7, "File Access."

+ *operand* specifies to skip the indicated number of columns. The *operand* is a literal number, a variable name, or an expression in parentheses.

INSERT Function

inserts one matrix inside another

INSERT(x, y, *row*<, *column*>**)**

The inputs to the INSERT function are as follows:

x is the target matrix. It can be either numeric or character.

y is the matrix to be inserted into the target. It can be either numeric or character, depending on the type of the target matrix.

row is the row where the insertion is to be made.

column is the column where the insertion is to be made.

The INSERT function returns the result of inserting the matrix y inside the matrix x at the place specified by the *row* and *column* arguments. This is done by splitting x either horizontally or vertically before the row or column specified and concatenating y between the two pieces. Thus, if x is m rows by n columns, *row* can range from 0 to $m + 1$ and *column* can range from 0 to $n + 1$. However, it is not possible to insert in both dimensions simultaneously, so either *row* or *column* must be 0, but not both. The *column* argument is optional and defaults to 0. Also, the matrices must conform in the dimension in which they are joined.

For example, the statements

```
a={1 2, 3 4};
b={5 6, 7 8};
c=insert(a, b, 2, 0);
d=insert(a, b, 0, 3);
```

produce the result

```
C              4 rows      2 cols    (numeric)

                       1         2
                       5         6
                       7         8
                       3         4

D              2 rows      4 cols    (numeric)

               1         2         5         6
               3         4         7         8
```

C shows the result of insertion in the middle, while **D** shows insertion on an end.

INT Function

truncates a value

INT(*matrix***)**

where *matrix* is a numeric matrix or literal.

The INT function truncates the decimal portion of the value of the argument. The integer portion of the value of the argument remains. The INT function takes the integer value of each element of the argument matrix.

An example using the INT function follows:

```
c=2.8;
b=int(c);
```

```
B              1 row       1 col     (numeric)

                                 2
```

In the next example, notice that a value of 11 is returned. This is because of the maximal machine precision. If the difference is less than 1E−12, the INT function rounds up.

```
x={12.95  10.9999999999999,
   -30.5  1e-6};
```

```
b=int(x);

             B             2 rows      2 cols    (numeric)

                                12          11
                               -30           0
```

INV Function

computes the matrix inverse

INV(*matrix***)**

where *matrix* is a square nonsingular matrix.

The INV function produces a matrix that is the inverse of *matrix*, which must be square and nonsingular.

For $\mathbf{G} = \mathrm{INV}(\mathbf{A})$ the inverse has the properties

$$\mathbf{GA} = \mathbf{AG} = \text{identity} .$$

To solve a system of linear equations $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$, you can use the statement

```
x=inv(a)*b;
```

However, the SOLVE function is more accurate and efficient for this task.

The INV function uses an LU decomposition followed by backsubstitution to solve for the inverse, as described in Forsythe, Malcolm, and Moler (1967).

The INV function (as well as the DET and SOLVE functions) uses the following criterion to decide whether the input matrix, $\mathbf{A} = [a_{ij}]_{i,j=1,\ldots,n}$, is singular:

$$\mathit{sing} = 100 \times \mathit{MACHEPS} \times \max_{1 \le i,j \le n} |a_{ij}|$$

where *MACHEPS* is the relative machine precision.

All matrix elements less than or equal to *sing* are now considered rounding errors of the largest matrix elements, so they are taken to be zero. For example, if a diagonal or triangular coefficient matrix has a diagonal value less than or equal to *sing*, the matrix is considered singular by the DET, inv, and SOLVE functions.

Previously, a much smaller singularity criterion was used, which caused algebraic operations to be performed on values that were essentially floating point error. This occasionally yielded numerically unstable results. The new criterion is much more conservative, and it generates far fewer erroneous results. In some cases, you may need to scale the data to avoid singular matrices. If you think the new criterion is too strong,

- try the GINV function to compute the generalized inverse
- examine the size of the singular values returned by the SVD function. The SVD function can be used to compute a generalized inverse with a user-specified singularity criterion.

INVUPDT Function

updates a matrix inverse

INVUPDT(*matrix, vector*<, *scalar*>**)**

The inputs to the INVUPDT function are as follows:

matrix	is an $n \times n$ positive definite matrix.
vector	is an $n \times 1$ or $1 \times n$ vector.
scalar	is a numeric scalar.

The INVUPDT function updates a matrix inverse. For example, let

```
r=invupdt(a,x,w);
```

where $\mathbf{R}$ is an $n \times w$ matrix; $\mathbf{A}$ is an $n \times n$ positive definite matrix; $\mathbf{X}$ is an $n \times 1$ or $1 \times n$ vector; and w is an optional scalar (if not specified, w has default value 1).

The INVUPDT function computes the matrix expression

$$\mathbf{R} = \mathbf{A} - w\mathbf{A}\mathbf{X}(1 + w\mathbf{X}'\mathbf{A}\mathbf{X})^{-1}\mathbf{X}'\mathbf{A}^{-1}\mathbf{X}'\mathbf{A}$$

or, in matrix language,

```
r=a-w*a*x*inv(1+w*x'*a*x)*x'*a;
```

The INVUPDT function is used primarily to update a matrix inverse because the function has the property

$$\text{INVUPDT}(\mathbf{B}^{-1}, \mathbf{X}, w) = (\mathbf{B} + w\mathbf{X}\mathbf{X}')^{-1} .$$

If $\mathbf{Z}$ is a design matrix and $\mathbf{X}$ is a new observation to be used in estimating the parameters of a linear model, then the inverse crossproducts matrix that includes the new observation can be updated from the old inverse by

```
c2=invupdt(c,x);
```

where $\mathbf{C} = \text{INV}(\mathbf{Z}'\mathbf{Z})$. Note that

```
c2=inv((z//x)'*(z//x));
```

If w is 1, the function adds an observation to the inverse; if w is -1, the function removes an observation from the inverse. If weighting is used, w is the weight.

To perform the computation, the INVUPDT function uses about $2n^2$ multiplications and additions, where n is the row dimension of the positive definite argument matrix.

IPF Call

performs an iterative proportional fit

CALL IPF(*fit, status, dim, table, config*<*, initab*><*, mod*>**);**

The inputs to the IPF subroutine are as follows:

fit is a returned matrix. The argument *fit* specifies an array of the estimates of the expected number in each cell under the model specified in *config*. This matrix conforms to *table*.

status is a returned matrix. The *status* argument specifies a row vector of length 3. If you specify STATUS={*error, obs-maxdev, no-iterate*}, then *error* is 0 if there is convergence to the desired accuracy and is 3 if there is no convergence to the desired accuracy; *obs-maxdev* is the maximum difference between estimates of the last two iterations; and *no-iterate* is the number of iterations performed.

dim is an input matrix. The *dim* argument is a vector specifying the number of variables and the number of their possible levels in a contingency table. If *dim* is $1 \times v$, then there are v variables, and the value of the ith element is the number of levels of the ith variable.

table is an input matrix. The *table* argument specifies an array of the number of observations at each level of each variable. Variables are nested across columns and then across rows.

config an input matrix. The *config* argument gives an array specifying which marginal totals to fit. Each column specifies a distinct marginal in the model under consideration. Because the model is hierarchical, all subsets of specified marginals are included in fitting.

initab is an input matrix. The *initab* argument is an array of initial values for the iterative procedure. If you do not specify values, 1s are used. For incomplete tables, *initab* is set to 1 if the cell is included in the design, and 0 if it is not.

mod is an input matrix. The *mod* argument is a two-element vector specifying the stopping criteria. If *mod*= {*maxdev*, *maxit*}, then the procedure iterates until either *maxit* iterations are completed or the maximum difference between estimates of the last two iterations is less than *maxdev*. Default values are *maxdev*=0.25 and *maxit*=15.

The IPF subroutine performs an iterative proportional fit of the marginal totals of a contingency table. The arguments used with the IPF function can be matrix names or literals.

The matrix *table* must conform in size to the contingency table as specified in *dim*. In particular, if *table* is $n \times m$, the product of the entries in *dim* must equal nm. Furthermore, there must be some integer k such that the product of the first k entries in *dim* equals m. If you specify *initab*, then it must be the same size as *table*.

For example, consider the no-three-factor-effect model for interpreting Bartlett's data as described in Bishop, Fienberg, and Holland (1975):

```
dim={2 2 2};
table={156  84  84  156,
       107 133  31  209};
config={1  1  2,
        2  3  3};
call ipf(fit,status,dim,table,config);
```

The result is

```
            FIT
         161.062   78.938  78.907  161.093
         101.905  138.095  36.119  203.881


          STATUS
               0   .166966     4
```

Equivalent results are obtained by the statement

```
table={156  84,
        84 156,
       107 133,
        31 209};
```

or the statement

```
table={156 84 84 156 107 133 31 209};
```

In the first specification, TABLE is interpreted as

	variable 2:	1		2	
variable 3	variable 1:	1	2	1	2
1		156	84	84	156
2		107	133	31	209

In the second specification, TABLE is interpreted as

```
variable 3  variable 2    variable 1:         1            2
_________________________________________________________________
        1           1                       156           84
                    2                        84          156
        2           1                       107          133
                    2                        31          209
```

And in the third specification, TABLE is interpreted as

```
 variable 3:            1                              2
             _______________________       ________________________
 variable 2:    1             2               1               2
             _________     _________       __________     __________
 variable 1: 1      2      1      2        1       2      1       2
            _____________________________________________________
            156     84     84     156     107     133     31     209 .
```

J Function

creates a matrix of identical values

J(*nrow*<, *ncol*<, *value*>>**)**

The inputs to the J function are as follows:

nrow	is a numeric matrix or literal giving the number of rows.
ncol	is a numeric matrix or literal giving the number of columns.
value	is a numeric or character matrix or literal for filling the rows and columns of the matrix.

The J function creates a matrix with *nrow* rows and *ncol* columns with all elements equal to *value*. If *ncol* is not specified, it defaults to *nrow*. If *value* is not specified, it defaults to 1. The REPEAT and SHAPE functions can also perform this operation, and they are more general.

Examples of the J function are as follows.

```
b=j(3);
```

```
             B             3 rows      3 cols    (numeric)

                             1           1           1
                             1           1           1
                             1           1           1
```

```
r=j(5,2,'xyz');

        R          5 rows      2 cols    (character, size 3)

                                    xyz xyz
                                    xyz xyz
                                    xyz xyz
                                    xyz xyz
                                    xyz xyz
```

JROOT Function

computes the first nonzero roots of a Bessel function of the first kind and the derivative of the Bessel function at each root

JROOT(ν, n**)**

The JROOT function returns an $n \times 2$ matrix with the calculated roots in the first column and the derivatives in the second column.

The inputs to the JROOT function are as follows:

ν	is a scalar denoting the order of the Bessel function, with $\nu > -1$.
n	is a positive integer denoting the number of roots.

The JROOT function returns a matrix in which the first column contains the first n roots of the Bessel function; these roots are the solutions to the equation

$$J_\nu(x_i) = 0, \quad i = 1, \ldots, n$$

The second column of this matrix contains the derivatives $J'_\nu(x_i)$ of the Bessel function at each of the roots x_i. The expression $J_\nu(x)$ is a solution to the differential equation

$$x^2 \frac{d^2 J_\nu}{dx^2} + x \frac{dJ_\nu}{dx} + (x^2 - \nu^2) J_\nu = 0$$

One of the expressions for such a function is given by the series

$$J_\nu(x) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu + k + 1)}$$

where $\Gamma(\cdot)$ is the gamma function. Refer to Abramowitz and Stegun (1973) for more details concerning the Bessel and gamma functions. The algorithm is a Newton method coupled with a reasonable initial guess. For large values of n or ν, the algorithm could fail due to machine limitations. In this case, JROOT returns a matrix

with zero rows and zero columns. The values that cause the algorithm to fail are machine dependent.

The following code provides an example:

```
proc iml;
  x = jroot(1,4);
  print x;
```

To obtain only the roots, you can use the following statement, which extracts the first column of the returned matrix:

```
  x = jroot(1,4)[,1];
```

KALCVF Call

CALL KALCVF(*pred, vpred, filt, vfilt, data, lead,* a, f, b, h, *var* <, *z0, vz0*>**);**

The KALCVF call computes the one-step prediction $\mathbf{z}_{t+1|t}$ and the filtered estimate $\mathbf{z}_{t|t}$, as well as their covariance matrices. The call uses forward recursions, and you can also use it to obtain k-step estimates.

The inputs to the KALCVF subroutine are as follows:

data is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.

lead is the number of steps to forecast after the end of the data.

a is an $N_z \times 1$ vector for a time-invariant input vector in the transition equation, or a $(T + \textit{lead})N_z \times 1$ vector containing input vectors in the transition equation.

f is an $N_z \times N_z$ matrix for a time-invariant transition matrix in the transition equation, or a $(T + \textit{lead})N_z \times N_z$ matrix containing transition matrices in the transition equation.

b is an $N_y \times 1$ vector for a time-invariant input vector in the measurement equation, or a $(T + \textit{lead})N_y \times 1$ vector containing input vectors in the measurement equation.

h is an $N_y \times N_z$ matrix for a time-invariant measurement matrix in the measurement equation, or a $(T + \textit{lead})N_y \times N_z$ matrix containing measurement matrices in the measurement equation.

var is an $(N_y + N_z) \times (N_y + N_z)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + \textit{lead})(N_y + N_z) \times (N_y + N_z)$ matrix containing variance matrices for the error in the transition equation and the error in the measurement equation, that is, $(\eta_t', \epsilon_t')'$.

$z0$	is an optional $1 \times N_z$ initial state vector $\mathbf{z}'_{1\|0}$.
$vz0$	is an optional $N_z \times N_z$ covariance matrix of an initial state vector $\mathbf{P}_{1\|0}$.

The KALCVF call returns the following values:

pred	is a $(T + \mathit{lead}) \times N_z$ matrix containing one-step predicted state vectors $(\mathbf{z}_{1\|0}, \cdots, \mathbf{z}_{T+1\|T}, \mathbf{z}_{T+2\|T}, \cdots, \mathbf{z}_{T+\mathit{lead}\|T})'$.
vpred	is a $(T+\mathit{lead})N_z \times N_z$ matrix containing mean square errors of predicted state vectors $(\mathbf{P}_{1\|0}, \cdots, \mathbf{P}_{T+1\|T}, \mathbf{P}_{T+2\|T}, \cdots, \mathbf{P}_{T+\mathit{lead}\|T})'$.
filt	is a $T \times N_z$ matrix containing filtered state vectors $(\mathbf{z}_{1\|1}, \cdots, \mathbf{z}_{T\|T})'$.
vfilt	is a $TN_z \times N_z$ matrix containing mean square errors of filtered state vectors $(\mathbf{P}_{1\|1}, \cdots, \mathbf{P}_{T\|T})'$.

The KALCVF call computes the conditional expectation of the state vector $\mathbf{z}_t$ given the observations, assuming that the mean and the variance of the initial state vector are known. The filtered value is the conditional expectation of the state vector $\mathbf{z}_t$ given the observations up to time t. For k-step forecasting where $k > 0$, the conditional expectation at time $t+k$ is computed given observations up to t. For notation, $\mathbf{V}_t$ and $\mathbf{R}_t$ are variances of η_t and ϵ_t, respectively, and $\mathbf{G}_t$ is a covariance of η_t and ϵ_t. $\mathbf{A}^-$ stands for the generalized inverse of $\mathbf{A}$. The filtered value and its covariance matrix are denoted $\mathbf{z}_{t|t}$ and $\mathbf{P}_{t|t}$, respectively. For $k > 0$, $\mathbf{z}_{t+k|t}$ and $\mathbf{P}_{t+k|t}$ stand for the k-step forecast of $\mathbf{z}_{t+k}$ and its mean square error. The Kalman filtering algorithm for one-step prediction and filtering is given as follows:

$$
\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t\mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{P}_{t|t-1}\mathbf{H}'_t + \mathbf{R}_t \\
\mathbf{z}_{t|t} &= \mathbf{z}_{t|t-1} + \mathbf{P}_{t|t-1}\mathbf{H}'_t\mathbf{D}_t^-\hat{\epsilon}_t \\
\mathbf{P}_{t|t} &= \mathbf{P}_{t|t-1} - \mathbf{P}_{t|t-1}\mathbf{H}'_t\mathbf{D}_t^-\mathbf{H}_t\mathbf{P}_{t|t-1} \\
\mathbf{K}_t &= (\mathbf{F}_t\mathbf{P}_{t|t-1}\mathbf{H}'_t + \mathbf{G}_t)\mathbf{D}_t^- \\
\mathbf{z}_{t+1|t} &= \mathbf{a}_t + \mathbf{F}_t\mathbf{z}_{t|t-1} + \mathbf{K}_t\hat{\epsilon}_t \\
\mathbf{P}_{t+1|t} &= \mathbf{F}_t\mathbf{P}_{t|t-1}\mathbf{F}'_t + \mathbf{V}_t - \mathbf{K}_t\mathbf{D}_t\mathbf{K}'_t
\end{aligned}
$$

And for k-step forecasting for $k > 1$,

$$
\begin{aligned}
\mathbf{z}_{t+k|t} &= \mathbf{a}_{t+k-1} + \mathbf{F}_{t+k-1}\mathbf{z}_{t+k-1|t} \\
\mathbf{P}_{t+k|t} &= \mathbf{F}_{t+k-1}\mathbf{P}_{t+k-1|t}\mathbf{F}'_{t+k-1} + \mathbf{V}_{t+k-1}
\end{aligned}
$$

When you use the alternative transition equation

$$\mathbf{z}_t = \mathbf{a}_t + \mathbf{F}_t\mathbf{z}_{t-1} + \eta_t$$

the forward recursion algorithm is written

$$\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t\mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{P}_{t|t-1}\mathbf{H}_t' + \mathbf{H}_t\mathbf{G}_t + \mathbf{G}_t'\mathbf{H}_t' + \mathbf{R}_t \\
\mathbf{z}_{t|t} &= \mathbf{z}_{t|t-1} + (\mathbf{P}_{t|t-1}\mathbf{H}_t' + \mathbf{G}_t)\mathbf{D}_t^-\hat{\epsilon}_t \\
\mathbf{P}_{t|t} &= \mathbf{P}_{t|t-1} - (\mathbf{P}_{t|t-1}\mathbf{H}_t' + \mathbf{G}_t)\mathbf{D}_t^-(\mathbf{H}_t\mathbf{P}_{t|t-1} + \mathbf{G}_t') \\
\mathbf{K}_t &= (\mathbf{F}_{t+1}\mathbf{P}_{t|t-1}\mathbf{H}_t' + \mathbf{G}_t)\mathbf{D}_t^- \\
\mathbf{z}_{t+1|t} &= \mathbf{a}_{t+1} + \mathbf{F}_{t+1}\mathbf{z}_{t|t-1} + \mathbf{K}_t\hat{\epsilon}_t \\
\mathbf{P}_{t+1|t} &= \mathbf{F}_{t+1}\mathbf{P}_{t|t-1}\mathbf{F}_{t+1}' + \mathbf{V}_{t+1} - \mathbf{K}_t\mathbf{D}_t\mathbf{K}_t'
\end{aligned}$$

And for k-step forecasting $(k > 1)$,

$$\begin{aligned}
\mathbf{z}_{t+k|t} &= \mathbf{a}_{t+k} + \mathbf{F}_{t+k}\mathbf{z}_{t+k-1|t} \\
\mathbf{P}_{t+k|t} &= \mathbf{F}_{t+k}\mathbf{P}_{t+k-1|t}\mathbf{F}_{t+k}' + \mathbf{V}_{t+k}
\end{aligned}$$

You can use the KALCVF call when you specify the alternative transition equation and $\mathbf{G}_t = \mathbf{0}$.

The initial state vector and its covariance matrix of the time invariant Kalman filters are computed under the stationarity condition

$$\begin{aligned}
\mathbf{z}_{1|0} &= (\mathbf{I} - \mathbf{F})^-\mathbf{a} \\
\mathbf{P}_{1|0} &= (\mathbf{I} - \mathbf{F} \otimes \mathbf{F})^-\text{vec}(\mathbf{V})
\end{aligned}$$

where $\mathbf{F}$ and $\mathbf{V}$ are the time invariant transition matrix and the covariance matrix of transition equation noise, and vec$(\mathbf{V})$ is an $N_z^2 \times 1$ column vector that is constructed by the stacking N_z columns of matrix $\mathbf{V}$. Note that all eigenvalues of the matrix $\mathbf{F}$ are inside the unit circle when the SSM is stationary. When the preceding formula cannot be applied, the initial state vector estimate $\mathbf{z}_{1|0}$ is set to $\mathbf{a}_1$ and its covariance matrix $\mathbf{P}_{1|0}$ is given by $10^6\mathrm{I}$. Optionally, you can specify initial values.

The KALCVF call allows missing values in observations. If there is a missing observation, the filtered state vector for the missing observation is given by the one-step forecast.

KALCVS Call

CALL KALCVS(*sm, vsm, data, a, f, b, h, var, pred, vpred <,un, vun>***);**

The KALCVS call uses backward recursions to compute the smoothed estimate $\mathbf{z}_{t|T}$ and its covariance matrix, $\mathbf{P}_{t|T}$, where T is the number of observations in the complete data set.

The inputs to the KALCVS subroutine are as follows.

data is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.

a is an $N_z \times 1$ vector for a time-invariant input vector in the transition equation, or a $TN_z \times 1$ vector containing input vectors in the transition equation.

f is an $N_z \times N_z$ matrix for a time-invariant transition matrix in the transition equation, or a $TN_z \times N_z$ matrix containing T transition matrices.

b is an $N_y \times 1$ vector for a time-invariant input vector in the measurement equation, or a $TN_y \times 1$ vector containing input vectors in the measurement equation.

h is an $N_y \times N_z$ matrix for a time-invariant measurement matrix in the measurement equation, or a $TN_y \times N_z$ matrix containing T time variant $\mathbf{H}_t$ matrices in the measurement equation.

var is an $(N_y + N_z) \times (N_y + N_z)$ covariance matrix for the errors in the transition and the measurement equations, or a $T(N_y + N_z) \times (N_y + N_z)$ matrix containing covariance matrices in the transition equation and measurement equation noises, that is, $(\eta_t', \epsilon_t')'$.

pred is a $T \times N_z$ matrix containing one-step forecasts $(\mathbf{z}_{1|0}, \cdots, \mathbf{z}_{T|T-1})'$.

vpred is a $TN_z \times N_z$ matrix containing mean square error matrices of predicted state vectors $(\mathbf{P}_{1|0}, \cdots, \mathbf{P}_{T|T-1})'$.

un is an optional $1 \times N_z$ vector containing $\mathbf{u}_T$. The returned value is $\mathbf{u}_0$.

vun is an optional $N_z \times N_z$ matrix containing $\mathbf{U}_T$. The returned value is $\mathbf{U}_0$.

The KALCVS call returns the following values:

sm is a $T \times N_z$ matrix containing smoothed state vectors $(\mathbf{z}_{1|T}, \cdots, \mathbf{z}_{T|T})'$.

vsm is a $TN_z \times N_z$ matrix containing covariance matrices of smoothed state vectors $(\mathbf{P}_{1|T}, \cdots, \mathbf{P}_{T|T})'$.

When the Kalman filtering is performed in the KALCVF call, the KALCVS call computes smoothed state vectors and their covariance matrices. The fixed-interval smoothing state vector at time t is obtained by the conditional expectation given all observations.

The smoothing algorithm uses one-step forecasts and their covariance matrices, which are given in the KALCVF call. For notation, $\mathbf{z}_{t|T}$ is the smoothed value of the state vector $\mathbf{z}_t$, and the mean square error matrix is denoted $\mathbf{P}_{t|T}$. For smoothing,

$$\begin{aligned} \hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t \mathbf{z}_{t|t-1} \\ \mathbf{D}_t &= \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}_t' + \mathbf{R}_t \\ \mathbf{K}_t &= (\mathbf{F}_t \mathbf{P}_{t|t-1} \mathbf{H}_t' + \mathbf{G}_t) \mathbf{D}_t^- \end{aligned}$$

$$
\begin{aligned}
\mathbf{L}_t &= \mathbf{F}_t - \mathbf{K}_t\mathbf{H}_t \\
\mathbf{u}_{t-1} &= \mathbf{H}_t'\mathbf{D}_t^-\hat{\epsilon}_t + \mathbf{L}_t'\mathbf{u}_t \\
\mathbf{U}_{t-1} &= \mathbf{H}_t'\mathbf{D}_t^-\mathbf{H}_t + \mathbf{L}_t'\mathbf{U}_t\mathbf{L}_t \\
\mathbf{z}_{t|T} &= \mathbf{z}_{t|t-1} + \mathbf{P}_{t|t-1}\mathbf{u}_{t-1} \\
\mathbf{P}_{t|T} &= \mathbf{P}_{t|t-1} - \mathbf{P}_{t|t-1}\mathbf{U}_{t-1}\mathbf{P}_{t|t-1}
\end{aligned}
$$

where $t = T, T-1, \ldots, 1$. The initial values are $\mathbf{u}_T = \mathbf{0}$ and $\mathbf{U}_T = \mathbf{0}$.

When the SSM is specified using the alternative transition equation

$$\mathbf{z}_t = \mathbf{a}_t + \mathbf{F}_t\mathbf{z}_{t-1} + \eta_t$$

the fixed-interval smoothing is performed using the following backward recursions:

$$
\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t\mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{P}_{t|t-1}\mathbf{H}_t' + \mathbf{R}_t \\
\mathbf{K}_t &= \mathbf{F}_{t+1}\mathbf{P}_{t|t-1}\mathbf{H}_t'\mathbf{D}_t^- \\
\mathbf{L}_t &= \mathbf{F}_{t+1} - \mathbf{K}_t\mathbf{H}_t \\
\mathbf{u}_{t-1} &= \mathbf{H}_t'\mathbf{D}_t^-\hat{\epsilon}_t + \mathbf{L}_t'\mathbf{u}_t \\
\mathbf{U}_{t-1} &= \mathbf{H}_t'\mathbf{D}_t^-\mathbf{H}_t + \mathbf{L}_t'\mathbf{U}_t\mathbf{L}_t \\
\mathbf{z}_{t|T} &= \mathbf{z}_{t|t-1} + \mathbf{P}_{t|t-1}\mathbf{u}_{t-1} \\
\mathbf{P}_{t|T} &= \mathbf{P}_{t|t-1} - \mathbf{P}_{t|t-1}\mathbf{U}_{t-1}\mathbf{P}_{t|t-1}
\end{aligned}
$$

where it is assumed that $\mathbf{G}_t = \mathbf{0}$.

You can use the KALCVS call regardless of the specification of the transition equation when $\mathbf{G}_t = \mathbf{0}$. Harvey (1989) gives the following fixed-interval smoothing formula, which produces the same smoothed value:

$$
\begin{aligned}
\mathbf{z}_{t|T} &= \mathbf{z}_{t|t} + \mathbf{P}_t^*(\mathbf{z}_{t+1|T} - \mathbf{z}_{t+1|t}) \\
\mathbf{P}_{t|T} &= \mathbf{P}_{t|t} + \mathbf{P}_t^*(\mathbf{P}_{t+1|T} - \mathbf{P}_{t+1|t})\mathbf{P}_t^{*\prime}
\end{aligned}
$$

where

$$\mathbf{P}_t^* = \mathbf{P}_{t|t}\mathbf{F}_t'\mathbf{P}_{t+1|t}^-$$

under the shifted transition equation, but

$$\mathbf{P}_t^* = \mathbf{P}_{t|t}\mathbf{F}_{t+1}'\mathbf{P}_{t+1|t}^-$$

under the alternative transition equation.

The KALCVS call is accompanied by the KALCVF call, as shown in the following code. Note that you do not need to specify UN and VUN.

```
call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var);
call kalcvs(sm,vsm,y,a,f,b,h,var,pred,vpred);
```

You can also compute the smoothed estimate and its covariance matrix on an observation-by-observation basis. When the SSM is time invariant, the following example performs smoothing. In this situation, you should initialize UN and VUN as matrices of value 0.

```
call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var);
n = nrow(y);
nz = ncol(f);
un = j(1,nz,0);
vun = j(nz,nz,0);
do i = 1 to n;
  y_i = y[n-i+1,];
  pred_i  = pred[n-i+1,];
  vpred_i = vpred[(n-i)*nz+1:(n-i+1)*nz,];
  call kalcvs(sm_i,vsm_i,y_i,a,f,b,h,var,pred_i,vpred_i,un,vun);
  sm  = sm_i // sm;
  vsm = vsm_i // vsm;
end;
```

KALDFF Call

CALL KALDFF(*pred, vpred, initial, s2, data, lead, int, coef, var, intd, coefd* $<$*, n0, at, mt, qt*$>$**);**

The KALDFF call computes the one-step forecast of state vectors in an SSM using the diffuse Kalman filter. The call estimates the conditional expectation of $\mathbf{z}_t$, and it also estimates the initial random vector, δ, and its covariance matrix.

The inputs to the KALDFF subroutine are as follows:

data is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.

lead is the number of steps to forecast after the end of the data set.

int is an $(N_y + N_z) \times N_\beta$ matrix for a time-invariant fixed matrix, or a $(T + \textit{lead})(N_y + N_z) \times N_\beta$ matrix containing fixed matrices for the time-variant model in the transition equation and the measurement equation, that is, $(\mathbf{W}_t', \mathbf{X}_t')'$.

coef is an $(N_y + N_z) \times N_z$ matrix for a time-invariant coefficient, or a $(T + \textit{lead})(N_y + N_z) \times N_z$ matrix containing coefficients at each time in the transition equation and the measurement equation, that is, $(\mathbf{F}_t', \mathbf{H}_t')'$.

var is an $(N_y + N_z) \times (N_y + N_z)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + \textit{lead})(N_y + N_z) \times (N_y + N_z)$

matrix containing covariance matrices for the error in the transition equation and the error in the measurement equation, that is, $(\eta_t', \epsilon_t')'$.

intd is an $(N_z + N_\beta) \times 1$ vector containing the intercept term in the equation for the initial state vector $\mathbf{z}_0$ and the mean effect β, that is, $(\mathbf{a}', \mathbf{b}')'$.

coefd is an $(N_z + N_\beta) \times N_\delta$ matrix containing coefficients for the initial state δ in the equation for the initial state vector $\mathbf{z}_0$ and the mean effect β, that is, $(\mathbf{A}', \mathbf{B}')'$.

$n0$ is an optional scalar including an initial denominator. If $n0 > 0$, the denominator for $\hat{\sigma}_t^2$ is $n0$ plus the number n_t of elements of $(\mathbf{y}_1, \cdots, \mathbf{y}_t)'$. If $n0 \leq 0$ or $n0$ is not specified, the denominator for $\hat{\sigma}_t^2$ is n_t. With $n0 \geq 0$, the initial values, $\mathbf{A}_1, \mathbf{M}_1$, and $\mathbf{Q}_1$, are assumed to be known and, hence, at, mt, and qt are used for input containing the initial values. If the value of $n0$ is negative or $n0$ is not specified, the initial values for at, mt, and qt are computed. The value of $n0$ is updated as $\max(n0, 0) + n_t$ after the KALDFF call.

at is an optional $kN_z \times (N_\delta + 1)$ matrix. If $n0 \geq 0$, at contains $(\mathbf{A}_1', \cdots, \mathbf{A}_k')'$. However, only the first matrix $\mathbf{A}_1$ is used as input. When you specify the KALDFF call, at returns $(\mathbf{A}'_{T-k+\mathit{lead}+1}, \cdots, \mathbf{A}'_{T+\mathit{lead}})'$. If $n0$ is negative or the matrix $\mathbf{A}_1$ contains missing values, $\mathbf{A}_1$ is automatically computed.

mt is an optional $kN_z \times N_z$ matrix. If $n0 \geq 0$, mt contains $(\mathbf{M}_1, \cdots, \mathbf{M}_k)'$. However, only the first matrix $\mathbf{M}_1$ is used as input. If $n0$ is negative or the matrix $\mathbf{M}_1$ contains missing values, mt is used for output, and it contains $(\mathbf{M}_{T-k+\mathit{lead}+1}, \cdots, \mathbf{M}_{T+\mathit{lead}})'$. Note that the matrix $\mathbf{M}_1$ can be used as an input matrix if either of the off-diagonal elements is not missing. The missing element $\mathbf{M}_1(i, j)$ is replaced by the nonmissing element $\mathbf{M}_1(j, i)$.

qt is an optional $k(N_\delta + 1) \times (N_\delta + 1)$ matrix. If $n0 \geq 0$, qt contains $(\mathbf{Q}_1, \cdots, \mathbf{Q}_k)'$. However, only the first matrix $\mathbf{Q}_1$ is used as input. If $n0$ is negative or the matrix $\mathbf{Q}_1$ contains missing values, qt is used for output and contains $(\mathbf{Q}_{T-k+\mathit{lead}+1}, \cdots, \mathbf{Q}_{T+\mathit{lead}})'$. The matrix $\mathbf{Q}_1$ can also be used as an input matrix if either of the off-diagonal elements is not missing since the missing element $\mathbf{Q}_1(i, j)$ is replaced by the nonmissing element $\mathbf{Q}_1(j, i)$.

The KALCVF call returns the following values:

pred is a $(T + \mathit{lead}) \times N_z$ matrix containing estimated predicted state vectors $(\hat{\mathbf{z}}_{1|0}, \cdots, \hat{\mathbf{z}}_{T+1|T}, \hat{\mathbf{z}}_{T+2|T}, \cdots, \hat{\mathbf{z}}_{T+\mathit{lead}|T})'$.

vpred is a $(T+\mathit{lead})N_z \times N_z$ matrix containing estimated mean square errors of predicted state vectors $(\mathbf{P}_{1|0}, \cdots, \mathbf{P}_{T+1|T}, \mathbf{P}_{T+2|T}, \cdots, \mathbf{P}_{T+\mathit{lead}|T})'$.

initial is an $N_d \times (N_d + 1)$ matrix containing an estimate and its variance for initial state δ, that is, $(\hat{\delta}_T, \hat{\Sigma}_\delta, T)$.

$s2$ is a scalar containing the estimated variance $\hat{\sigma}_T^2$.

The KALDFF call computes the one-step forecast of state vectors in an SSM using the diffuse Kalman filter. The SSM for the diffuse Kalman filter is written

$$\begin{aligned}
\mathbf{y}_t &= \mathbf{X}_t\beta + \mathbf{H}_t\mathbf{z}_t + \epsilon_t \\
\mathbf{z}_{t+1} &= \mathbf{W}_t\beta + \mathbf{F}_t\mathbf{z}_t + \eta_t \\
\mathbf{z}_0 &= \mathbf{a} + \mathbf{A}\delta \\
\beta &= \mathbf{b} + \mathbf{B}\delta
\end{aligned}$$

where $\mathbf{z}_t$ is an $N_z \times 1$ state vector, $\mathbf{y}_t$ is an $N_y \times 1$ observed vector, and

$$\begin{aligned}
\begin{bmatrix} \eta_t \\ \epsilon_t \end{bmatrix} &\sim N\left(\mathbf{0}, \sigma^2 \begin{bmatrix} \mathbf{V}_t & \mathbf{G}_t \\ \mathbf{G}_t' & \mathbf{R}_t \end{bmatrix}\right) \\
\delta &\sim N(\mu, \sigma^2\Sigma)
\end{aligned}$$

It is assumed that the noise vector $(\eta_t', \epsilon_t')'$ is independent and δ is independent of the vector $(\eta_t', \epsilon_t')'$. The matrices, $\mathbf{W}_t$, $\mathbf{F}_t$, $\mathbf{X}_t$, $\mathbf{H}_t$, $\mathbf{a}$, $\mathbf{A}$, $\mathbf{b}$, $\mathbf{B}$, $\mathbf{V}_t$, $\mathbf{G}_t$, and $\mathbf{R}_t$, are assumed to be known. The KALDFF call estimates the conditional expectation of the state vector $\mathbf{z}_t$ given the observations. The KALDFF subroutine also produces the estimates of the initial random vector δ and its covariance matrix. For k-step forecasting where $k > 0$, the estimated conditional expectation at time $t+k$ is computed with observations given up to time t. The estimated k-step forecast and its estimated MSE are denoted $\mathbf{z}_{t+k|t}$ and $\mathbf{P}_{t+k|t}$ (for $k > 0$). $\mathbf{A}_{t+k(\delta)}$ and $\mathbf{E}_{t(\delta)}$ are last-column-deleted submatrices of $\mathbf{A}_{t+k}$ and $\mathbf{E}_t$, respectively. The algorithm for one-step prediction is given as follows:

$$\begin{aligned}
\mathbf{E}_t &= (\mathbf{X}_t\mathbf{B},\ \mathbf{y}_t - \mathbf{X}_t\mathbf{b}) - \mathbf{H}_t\mathbf{A}_t \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{R}_t \\
\mathbf{Q}_{t+1} &= \mathbf{Q}_t + \mathbf{E}_t'\mathbf{D}_t^-\mathbf{E}_t \\
&= \begin{bmatrix} \mathbf{S}_t & \backslash_t \\ \backslash_t' & q_t \end{bmatrix} \\
\hat{\sigma}_t^2 &= (q_t - \backslash_t'\mathbf{S}_t^-\backslash_t)/n_t \\
\hat{\delta}_t &= \mathbf{S}_t^-\backslash_t \\
\hat{\Sigma}_{\delta,t} &= \hat{\sigma}_t^2\mathbf{S}_t^- \\
\mathbf{K}_t &= (\mathbf{F}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{G}_t)\mathbf{D}_t^-
\end{aligned}$$

$$
\begin{aligned}
\mathbf{A}_{t+1} &= \mathbf{W}_t(-\mathbf{B}, \mathbf{b}) + \mathbf{F}_t\mathbf{A}_t + \mathbf{K}_t\mathbf{E}_t \\
\mathbf{M}_{t+1} &= (\mathbf{F}_t - \mathbf{K}_t\mathbf{H}_t)\mathbf{M}_t\mathbf{F}_t' + \mathbf{V}_t - \mathbf{K}_t\mathbf{G}_t' \\
\mathbf{z}_{t+1|t} &= \mathbf{A}_{t+1}(-\hat{\delta}_t', 1)' \\
\mathbf{P}_{t+1|t} &= \hat{\sigma}_t^2\mathbf{M}_{t+1} + \mathbf{A}_{t+1(\delta)}\hat{\Sigma}_{\delta,t}\mathbf{A}_{t+1(\delta)}'
\end{aligned}
$$

where n_t is the number of elements of $(\mathbf{y}_1, \cdots, \mathbf{y}_t)'$ plus $\max(n0, 0)$. Unless initial values are given and $n0 \geq 0$, initial values are set as follows:

$$
\begin{aligned}
\mathbf{A}_1 &= \mathbf{W}_1(-\mathbf{B}, \mathbf{b}) + \mathbf{F}_1(-\mathbf{A}, \mathbf{a}) \\
\mathbf{M}_1 &= \mathbf{V}_1 \\
\mathbf{Q}_1 &= \mathbf{0}
\end{aligned}
$$

For k-step forecasting where $k > 1$,

$$
\begin{aligned}
\mathbf{A}_{t+k} &= \mathbf{W}_{t+k-1}(-\mathbf{B}, \mathbf{b}) + \mathbf{F}_{t+k-1}\mathbf{A}_{t+k-1} \\
\mathbf{M}_{t+k} &= \mathbf{F}_{t+k-1}\mathbf{M}_{t+k-1}\mathbf{F}_{t+k-1}' + \mathbf{V}_{t+k-1} \\
\mathbf{D}_{t+k} &= \mathbf{H}_{t+k}\mathbf{M}_{t+k}\mathbf{H}_{t+k}' + \mathbf{R}_{t+k} \\
\mathbf{z}_{t+k|t} &= \mathbf{A}_{t+k}(-\hat{\delta}_t', 1)' \\
\mathbf{P}_{t+k|t} &= \hat{\sigma}_t^2\mathbf{M}_{t+k} + \mathbf{A}_{t+k(\delta)}\hat{\Sigma}_{\delta,t}\mathbf{A}_{t+k(\delta)}'
\end{aligned}
$$

Note that if there is a missing observation, the KALDFF call computes the one-step forecast for the observation following the missing observation as the two-step forecast from the previous observation.

KALDFS Call

CALL KALDFS(*sm, vsm, data, int, coef, var, bvec, bmat, initial, at, mt, s2 <, un, vun>***);**

KALDFS computes the smoothed state vector and its mean square error matrix from the one-step forecast and mean square error matrix computed by KALDFF.

The inputs to the KALDFS subroutine are as follows:

data is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.

int is an $(N_y + N_z) \times N_\beta$ vector for a time-invariant intercept, or a $(T + \textit{lead})(N_y + N_z) \times N_\beta$ vector containing fixed matrices for the time-variant model in the transition equation and the measurement equation, that is, $(\mathbf{W}_t', \mathbf{X}_t')'$.

coef is an $(N_y + N_z) \times N_z$ matrix for a time-invariant coefficient, or a $(T + \textit{lead})(N_y + N_z) \times N_z$ matrix containing coefficients at each time in the transition equation and the measurement equation, that is, $(\mathbf{F}_t', \mathbf{H}_t')'$.

var is an $(N_y + N_z) \times (N_y + N_z)$ matrix for a time-invariant variance matrix for transition equation noise and the measurement equation noise, or a $(T + \textit{lead})(N_y + N_z) \times (N_y + N_z)$ matrix containing covariance matrices for the transition equation and measurement equation errors, that is, $(\eta_t', \epsilon_t')'$.

bvec is an $N_\beta \times 1$ constant vector for the intercept for the mean effect β.

bmat is an $N_\beta \times N_\delta$ matrix for the coefficient for the mean effect β.

initial is an $N_\delta \times (N_\delta + 1)$ matrix containing an initial random vector estimate and its covariance matrix, that is, $(\hat{\delta}_T, \hat{\Sigma}_{\delta,T})$.

at is a $TN_z \times (N_\delta + 1)$ matrix containing $(\mathbf{A}_1', \cdots, \mathbf{A}_T')'$.

mt is a $(TN_z) \times N_z$ matrix containing $(\mathbf{M}_1, \cdots, \mathbf{M}_T)'$.

s2 is the estimated variance in the end of the data set, $\hat{\sigma}_T^2$.

un is an optional $N_z \times (N_\delta + 1)$ matrix containing $\mathbf{u}_T$. The returned value is $\mathbf{u}_0$.

vun is an optional $N_z \times N_z$ matrix containing $\mathbf{U}_T$. The returned value is $\mathbf{U}_0$.

The KALDFS call returns the following values:

sm is a $T \times N_z$ matrix containing smoothed state vectors $(\mathbf{z}_{1|T}, \cdots, \mathbf{z}_{T|T})'$.

vsm is a $TN_z \times N_z$ matrix containing mean square error matrices of smoothed state vectors $(\mathbf{P}_{1|T}, \cdots, \mathbf{P}_{T|T})'$.

Given the one-step forecast and mean square error matrix in the KALDFF call, the KALDFS call computes a smoothed state vector and its mean square error matrix. Then the KALDFS subroutine produces an estimate of the smoothed state vector at time t, that is, the conditional expectation of the state vector $\mathbf{z}_t$ given all observations. Using the notations and results from the KALDFF section, the backward recursion algorithm for smoothing is denoted for $t = T, T-1, \ldots, 1$,

$$\begin{aligned}
\mathbf{E}_t &= (\mathbf{X}_t\mathbf{B},\ \mathbf{y}_t - \mathbf{X}_t\mathbf{b}) - \mathbf{H}_t\mathbf{A}_t \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{R}_t \\
\mathbf{L}_t &= \mathbf{F}_t - (\mathbf{F}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{G}_t)\mathbf{D}_t^{-}\mathbf{H}_t
\end{aligned}$$

$$
\begin{aligned}
\mathbf{u}_{t-1} &= \mathbf{H}_t'\mathbf{D}_t^{-}\mathbf{E}_t + \mathbf{L}_t'\mathbf{u}_t \\
\mathbf{U}_{t-1} &= \mathbf{H}_t'\mathbf{D}_t^{-}\mathbf{H}_t + \mathbf{L}_t'\mathbf{U}_t\mathbf{L}_t \\
\mathbf{z}_{t|T} &= (\mathbf{A}_t + \mathbf{M}_t\mathbf{u}_{t-1})(-\hat{\delta}_T', 1)' \\
\mathbf{C}_t &= \mathbf{A}_t + \mathbf{M}_t\mathbf{u}_{t-1} \\
\mathbf{P}_{t|T} &= \hat{\sigma}_T^2(\mathbf{M}_t - \mathbf{M}_t\mathbf{R}_{t-1}\mathbf{M}_t) + \mathbf{C}_{t(\delta)}\hat{\Sigma}_{\delta,T}\mathbf{C}_{t(\delta)}'
\end{aligned}
$$

where the initial values are $\mathbf{u}_T = \mathbf{0}$ and $\mathbf{U}_T = \mathbf{0}$, and $\mathbf{C}_{t(\delta)}$ is the last-column-deleted submatrix of $\mathbf{C}_t$. Refer to De Jong (1991b) for details on smoothing in the diffuse Kalman filter.

The KALDFS call is accompanied by the KALDFF call as shown in the following code:

```
ny = ncol(y);
nz = ncol(coef);
nb = ncol(int);
nd = ncol(coefd);
at = j(nz,nd+1,.);
mt = j(nz,nz,.);
qt = j(nd+1,nd+1,.);
n0 = -1;
call kaldff(pred,vpred,initial,s2,y,0,int,coef,var,intd,coefd,
            n0,at,mt,qt);
bvec = intd[nz+1:nz+nb,];
bmat = coefd[nz+1:nz+nb,];
call kaldfs(sm,vsm,x,int,coef,var,bvec,bmat,initial,at,mt,s2);
```

You can also compute the smoothed estimate and its covariance matrix observation by observation. When the SSM is time invariant, the following code performs smoothing. You should initialize UN and VUN as matrices of value **0**.

```
n  = nrow(y);
ny = ncol(y);
nz = ncol(coef);
nb = ncol(int);
nd = ncol(coefd);
at = j(nz,nd+1,.);
mt = j(nz,nz,.);
qt = j(nd+1,nd+1,.);
n0 = -1;
call kaldff(pred,vpred,initial,s2,y,0,int,coef,var,intd,coefd,
            n0,at,mt,qt);
bvec = intd[nz+1:nz+nb,];
bmat = coefd[nz+1:nz+nb,];
un  = j(nz,nd+1,0);
vun = j(nz,nz,0);
do i = 1 to n;
   call kaldfs(sm_i,vsm_i,y[n-i+1],int,coef,var,bvec,bmat,
               initial,at,mt,s2,un,vun);
   sm  = sm_i // sm;
```

```
      vsm = vsm_i // vsm;
   end;
```

LAV Call

performs linear least absolute value regression by solving the L_1 norm minimization problem

CALL LAV(*rc, xr, a, b* <, *<x0>*<, *opt*>>**);**

The LAV subroutine returns the following values:

rc is a scalar return code indicating the reason for optimization termination.

rc	**Termination**
0	Successful
1	Successful, but approximate covariance matrix and standard errors cannot be computed
-1 or -3	Unsuccessful: error in the input arguments
-2	Unsuccessful: matrix $\mathbf{A}$ is rank deficient ($\mathrm{rank}(\mathbf{A}) < n$)
-4	Unsuccessful: maximum iteration limit exceeded
-5	Unsuccessful: no solution found for ill-conditioned problem

xr specifies a vector or matrix with n columns. If the optimization process is not successfully completed, *xr* is a row vector with n missing values. If termination is successful and the *opt*[3] option is not set, *xr* is the vector with the optimal estimate, x^*. If termination is successful and the *opt*[3] option is specified, *xr* is an $(n+2)\times n$ matrix that contains the optimal estimate, x^*, in the first row, the asymptotic standard errors in the second row, and the $n \times n$ covariance matrix of parameter estimates in the remaining rows.

The inputs to the LAV subroutine are as follows:

a specifies an $m \times n$ matrix $\mathbf{A}$ with $m \geq n$ and full column rank, $\mathrm{rank}(\mathbf{A}) = n$. If you want to include an intercept in the model, you must include a column of ones in the matrix $\mathbf{A}$.

b specifies the $m \times 1$ vector $\mathbf{b}$.

$x0$ specifies an optional $n \times 1$ vector that specifies the starting point of the optimization.

opt is an optional vector used to specify options.

opt[1] specifies the maximum number *maxi* of outer iterations (this corresponds to the number of changes of the Huber parameter γ). The default is *maxi* $= \min(100, 10n)$. (The number of inner iterations is restricted by an internal threshold. If the number of inner iterations exceeds this threshold, a new outer iteration is started with an increased value of γ.)

opt[2] specifies the amount of printed output. Higher values request additional output and include the output of lower values.

opt[2]	**Termination**
0	no output is printed
1	error and warning messages are printed
2	the iteration history is printed (this is the default)
3	the n least-squares (L_2 norm) estimates are printed if no starting point is specified; the L_1 norm estimates are printed; if *opt*[3] is set, the estimates are printed together with the asymptotic standard errors
4	the $n \times n$ approximate covariance matrix of parameter estimates is printed if *opt*[3] is set
5	the residual and predicted values for all m rows (equations) of $\mathbf{A}$ are printed

opt[3] specifies which estimate of the variance of the median of nonzero residuals is to be used as a factor for the approximate covariance matrix of parameter estimates and for the approximate standard errors (ASE). If *opt*$[3] = 0$, the McKean-Schrader (1987) estimate is used, and if *opt*$[3] > 0$, the Cox-Hinkley (1974) estimate, with $v =$*opt*[3], is used. The default is *opt*$[3] = -1$ or *opt*$[3] =$., which means that the covariance matrix is not computed.

opt[4] specifies whether a computationally expensive test for necessary and sufficient optimality of the solution x is executed. The default is *opt*$[4] = 0$ or *opt*$[4] =$., which means that the convergence test is not performed.

Missing values are not permitted in the a or b argument. The $x0$ argument is ignored if it contains any missing values. Missing values in the *opt* argument cause the default value to be used.

The Least Absolute Values (LAV) subroutine is designed for solving the unconstrained linear L_1 norm minimization problem,

$$\min_{\mathbf{x}} L_1(\mathbf{x}) \qquad \text{where} \qquad L_1(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 = \sum_{i=1}^{m} \left| \sum_{j=1}^{n} a_{ij}x_j - b_i \right|$$

for m equations with n (unknown) parameters $\mathbf{x} = (x_1, \ldots, x_n)$. This is equivalent

to estimating the unknown parameter vector, $\mathbf{x}$, by least absolute value regression in the model

$$\mathbf{b} = \mathbf{A}\mathbf{x} + \boldsymbol{\epsilon}$$

where $\mathbf{b}$ is the vector of n observations, $\mathbf{A}$ is the design matrix, and $\boldsymbol{\epsilon}$ is a random error term.

An algorithm by Madsen and Nielsen (1993) is used, which can be faster for large values of m and n than the Barrodale and Roberts (1974) algorithm. The current version of the algorithm assumes that $\mathbf{A}$ has full column rank. Also, constraints cannot be imposed on the parameters in this version.

The L_1 norm minimization problem is more difficult to solve than the least-squares (L_2 norm) minimization problem because the objective function of the L_1 norm problem is not continuously differentiable (the first derivative has jumps). A function that is continuous but not continuously differentiable is called *nonsmooth*. Using PROC NLP and the IML nonlinear optimization subroutines, you can obtain the estimates in linear and nonlinear L_1 norm estimation (even subject to linear or nonlinear constraints) as long as the number of parameters, n, is small. Using the nonlinear optimization subroutines, there are two ways to solve the nonlinear L_p-norm, $p \geq 1$, problem:

- For small values of n, you can implement the Nelder-Mead simplex algorithm with the NLPNMS subroutine to solve the minimization problem in its original specification. The Nelder-Mead simplex algorithm does not assume a smooth objective function, does not take advantage of any derivatives, and therefore does not require continuous differentiability of the objective function. See the "NLPNMS Call" section on page 648 for details.
- Gonin and Money (1989) describe how an original L_p norm estimation problem can be modified to an equivalent optimization problem with nonlinear constraints which has a simple differentiable objective function. You can invoke the NLPQN subroutine, which implements a quasi-Newton algorithm, to solve the nonlinearly constrained L_p norm optimization problem. See the section "NLPQN Call" on page 658 for details on the NLPQN subroutine.

Both approaches are successful only for a small number of parameters and good initial estimates. If you cannot supply good initial estimates, the optimal results of the corresponding nonlinear least-squares (L_2 norm) estimation may provide fairly good initial estimates.

Gonin and Money (1989, pp. 44−45) show that the nonlinear L_1 norm estimation problem

$$\min_{\mathbf{x}} \sum_{i=1}^{m} |f_i(\mathbf{x})|$$

can be reformulated as a linear optimization problem with nonlinear constraints in the following ways.

- $$\min_{\mathbf{x}} \sum_{i=1}^{m} u_i \qquad \text{subject to} \quad \left.\begin{array}{lcl} f_i(\mathbf{x}) - u_i & \leq & 0 \\ f_i(\mathbf{x}) + u_i & \geq & 0 \\ u_i & \geq & 0 \end{array}\right\} \quad i = 1, \ldots, m$$

  is a linear optimization problem with $2m$ nonlinear inequality constraints in $m + n$ variables u_i and x_j.

- $$\min_{\mathbf{x}} \sum_{i=1}^{m} (y_i + z_i) \qquad \text{subject to} \quad \left.\begin{array}{lcl} f_i(\mathbf{x}) + y_i - z_i & = & 0 \\ y_i & \geq & 0 \\ z_i & \geq & 0 \end{array}\right\} \quad i = 1, \ldots, m$$

  is a linear optimization problem with $2m$ nonlinear equality constraints in $2m+n$ variables y_i, z_i, and x_j.

For linear functions $f_i(\mathbf{x}) = \sum_{j=1}^{n}(a_{ij}x_j - b_i)$, $i = 1, \ldots, m$, you obtain linearly constrained linear optimization problems, for which the number of variables and constraints is on the order of the number of observations, m. The advantage that the algorithm by Madsen and Nielsen (1993) has over the Barrodale and Roberts (1974) algorithm is that its computational cost increases only linearly with m, and it can be faster for large values of m.

In addition to computing an optimal solution $\mathbf{x}^*$ that minimizes $L_1(\mathbf{x})$, you can also compute approximate standard errors and the approximate covariance matrix of $\mathbf{x}^*$. The standard errors may be used to compute confidence limits.

The following example is the same one used for illustrating the LAV procedure by Lee and Gentle (1986). $\mathbf{A}$ and $\mathbf{b}$ are as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & 2 \\ 1 & 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ -1 \\ 2 \\ 4 \end{bmatrix}$$

The following code specifies the matrix A, the vector B, and the options vector OPT. The options vector specifies that all output is printed ($opt[2] = 5$), that the asymptotic standard errors and covariance matrix are computed based on the McKean-Schrader (1987) estimate λ of the variance of the median ($opt[3] = 0$), and that the convergence test should be performed ($opt[4] = 1$).

```
proc iml;
   a = { 0,  1, -1, -1,  2,  2 };
   m = nrow(a);
   a = j(m,1,1.) || a;
   b = { 1,  2,  1, -1,  2,  4 };

   opt= { . 5  0 1 };
   call lav(rc,xr,a,b,,opt);
```

The first part of the printed output refers to the least-squares solution, which is used as the starting point. The estimates of the largest and smallest nonzero eigenvalues of

$\mathbf{A}'\mathbf{A}$ give only an idea of the magnitude of these values, and they can be very crude approximations.

The second part of the printed output shows the iteration history.

The third part of the printed output shows the L_1 norm solution (first row) together with asymptotic standard errors (second row) and the asymptotic covariance matrix of parameter estimates (the ASEs are the square roots of the diagonal elements of this covariance matrix).

The last part of the printed output shows the predicted values and residuals, as in Lee and Gentle (1986).

LCP Call

solves the linear complementarity problem

CALL LCP(*rc*, w, z, m, q <, *epsilon*> **);**

The inputs to the LCP subroutine are as follows:

m	is an $m \times m$ matrix.
q	is an $m \times 1$ matrix.
epsilon	is a scalar defining virtual zero. The default value of *epsilon* is 1.0E−8.
rc	returns one of the following scalar return codes: 0 solution found 1 no solution possible 5 solution is numerically unstable 6 subroutine could not obtain enough memory.
w and z	return the solution in an m-element column vector.

The LCP subroutine solves the linear complementarity problem:

$$\begin{aligned} \mathbf{w} &= \mathbf{M}\mathbf{z} + \mathbf{q} \\ \mathbf{w}'\mathbf{z} &= 0 \\ \mathbf{w}, \mathbf{z} &\geq 0 \end{aligned}$$

Consider the following example:

```
q={1, 1};
m={1 0,
   0 1};
call lcp(rc,w,z,m,q);
```

The result is

```
RC              1 row       1 col     (numeric)

                            0


W               2 rows      1 col     (numeric)

                            1
                            1


Z               2 rows      1 col     (numeric)

                            0
                            0
```

The next example shows the relationship between quadratic programming and the linear complementarity problem. Consider the linearly constrained quadratic program:

$$\begin{array}{rcll} \min \mathbf{c}'\mathbf{x} & + & \frac{1}{2}\mathbf{x}'\mathbf{H}\mathbf{x} & \\ \text{st. } \mathbf{G}\mathbf{x} & \geq & \mathbf{b} & \text{(QP)} \\ \mathbf{x} & \geq & 0 & \end{array}$$

If $\mathbf{H}$ is positive semidefinite, then a solution to the Kuhn-Tucker conditions solves QP. The Kuhn-Tucker conditions for QP are

$$\begin{array}{rcl} \mathbf{c} + \mathbf{H}\mathbf{x} & = & \mu + \mathbf{G}'\lambda \\ \lambda'(\mathbf{G}\mathbf{x} - \mathbf{b}) & = & 0 \\ \mu'\mathbf{x} & = & 0 \\ \mathbf{G}\mathbf{x} & \geq & \mathbf{b} \\ \\ \mathbf{x}, \mu, \lambda & \geq & 0 \end{array}$$

In the linear complementarity problem, let

$$\begin{array}{rcl} \mathbf{M} & = & \left[\begin{array}{cc} H & -G' \\ G & 0 \end{array} \right] \\ \mathbf{w}' & = & (\mu'\mathbf{s}') \\ \mathbf{z}' & = & (\mathbf{x}'\lambda') \\ \mathbf{q}' & = & (\mathbf{c}' - \mathbf{b}) \end{array}$$

Then the Kuhn-Tucker conditions are expressed as finding $\mathbf{w}$ and $\mathbf{z}$ that satisfy

$$\begin{array}{rcl} \mathbf{w} & = & \mathbf{M}\mathbf{z} + \mathbf{q} \\ \mathbf{w}'\mathbf{z} & = & 0 \\ \mathbf{w}, \mathbf{z} & \geq & 0 \end{array}$$

From the solution $\mathbf{w}$ and $\mathbf{z}$ to this linear complementarity problem, the solution to QP is obtained; namely, $\mathbf{x}$ is the primal structural variable, $\mathbf{s} = \mathbf{G}\mathbf{x} - \mathbf{b}$ the surpluses, and μ and λ are the dual variables. Consider a quadratic program with the following data:

$$\mathbf{C}' = (1245) \quad \mathbf{B}' = (11)$$

$$\mathbf{H} = \begin{bmatrix} 100 & 10 & 1 & 0 \\ 10 & 100 & 10 & 1 \\ 1 & 10 & 100 & 10 \\ 0 & 1 & 10 & 100 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 10 & 20 & 30 & 40 \end{bmatrix}$$

This problem is solved using the LCP subroutine in PROC IML as follows:

```
   /*---- Data for the Quadratic Program -----*/
c={1,2,3,4};
h={100 10 1 0, 10 100 10 1, 1 10 100 10, 0 1 10 100};
g={1 2 3 4, 10 20 30 40};
b={1, 1};

   /*----- Express the Kuhn-Tucker Conditions as an LCP ----*/
m=h||-g^{\prime} ;
m=m//(g||j(nrow(g),nrow(g),0));
q=c//-b ;

   /*----- Solve for a Kuhn-Tucker Point --------*/
call lcp(rc,w,z,m,q);

   /*------ Extract the Solution to the Quadratic Program ----*/
x=z[1:nrow(h)];
print rc x;
```

The printed solution is

```
RC            1 row       1 col     (numeric)

                          0

X             4 rows      1 col     (numeric)

                  0.0307522
                  0.0619692
                  0.0929721
                  0.1415983
```

LENGTH Function

finds the lengths of character matrix elements

LENGTH(*matrix***)**

where *matrix* is a character matrix or quoted literal.

The LENGTH function takes a character matrix as an argument and produces a numeric matrix as a result. The result matrix has the same dimensions as the argument and contains the lengths of the corresponding string elements in *matrix*. The length of a string is equal to the position of the rightmost nonblank character in the string. If a string is entirely blank, its length value is set to 1. An example of the LENGTH function is shown below:

```
c={'Hello' 'My name is Jenny'};
b=length(c);

            B             1 row       2 cols    (numeric)

                                  5        16
```

See also the description of the NLENG function.

LINK and RETURN Statements

jump to another statement

LINK *label***;**
statements

label:statements

RETURN;

The LINK statement, like the GOTO statement, directs IML to jump to the statement with the specified label. Unlike the GOTO statement, however, IML remembers where the LINK statement was issued and returns to that point when a RETURN statement is executed. This statement can only be used inside modules and DO groups.

The LINK statement provides a way of calling sections of code as if they were subroutines. The LINK statement calls the routine. The routine begins with the label and ends with a RETURN statement. LINK statements can be nested within other LINK statements to any level. A RETURN statement without a LINK statement is executed the same as the STOP statement.

Any time you use a LINK statement, you may consider using a RUN statement and a module defined using the START and FINISH statements instead.

An example using the LINK statement is shown below:

```
start a;
   x=1;
   y=2;
   link sum1;
   print z;
   stop;
   sum1:
      z=x+y;
   return;
finish a;
run a;
```

```
             Z             1 row       1 col     (numeric)

                                       3
```

LIST Statement

displays observations of a data set

LIST <*range*> <**VAR** *operand*> <**WHERE(***expression***)**>**;**

The inputs to the LIST statement are as follows:

range	specifies a range of observations
operand	specifies a set of variables
expression	is an expression evaluated to be true or false.

The LIST statement prints selected observations of a data set. If all data values for variables in the VAR clause fit on a single line, values are displayed in columns headed by the variable names. Each record occupies a separate line. If the data values do not fit on a single line, values from each record are grouped into paragraphs. Each element in the paragraph has the form *name=value*.

You can specify a *range* of observations with a keyword or by record number using the POINT option. You can use any of the following keywords to specify a *range*:

ALL	all observations
CURRENT	the current observation (this is the default for the LIST statement)
NEXT <*number*>	the next observation or the next *number* of observations
AFTER	all observations after the current one

POINT *operand* observations specified by number, where *operand* can be one of the following:

Operand	Example
a single record number	`point 5`
a literal giving several record numbers	`point {2 5 10}`
the name of a matrix containing record numbers	`point p`
an expression in parentheses	`point (p+1)`

If the current data set has an index in use, the POINT option is invalid.

You can specify a set of variables to use with the VAR clause. The *operand* can be specified as one of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of the keywords described below:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

Below are examples showing each possible way you can use the VAR clause:

```
var {time1 time5 time9};  /* a literal giving the variables */
var time;                 /* a matrix containing the names  */
var('time1':'time9');     /* an expression                  */
var _all_;                /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the *range* specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable is a variable in the SAS data set.

comparison-op is any one of the following comparison operators:

< less than

<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables and the expression on the right-hand side refers to matrix values.

Below are several examples on using the LIST statement:

```
list all;                                /* lists whole data set */
list;                                 /* lists current observation */
list var{name addr};     /* lists NAME and ADDR in current obs */
list all where(age>30); /* lists all obs where condition holds */
list next;                              /* lists next observation */
list point 24;                          /* lists observation 24 */
list point (10:15);        /* lists observations 10 through 15 */
```

LMS and LTS Calls

performs robust regression

CALL LMS(*sc, coef, wgt, opt,* $y <$, $< x >$<, *sorb*>>**);**

robust (resistant) regression method, defined by minimizing the hth ordered squared residual.

CALL LTS(*sc, coef, wgt, opt,* $y <$, $< x >$<, *sorb*>>**);**

robust (resistant) regression method, defined by minimizing the sum of the h smallest squared residuals.

The Least Median of Squares (LMS) and Least Trimmed Squares (LTS) subroutines perform *robust* regression (sometimes called *resistant* regression). They are able to detect outliers and perform a least-squares regression on the remaining observations.

The value of h may be specified, but in most applications the default value works just fine and the results seem to be quite stable toward different choices of h.

The inputs to the LMS and LTS subroutines are as follows:

opt refers to an options vector with the following components (missing values are treated as default values). The options vector can be a null vector.

opt[1] specifies whether an intercept is used in the model (*opt*[1]=0) or not (*opt*[1]$\neq$ 0). If *opt*[1]=0, then a column of 1s is added as the last column to the input matrix $\mathbf{X}$; that is, you do not need to add this column of 1s yourself. The default is *opt*[1]=0.

opt[2] specifies the amount of printed output. Higher values request additional output and include the output of lower values.

opt[2]=0 prints no output except error messages.

opt[2]=1 prints all output except (1) arrays of $O(N)$, such as weights, residuals, and diagnostics; (2) the history of the optimization process; and (3) subsets that result in singular linear systems.

opt[2]=2 additionally prints arrays of $O(N)$, such as weights, residuals, and diagnostics; also prints the case numbers of the observations in the best subset and some basic history of the optimization process.

opt[2]=3 additionally prints subsets that result in singular linear systems.

The default is *opt*[2]=0.

opt[3] specifies whether only LMS or LTS is computed or whether, additionally, least-squares (LS) and weighted least-squares (WLS) regression are computed:

opt[3]=0 computes only LMS or LTS.

opt[3]=1 computes, in addition to LMS or LTS, weighted least-squares regression on the observations with *small* LMS or LTS residuals (where *small* is defined by opt[8]).

opt[3]=2 computes, in addition to LMS or LTS, unweighted least-squares regression.

opt[3]=3 adds both unweighted and weighted least-squares regression to LMS and LTS regression.

The default is *opt*[3]=0.

opt[4] specifies the quantile h to be minimized. This is used in the objective function. The default is *opt*[5]$= h = [\frac{N+n+1}{2}]$, which corresponds to the highest possible breakdown value. This is also the default of the PROGRESS program. The value of h should be in the range $\frac{N}{2}+1 \leq h \leq \frac{3N}{4}+\frac{n+1}{4}$

opt[5] specifies the number N_{Rep} of generated subsets. Each subset consists of n observations $(k_1, \ldots, k_n)$, where $1 \leq k_i \leq N$. The total number of subsets consisting of n observations out of N observations is

$$N_{\text{tot}} = \binom{N}{n} = \frac{\prod_{j=1}^{n}(N-j+1)}{\prod_{j=1}^{n} j}$$

where n is the number of parameters including the intercept.

Due to computer time restrictions, not all subset combinations of n observations out of N can be inspected for larger values of N and n. Specifying a value of $N_{\text{Rep}} < N_{\text{tot}}$ enables you to save computer time at the expense of computing a suboptimal solution.

If *opt*[5] is zero or missing, the default number of subsets is taken from the following table.

n	1	2	3	4	5	6	7	8	9	10
N_{lower}	500	50	22	17	15	14	0	0	0	0
N_{upper}	1000000	1414	182	71	43	32	27	24	23	22
N_{Rep}	500	1000	1500	2000	2500	3000	3000	3000	3000	3000

n	11	12	13	14	15
N_{lower}	0	0	0	0	0
N_{upper}	22	22	22	23	23
N_{Rep}	3000	3000	3000	3000	3000

If the number of cases (observations) N is smaller than N_{lower}, then all possible subsets are used; otherwise, N_{Rep} subsets are drawn

randomly. This means that an exhaustive search is performed for *opt*[6]=−1. If N is larger than N_{upper}, a note is printed in the log file indicating how many subsets exist.

opt[7] specifies that the latest argument *sorb* contains a given parameter vector $\mathbf{b}$ rather than a given subset for which the objective function should be evaluated.

opt[8] is relevant only for LS and WLS regression (*opt*[3] > 0). It specifies whether the covariance matrix of parameter estimates and approximate standard errors (ASEs) are computed and printed.

opt[8]=0 does not compute covariance matrix and ASEs.
opt[8]=1 computes the covariance matrix and ASEs but prints only the ASEs.
opt[8]=3 computes and prints both the covariance matrix and the ASEs.

The default is *opt*[8]=0.

y refers to an N response vector $\mathbf{y}$.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors. If opt[1] is zero or missing, an intercept $\mathbf{x}_{n+1} \equiv 1$ is added by default as the last column of $\mathbf{X}$. If the matrix $\mathbf{X}$ is not specified, $\mathbf{y}$ is analyzed as a univariate data set.

sorb refers to an n vector containing either of the following:

- n observation numbers of a subset for which the objective function should be evaluated; this subset can be the start for a pairwise exchange algorithm if *opt*[4] is specified.
- n given parameters $\mathbf{b} = (b_1, \ldots, b_n)$ (including the intercept, if necessary) for which the objective function should be evaluated.

Missing values are not permitted in x or y. Missing values in *opt* cause the default value to be used.

The LMS and LTS subroutines return the following values:

sc is a column vector containing the following scalar information, where rows 1–9 correspond to LMS or LTS regression and rows 11–14 correspond to either LS or WLS:

sc[1] the quantile h used in the objective function
sc[2] number of subsets generated
sc[3] number of subsets with singular linear systems
sc[4] number of nonzero weights w_i
sc[5] lowest value of the objective function F_{LMS} or F_{LTS} attained
sc[6] preliminary LMS or LTS scale estimate S_P
sc[7] final LMS or LTS scale estimate S_F
sc[8] robust R^2 (*coefficient of determination*)

sc[9] asymptotic consistency factor

If *opt*[3] > 0, then the following can also be set:

sc[11] LS or WLS objective function (sum of squared residuals)
sc[12] LS or WLS scale estimate
sc[13] R^2 value for LS or WLS
sc[14] F value for LS or WLS

For *opt*[3]=1 or *opt*[3]=3, these rows correspond to WLS estimates; for *opt*[3]=2, these rows correspond to LS estimates.

coef is a matrix with n columns containing the following results in its rows:

coef[1,] LMS or LTS parameter estimates
coef[2,] indices of observations in the best subset

If *opt*[3] > 0, then the following can also be set:

coef[3] LS or WLS parameter estimates
coef[4] approximate standard errors of LS or WLS estimates
coef[5] t-values
coef[6] p-values
coef[7] lower boundary of Wald confidence intervals
coef[8] upper boundary of Wald confidence intervals

For *opt*[3]=1 or *opt*[3]=3, these rows correspond to WLS estimates; for *opt*[3]=2, to LS estimates.

wgt is a matrix with N columns containing the following results in its rows:

wgt[1] weights (=1 for small, =0 for large residuals)
wgt[2] residuals $r_i = y_i - \mathbf{x}_i\mathbf{b}$
wgt[3] resistant diagnostic u_i (note that the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider results for Brownlee's (1965) stackloss data. The three explanatory variables correspond to measurements for a plant oxidizing ammonia to nitric acid:

- x_1 air flow to the plant
- x_2 cooling water inlet temperature
- x_3 acid concentration

on 21 consecutive days. The response variable y_i gives the permillage of ammonia lost (stackloss). The data are also given by Rousseeuw & Leroy (1987, p.76) and Osborne (1985, p.267):

```
print "Stackloss Data";
aa = { 1  80  27  89  42,
       1  80  27  88  37,
       1  75  25  90  37,
       1  62  24  87  28,
       1  62  22  87  18,
       1  62  23  87  18,
       1  62  24  93  19,
       1  62  24  93  20,
       1  58  23  87  15,
       1  58  18  80  14,
       1  58  18  89  14,
       1  58  17  88  13,
       1  58  18  82  11,
       1  58  19  93  12,
       1  50  18  89   8,
       1  50  18  86   7,
       1  50  19  72   8,
       1  50  19  79   8,
       1  50  20  80   9,
       1  56  20  82  15,
       1  70  20  91  15 };
```

Rousseeuw & Leroy (1987, p.76) cite a large number of papers where this data set was analyzed before and state that most researchers "concluded that observations 1, 3, 4, and 21 were outliers," and some people also reported observation 2 as outlier.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5985 different subsets of 4 observations out of 21. If you decide not to specify `optn[5]`, your LMS and LTS algorithms draw $N_{rep} = 2000$ random sample subsets. Since there is a large number of subsets with singular linear systems, which you do not want to print, you chose `optn[2]=2;` for reduced printed output.

```
a = aa[,2:4]; b = aa[,5];
optn = j(8,1,.);
optn[2]= 2;    /* ipri */
optn[3]= 3;    /* ilsq */
optn[8]= 3;    /* icov */
```

```
CALL LMS(sc,coef,wgt,optn,b,a);
```

```
 LMS: The 13th ordered squared residual will be minimized.
```

Median and Mean

	Median	Mean
VAR1	58	60.428571429
VAR2	20	21.095238095
VAR3	87	86.285714286
Intercep	1	1
Response	15	17.523809524

Dispersion and Standard Deviation

	Dispersion	StdDev
VAR1	5.930408874	9.1682682584
VAR2	2.965204437	3.160771455
VAR3	4.4478066555	5.3585712381
Intercep	0	0
Response	5.930408874	10.171622524

The following are the results of LS regression:

Unweighted Least-Squares Estimation

LS Parameter Estimates

Variable	Estimate	Approx Std Err	t Value	Pr > |t|
VAR1	0.715640	0.134858	5.31	<.0001
VAR2	1.295286	0.368024	3.52	0.0026
VAR3	-0.152123	0.156294	-0.97	0.3440
Intercep	-39.919674	11.895997	-3.36	0.0038

Variable	Lower WCI	Upper WCI
VAR1	0.451323	0.979957
VAR2	0.573972	2.016600
VAR3	-0.458453	0.154208
Intercep	-63.2354	-16.603949

```
                    Sum of Squares = 178.8299616
                      Degrees of Freedom = 17
                    LS Scale Estimate = 3.2433639182

                    Cov Matrix of Parameter Estimates

             VAR1         VAR2         VAR3        Intercep

VAR1         0.018187    -0.036511    0.007144    0.287587
VAR2        -0.036511     0.135442    0.000010   -0.651794
VAR3        -0.007144     0.000011    0.024428   -1.676321
Intercep     0.287587    -0.651794    1.676321  141.514741

                    R-squared = 0.9135769045
                  F(3,17) Statistic = 59.9022259
                    Probability = 3.0163272E-9
```

These are the LMS results for the 2000 random subsets:

```
                Random Subsampling for LMS


                                    Best
   Subset     Singular       Criterion     Percent

      500           23        0.163262          25
     1000           55        0.140519          50
     1500           79        0.140519          75
     2000          103        0.126467         100

          Minimum Criterion= 0.1264668282
       Least Median of Squares (LMS) Method
    Minimizing 13th Ordered Squared Residual.
   Highest Possible Breakdown Value =  42.86 %
         Random Selection of 2103 Subsets
       Among 2103 subsets 103 are singular.

             Observations of Best Subset

  15                11                19                10

               Estimated Coefficients

  VAR1              VAR2              VAR3           Intercep

  0.75               0.5                 0            -39.25
```

```
LMS Objective Function = 0.75
 Preliminary LMS Scale = 1.0478510755
      Robust R Squared = 0.96484375
       Final LMS Scale = 1.2076147288
```

For LMS observations, 1, 3, 4, and 21 have scaled residuals larger than 2.5 (table not shown) and are considered outliers. These are the corresponding WLS results:

Weighted Least-Squares Estimation

RLS Parameter Estimates Based on LMS

Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.797686	0.067439	11.83	<.0001	0.665507	0.929864
VAR2	0.577340	0.165969	3.48	0.0041	0.252047	0.902634
VAR3	-0.067060	0.061603	-1.09	0.2961	-0.187800	0.053680
Intercep	-37.652459	4.732051	-7.96	<.0001	-46.927108	-28.37781

```
Weighted Sum of Squares = 20.400800254
      Degrees of Freedom = 13
  RLS Scale Estimate = 1.2527139846
```

Cov Matrix of Parameter Estimates

	VAR1	VAR2	VAR3	Intercep
VAR1	0.004548	-0.007921	-0.001199	0.001568
VAR2	-0.007921	0.027546	-0.000463	-0.065018
VAR3	-0.001199	-0.000463	0.003795	-0.246102
Intercep	0.001568	-0.065018	-0.246102	22.392305

```
Weighted R-squared = 0.9750062263
 F(3,13) Statistic = 169.04317954
    Probability = 1.158521E-10
There are 17 points with nonzero weight.
     Average Weight = 0.8095238095
```

References

- Brownlee, K.A. (1965), *Statistical Theory and Methodology in Science and Engineering*, New York: John Wiley & Sons, Inc.
- Davies, L. (1992), "The Asymptotics of Rousseeuw's Minimum Volume Ellipsoid Estimator," *The Annals of Statistics*, 20, 1828 –1843.
- Rousseeuw, P.J. (1984), "Least Median of Squares Regression," *Journal of the American Statistical Association*, 79, 871 –880.
- Rousseeuw, P.J. (1985), "Multivariate Estimation with High Breakdown Point," in *Mathematical Statistics and Applications*, ed. by W. Grossmann, G. Pflug, I. Vincze, and W. Wertz, Dordrecht: Reidel Publishing Company, 283 –297.
- Rousseeuw, P.J. and Croux, C. (1993), "Alternatives to the Median Absolute Deviation," *Journal of the American Statistical Association*, 88, 1273 –1283.
- Rousseeuw, P.J. and Hubert, M. (1997), "Recent Developments in PROGRESS," in L_1*-Statistical Procedures and Related Topics*, ed. by Y. Dodge, IMS Lecture Notes - Monograph Series, No. 31, 201 –214.
- Rousseeuw, P.J. and Leroy, A.M. (1987), *Robust Regression and Outlier Detection*, New York: John Wiley & Sons, Inc.
- Rousseeuw, P.J. and Van Driessen, K. (1997), "A fast Algorithm for the Minimum Covariance Determinant Estimator," submitted for publication.
- Rousseeuw, P.J. and Van Zomeren, B.C. (1990), "Unmasking Multivariate Outliers and Leverage Points," *Journal of the American Statistical Association*, 85, 633 –639.

LOAD Statement

loads modules and matrices from library storage

LOAD <**MODULE=(***module-list***)**> <*matrix-list*>**;**

The inputs to the LOAD statement are as follows:

module-list	is a list of modules.
matrix-list	is a list of matrices

The LOAD statement loads modules or matrix values from the current library storage into the current workspace. For example, to load three modules A, B, and C and one matrix X, specify the statement

```
load module=(A B C) X;
```

The special operand _ALL_ can be used to load all matrices or all modules. For example, if you want to load all matrices, specify

```
load _all_;
```

If you want to load all modules, specify

```
load module=_all_;
```

To load all matrices and modules stored in the library storage, you can enter the LOAD command without any arguments:

```
load;
```

The storage library can be specified using a RESET *storage* command The default library is SASUSER.IMLSTOR. For more information, see Chapter 14, "Storage Features," and the descriptions of the STORE, REMOVE, RESET, and SHOW statements.

LOC Function

finds nonzero elements of a matrix

LOC(*matrix***)**

where *matrix* is a numeric matrix or literal. The LOC function creates a $1 \times n$ row vector, where n is the number of nonzero elements in the argument. Missing values are treated as zeros. The values in the resulting row vector are the locations of the nonzero elements in the argument (in row-major order, like subscripting). For example, the statements

```
a={1 0 2 3 0};
b=loc(a);
```

result in the row vector

```
B             1 row       3 cols    (numeric)

                1         3         4
```

since the first, third, and fourth elements of $\mathbf{A}$ are nonzero. If every element of the argument vector is 0, the result is empty; that is, $\mathbf{B}$ has zero rows and zero columns.

The LOC function is useful for subscripting parts of a matrix that satisfy some condition. For example, suppose you want to create a matrix $\mathbf{Y}$ containing the rows of $\mathbf{X}$ that have a positive element in the diagonal of $\mathbf{X}$. Specify the following statements.

```
x={1  1  0,
   0 -2  2,
   0  0  3};
y=x[loc(vecdiag(x)>0),];
```

The result is

$$\mathbf{Y} = \mathbf{X}[\{13\},]$$

or the matrix

```
Y             2 rows      3 cols    (numeric)

                1         1         0
                0         0         3
```

since the first and third rows of $\mathbf{X}$ have positive elements on the diagonal of $\mathbf{X}$.

The next example selects all positive elements of a column vector $\mathbf{A}$:

```
a={0,
  -1,
   2,
   0};
y=a[loc(a>0),];
```

The result is

$$\mathbf{Y} = \mathbf{A}[3,]$$

or the scalar

```
Y             1 row       1 col     (numeric)

                          2
```

LOG Function

takes the natural logarithm

LOG(*matrix***)**

where *matrix* is a numeric matrix or literal.

The LOG function is the scalar function that takes the natural logarithm of each element of the argument matrix. An example of a valid statement is shown below:

```
b=log(c);
```

LP Call

solves the linear programming problem

CALL LP(*rc, x, dual, a, b <, cntl><, u><, l><, basis>***);**

The inputs to the LP subroutine are as follows:

a is an $m \times n$ vector specifying the technological coefficients, where m is less than or equal to n.

b is an $m \times 1$ vector specifying the right-side vector.

cntl is an optional row vector with 1 to 5 elements. If CNTL=(*indx, nprimal, ndual, epsilon, infinity*), then

indx	is the subscript of nonzero objective coefficient.
nprimal	is the maximum number of primal iterations.
ndual	is the maximum number of dual iterations.
epsilon	is the value of virtual zero.
infinity	is the value of virtual infinity.

The default values are as follows: *indx* equals n, *nprimal* equals 999999, *ndual* equals 999999, *epsilon* equals 1.0E−8, and *infinity* is machine dependent. If you specify *ndual* or *nprimal* or both, then on return they contain the number of iterations actually performed.

u is an optional array of dimension n specifying upper bounds on the decision variables. If you do not specify u, the upper bounds are assumed to be *infinity*.

l is an optional array of dimension n specifying lower bounds on the decision variables. If l is not given, then the lower bounds are assumed to be 0 for all the decision variables. This includes the decision variable associated with the objective value, which is specified by the value of *indx*.

basis is an optional array of dimension n specifying the current basis. This is given by identifying which columns are explicitly in the basis and which columns are at their upper bound, as given in u. The absolute value of the elements in this vector is a permutation of the column indices. The columns specified in the first m elements of *basis* are considered the explicit basis. The absolute value of the last $n-m$ elements of *basis* are the indices of the nonbasic variables. Any of the last $n-m$ elements of *basis* that are negative indicate that that nonbasic variable is at its upper bound. On return from the LP subroutine, the *basis* vector contains the final basis encountered. If you do not specify *basis*, then the subroutine assumes that an initial basis is in the last m columns of $\mathbf{A}$ and that no nonbasic variables are at their upper bound.

rc returns one of the following scalar return codes:

0	solution is optimal
1	solution is primal infeasible and dual feasible
2	solution is dual infeasible and primal feasible
3	solution is neither primal nor dual feasible
4	singular basis encountered
5	solution is numerically unstable
6	subroutine could not obtain enough memory
7	number of iterations exceeded

x returns the current primal solution in a column vector of length n.

dual returns the current dual solution in a row vector of length m.

The LP subroutine solves the linear program:

$$\max(0, \ldots, 0, 1, 0, \ldots, 0)\mathbf{x}$$
$$\text{st. } \mathbf{A}\mathbf{x} = \mathbf{b}$$
$$l \leq \mathbf{x} \leq \mathbf{u}$$

The subroutine first inverts the initial basis. If the **BASIS** vector is given, then the initial basis is the $m \times m$ submatrix identified by the first m elements in **BASIS**; otherwise, the initial basis is defined by the last m columns of $\mathbf{A}$. If the initial basis is singular, the subroutine returns with RC=4. If the basis is nonsingular, then the current dual and primal solutions are evaluated. If neither is feasible, then the subroutine returns with RC=3. If the primal solution is feasible, then the primal algorithm iterates until either a dual feasible solution is encountered or the number of NPRIMAL iterations is exceeded. If the dual solution is feasible, then the dual algorithm iterates until either a primal feasible solution is encountered or the number of NDUAL iterations is exceeded. When a basis is identified that is both primal and dual feasible, then the subroutine returns with RC=0.

Note that care must be taken when solving a sequence of linear programs and using the NPRIMAL or NDUAL control parameters or both. Because the LP subroutine resets the NPRIMAL and NDUAL parameters to reflect the number of iterations executed, subsequent invocations of the LP subroutine will have the number of iterations limited to the number used by the last LP subroutine executed. In these cases you should consider resetting these parameters prior to each LP call.

Consider the following example to maximize X_1 subject to the constraints $X_1 - X_2 = 10$ and $X_1 \geq 0$. The problem is solved as follows:

```
   /* the problem data */
obj={1 0};
coef={1 1};
```

```
b={0, 10};

   /* embed the objective function */
   /* in the coefficient matrix    */
a=obj//coef;
a=a||{-1, 0};

   /* solve the problem */
call lp(rc,x,dual,a,b);
```

The result is

```
RC            1 row       1 col     (numeric)

                          0


X             3 rows      1 col     (numeric)

                         10
                          0
                         10


DUAL          1 row       2 cols    (numeric)

                       -1         1
```

LTS Call

performs robust regression

CALL LTS(*sc, coef, wgt, opt,* $y <$, $< x >$<, *sorb*>>**);**

See the entry for the LMS subroutine for details.

LUPDT Call

provides updating and downdating for rank deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses

CALL LUPDT(*lup, bup, sup,* l, $z <$, b, $y <$, *ssq*>>**);**

The LUPDT subroutine returns the following values:

lup is an $n \times n$ lower triangular matrix $\mathbf{L}$ that is updated or downdated by using the q rows in $\mathbf{Z}$.

bup is an $n \times p$ matrix $\mathbf{B}$ of right-hand sides that is updated or downdated by using the q rows in $\mathbf{Y}$. If b is not specified, *bup* is not accessible.

sup is a p vector of square roots of residual sum of squares that is updated or downdated by using the q rows in $\mathbf{Y}$. If *ssq* is not specified, *sup* is not accessible.

The inputs to the LUPDT subroutine are as follows:

l specifies an $n \times n$ lower triangular matrix $\mathbf{L}$ to be updated or downdated by q row vectors z stored in the $q \times n$ matrix $\mathbf{Z}$. Only the lower triangle of l is used; the upper triangle may contain any information.

z is a $q \times n$ matrix $\mathbf{Z}$ used rowwise to update or downdate the matrix $\mathbf{L}$.

b specifies an optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with $\mathbf{L}$. If b is specified, the argument y must be specified.

y specifies an optional $q \times p$ matrix $\mathbf{Y}$ used rowwise to update or downdate the right-hand-side matrix $\mathbf{B}$.

ssq specifies an optional $p \times 1$ vector that, if b is specified, specifies the square root of the error sum of squares that should be updated or downdated simultaneously with $\mathbf{L}$ and b.

The relevant formula for the LUPDT call is $\tilde{\mathbf{L}}\tilde{\mathbf{L}}' = \mathbf{L}\mathbf{L}' + \mathbf{Z}\mathbf{Z}'$. See the LUPDT Example section for an example of the LUPDT call.

MAD Function

finds the univariate (scaled) median-absolute-deviation

MAD(*$x <, spt >$***)**

where

x is an $n \times p$ input data matrix.

spt is an optional string argument with the following values:

"MAD" for computing the MAD (which is the default)

"NMAD" for computing the normalized version of MAD

"SN" for computing S_n

"QN" for computing Q_n

The MAD function can be used for computing one of the following three robust scale estimates:

- Median Absolute Deviation (MAD) or normalized form of MAD:

$$MAD_n = b * med_i^n \ |x_i - med_j^n \ x_j|$$

where $b = 1$ is the unscaled default and $b = 1.4826$ is used for the scaled version (consistency with the Gaussian distribution).

- S_n which is a more efficient alternative to MAD:

$$S_n = c_n * med_i \; med_{j \neq i} \; |x_i - x_j|$$

where the outer median is a low median (order statistic of rank $\left[\frac{n+1}{2}\right]$) and the inner median is a high median (order statistic of rank $\left[\frac{n}{2} + 1\right]$), and where c_n is a scalar depending on sample size n.

- Q_n is another efficient alternative to MAD. It is based on the kth order statistic of the $\begin{pmatrix} n \\ 2 \end{pmatrix}$ inter-point distances:

$$Q_n = d_n * \{|x_i - x_j|; \quad i < j\}_{(k)} \quad \text{with} \quad k \approx \begin{pmatrix} n \\ 2 \end{pmatrix} / 4$$

where d_n is a scalar similar to, but different from c_n. See Rousseeuw and Croux (1993) for more details.

The scalars c_n and d_n are defined as follows:

$$c_n = 1.1926 * \begin{cases} .743 & \text{for n=2} \\ 1.851 & \text{for n=3} \\ .954 & \text{for n=4} \\ 1.351 & \text{for n=5} \\ .993 & \text{for n=6} \\ 1.198 & \text{for n=7} \\ 1.005 & \text{for n=8} \\ 1.131 & \text{for n=9} \\ n/(n-0.9) & \text{odd n} \\ 1.0 & \text{otherwise} \end{cases} \qquad d_n = 2.2219 * \begin{cases} .399 & \text{for n=2} \\ .994 & \text{for n=3} \\ .512 & \text{for n=4} \\ .844 & \text{for n=5} \\ .611 & \text{for n=6} \\ .857 & \text{for n=7} \\ .669 & \text{for n=8} \\ .872 & \text{for n=9} \\ n/(n+1.4) & \text{uneven n} \\ n/(n+3.8) & \text{even n} \end{cases}$$

Example

This example uses the univariate data set of Barnett & Lewis (1978) that is used above to illustrate the univariate LMS and LTS estimates:

```
b = { 3, 4, 7, 8, 10, 949, 951 };

rmad1 = mad(b);
rmad2 = mad(b,"mad");
rmad3 = mad(b,"nmad");
rmad4 = mad(b,"sn");
rmad5 = mad(b,"qn");
print "Default MAD=" rmad1,
      "Common MAD =" rmad2,
      "MAD*1.4826 =" rmad3,
      "Robust S_n =" rmad4,
      "Robust Q_n =" rmad5;
```

This program produces the following:

```
Default MAD=         4
Common MAD =         4
MAD*1.4826 = 5.9304089
Robust S_n =  7.143674
Robust Q_n = 5.7125049
```

References

- Barnett, V. and Lewis, T. (1978), *Outliers in Statistical Data*, New York: John Wiley & Sons, Inc.
- Rousseeuw, P.J. and Croux, C. (1993), "Alternatives to the Median Absolute Deviation," *Journal of the American Statistical Association*, 88, 1273 –1283.

MARG Call

evaluates marginal totals in a multiway contingency table

CALL MARG(*locmar, marginal, dim, table, config***);**

The inputs to the MARG subroutine are as follows:

dim	specifies a vector containing the number of variables and the number of their possible levels in a contingency table. If *dim* is $1 \times v$, then there are v variables, and the value of the ith element is the number of levels of the ith variable.
table	specifies an array containing the number of observations at each level of each variable. Variables are nested across columns and then across rows.
config	specifies an array containing the marginal totals to be evaluated. Each column specifies a distinct marginal.
locmar	returns a vector of indices to each new set of marginal totals specified by *config*. A marginal total is exhibited for each level of the specified marginal. These indices help locate particular totals.
marginal	returns a vector of marginal totals.

The matrix *table* must conform in size to the contingency table specified in *dim*. In particular, if *table* is $n \times m$, the product of the entries in the *dim* vector must equal nm. In addition, there must be some integer k such that the product of the first k entries in *dim* equals m. See the description of the IPF function for more information on specifying *table*.

For example, consider the no-three-factor-effect model for Bartlett's data as described in Bishop, Fienberg, and Holland (1975).

```
dim={2 2 2};
table={156   84   84  156,
       107  133   31  209};
config={1 1 2,
        2 3 3};
call marg(locmar,marginal,dim,table,config);
```

These statements return

```
      LOCMAR      1 row     3 cols  (numeric)

                    1       5       9

     MARGINAL    1 row    12 cols  (numeric)

  263     217     115     365     240     240     138
:342     240     240     240     240
```

MATTRIB Statement

associates printing attributes with matrices

MATTRIB *name* <**ROWNAME=***row-name*>
<**COLNAME=***column-name*> <**LABEL=***label*> <**FORMAT=***format*>**;**

The inputs to the MATTRIB subroutine are as follows:

name	is a character matrix or quoted literal giving the name of a matrix.
row-name	is a character matrix or quoted literal specifying row names.
column-name	is a character matrix or quoted literal specifying column names.
label	is a character matrix or quoted literal associating a label with the matrix. The *label* argument has a maximum length of 40 characters.
format	is a valid SAS format.

The MATTRIB statement associates printing attributes with matrices. Each matrix can be associated with a ROWNAME= matrix and a COLNAME= matrix, which is used whenever the matrix is printed to label the rows and columns, respectively. The statement is written as the keyword MATTRIB followed by a list of one or more names and attribute associations. It is not necessary to specify all attributes. The attribute associations are applied to the previous *name*. Thus, the following statement gives a row name RA and a column name CA to **A**, and a column name CB to **B**:

```
mattrib a rowname=ra colname=ca b colname=cb;
```

You cannot group names; although the following statement is valid, it does not associate anything with **A**.

```
mattrib a b rowname=n;
```

The values of the associated matrices are not looked up until they are needed. Thus, they need not have values at the time the MATTRIB statement is specified. They can be specified later when the object matrix is printed. The attributes continue to bind with the matrix until reassigned with another MATTRIB statement. To eliminate an attribute, specify EMPTY as the name, for example, ROWNAME=EMPTY. Labels can be up to 40 characters long. Longer labels are truncated. Use the SHOW *names* statemento view current matrix attributes.

An example using the MATTRIB statement follows:

```
rows='xr1':'xr5';
print rows;

                          ROWS
                          xr1 xr2 xr3 xr4 xr5


cols='cl1':'cl5';
print cols;

                          COLS
                          cl1 cl2 cl3 cl4 cl5


x={1 1 1 1,2 2 2 2,3 3 3 3};
mattrib x rowname=(rows [1:3 ])
          colname=(cols [1:4])
          label={'matrix,x'}
          format=5.2;
print x;

                    matrix,x   cl1   cl2   cl3   cl4

                    xr1       1.00  1.00  1.00  1.00
                    xr2       2.00  2.00  2.00  2.00
                    xr3       3.00  3.00  3.00  3.00
```

MAX Function

finds the maximum value of matrix

MAX(*matrix1*<, *matrix2*,. . ., *matrix15*>**)**

where *matrix* is a numeric or character matrix or literal.

The MAX function produces a single numeric value (or a character string value) that is the largest element (or highest character string value) in all arguments. There can be as many as 15 argument matrices. The function checks for missing numeric

values and does not include them in the result. If all arguments are missing, then the machine's most negative representable number is the result.

If you want to find the elementwise maximums of the corresponding elements of two matrices, use the maximum operator (<>).

For character arguments, the size of the result is the size of the largest of all arguments.

An example using the MAX function is shown below:

```
b=max(c);
```

MAXQFORM Call

computes the subsets of a matrix system that maximize the quadratic form

CALL MAXQFORM(*rc, maxq,* V *,* b *<,* *best*$>$**);**

If $\mathbf{V}$ and $\mathbf{b}$ are an $n \times n$ matrix and an $n \times 1$ vector, respectively, then the MAXQFORM function computes the subsets of components s such that $\mathbf{b}'[s]\mathbf{V}^{-1}[s,s]\mathbf{b}[s]$ is maximized.

The MAXQFORM subroutine returns the following values:

rc is one of the following scalar return codes:

0	normal return
1	error: the number of elements of $\mathbf{b}$ is too large to process
2	error: $\mathbf{V}$ is not positive semidefinite

maxq is an $m \times (n+2)$ matrix, where m is the total number of subsets computed and n is the number of elements in $\mathbf{b}$. The value of m depends on the value of *best* and is equal to $2^n - 1$ if *best* is not specified. Each row of *maxq* contains information for a selected subset of $\mathbf{V}$ and $\mathbf{b}$. The first element of the row is the number of components in the subset. The second element is the value of the quadratic form. The following elements of the row are either 0 or 1, to indicate whether the corresponding components of $\mathbf{V}$ and $\mathbf{b}$ are included in the subset.

The inputs to the MAXQFORM subroutine are as follows:

V specifies an $n \times n$ positive semidefinite matrix. Often this is generated as a crossproduct matrix, $\mathbf{X}'\mathbf{X}$, where $\mathbf{X}$ is a $k \times n$ matrix.

b specifies an $n \times 1$ vector. Often this arises as $\mathbf{X}'\mathbf{y}$, where $\mathbf{X}$ is a $k \times n$ matrix, and $\mathbf{y}$ is a $k \times 1$ vector.

best specifies an optional scalar. If *best* is specified with the value p, then the p subsets with the largest value for the quadratic form are returned for each subset size.

The leaps and bounds algorithm by Furnival and Wilson (1974) computes the maximum value of quadratic forms for subsets of components. Many statistics computed as a quadratic form can then be used as the criterion for the method of subset selection. These include the regression sum of squares, Wald statistics, and score statistics.

Consider the following fitness data, which consists of observations with values for age measured in years, weight measured in kilograms, time to run 1.5 miles measured in minutes, heart rate while resting, heart rate while running, maximum heart rate recorded while running, and oxygen intake rate while running measured in milliliters per kilogram of body weight per minute.

```
proc iml;
fit = {
  44  89.47  11.37  62  178  182  44.609,
  40  75.07  10.07  62  185  185  45.313,
  44  85.84   8.65  45  156  168  54.297,
  42  68.15   8.17  40  166  172  59.571,
  38  89.02   9.22  55  178  180  49.874,
  47  77.45  11.63  58  176  176  44.811,
  40  75.98  11.95  70  176  180  45.681,
  43  81.19  10.85  64  162  170  49.091,
  44  81.42  13.08  63  174  176  39.442,
  38  81.87   8.63  48  170  186  60.055,
  44  73.03  10.13  45  168  168  50.541,
  45  87.66  14.03  56  186  192  37.388,
  45  66.45  11.12  51  176  176  44.754,
  47  79.15  10.60  47  162  164  47.273,
  54  83.12  10.33  50  166  170  51.855,
  49  81.42   8.95  44  180  185  49.156,
  51  69.63  10.95  57  168  172  40.836,
  51  77.91  10.00  48  162  168  46.672,
  48  91.63  10.25  48  162  164  46.774,
  49  73.37  10.08  67  168  168  50.388,
  57  73.37  12.63  58  174  176  39.407,
  54  79.38  11.17  62  156  165  46.080,
  52  76.32   9.63  48  164  166  45.441,
  50  70.87   8.92  48  146  155  54.625,
  51  67.25  11.08  48  172  172  45.118,
  54  91.63  12.88  44  168  172  39.203,
  51  73.71  10.47  59  186  188  45.790,
  57  59.08   9.93  49  148  155  50.545,
  49  76.32   9.40  56  186  188  48.673,
  48  61.24  11.50  52  170  176  47.920,
  52  82.78  10.50  53  170  172  47.467 };
```

Use the following IML statement to center the data.

```
fit = fit - j(31,1,1) * fit[:,];
```

Now compute the crossproduct matrices, as follows:

```
x = fit[,1:6];
y = fit[,7];
xpx = x'*x;
xpy = x'*y;
```

The following statements compute the best three regression sums of squares for each size of regressor set:

```
call maxqform( rc, maxq, xpx, xpy, 3 );
print maxq;
```

MCD and MVE Calls

finds the minimum covariance determinant estimator and the minimum volume ellipsoid estimator

CALL MCD(*sc, coef, dist, opt,* $x <$, $s >$**);**

The MDC call is the robust (resistent) estimation of multivariate location and scatter, defined by minimizing the determinant of the covariance matrix computed from h points.

CALL MVE(*sc, coef, dist, opt,* $x <$, $s >$**);**

The MVE call is the robust (resistent) estimation of multivariate location and scatter, defined by minimizing the volume of an ellipsoid containing h points.

The MVE and MCD subroutines compute the minimum volume ellipsoid estimator and the minimum covariance determinant estimator. These robust locations and covariance matrices can be used to detect multivariate outliers and leverage points. For this purpose, the MVE and MCD subroutines provide a table of robust distances.

The inputs to the MCD and MVE subroutine are as follows:

opt refers to an options vector with the following components (missing values are treated as default values):

opt[1] specifies the amount of printed output. Higher option values request additional output and include the output of lower values.

opt[1]=0 prints no output except error messages.
opt[1]=1 prints most of the output.

opt[1]=2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.

opt[1]=3 additionally prints how many subsets result in singular linear systems.

The default is *opt*[1]=0.

opt[2] specifies whether the classical, initial, and final robust covariance matrices are printed. The default is *opt*[2]=0. Note that the final robust covariance matrix is always returned in *coef*.

opt[3] specifies whether the classical, initial, and final robust correlation matrices are printed or returned:

opt[3]=0 does not return or print.
opt[3]=1 prints the robust correlation matrix.
opt[3]=2 returns the final robust correlation matrix in *coef*.
opt[3]=3 prints and returns the final robust correlation matrix.

opt[4] specifies the quantile h used in the objective function. The default is *opt*[5]= $h = \frac{N+n+1}{2}$. If the value of h is specified outside the range $\frac{N}{2}+1 \leq h \leq \frac{3N}{4}+\frac{n+1}{4}$, it is reset to the closest boundary of this region.

opt[5] specifies the number N_{Rep} of subset generations. This option is the same as described previously for the LMS and LTS subroutines. Due to computer time restrictions, not all subset combinations can be inspected for larger values of N and n. If *opt*[6] is zero or missing, the default number of subsets is taken from the following table.

n	1	2	3	4	5	6	7	8	9	10
N_{lower}	500	50	22	17	15	14	0	0	0	0
N_{upper}	1000000	1414	182	71	43	32	27	24	23	22
N_{Rep}	500	1000	1500	2000	2500	3000	3000	3000	3000	3000

n	11	12	13	14	15
N_{lower}	0	0	0	0	0
N_{upper}	22	22	22	23	23
N_{Rep}	3000	3000	3000	3000	3000

If the number of cases (observations) N is smaller than N_{lower}, then all possible subsets are used; otherwise, N_{Rep} subsets are drawn randomly. This means that an exhaustive search is performed for *opt*[6]=-1. If N is larger than N_{upper}, a note is printed in the log file indicating how many subsets exist.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors.

s refers to an n vector containing n observation numbers of a subset for which the objective function should be evaluated. This subset can be the start for a pairwise exchange algorithm if *opt*[4] is specified.

Missing values are not permitted in x. Missing values in *opt* cause the default value to be used.

The MCD and MVE subroutines return the following values:

sc is a column vector containing the following scalar information:

- *sc*[1] the quantile h used in the objective function
- *sc*[2] number of subsets generated
- *sc*[3] of subsets with singular linear systems
- *sc*[4] number of nonzero weights w_i
- *sc*[5] lowest value of the objective function F_{MVE} attained (volume of smallest ellipsoid found)
- *sc*[6] Mahalanobis-like distance used in the computation of the lowest value of the objective function F_{MVE}
- *sc*[7] the cutoff value used for the outlier decision
- *sc*[8] the correction factor $c(N, n)$ used in scaling the initial MVE scatter matrix
- *sc*[9] scaling factor for the initial MVE scatter matrix

$$f = \frac{c^2(N,n)}{\mathrm{CINV}(0.5,n)}$$

coef is a matrix with n columns containing the following results in its rows:

- *coef*[1] location of ellipsoid center
- *coef*[2] eigenvalues of final robust scatter matrix
- *coef*[3:2+n] the final robust scatter matrix for *opt*[2]=1 or *opt*[2]=3

dist is a matrix with N columns containing the following results in its rows:

- *dist*[1] Mahalanobis distances
- *dist*[2] robust distances based on the final estimates
- *dist*[3] weights (=1 for small, =0 for large robust distances)

Example

Consider results for Brownlee's (1965) stackloss data. The three explanatory variables correspond to measurements for a plant oxidizing ammonia to nitric acid on 21 consecutive days.

- x_1 air flow to the plant
- x_2 cooling water inlet temperature
- x_3 acid concentration

The response variable y_i gives the permillage of ammonia lost (stackloss). These data are also given by Rousseeuw & Leroy (1987, p.76).

```
print "Stackloss Data";
aa = { 1  80  27  89  42,
       1  80  27  88  37,
       1  75  25  90  37,
       1  62  24  87  28,
       1  62  22  87  18,
       1  62  23  87  18,
       1  62  24  93  19,
       1  62  24  93  20,
       1  58  23  87  15,
       1  58  18  80  14,
       1  58  18  89  14,
       1  58  17  88  13,
       1  58  18  82  11,
       1  58  19  93  12,
       1  50  18  89   8,
       1  50  18  86   7,
       1  50  19  72   8,
       1  50  19  79   8,
       1  50  20  80   9,
       1  56  20  82  15,
       1  70  20  91  15 };
```

Rousseeuw & Leroy (1987, p.76) cite a large number of papers where this data set was analyzed before and state that most researchers "concluded that observations 1, 3, 4, and 21 were outliers"; some people also reported observation 2 as an outlier.

By default, subroutine MVE tries only 2000 randomly selected subsets in its search. There are in total 5985 subsets of 4 cases out of 21 cases.

```
a = aa[,2:4];
optn = j(8,1,.);
optn[1]= 2;              /* ipri */
optn[2]= 1;              /* pcov: print COV */
optn[3]= 1;              /* pcor: print CORR */
optn[5]= -1;             /* nrep: use all subsets */

CALL MVE(sc,xmve,dist,optn,a);
```

The first part of the output shows the classical scatter and correlation matrix:

Minimum Volume Ellipsoid (MVE) Estimation

Consider Ellipsoids Containing 12 Cases.

Classical Covariance Matrix

	VAR1	VAR2	VAR3
VAR1	84.057142857	22.657142857	24.571428571
VAR2	22.657142857	9.9904761905	6.6214285714
VAR3	24.571428571	6.6214285714	28.714285714

Classical Correlation Matrix

	VAR1	VAR2	VAR3
VAR1	1	0.781852333	0.5001428749
VAR2	0.781852333	1	0.3909395378
VAR3	0.5001428749	0.3909395378	1

Classical Mean

VAR1	60.428571429
VAR2	21.095238095
VAR3	86.285714286

There are 5985 subsets of 4 cases out of 21 cases.
All 5985 subsets will be considered.

The second part of the output shows the results of the optimization (complete subset sampling):

Complete Enumeration for MVE

Subset	Singular	Best Criterion	Percent
1497	22	253.312431	25
2993	46	224.084073	50
4489	77	165.830053	75
5985	156	165.634363	100

Minimum Criterion= 165.63436284

Among 5985 subsets 156 are singular.

Observations of Best Subset

7	10	14	20

Initial MVE Location Estimates

VAR1	58.5
VAR2	20.25
VAR3	87

Initial MVE Scatter Matrix

	VAR1	VAR2	VAR3
VAR1	34.829014749	28.413143611	62.32560534
VAR2	28.413143611	38.036950318	58.659393261
VAR3	62.32560534	58.659393261	267.63348175

The third part of the output shows the optimization results after local improvement:

Final MVE Estimates (Using Local Improvement)

Number of Points with Nonzero Weight=17

Robust MVE Location Estimates

VAR1	56.705882353
VAR2	20.235294118
VAR3	85.529411765

Robust MVE Scatter Matrix

	VAR1	VAR2	VAR3
VAR1	23.470588235	7.5735294118	16.102941176
VAR2	7.5735294118	6.3161764706	5.3676470588
VAR3	16.102941176	5.3676470588	32.389705882

Eigenvalues of Robust Scatter Matrix

VAR1	46.597431018
VAR2	12.155938483
VAR3	3.423101087

Robust Correlation Matrix

	VAR1	VAR2	VAR3
VAR1	1	0.6220269501	0.5840361335
VAR2	0.6220269501	1	0.375278187
VAR3	0.5840361335	0.375278187	1

The final output presents a table containing the classical Mahalanobis distances, the robust distances, and the weights identifying the outlying observations (that is leverage points when explaining y with these three regressor variables):

Classical Distances and Robust (Rousseeuw) Distances

Unsquared Mahalanobis Distance and
Unsquared Rousseeuw Distance of Each Observation

N	Mahalanobis Distances	Robust Distances	Weight
1	2.253603	5.528395	0
2	2.324745	5.637357	0
3	1.593712	4.197235	0
4	1.271898	1.588734	1.000000
5	0.303357	1.189335	1.000000
6	0.772895	1.308038	1.000000
7	1.852661	1.715924	1.000000
8	1.852661	1.715924	1.000000
9	1.360622	1.226680	1.000000
10	1.745997	1.936256	1.000000
11	1.465702	1.493509	1.000000
12	1.841504	1.913079	1.000000
13	1.482649	1.659943	1.000000
14	1.778785	1.689210	1.000000
15	1.690241	2.230109	1.000000
16	1.291934	1.767582	1.000000
17	2.700016	2.431021	1.000000
18	1.503155	1.523316	1.000000
19	1.593221	1.710165	1.000000
20	0.807054	0.675124	1.000000
21	2.176761	3.657281	0

Distribution of Robust Distances

MinRes	1st Qu.	Median
0.6751244996	1.5084120761	1.7159242054
Mean	**3rd Qu.**	**MaxRes**
2.2282960174	2.0831826658	5.6373573538

Cutoff Value = 3.0575159206

The cutoff value is the square root of the 0.975 quantile of the chi square distribution with 3 degrees of freedom.

There are 4 points with large robust distances receiving zero weights. These may include boundary cases. Only points whose robust distances are substantially larger than the cutoff value should be considered outliers.

References

- Brownlee, K.A. (1965), *Statistical Theory and Methodology in Science and Engineering*, New York: John Wiley & Sons, Inc.
- Davies, L. (1992), "The Asymptotics of Rousseeuw's Minimum Volume Ellipsoid Estimator," *The Annals of Statistics*, 20, 1828 –1843.
- Rousseeuw, P.J. (1984), "Least Median of Squares Regression," *Journal of the American Statistical Association*, 79, 871 –880.
- Rousseeuw, P.J. (1985), "Multivariate Estimation with High Breakdown Point," in *Mathematical Statistics and Applications*, ed. by W. Grossmann, G. Pflug, I. Vincze, and W. Wertz, Dordrecht: Reidel Publishing Company, 283 –297.
- Rousseeuw, P.J. and Croux, C. (1993), "Alternatives to the Median Absolute Deviation," *Journal of the American Statistical Association*, 88, 1273 –1283.
- Rousseeuw, P.J. and Hubert, M. (1997), "Recent developments in PROGRESS," in L_1-*Statistical Procedures and Related Topics*, ed. by Y. Dodge, IMS Lecture Notes - Monograph Series, No. 31, 201 –214.
- Rousseeuw, P.J. and Leroy, A.M. (1987), *Robust Regression and Outlier Detection*, New York: John Wiley & Sons, Inc.
- Rousseeuw, P.J. and Van Driessen, K. (1997), "A fast Algorithm for the Minimum Covariance Determinant Estimator," submitted for publication.
- Rousseeuw, P.J. and Van Zomeren, B.C. (1990), "Unmasking Multivariate Outliers and Leverage Points," *Journal of the American Statistical Association*, 85, 633 –639.

MIN Function

finds the smallest element of a matrix

MIN(*matrix1*<*, matrix2,. . ., matrix15*>**)**

where *matrix* is a numeric or character matrix or literal.

The MIN function produces a single numeric value (or a character string value) that is the smallest element (lowest character string value) in all arguments. There can be as many as 15 argument matrices. The function checks for missing numeric values and excludes them from the result. If all arguments are missing, then the machine's largest representable number is the result.

If you want to find the elementwise minimums of the corresponding elements of two matrices, use the element minimum operator (><).

For character arguments, the size of the result is the size of the largest of all arguments.

An example using the MIN function is shown below.

```
b=min(c);
```

MOD Function

computes the modulo (remainder)

MOD(*value, divisor***)**

The inputs to the MOD function are as follows:

value	is a numeric matrix or literal giving the dividend.
divisor	is a numeric matrix or literal giving the divisor.

The MOD function is the scalar function returning the remainder of the division of elements of the first argument by elements of the second argument. An example of a valid statement follows.

```
b=mod(4,1);
```

NAME Function

lists the names of arguments

NAME(*arguments***);**

where *arguments* are the names of existing matrices.

The NAME function returns the names of the arguments in a column vector. In the following example, $\mathbf{N}$ is a 3×1 character matrix of element size 8 containing the character values A, B, and C:

```
n=name(a,b,c);
```

The main use of the NAME function is with macros when you want to use an argument for both its name and its value.

NCOL Function

finds the number of columns of a matrix

NCOL(*matrix***)**

where *matrix* is a numeric or character matrix.

The NCOL function returns a single numeric value that is the number of columns in *matrix*. If the matrix has not been given a value, the NCOL function returns a value of 0.

For example, to let B contain the number of columns of matrix **S**, use the statement

```
b=ncol(s);
```

NLENG Function

finds the size of an element

NLENG(*matrix***)**

where *matrix* is a numeric or character matrix.

The NLENG function returns a single numeric value that is the size in bytes of each element in *matrix*. All matrix elements have the same size. If the matrix does not have a value, then the NLENG function returns a value of 0. This function is different from the LENGTH function, which returns the size of each element of a character matrix, omitting the trailing blanks.

The following statement returns the value 7:

```
a=nleng({"ab " "ijklm  ",
         "x"   " "});
```

Nonlinear Optimization and Related Subroutines

Table 17.1. Nonlinear Optimization and Related Subroutines

Optimization Subroutines
Conjugate Gradient Optimization Method **CALL NLPCG(***rc, xr, "fun", x0 <, opt, blc, tc, par, "ptit", "grd">***);** Double Dogleg Optimization Method **CALL NLPDD(***rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "grd">***);** Nelder-Mead Simplex Optimization Method **CALL NLPNMS(***rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "nlc">***);** Newton-Raphson Optimization Method **CALL NLPNRA(***rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "grd", "hes">***);** Newton-Raphson Ridge Optimization Method **CALL NLPNRR(***rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "grd", "hes">***);** (Dual) Quasi-Newton Optimization Method **CALL NLPQN(***rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "grd", "nlc", "jacnlc">***);** Quadratic Optimization Method **CALL NLPQUA(***rc, xr, quad, x0 <,opt, blc, tc, par, "ptit", lin>***);** Trust-Region Optimization Method **CALL NLPTR(***rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "grd", "hes">***);**

Least-Squares Subroutines
Hybrid Quasi-Newton Least-Squares Methods **CALL NLPHQN(***rc, xr, "fun", x0, opt <,blc, tc, par, "ptit", "jac">***);** Levenberg-Marquardt Least-Squares Method **CALL NLPLM(***rc, xr, "fun", x0, opt <,blc, tc, par, "ptit", "jac">***);**

Supplementary Subroutines
Approximate Derivatives by Finite Differences **CALL NLPFDD(***f, g, h, "fun", x0* <*,par, "grd"*>**);** Feasible Point Subject to Constraints **CALL NLPFEA(***xr, x0, blc* <*,par*>**);**

Note: The names of the optional arguments can be used as keywords. For example, the following statements are equivalent:

```
call nlpnrr(rc,xr,"fun",x0,,,ter,,,"grad");
call nlpnrr(rc,xr,"fun",x0) tc=ter grd="grad";
```

All the optimization subroutines require at least two input arguments.

- The NLPQUA subroutine requires the *quad* matrix argument, which specifies the symmetric matrix **G** of the quadratic problem. The input can be dense or sparse. Other optimization subroutines require the *fun* module argument, which specifies an IML module that defines the objective function or functions. For least-squares subroutines, the FUN module must return a column vector of length m that corresponds to the values of the m functions $f_1(x), \ldots, f_m(x)$, each evaluated at the point $x = (x_1, \ldots, x_n)$. For other subroutines, the FUN module must return the value of the objective function $f = f(x)$ evaluated at the point x.
- The argument *x0* specifies a row vector that defines the number of parameters n. If *x0* is a feasible point, it represents a starting point for the iterative optimization process. Otherwise, a linear programming algorithm is called at the start of each optimization subroutine to replace the input *x0* by a feasible starting point.

The other arguments that can be used as input are described in the following list. As indicated in Table 17.1, not all input arguments apply to each subroutine. Note that you can specify optional arguments with the *keyword=argument* syntax.

- The *opt* argument indicates an options vector that specifies details of the optimization process, such as particular updating techniques and whether the objective function is to be maximized instead of minimized. See “Options Vector” for details.
- The *blc* argument specifies a constraint matrix that defines lower and upper bounds for the n parameters as well as general linear equality and inequality constraints. For details, see
- The *tc* argument specifies a vector of thresholds corresponding to the termination criteria tested in each iteration. See “Termination Criteria” for details.

- The *par* argument specifies a vector of control parameters that can be used to modify the algorithms if the default settings do not complete the optimization process successfully. For details, see
- The *"ptit"* module argument specifies an IML module that replaces the subroutine used to print the iteration history and test the termination criteria. If the *"ptit"* module is specified, the matrix specified by the *tc* argument has no effect. See "Termination Criteria" for details.
- The *"grd"* module argument specifies an IML module that computes the gradient vector, $g = \nabla f$, at a given input point x. See "Objective Function and Derivatives" for details.
- The *"hes"* module argument specifies an IML module that computes the $n \times n$ Hessian matrix, $\mathbf{G} = \nabla^2 f$, at a given input point x. See "Objective Function and Derivatives" for details.
- The *"jac"* module argument specifies an IML module that computes the $m \times n$ Jacobian matrix, $\mathbf{J} = (\nabla f_i)$, of the m least-squares functions at a given input point x. See "Objective Function and Derivatives" for details.
- The *"nlc"* module argument specifies an IML module that allows the computation of general equality and inequality constraints. This is the method by which nonlinear constraints must be specified. For details, see
- The *"jacnlc"* module argument specifies an IML module that computes the Jacobian matrix of first-order derivatives of the equality and inequality constraints specified by the NLC module. For details, see "Parameter Constraints".
- The *lin* argument specifies the linear part of the quadratic optimization problem. See "NLPQUA Call" for details.

The modules that can be used as input arguments for the subroutines (*"fun", "grd", "hes", "jac", "ptit", "nlc",* and *"jacnlc"*) allow only a single input parameter $x = (x_1, \ldots, x_n)$. You can provide more input parameters for these modules by using the GLOBAL clause. See "Using the GLOBAL Clause" for an example.

All the optimization subroutines return the following results:

- The scalar return code *rc* indicates the reason for the termination of the optimization process. A return code *rc* > 0 indicates successful termination corresponding to one of the specified termination criteria. A return code *rc* < 0 indicates unsuccessful termination, that is, that the result *xr* is unreliable. See "Definition of Return Codes" for more details.
- The row vector *xr*, which has length n, the number of parameters, contains the optimal point when *rc* > 0.

NLPCG Call

nonlinear optimization by conjugate gradient method

CALL NLPCG(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPCG subroutine requires function and gradient calls; it does not need second-order derivatives. The gradient vector contains the first derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$g(x) = \nabla f(x) = \left(\frac{\partial f}{\partial x_j} \right)$$

If you do not specify an IML module with the *"grd"* argument, the first-order derivatives are approximated by finite difference formulas using only function calls. The NLPCG algorithm can require many function and gradient calls, but it requires less memory than other subroutines for unconstrained optimization. In general, many iterations are needed to obtain a precise solution, but each iteration is computationally inexpensive. You can specify one of four update formulas for generating the conjugate directions with the fourth element of the *opt* input argument.

Value of *opt*[4]	**Update Method**
1	Automatic restart method of Powell (1977) and Beale (1972). This is the default.
2	Fletcher-Reeves update (Fletcher 1987)
3	Polak-Ribiere update (Fletcher 1987)
4	Conjugate-descent update of Fletcher (1987)

The NLPCG subroutine is useful for optimization problems with large n. For the unconstrained or boundary constrained case, the NLPCG method needs only order n bytes of working memory, whereas the other optimization methods require order n^2 bytes of working memory. During n successive iterations, uninterrupted by restarts or changes in the working set, the conjugate gradient algorithm computes a cycle of n conjugate search directions. In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step size α that satisfies the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit for the step size. You can specify other line-search algorithms with the fifth element of the *opt* argument.

For an example of the NLPCG subroutine, see "Constrained Betts Function".

NLPDD Call

nonlinear optimization by double dogleg method

CALL NLPDD(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The double dogleg optimization method combines the ideas of the quasi-Newton and trust-region methods. In each iteration, the algorithm computes the step, $s^{(k)}$, as a linear combination of the steepest descent or ascent search direction, $s_1^{(k)}$, and a quasi-Newton search direction, $s_2^{(k)}$, as follows:

$$s^{(k)} = \alpha_1 s_1^{(k)} + \alpha_2 s_2^{(k)}.$$

The step $s^{(k)}$ must remain within a specified trust-region radius (refer to Fletcher 1987). Hence, the NLPDD subroutine uses the dual quasi-Newton update but does not perform a line search. You can specify one of two update formulas with the fourth element of the *opt* input argument.

Value of *opt*[4]	Update Method
1	Dual BFGS update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual DFP update of the Cholesky factor of the Hessian matrix

The double dogleg optimization technique works well for medium to moderately large optimization problems, in which the objective function and the gradient are much faster to compute than the Hessian. The implementation is based on Dennis and Mei (1979) and Gay (1983), but it is extended for boundary and linear constraints. The NLPDD subroutine generally needs more iterations than the techniques that require second-order derivatives (NLPTR, NLPNRA, and NLPNRR), but each of the NLPDD iterations is computationally inexpensive. Furthermore, the NLPDD subroutine needs only gradient calls to update the Cholesky factor of an approximate Hessian.

In addition to the standard iteration history, the NLPDD routine prints the following information:

- The heading *lambda* refers to the parameter λ of the double dogleg step. A value of 0 corresponds to the full (quasi-) Newton step.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero.

The following statements invoke the NLPDD subroutine to solve the constrained Betts optimization problem (see "Constrained Betts Function").

```
proc iml;
   start F_BETTS(x);
      f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
      return(f);
   finish F_BETTS;

   con = {  2. -50.  .   .,
           50.  50.  .   .,
           10.  -1. 1. 10.};
   x = {-1. -1.};
   optn = {0 1};
   call nlpdd(rc,xres,"F_BETTS",x,optn,con);
   quit;
```

The preceding statements produce the following iteration history.

```
                  Double Dogleg Optimization

       Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

                      Without Parameter Scaling
                Gradient Computed by Finite Differences

                  Parameter Estimates              2
                  Lower Bounds                     2
                  Upper Bounds                     2
                  Linear Constraints               1

                       Optimization Start

Active Constraints            0  Objective Function     -98.5376
Max Abs Gradient Element      2  Radius                        1


                    Function        Active       Objective
 Iter    Restarts      Calls   Constraints        Function

    1           0          2             0       -99.54678
    2           0          3             0       -99.59120
    3           0          5             0       -99.90252
    4           0          6             1       -99.96000
    5           0          7             1       -99.96000
    6           0          8             1       -99.96000


             Objective    Max Abs                Slope of
              Function   Gradient                  Search
 Iter          Change    Element    Lambda    Direction

    1          1.0092     0.1346     6.012       -1.805
    2          0.0444     0.1279         0      -0.0228
    3          0.3113     0.0624         0       -0.209
    4          0.0575    0.00432         0      -0.0975
```

```
     5          4.66E-6   0.000079         0     -458E-8
     6        1.559E-9          0         0     -16E-10


                         Optimization Results

Iterations                                6  Function Calls              9
Gradient Calls                            8  Active Constraints          1
Objective Function                   -99.96  Max Abs Gradient Element    0
Slope of Search Direction     -1.56621E-9  Radius                      1


   GCONV convergence criterion satisfied.
```

NLPFDD Call

approximates derivatives by finite differences method

CALL NLPFDD(*f, g, h, "fun", x0, <,par, "grd">***);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPFDD subroutine can be used for the following tasks:

- If the module *"fun"* returns a scalar, the NLPFDD subroutine computes the function value *f*, the gradient vector *g*, and the Hessian matrix *h*, all evaluated at the point *x0*.
- If the module *"fun"* returns a column vector of m function values, the subroutine assumes that a least-squares function is specified, and it computes the function vector *f*, the Jacobian matrix **J**, and the crossproduct of the Jacobian matrix $\mathbf{J}'\mathbf{J}$ at the point *x0*. Note that in this case, you must set the first element of the *par* argument to m.

If any of the results cannot be computed, the subroutine returns a missing value for that result.

You can specify the following input arguments with the NLPFDD subroutine:

- The *"fun"* argument refers to an IML module that returns either a scalar value or a column vector of length m. This module returns the value of the objective function or, for least-squares problems, the values of the m functions that comprise the objective function.
- The *x0* argument is a vector of length n that defines the point at which the functions and derivatives should be computed.

- The *par* argument is a vector that defines options and control parameters. Note that the *par* argument in the NLPFDD call is different from the one used in the optimization subroutines.
- The *"grd"* argument is optional and refers to an IML module that returns a vector defining the gradient of the function at *x0*. If the *"fun"* argument returns a vector of values instead of a scalar, the *"grd"* argument is ignored.

If the *"fun"* module returns a scalar, the subroutine returns the following values:

- *f* is the value of the function at the point *x0*.
- *g* is a vector containing the value of the gradient at the point *x0*. If you specify the *"grd"* argument, the gradient is computed from that module. Otherwise, the approximate gradient is computed by a finite difference approximation using calls of the function module in a neighborhood of *x0*.
- *h* is a matrix containing a finite difference approximation of the value of the Hessian at the point *x0*. If you specify the *"grd"* argument, the Hessian is computed by calls of that module in a neighborhood of *x0*. Otherwise, it is computed by calls of the function module in a neighborhood of *x0*.

If the *"fun"* module returns a vector, the subroutine returns the following values:

- *f* is a vector containing the values of the m functions comprising the objective function at the point *x0*.
- *g* is the $m \times n$ Jacobian matrix $\mathbf{J}$, which contains the first-order derivatives of the functions with respect to the parameters, evaluated at *x0*. It is computed by finite difference approximations in a neighborhood of *x0*.
- *h* is the $n \times n$ crossproduct of the Jacobian matrix, $\mathbf{J}^T\mathbf{J}$. It is computed by finite difference approximations in a neighborhood of *x0*.

The *par* argument is a vector of length 3.

- *par*[1] corresponds to the *opt*[1] argument in the optimization subroutines. This argument is relevant only to least-squares optimization methods, in which case it specifies the number of functions returned by the module *"fun"*. If *par*[1] is missing or is smaller than 1, it is set to 1.
- *par*[2] corresponds to the *opt*[8] argument in the optimization subroutines. It determines what type of approximation is to be used and how the finite difference interval, h, is to be computed. See "Finite Difference Approximations of Derivatives" for details.
- *par*[3] corresponds to the *par*[8] argument in the optimization subroutines. It specifies the number of accurate digits in evaluating the objective function. The default is $-\log_{10}(\epsilon)$, where ϵ is the machine precision.

If you specify a missing value in the *par* argument, the default value is used.

The NLPFDD subroutine is particularly useful for checking your analytical derivative specifications of the *"grd"*, *"hes"*, and *"jac"* modules. You can compare the results of the modules with the finite difference approximations of the derivatives of *f* at the point *x0* to verify your specifications.

In the unconstrained Rosenbrock problem (see "Unconstrained Rosenbrock Function"), the objective function is

$$f(x) = 50(x_2 - x_1^2)^2 + \frac{1}{2}(1 - x_1)^2$$

Then the gradient and the Hessian, evaluated at the point $x = (2, 7)$, are

$$g' = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 200x_1^3 - 200x_1x_2 + x_1 - 1 \\ -100x_1^2 + 100x_2 \end{bmatrix} = \begin{bmatrix} -1199 \\ 300 \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 600x_1^2 - 200x_2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix} = \begin{bmatrix} 1001 & -400 \\ -400 & 100 \end{bmatrix}$$

The following statements define the Rosenbrock function and use the NLPFDD call to compute the gradient and the Hessian.

```
proc iml;
   start F_ROSEN(x);
      y1 = 10. * (x[2] - x[1] * x[1]);
      y2 = 1. - x[1];
      f  = .5 * (y1 * y1 + y2 * y2);
      return(f);
   finish F_ROSEN;
   x = {2 7};
   CALL NLPFDD(crit,grad,hess,"F_ROSEN",x);
   print grad;
   print hess;
```

```
GRAD

-1199 300.00001


HESS

1000.9998 -400.0018
-400.0018 99.999993
```

If the Rosenbrock problem is considered from a least-squares perspective, the two functions are

$$\begin{aligned} f_1(x) &= 10(x_2 - x_1^2) \\ f_2(x) &= 1 - x_1 \end{aligned}$$

Then the Jacobian and the crossproduct of the Jacobian, evaluated at the point $x = (2, 7)$, are

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -40 & 10 \\ -1 & 0 \end{bmatrix}$$

$$\mathbf{J}^T\mathbf{J} = \begin{bmatrix} 400x_1^2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix} = \begin{bmatrix} 1601 & -400 \\ -400 & 100 \end{bmatrix}$$

The following statements define the Rosenbrock problem in a least-squares framework and use the NLPFDD call to compute the Jacobian and the crossproduct matrix. Since the value of the PARMS variable, which is used for the *par* argument, is 2, the NLPFDD subroutine allocates memory for a least-squares problem with two functions, $f_1(x)$ and $f_2(x)$.

```
proc iml;
   start F_ROSEN(x);
      y = j(2,1,0.);
      y[1] = 10. * (x[2] - x[1] * x[1]);
      y[2] = 1. - x[1];
      return(y);
   finish F_ROSEN;
   x     = {2 7};
   parms = 2;
   CALL NLPFDD(fun,jac,crpj,"F_ROSEN",x,parms);
   print jac;
   print crpj;
```

The finite difference approximations for Jacobian follow.

```
                JAC

            -40          10
             -1           0


               CRPJ

           1601        -400
           -400         100
```

NLPFEA Call

computes feasible points subject to constraints

CALL NLPFEA(*xr, x0, blc* <*,par*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPFEA subroutine tries to compute a point that is feasible subject to a set of boundary and linear constraints. You can specify boundary and linear constraints that define an empty feasible region, in which case the subroutine will return missing values.

You can specify the following input arguments with the NLPFEA subroutine:

- *x0* is a row vector defining the coordinates of a point that is not necessarily feasible for a set of linear and boundary constraints.
- *blc* is an $m \times n$ matrix defining a set of m boundary and linear constraints. See "Parameter Constraints" for details.
- *par* is a vector of length two. The argument is different from the one used in the optimization subroutines. The first element sets the LCEPS parameter, which controls how precisely the returned point must satisfy the constraints. The second element sets the LCSING parameter, which specifies the criterion for deciding when constraints are considered linearly dependent. For details, see

The NLPFEA subroutine returns the *xr* argument. The result is a vector containing either the n coordinates of a feasible point, which indicates that the subroutine was successful, or missing values, which indicates that the subroutine could not find a feasible point.

The following statements call the NLPFEA subroutine with the constraints from the Betts problem (see "Constrained Betts Function") and an initial infeasible point $x_0 = (-17, -61)$. The subroutine returns the feasible point $(2, -50)$ as the vector XFEAS.

```
proc iml;
   con = {  2. -50.  .   .,
           50.  50.  .   .,
           10.  -1. 1. 10.};
   x = {-17. -61};
   call nlpfea(xfeas,x,con);
```

NLPHQN Call

calculates hybrid quasi-Newton least squares

CALL NLPHQN(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "jac"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPHQN subroutine uses one of the Fletcher and Xu (1987) hybrid quasi-Newton methods. Refer also to Al-Baali and Fletcher (1985, 1986). In each iteration, the subroutine uses a criterion to decide whether a Gauss-Newton or a dual quasi-Newton search direction is appropriate. You can choose one of three criteria (HY1, HY2, or HY3) proposed by Fletcher and Xu (1987) with the sixth element of the *opt* vector. The default is HY2. The subroutine computes the crossproduct Jacobian (for the Gauss-Newton step), updates the Cholesky factor of an approximate Hessian (for the quasi-Newton step), and performs a line search to compute an approximate minimum along the search direction. The default line-search technique used by the NLPHQN method is designed for least-squares problems (refer to Lindström and Wedin 1984, and Al-Baali and Fletcher 1986), but you can specify a different line-search algorithm with the fifth element of the *opt* argument. See "Options Vector" for details.

You can specify two update formulas with the fourth element of the *opt* argument as indicated in the following table.

Value of *opt*[4]	**Update Method**
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.

The NLPHQN subroutine needs approximately the same amount of working memory as the NLPLM subroutine, and in most applications, the latter seems to be superior. Hence, the NLPHQN method is recommended only when the NLPLM method encounters problems.

Note: In least-squares subroutines, you must set the first element of the *opt* vector to m, the number of functions.

In addition to the standard iteration history, the NLPHQN subroutine prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that, on the basis of the Fletcher and Xu (1987) criterion, the subroutine used a Gauss-Newton search direction instead of a quasi-Newton search direction.
- The heading *alpha* is the step size, α, computed with the line-search algorithm.

- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The following statements use the NLPHQN call to solve the unconstrained Rosenbrock problem (see "Unconstrained Rosenbrock Function").

```
proc iml;
   title 'Test of NLPHQN subroutine: No Derivatives';
   start F_ROSEN(x);
     y = j(1,2,0.);
     y[1] = 10. * (x[2] - x[1] * x[1]);
     y[2] = 1. - x[1];
    return(y);
   finish F_ROSEN;

   x = {-1.2 1.};
   optn = {2 2};
   call nlphqn(rc,xr,"F_ROSEN",x,optn);
```

The iteration history for the subroutine follows.

```
                  Optimization Start
                  Parameter Estimates
                                          Gradient
                                         Objective
        N Parameter         Estimate        Function

        1 X1               -1.200000     -107.799999
        2 X2                1.000000      -44.000000

           Value of Objective Function = 12.1


           Hybrid Quasi-Newton LS Minimization

Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
          Version HY2 of Fletcher & Xu (1987)
        Gradient Computed by Finite Differences
      CRP Jacobian Computed by Finite Differences

         Parameter Estimates                  2
         Functions (Observations)             2

                  Optimization Start

Active Constraints                    0  Objective Function    12.1
Max Abs Gradient Element     107.7999987


                 Function        Active       Objective
Iter    Restarts      Calls   Constraints       Function
```

```
 1          0          3             0         7.22423
 2*         0          5             0         0.97090
 3*         0          7             0         0.81911
 4          0          9             0         0.69103
 5          0         19             0         0.47345
 6*         0         21             0         0.35906
 7*         0         22             0         0.23342
 8*         0         24             0         0.14799
 9*         0         26             0         0.00948
10*         0         28             0       1.98834E-6
11*         0         30             0       7.0768E-10
12*         0         32             0       2.0246E-21

          Objective     Max Abs                Slope of
           Function    Gradient       Step      Search
Iter         Change     Element       Size   Direction

 1           4.8758     56.9322     0.0616       -628.8
 2*          6.2533      2.3017      0.266      -14.448
 3*          0.1518      3.7839      0.119       -1.942
 4           0.1281      5.5103      2.000       -0.144
 5           0.2176      8.8638     11.854       -0.194
 6*          0.1144      9.8734      0.253       -0.947
 7*          0.1256     10.1490      0.398       -0.718
 8*          0.0854     11.6248      1.346       -0.467
 9*          0.1385      2.6275      1.443       -0.296
10*         0.00947     0.00609      0.938      -0.0190
11*       1.988E-6    0.000748      1.003      -398E-8
12*       7.08E-10    1.82E-10      1.000      -14E-10


                         Optimization Results

Iterations                           12  Function Calls                33
Jacobian Calls                       13  Gradient Calls                19
Active Constraints                    0  Objective Function        2.024612E-21
Max Abs Gradient Element  1.816863E-10  Slope of Search Direction -1.415366E-9

   ABSGCONV convergence criterion satisfied.

                         Optimization Results
                         Parameter Estimates
                                          Gradient
                                          Objective
          N Parameter         Estimate     Function

          1 X1                1.000000   1.816863E-10
          2 X2                1.000000   -1.22069E-10

           Value of Objective Function = 2.024612E-21
```

NLPLM Call

calculates Levenberg-Marquardt least squares

CALL NLPLM(*rc, xr, "fun", x0, opt, blc, tc, par, "ptit", "jac"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPLM subroutine uses the Levenberg-Marquardt method, which is an efficient modification of the trust-region method for nonlinear least-squares problems and is implemented as in Moré (1978). This is the recommended algorithm for small to medium least-squares problems. Large least-squares problems can often be processed more efficiently with other subroutines, such as the NLPCG and NLPQN methods. In each iteration, the NLPLM subroutine solves a quadratically-constrained quadratic minimization problem that restricts the step to the boundary or interior of an n-dimensional elliptical trust region.

The m functions $f_1(x), \ldots, f_m(x)$ are computed by the module specified with the *"fun"* module argument. The $m \times n$ Jacobian matrix, **J**, contains the first-order derivatives of the m functions with respect to the n parameters, as follows:

$$\mathbf{J}(x) = (\nabla f_1, \ldots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

You can specify **J** with the *"jac"* module argument; otherwise, the subroutine will compute it with finite difference approximations. In each iteration, the subroutine computes the crossproduct of the Jacobian matrix, $\mathbf{J}^T\mathbf{J}$, to be used as an approximate Hessian.

Note: In least-squares subroutines, you must set the first element of the *opt* vector to m, the number of functions.

In addition to the standard iteration history, the NLPLM subroutine also prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that the computed Hessian approximation was singular and had to be ridged with a positive value.
- The heading *lambda* represents the Lagrange multiplier, λ. This has a value of zero when the optimum of the quadratic function approximation is inside the trust region, in which case a trust-region-scaled Newton step is performed. It is greater than zero when the optimum is at the boundary of the trust region, in which case the scaled Newton step is too long to fit in the trust region and a quadratically-constrained optimization is done. Large values indicate optimization difficulties, and as in Gay (1983), a negative value indicates the special case of an indefinite Hessian matrix.

- The heading *rho* refers to ρ, the ratio between the achieved and predicted difference in function values. Values that are much smaller than one indicate optimization difficulties. Values close to or larger than one indicate that the trust region radius can be increased.

The iteration history for the solution of the unconstrained Rosenbrock problem follows. See the section "Unconstrained Rosenbrock Function" for the statements that generate this output.

Optimization Start
Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	-1.200000	-107.799999
2	X2	1.000000	-44.000000

Value of Objective Function = 12.1

Levenberg-Marquardt Optimization

Scaling Update of More (1978)
Gradient Computed by Finite Differences
CRP Jacobian Computed by Finite Differences

Parameter Estimates	2
Functions (Observations)	2

Optimization Start

Active Constraints	0	Objective Function	12.1
Max Abs Gradient Element	107.7999987	Radius	2626.5613171

Iter	Restarts	Function Calls	Active Constraints	Objective Function
1	0	4	0	2.18185
2	0	6	0	1.59370
3	0	7	0	1.32848
4	0	8	0	1.03891
5	0	9	0	0.78943
6	0	10	0	0.58838
7	0	11	0	0.34224
8	0	12	0	0.19630
9	0	13	0	0.11626
10	0	14	0	0.0000396
11	0	15	0	2.4652E-30

Iter	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	9.9181	17.4704	0.00804	0.964
2	0.5881	3.7015	0.0190	0.988
3	0.2652	7.0843	0.00830	0.678
4	0.2896	6.3092	0.00753	0.593
5	0.2495	7.2617	0.00634	0.486
6	0.2011	7.8837	0.00462	0.393
7	0.2461	6.6815	0.00307	0.524
8	0.1459	8.3857	0.00147	0.469
9	0.0800	9.3086	0.00016	0.409
10	0.1162	0.1781	0	1.000
11	0.000040	4.44E-14	0	1.000

Optimization Results

Iterations	11	Function Calls	16
Jacobian Calls	12	Active Constraints	0
Objective Function	2.46519E-30	Max Abs Gradient Element	4.440892E-14
Lambda	0	Actual Over Pred Change	1
Radius	0.0178062912		

ABSGCONV convergence criterion satisfied.

Optimization Results
Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	1.000000	-4.44089E-14
2	X2	1.000000	2.220446E-14

Value of Objective Function = 2.46519E-30

NLPNMS Call

nonlinear optimization by Nelder-Mead simplex method

CALL NLPNMS(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "nlc"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The Nelder-Mead simplex method is one of the subroutines that can solve optimization problems with nonlinear constraints. It does not use any derivatives, and it does not assume that the objective function has continuous derivatives. However, the objective function must be continuous. The NLPNMS technique uses a large number of function calls, and it may be unable to generate precise results when $n > 40$.

The NLPNMS subroutine uses the following simplex algorithms:

- For unconstrained or only boundary-constrained problems, the original Nelder-Mead simplex algorithm is implemented and extended to boundary constraints. This algorithm does not compute the objective for infeasible points, and it is invoked if the *"nlc"* module argument is not specified and the *blc* argument contains at most two rows (corresponding to lower and upper bounds).
- For linearly or nonlinearly constrained problems, a slightly modified version of Powell's (1992) Constrained Optimization BY Linear Approximations (COBYLA) implementation is used. This algorithm is invoked if the *"nlc"* module argument is specified or if at least one linear constraint is specified with the *blc* argument.

The original Nelder-Mead algorithm cannot be used for general linear or nonlinear constraints, but in the unconstrained or boundary-constrained cases, it can be faster. It changes the shape of the simplex by adapting the nonlinearities of the objective function; this contributes to an increased speed of convergence.

Powell's COBYLA Algorithm

Powell's COBYLA algorithm is a sequential trust-region algorithm that tries to maintain a regularly-shaped simplex throughout the iterations. The algorithm uses a monotone-decreasing radius, ρ, of a spheric trust region. The modification implemented in the NLPNMS call permits an increase of the trust-region radius ρ in special situations. A sequence of iterations is performed with a constant trust-region radius ρ until the computed function reduction is much less than the predicted reduction. Then, the trust-region radius ρ is reduced. The trust-region radius is increased only if the computed function reduction is relatively close to the predicted reduction and if the simplex is well-shaped. The start radius, ρ_{beg}, can be specified with the second element of the *par* argument, and the final radius, ρ_{end}, can be specified with the ninth element of the *tc* argument. Convergence to small values of ρ_{end}, or high-precision convergence, may require many calls of the function and constraint modules and may result in numerical problems. The main reasons for the slow convergence of the

COBYLA algorithm are as follows:

- Linear approximations of the objective and constraint functions are used locally.
- Maintaining the regularly-shaped simplex and not adapting its shape to nonlinearities yields very small simplexes for highly nonlinear functions, such as fourth-order polynomials.

To allocate memory for the vector returned by the *"nlc"* module argument, you must specify the total number of nonlinear constraints with the tenth element of the *opt* argument. If any of the constraints are equality constraints, the number of equality constraints must be specified by the eleventh element of the *opt* argument. See "Parameter Constraints" for details.

For more information on the special sets of termination criteria used by the NLPNMS algorithms, see

In addition to the standard iteration history, the NLPNMS subroutine prints the following information. For unconstrained or boundary-constrained problems, the subroutine also prints

- *difcrit*, which, in this subroutine, refers to the difference between the largest and smallest function values of the $n + 1$ simplex vertices
- *std*, which is the standard deviation of the function values of the simplex vertices
- *deltax*, which is the vertex length of a restarted simplex. If there are convergence problems, the algorithm restarts the iteration process with a simplex of smaller vertex length.
- *size*, which is the average L_1 distance of the simplex vertex with the smallest function value to the other simplex vertices

For linearly and nonlinearly constrained problems, the subroutine prints the following:

- *conmax* is the maximum constraint violation.
- *meritf* is the value of the merit function, Φ.
- *difmerit* is the difference between adjacent values of the merit function.
- ρ is the trust-region radius.

The following code uses the NLPNMS call to solve the Rosen-Suzuki problem (see "Rosen-Suzuki Problem"), which has three nonlinear constraints:

```
proc iml;
   start F_HS43(x);
      f = x*x‘ + x[3]*x[3] - 5*(x[1] + x[2]) - 21*x[3] + 7*x[4];
      return(f);
   finish F_HS43;
```

```
start C_HS43(x);
   c = j(3,1,0.);
   c[1] = 8 - x*x‘ - x[1] + x[2] - x[3] + x[4];
   c[2] = 10 - x*x‘ - x[2]*x[2] - x[4]*x[4] + x[1] + x[4];
   c[3] = 5 - 2.*x[1]*x[1] - x[2]*x[2] - x[3]*x[3]
            - 2.*x[1] + x[2] + x[4];
   return(c);
finish C_HS43;
x = j(1,4,1);
optn= j(1,11,.); optn[2]= 3; optn[10]= 3; optn[11]=0;
call nlpnms(rc,xres,"F_HS43",x,optn,,,,,"C_HS43");
```

Part of the output produced by the preceding code follows.

```
          Optimization Start
          Parameter Estimates
     N Parameter         Estimate

     1 X1                1.000000
     2 X2                1.000000
     3 X3                1.000000
     4 X4                1.000000

   Value of Objective Function = -19


          Values of Nonlinear Constraints

     Constraint         Residual

     [         1 ]      4.0000
     [         2 ]      6.0000
     [         3 ]      1.0000


     Nelder-Mead Simplex Optimization

  COBYLA Algorithm by M.J.D. Powell (1992)


Minimum Iterations                                    0
Maximum Iterations                                 1000
Maximum Function Calls                             3000
Iterations Reducing Constraint Violation              0
ABSFCONV Function Criterion                           0
FCONV Function Criterion                  2.220446E-16
FCONV2 Function Criterion                         1E-6
FSIZE Parameter                                       0
ABSXCONV Parameter Change Criterion              0.0001
XCONV Parameter Change Criterion                      0
XSIZE Parameter                                       0
ABSCONV Function Criterion                -1.34078E154
Initial Simplex Size (INSTEP)                       0.5
Singularity Tolerance (SINGULAR)                  1E-8
      Nelder-Mead Simplex Optimization
```

```
    COBYLA Algorithm by M.J.D. Powell (1992)

                Parameter Estimates               4
                Nonlinear Constraints             3

               Optimization Start

Objective Function   -29.5  Maximum Constraint Violation   4.5
```

Iter	Restarts	Function Calls	Objective Function	Maximum Constraint Violation
1	0	12	-52.80342	4.3411
2	0	17	-39.51475	0.0227
3	0	53	-44.02098	0.00949
4	0	62	-44.00214	0.000833
5	0	72	-44.00009	0.000033
6	0	79	-44.00000	1.783E-6
7	0	90	-44.00000	1.363E-7
8	0	94	-44.00000	1.543E-8

Iter	Merit Function	Merit Function Change	Between Actual and Predicted Change
1	-42.3031	12.803	1.000
2	-39.3797	-2.923	0.250
3	-43.9727	4.593	0.0625
4	-43.9977	0.0249	0.0156
5	-43.9999	0.00226	0.0039
6	-44.0000	0.00007	0.0010
7	-44.0000	1.74E-6	0.0002
8	-44.0000	5.33E-7	0.0001

```
                                  Optimization Results

Iterations                                 8  Function Calls                95
Restarts                                   0  Objective Function  -44.00000003
Maximum Constraint Violation  1.543059E-8  Merit Function      -43.99999999
Actual Over Pred Change             0.0001

   ABSXCONV convergence criterion satisfied.

WARNING: The point x is feasible only at the LCEPSILON= 1E-7 range.

            Optimization Results
            Parameter Estimates
       N Parameter         Estimate
```

```
      1 X1            -0.000034167
      2 X2                1.000004
      3 X3                2.000023
      4 X4               -0.999971

Value of Objective Function = -44.00000003


        Values of Nonlinear Constraints

         Constraint       Residual

         [         1 ]    -1.54E-8    *?*
         [         2 ]      1.0000
         [         3 ]     -1.5E-8    *?*
```

NLPNRA Call

nonlinear optimization by Newton-Raphson method

CALL NLPNRA(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd", "hes"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPNRA algorithm uses a pure Newton step at each iteration when both the Hessian is positive definite and the Newton step successfully reduces the value of the objective function. Otherwise, it performs a combination of ridging and line-search to compute successful steps. If the Hessian is not positive definite, a multiple of the identity matrix is added to the Hessian matrix to make it positive definite (refer to Eskow & Schnabel 1991).

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$, and it requires continuous first- and second-order derivatives of the objective function inside the feasible region. If second-order derivatives are computed efficiently and precisely, the NLPNRA method does not need many function, gradient, and Hessian calls, and it may perform well for medium to large problems.

Note that using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and may contain significant rounding errors. If you use the *"grd"* input argument to specify a module that computes first-order derivatives analytically, you can reduce drastically the computation time for numerical second-order derivatives. The computation of the finite difference approximation for the Hessian matrix generally uses only n calls of the module that specifies the gradient.

In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation. You can specify other line-search algorithms with the fifth element of the *opt* argument. See "Options Vector" for details.

In unconstrained and boundary constrained cases, the NLPNRA algorithm can take advantage of diagonal or sparse Hessian matrices that are specified by the input argument *"hes"*. To use sparse Hessian storage, the value of the ninth element of the *opt* argument must specify the number of nonzero Hessian elements returned by the Hessian module. See "Objective Function and Derivatives" for more details.

In addition to the standard iteration history, the NLPNRA subroutine prints the following information:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The following statements invoke the NLPNRA subroutine to solve the constrained Betts optimization problem (see "Constrained Betts Function"). The iteration history follows.

```
proc iml;
   start F_BETTS(x);
      f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
      return(f);
   finish F_BETTS;

   con = {  2. -50.  .   .,
           50.  50.  .   .,
           10.  -1. 1. 10.};
   x = {-1. -1.};
   optn = {0 2};
   call nlpnra(rc,xres,"F_BETTS",x,optn,con);
   quit;
```

```
                        Optimization Start
                        Parameter Estimates

                           Gradient    Lower        Upper
                           Objective   Bound        Bound
  N Parameter    Estimate   Function    Constraint   Constraint
  1 X1           6.800000   0.136000    2.000000     50.000000
  2 X2          -1.000000  -2.000000   -50.000000    50.000000

                Value of Objective Function = -98.5376
```

```
                        Linear Constraints

 1   59.00000 :      10.0000  <=   +   10.0000 * X1        -    1.0000 * X2

          Newton-Raphson Optimization with Line Search

                   Without Parameter Scaling
              Gradient Computed by Finite Differences
         CRP Jacobian Computed by Finite Differences
                 Parameter Estimates               2
                 Lower Bounds                      2
                 Upper Bounds                      2
                 Linear Constraints                1

                  Optimization Start

  Active Constraints        0  Objective Function   -98.5376
  Max Abs Gradient Element  2


                    Function        Active       Objective
Iter    Restarts       Calls   Constraints        Function

   1           0           2             0       -98.81551
   2*          0           3             0       -99.40840
   3*          0           4             1       -99.87504
   4           0           5             1       -99.96000
   5           0           6             1       -99.96000


       Objective    Max Abs             Slope of
        Function   Gradient      Step      Search
Iter      Change    Element      Size   Direction

   1      0.2779     1.8000     0.100      -2.925
   2*     0.5929     1.2713     0.294      -2.365
   3*     0.4666     0.5829     0.542      -1.181
   4      0.0850   0.000039     1.000      -0.170
   5     3.9E-10   9.537E-7     1.000     -76E-11

                  Optimization Results

  Iterations                           5  Function Calls            7
  Hessian Calls                        6  Active Constraints        1
  Objective Function             -99.96  Max Abs Gradient Element  0
  Slope of Search Direction -7.64376E-10  Ridge                     0

  GCONV convergence criterion satisfied.


                  Optimization Results
                  Parameter Estimates

                                      Gradient    Active
                                      Objective    Bound
 N Parameter         Estimate         Function    Constraint

 1 X1                2.000000         0.040000     Lower BC
 2 X2           -0.000000196                0
```

```
Value of Objective Function = -99.96


        Linear Constraints Evaluated at Solution

 1     10.00000  =  -10.0000 + 10.0000 * X1 - 1.0000 * X2
```

NLPNRR Call

nonlinear optimization by Newton-Raphson ridge method

CALL NLPNRR(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd", "hes"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPNRR algorithm uses a pure Newton step when both the Hessian is positive definite and the Newton step successfully reduces the value of the objective function. Otherwise, a multiple of the identity matrix is added to the Hessian matrix.

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$, and it requires continuous first- and second-order derivatives of the objective function inside the feasible region.

Note that using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and may contain significant rounding errors. If you use the *"grd"* input argument to specify a module that computes first-order derivatives analytically, you can reduce drastically the computation time for numerical second-order derivatives. The computation of the finite difference approximation for the Hessian matrix generally uses only n calls of the module that specifies the gradient.

The NLPNRR method performs well for small to medium-sized problems, and it does not need many function, gradient, and Hessian calls. However, if the gradient is not specified analytically by using the *"grd"* module argument, or if the computation of the Hessian module specified with the *"hes"* argument is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms may be more efficient.

In addition to the standard iteration history, the NLPNRR subroutine prints the following information:

- The heading *ridge* refers to the value of the nonnegative ridge parameter. A value of zero indicates that a Newton step is performed. A value greater than zero indicates either that the Hessian approximation is zero or that the Newton step fails to reduce the optimization criterion. A large value can indicate optimization difficulties.

- The heading *rho* refers to ρ, the ratio of the achieved difference in function values and the predicted difference, based on the quadratic function approximation. A value that is much smaller than one indicates possible optimization difficulties.

The following statements invoke the NLPNRR subroutine to solve the constrained Betts optimization problem (see "Constrained Betts Function"). The iteration history follows.

```
proc iml;
   start F_BETTS(x);
      f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
      return(f);
   finish F_BETTS;

   con = {  2. -50.  .   .,
           50.  50.  .   .,
           10.  -1. 1. 10.};
   x = {-1. -1.};
   optn = {0 2};
   call nlpnrr(rc,xres,"F_BETTS",x,optn,con);
   quit;
```

```
                  Optimization Start
                  Parameter Estimates

                           Gradient    Lower        Upper
                           Objective   Bound        Bound
 N Parameter   Estimate    Function    Constraint   Constraint

 1 X1          6.800000    0.136000    2.000000     50.000000
 2 X2         -1.000000   -2.000000   -50.000000    50.000000

          Value of Objective Function = -98.5376


                    Linear Constraints

1   59.00000 :      10.0000  <=   +   10.0000 * X1        -    1.0000 * X2


           Newton-Raphson Ridge Optimization

              Without Parameter Scaling
     Gradient Computed by Finite Differences
    CRP Jacobian Computed by Finite Differences

       Parameter Estimates              2
       Lower Bounds                     2
       Upper Bounds                     2
       Linear Constraints               1
```

Optimization Start

Active Constraints	0	Objective Function	-98.5376
Max Abs Gradient Element	2		

Iter	Restarts	Function Calls	Active Constraints	Objective Function
1	0	2	1	-99.87337
2	0	3	1	-99.96000
3	0	4	1	-99.96000

Iter	Objective Function Change	Max Abs Gradient Element	Ridge	Ratio Actual and Predicted Change
1	1.3358	0.5887	0	0.706
2	0.0866	0.000040	0	1.000
3	4.07E-10	0	0	1.014

Optimization Results

Iterations	3	Function Calls	5
Hessian Calls	4	Active Constraints	1
Objective Function	-99.96	Max Abs Gradient Element	0
Ridge	0	Actual Over Pred Change	1.0135158294

GCONV convergence criterion satisfied.

Optimization Results
Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	X1	2.000000	0.040000	Lower BC
2	X2	0.000000134	0	

Value of Objective Function = -99.96

Linear Constraints Evaluated at Solution

1	10.00000	=	-10.0000 + 10.0000 * X1 - 1.0000 * X2

NLPQN Call

nonlinear optimization by quasi-Newton method

CALL NLPQN(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit",*
"grd", "nlc", "jacnlc">**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPQN subroutine uses (dual) quasi-Newton optimization techniques, and it is one of the two subroutines available that can solve problems with nonlinear constraints. These techniques work well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian matrix. The NLPQN subroutine does not need to compute second-order derivatives, but it generally requires more iterations than the techniques that compute second-order derivatives.

The two categories of problems solved by the NLPQN subroutine are unconstrained or linearly constrained problems and nonlinearly constrained problems. Unconstrained or linearly constrained problems do not use the *"nlc"* or *"jacnlc"* module arguments, whereas nonlinearly constrained problems use the arguments to specify the nonlinear constraints and the Jacobian matrix of their first-order derivatives, respectively.

The type of optimization problem specified determines the algorithm that the subroutine invokes. The algorithms are very different, and they use different sets of termination criteria. For more details, see

Unconstrained or Linearly Constrained QN Optimization

The NLPQN subroutine invokes this algorithm if you do not specify the *"nlc"* argument. Using the fourth element of the *opt* argument, you can specify two update formulas for either the original quasi-Newton algorithm or the dual quasi-Newton algorithm, as indicated in the following table:

Value of *opt*[4]	**Update Method**
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.
3	Original Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update of the inverse Hessian matrix.
4	Original Davidon, Fletcher, and Powell (DFP) update of the inverse Hessian matrix.

In each iteration, a line search is performed along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step size that satisfies the

Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit of the step size. Violating the left-side Goldstein condition can affect the positive definiteness of the quasi-Newton update. In these cases, either the update is skipped or the iterations are restarted with an identity matrix resulting in the steepest descent or ascent search direction. You can specify line-search algorithms different from the default method with the fifth element of the *opt* argument.

Note: In Release 6.08, the DBFGS and DDFP updates were implemented with the NLPDQN subroutine. In Release 6.09 and in later releases, these updates are specified with the NLPQN subroutine, and the NLPDQN subroutine is not permitted.

The following statements invoke the NLPQN subroutine to solve the Rosenbrock problem (see "Unconstrained Rosenbrock Function"):

```
proc iml;
   start F_ROSEN(x);
      y1 = 10. * (x[2] - x[1] * x[1]);
      y2 = 1. - x[1];
      f  = .5 * (y1 * y1 + y2 * y2);
      return(f);
   finish F_ROSEN;
   x = {-1.2 1.};
   optn = {0 2 . 2};
   call nlpqn(rc,xr,"F_ROSEN",x,optn);
   quit;
```

Since OPTN$[4] = 2$, the DDFP update is performed. The gradient is approximated by finite differences since no module is specified that computes the first-order derivatives. Part of the iteration history follows. In addition to the standard iteration history, the NLPQN subroutine prints the following information for unconstrained or linearly constrained problems:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

```
                Optimization Start
                Parameter Estimates

                                        Gradient
                                        Objective
N Parameter          Estimate           Function

1 X1                -1.200000        -107.799989
2 X2                 1.000000         -43.999999

          Value of Objective Function = 12.1
```

Dual Quasi-Newton Optimization

Dual Davidon - Fletcher - Powell Update (DDFP)
Gradient Computed by Finite Differences

Parameter Estimates 2

Optimization Start

Active Constraints 0 Objective Function 12.1
Max Abs Gradient Element 107.79998927

Iter	Restarts	Function Calls	Active Constraints	Objective Function
1	0	4	0	2.06405
2	0	7	0	1.92035
3	0	10	0	1.78089
4	0	13	0	1.33331
5	0	17	0	1.13400
6	0	22	0	0.93915
7	0	24	0	0.84821
8	0	30	0	0.54334
9	0	32	0	0.46593
10	0	37	0	0.35322
12	0	41	0	0.20282
12	0	41	0	0.20282
13	0	46	0	0.11714
14	0	51	0	0.07149
15	0	53	0	0.04746
16	0	58	0	0.02759
17	0	60	0	0.01625
18	0	62	0	0.00475
19	0	66	0	0.00167
20	0	70	0	0.0005952
21	0	72	0	0.0000771
23	0	78	0	2.39914E-8
23	0	78	0	2.39914E-8
24	0	80	0	5.0936E-11
25	0	119	0	3.9538E-11

Iter	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
1	10.0359	0.7917	0.0340	-628.8
2	0.1437	8.6301	6.557	-0.0363
3	0.1395	11.0943	8.193	-0.0288
4	0.4476	7.6069	33.376	-0.0269
5	0.1993	0.9386	15.438	-0.0260
6	0.1948	3.5290	11.537	-0.0233
7	0.0909	4.8308	8.124	-0.0193
8	0.3049	4.1770	35.143	-0.0186
9	0.0774	0.9479	8.708	-0.0178

```
 10      0.1127     2.5981    10.964     -0.0147
 11      0.0894     3.3028    13.590     -0.0121
 12      0.0610     0.6451    10.000     -0.0116
 13      0.0857     1.6603    11.395     -0.0102
 14      0.0456     2.4050    11.559     -0.0074
 15      0.0240     0.5628     6.868     -0.0071
 16      0.0199     1.3282     5.365     -0.0055
 17      0.0113     1.9246     5.882     -0.0035
 18      0.0115     0.6357     8.068     -0.0032
 19     0.00307     0.4810     2.336     -0.0022
 20     0.00108     0.6043     3.287     -0.0006
 21    0.000518     0.0289     2.329     -0.0004
 22    0.000075     0.0365     1.772     -0.0001
 23    1.897E-6    0.00158     1.159     -331E-8
 24    2.394E-8   0.000016     0.967      -46E-9
 25    1.14E-11   7.962E-7     1.061     -19E-13


            Optimization Results

Iterations                           25  Function Calls                   120
Gradient Calls                      107  Active Constraints                 0
Objective Function         3.953804E-11  Max Abs Gradient Element  7.9622469E-7
Slope of Search Direction  -1.88032E-12


   ABSGCONV convergence criterion satisfied.


            Optimization Results
            Parameter Estimates

                                    Gradient
                                    Objective
N Parameter         Estimate        Function

1 X1                0.999991    -0.000000796
2 X2                0.999982     0.000000430

     Value of Objective Function = 3.953804E-11
```

Nonlinearly Constrained QN Optimization

The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's (1978, 1982) Variable Metric Constrained WatchDog (VMCWD) algorithm. A similar but older algorithm (VF02AD) is part of the Harwell library. Both the VMCWD and VF02AD algorithms use Fletcher's VE02AD algorithm, which is also part of the Harwell library, for positive definite quadratic programming. This NLPQN implementation uses a quadratic programming subroutine that updates and downdates the Cholesky factor when the active set changes (refer to Gill, Murray, Saunders, and Wright 1984). The nonlinear NLPQN algorithm is not a feasible point algorithm, and the value of the objective function is not required to decrease monotonically. Instead, the algorithm tries to reduce a linear combination of objective function and constraint violations.

The following are similarities and differences between this algorithm and Powell's VMCWD algorithm:

- You can use the sixth element of the *opt* argument to modify the algorithm used by the NLPQN subroutine. If you specify $opt[6] = 2$, which is the default, the evaluation of the Lagrange vector μ is performed the same way as described in Powell (1982b). Note, however, that the VMCWD program seems to have a bug in the implementation of formula (4.4) in Powell (1982b). If you specify $opt[6] = 1$, the original update of μ used in the VF02AD algorithm in Powell (1978a) is performed.
- Instead of updating an approximate Hessian matrix, this algorithm uses the dual BFGS or dual DFP update that updates the Cholesky factor of an approximate Hessian. If the condition of the updated matrix gets too bad, a restart is done with a positive diagonal matrix. At the end of the first iteration after each restart, the Cholesky factor is scaled.
- The Cholesky factor is loaded into the quadratic programming subroutine, which ensures positive definiteness of the problem. During the quadratic programming step, the Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ is updated simultaneously with QT decomposition when the active set changes. Refer to Gill, Murray, Saunders, and Wright (1984) for more information.
- The line-search strategy is very similar to that of Powell's algorithm, but this algorithm does not call for derivatives during the line search. For that reason, this algorithm generally needs fewer derivative calls than function calls, whereas the VMCWD algorithm always requires the same number of derivative calls as function calls. Also, Powell's line-search method sometimes uses steps that are too long during the early iterations. In those cases, you can use the second element of the *par* argument to restrict the step length α in the first five iterations. See "Control Parameters Vector" for more details.
- The watchdog strategy is also similar to that of Powell's algorithm. However, this algorithm does not return automatically after a fixed number of iterations to a previous, more optimal point. A return to such a point is further delayed if the observed function reduction is close to the expected function reduction of the quadratic model.
- Although Powell's termination criterion, the FTOL2 criterion, can still be used, the NLPQN implementation uses, by default, two other termination criteria (GTOL and ABSGTOL).

This algorithm is automatically invoked if the *"nlc"* argument is specified. The module specified with the *"nlc"* argument must return a vector of length nc, where nc is the total number of constraints. Letting nec be the number of equality constraints, the constraints must be of the following form:

$$\begin{aligned} c_i(x) &= 0, \quad i = 1, \ldots, nec \\ c_i(x) &\geq 0, \quad i = nec + 1, \ldots, nc \end{aligned}$$

The first nec elements of the returned vector contain the c_i for the equality constraints, and the remaining elements contain the c_i for the inequality constraints.

Note: You must specify the total number of constraints with the tenth element of the *opt* argument, and if there are any equality constraints, you must specify that number, nec, with the eleventh element of the *opt* argument.

The nonlinear NLPQN algorithm requires the Jacobian matrix of the first-order derivatives of the nc constraints returned by the module specified by the *"nlc"* argument. You can provide these derivatives by specifying a module with the *"jacnlc"* argument. This module must return the Jacobian matrix **J** of first-order partial derivatives. That is, **J** is an $nc \times n$ matrix such that the entry in the i^{th} row and j^{th} column is given by

$$\mathbf{J}(i,j) = \frac{\partial c_i}{\partial x_j}$$

If you specify an *"nlc"* module without specifying a *"jacnlc"* argument, finite difference approximations of the first-order derivatives of the constraints are used. You can use the ninth element of the *par* argument to specify the number of accurate digits used in evaluating the constraints.

You can specify two update formulas with the fourth element of the *opt* argument as indicated in the following table:

Value of *opt*[4]	**Update Method**
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.

This algorithm uses its own line-search technique. None of the options and parameters that control the line search in the other algorithms apply in the nonlinear NLPQN algorithm, with the exception of the second element of the *par* vector, which can be used to restrict the length of the step size in the first five iterations.

See Example 11.8 for an example where you need to specify a value for the second element of the *par* argument. The values of the fourth, fifth, and sixth elements of the *par* vector, which control the processing of linear and boundary constraints, are valid only for the quadratic programming subroutine used in each iteration of the NLPQN call. For a simple example of the NLPQN subroutine, see "Rosen-Suzuki Problem".

NLPQUA Call

nonlinear optimization by quadratic method

CALL NLPQUA(*rc, xr, quad, x0* <*,opt, blc, tc, par, "ptit", lin*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPQUA subroutine uses a fast algorithm for maximizing or minimizing the quadratic objective function

$$\frac{1}{2}x^T Gx + g^T x + con$$

subject to boundary constraints and general linear equality and inequality constraints. The algorithm is memory-consuming for problems with general linear constraints.

The matrix G must be symmetric but not necessarily positive definite (or negative definite for maximization problems). The constant term con affects only the value of the objective function, not its derivatives or the optimal point x^*.

The algorithm is an active-set method in which the update of active boundary and linear constraints is done separately. The QT decomposition of the matrix A_k of active linear constraints is updated iteratively (refer to Gill, Murray, Saunders, and Wright, 1984). If n_f is the number of free parameters (that is, n minus the number of active boundary constraints), and n_a is the number of active linear constraints, then Q is an $n_f \times n_f$ orthogonal matrix containing null space Z in its first $n_f - n_a$ columns and range space Y in its last n_a columns. The matrix T is an $n_a \times n_a$ triangular matrix of the form $t_{ij} = 0$ for $i < n - j$. The Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ is updated simultaneously with the QT decomposition when the active set changes.

The objective function is specified by the input arguments *quad* and *lin*, as follows:

- The *quad* argument specifies the symmetric $n \times n$ Hessian matrix, G, of the quadratic term. The input can be in dense or sparse form. In dense form, all n^2 entries of the *quad* matrix must be specified. If $n \leq 3$, the dense specification must be used. The sparse specification can be useful when G has many zero elements. You can specify an $nn \times 3$ matrix in which each row represents one of the nn nonzero elements of G. The first column specifies the row location in G, the second column specifies the column location, and the third column specifies the value of the nonzero element.
- The *lin* argument specifies the linear part of the quadratic optimization problem. It must be a vector of length n or $n + 1$. If *lin* is a vector of length n, it specifies the vector g of the linear term, and the constant term con is considered zero. If *lin* is a vector of length $n + 1$, then the first n elements of the argument specify the vector g and the last element specifies the constant term con of the objective function.

As in the other optimization subroutines, you can use the *blc* argument to specify boundary and general linear constraints, and you must provide a starting point *x0* to determine the number of parameters. If *x0* is not feasible, a feasible initial point is computed by linear programming, and the elements of *x0* can be missing values.

Assuming nonnegativity constraints $x \geq 0$, the quadratic optimization problem solved with the LCP call, which solves the linear complementarity problem. Refer to *SAS/IML Software: Usage and Reference, Version 6, First Edition* for details.

Choosing a sparse (or dense) input form of the *quad* argument does not mean that the algorithm used in the NLPQUA subroutine is necessarily sparse (or dense). If the following conditions are satisfied, the NLPQUA algorithm will store and process the matrix G as sparse:

- No general linear constraints are specified.
- The memory needed for the sparse storage of G is less than 80% of the memory needed for dense storage.
- G is not a diagonal matrix. If G is diagonal, it is stored and processed as a diagonal matrix.

The sparse NLPQUA algorithm uses a modified form of minimum degree Cholesky factorization (George and Liu 1981).

In addition to the standard iteration history, the NLPNRA subroutine prints the following information:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The Betts problem (see "Constrained Betts Function") can be expressed as a quadratic problem in the following way:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad G = \begin{bmatrix} 0.02 & 0 \\ 0 & 2 \end{bmatrix}, \quad g = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad con = -100$$

Then

$$\frac{1}{2}x^T Gx - g^T x + con = 0.5[0.02x_1^2 + 2x_2^2] - 100 = 0.01x_1^2 + x_2^2 - 100$$

The following statements use the NLPQUA subroutine to solve the Betts problem:

```
proc iml;
   lin  = { 0. 0. -100};
   quad = {  0.02   0.0 ,
             0.0    2.0 };
   c    = {  2. -50.  .   .,
            50.  50.  .   .,
            10.  -1. 1. 10.};
   x = { -1. -1.};
   optn = {0 2};
   CALL NLPQUA(rc,xres,quad,x,optn,c,,,,lin);
```

The *quad* argument specifies the G matrix, and the *lin* argument specifies the g vector with the value of *con* appended as the last element. The matrix C specifies the boundary constraints and the general linear constraint.

The iteration history follows.

```
                      Optimization Start
                      Parameter Estimates
                                        Gradient          Lower          Upper
                                       Objective          Bound          Bound
 N Parameter         Estimate          Function      Constraint     Constraint

 1 X1                6.800000          0.136000        2.000000      50.000000
 2 X2               -1.000000         -2.000000      -50.000000      50.000000

        Value of Objective Function = -98.5376


                    Linear Constraints

 1   59.00000 :  10.0000 <= + 10.0000 * X1 - 1.0000 * X2


               Null Space Method of Quadratic Problem

                Parameter Estimates               2
                Lower Bounds                      2
                Upper Bounds                      2
                Linear Constraints                1
                Using Sparse Hessian              _

                 Optimization Start

  Active Constraints        0  Objective Function  -98.5376
  Max Abs Gradient Element  2


                   Function        Active       Objective
 Iter    Restarts      Calls   Constraints        Function

    1           0          2             1       -99.87349
    2           0          3             1       -99.96000

        Objective    Max Abs              Slope of
         Function   Gradient      Step      Search
 Iter      Change    Element      Size   Direction

    1      1.3359     0.5882     0.706      -2.925
    2      0.0865          0     1.000      -0.173


                         Optimization Results

Iterations                           2  Function Calls             4
Gradient Calls                       3  Active Constraints         1
Objective Function             -99.96  Max Abs Gradient Element   0
Slope of Search Direction  -0.173010381

   ABSGCONV convergence criterion satisfied.
```

```
                    Optimization Results
                    Parameter Estimates

                                       Gradient    Active
                                       Objective    Bound
 N Parameter         Estimate          Function    Constraint

 1 X1                2.000000          0.040000     Lower BC
 2 X2                       0                 0

            Value of Objective Function = -99.96


         Linear Constraints Evaluated at Solution

 1    10.00000 = -10.0000 + 10.0000 * X1 - 1.0000 * X2
```

NLPTR Call

nonlinear optimization by trust region method

CALL NLPTR(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd", "hes"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPTR subroutine is a trust-region method that uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$. It requires that the objective function $f = f(x)$ has continuous first- and second-order derivatives inside the feasible region.

The $n \times n$ Hessian matrix **G** contains the second derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$G(x) = \nabla^2 f(x) = \left(\frac{\partial^2 f}{\partial x_j \partial x_k} \right)$$

The trust-region method works by optimizing a quadratic approximation to the nonlinear objective function within a hyperelliptic trust region. This trust region has a radius, Δ, that constrains the step size corresponding to the quality of the quadratic approximation. The method is implemented using Dennis, Gay, and Welsch (1981), Gay (1983), and Moré and Sorensen (1983).

Note that finite difference approximations for second-order derivatives using only function calls are computationally very expensive. If you specify first-order derivatives analytically with the *"grd"* module argument, you can drastically reduce the computation time for numerical second-order derivatives. Computing the finite difference approximation for the Hessian matrix **G** generally uses only n calls of the module that computes the gradient analytically.

The NLPTR method performs well for small- to medium-sized problems and does not need many function, gradient, and Hessian calls. However, if the gradient is not specified analytically by using the *"grd"* argument or if the computation of the Hessian module, as specified by the *"hes"* module argument, is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms may be more efficient.

In addition to the standard iteration history, the NLPTR subroutine prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that the computed Hessian approximation was singular and had to be ridged with a positive value.
- The heading *lambda* represents the Lagrange multiplier, λ. This has a value of zero when the optimum of the quadratic function approximation is inside the trust region, in which case a trust-region-scaled Newton step is performed. It is greater than zero when the optimum is at the boundary of the trust region, in which case the scaled Newton step is too long to fit in the trust region and a quadratically-constrained optimization is done. Large values indicate optimization difficulties, and as in Gay (1983), a negative value indicates the special case of an indefinite Hessian matrix.
- The heading *radius* refers to Δ, the radius of the trust region. Small values of the radius combined with large values of λ in subsequent iterations indicate optimization problems.

For an example of the use of the NLPTR subroutine, see “Unconstrained Rosenbrock Function”.

NORMAL Function

generates a pseudo-random normal deviate

NORMAL(*seed*)

where *seed* is a numeric matrix or literal. The *seed* argument can be any integer value up to $2^{31} - 1$.

The NORMAL function is a scalar function that returns a pseudo-random number having a normal distribution with a mean of 0 and a standard deviation of 1. The NORMAL function returns a matrix with the same dimensions as the argument. The first argument on the first call is used for the seed (or if that is 0, the system clock is used for the seed). This function is synonymous with the DATA step function RANNOR. The Box-Muller transformation of the UNIFORM function deviates is used to generate the numbers.

NROW Function

finds the number of rows of a matrix

NROW(*matrix***)**

where *matrix* is a numeric or character matrix.

The NROW function returns a single numeric value that is the number of rows in matrix. If the matrix has not been given a value, the NROW function returns a value of 0.

For example, to let J contain the number of rows of the matrix **S**, use the statement

```
j=nrow(s);
```

NUM Function

produces a numeric representation of a character matrix

NUM(*matrix***)**

where *matrix* is a character matrix or a quoted literal.

The NUM function takes as an argument a character matrix with elements that are character numerics; and produces a numeric matrix with dimensions that are the same as the dimensions of the argument and with elements that are the numeric representations (double-precision floating-point) of the corresponding elements of the argument.

An example using the NUM function is shown below:

```
c={'1' '2' '3'};
j=num(c);
```

```
        C        1 row       3 cols    (character, size 1)

                                1 2 3


        J        1 row       3 cols    (numeric)
                    1         2         3
```

See also the description of the CHAR function, which does the reverse conversion.

ODE Call

performs numerical integration of vector differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}(t)) \quad \text{with } \mathbf{y}(0) = \mathbf{c}$$

CALL ODE(*r*, *"dername"*, *c*, *t*, *h* <, **J=***"jacobian"*><, **EPS=***eps*><, *"data"*>**);**

The ODE subroutine returns the following values:

r is a numeric matrix that contains the results of the integration over connected subintervals. The number of columns in r is equal to the number of subintervals of integration as defined by the argument t. In case of any error in the integration on any subinterval, partial results will not be reported in r.

The inputs to the ODE subroutine are as follows:

"dername" specifies the name of an IML module used to evaluate the integrand.

c specifies an initial value vector for the variable $\mathbf{y}$.

t specifies a sorted vector that describes the limits of integration over connected subintervals. The simplest form of the vector t contains only the limits of the integration on one interval. The first component of t should contain the initial value, and the second component should be the final value of the independent variable. For more advanced usage of the ODE subroutine, the vector t can contain more than two components. The components of the vector must be sorted in ascending order. Two consecutive components of the vector t are interpreted as a subinterval. The ODE call reports the final result of integration at the right endpoint of each subinterval. This information is vital if $f(\cdot)$ has internal points of discontinuity. To produce accurate solutions, it is essential that you provide the location of these points in the variable t, since the continuity of the forcing function is vital to the internal control of error.

h specifies a numeric vector that contains three components: the minimum allowable step size, *hmin*; the maximum allowable step size, *hmax*; and the initial step size to start the integration process, *hinit*.

"jacobian" optionally specifies the name of an IML module that is used to evaluate the Jacobian analytically. The Jacobian is the matrix J, with

$$J_{ij} = \frac{\partial f_i}{\partial y_j}$$

If the *"jacobian"* module is not specified, the ODE call uses a finite difference method to approximate the Jacobian. The keyword for this option is J.

eps specifies a scalar indicating the required accuracy. It has a default value of 1E−4. The keyword for this option is EPS.

data (scalar, optional, character, input) a valid predefined SAS Dataset name that is used to save the successful independent and dependent variables of the integration at each step.

The ODE subroutine is an adaptive, variable order, variable step-size, stiff integrator based on implicit backward-difference methods. Refer to Aiken (1985), Bickart and Picel (1973), Donelson and Hansen (1971), Gaffney (1984), and Shampine (1978). The integrator is an implicit predictor-corrector method that locally attempts to maintain the prescribed precision *eps* relative to

$$d = \max_{0 \le t \le T} (\|y(t)\|_\infty, 1)$$

As you can see from the expression, this quantity is dynamically updated during the integration process and can help you to understand the validity of the results reported by the subroutine.

Consider the differential equation

$$\frac{dy}{dt} = -ty \text{ with } y = 0.5 \text{ at } t = 0$$

The following statements attempt to find the solution at $t = 1$:

```
proc iml; ;
  /* Define the integrand */
  start fun(t,y);
      v = -t*y;
      return(v);
  finish;

  /* Call ODE */
  c   = .5;
  t   = { 0 1};
  h   = { 1E-12 1 1E-5};
  call ode(r,"FUN",c,t,h);
  print r[format=E21.14];
```

In this case, the integration is carried out over $(0, 1)$ to give the value of y at $t = 1$. The optional parameter *eps* has not been specified, so it is internally set to 1E−4. Also, the optional parameter *"jacobian"* has not been specified, so finite difference methods are used to estimate the Jacobian. The accuracy of the answer can be increased by specifying *eps*. For example, set *eps*=1E−7 as follows.

```
proc iml;
  /* Define the integrand */
  start fun(t,y);
      v = -t*y;
      return(v);
  finish;

  /* Call ODE */
  c   = .5;
  t   =  {0 1};
  h   =  {1E-12 1. 1E-5};
  call ode(r,"FUN",c,t,h) eps=1E-7;

  print r[format =E21.14];
```

Compare this value to $0.5e^{-0.5} = 3.03265329856310\text{E}{-}01$ and observe that the result is correct through the sixth decimal digit and has an error relative to 1 that is $O(1\text{E}{-}7)$.

If the solution was desired at 1 and 2 with an accuracy of $1\text{E}{-}7$, you would use the following statements:

```
proc iml;
  /* Define the integrand */
  start fun(t,y);
      v = -t*y;
      return(v);
  finish;

  /* Call ODE */
  c   = .5;
  t   = { 0 1 2};
  h   = { 1E-12 1. 1E-5};
  call ode(r,"FUN",c,t,h) eps=1E-7;

  print r[format=E21.14];
```

Note that R contains the solution at $t = 1$ in the first column and at $t = 2$ in the second column.

Now consider the smoothness of the forcing function $f(\cdot)$. For the purpose of estimating errors, adaptive methods require some degree of smoothness in the function $f(\cdot)$. If this smoothness is not present in $f(\cdot)$ over the interior and including the left endpoint of the subinterval, the reported result will not have the desired accuracy. The function $f(\cdot)$ must be at least continuous. If the function does not meet this requirement, you should specify the discontinuity as an intermediate point. For example, consider the differential equation

$$\frac{dy}{dt} = \begin{cases} t & \text{if } t < 1 \\ 0.5t^2 & \text{if } t \geq 1 \end{cases}$$

To find the solution at $t = 2$, use the following statements:

```
proc iml;
  /* Define the integrand */
  start fun(t,y);
      if t < 1 then v = t;
      else   v = .5*t*t;
      return(v);
  finish;

  /* Call ODE */
  c   = 0;
  t   = { 0 2};
  h   = { 1E-12 1. 1E-5};
  call ode(r,"FUN",c,t,h) eps = 1E-12;
  print r[format =E21.14];
```

In the preceding case, the integration is carried out over a single interval, $(0, 2)$. The optional parameter *eps* is specified to be $1\mathrm{E}{-12}$. The optional parameter *jacobian* is not specified, so finite difference methods are used to estimate the Jacobian.

Note that the value of R does not have the required accuracy (it should contain a 12 decimal-place representation of 5/3), although no error message is produced. The reason is that the function is not continuous at the point $t = 1$. Even the lowest-order method cannot produce a local reliable error estimate near the point of discontinuity. To avoid this problem, you can create subintervals so that the integration is carried out first over $(0, 1)$ and then over $(1, 2)$. The following code implements this method:

```
proc iml;
  /* Define the integrand */
  start fun(t,y);
      if t < 1 then v = t;
      else          v = .5*t*t;
      return(v);
  finish;

  /* Call ODE */
  c   = 0;
  t   = { 0 1 2};
  h   = { 1E-12 1. 1E-5};
  call ode(r,"FUN",c,t,h) eps=1E-12;
  print r[format=E21.14];
```

The variable R contains the solutions at both $t = 1$ and $t = 2$, and the errors are of the specified order. Although there is no interest in the solution at the point $t = 1$, the advantage of specifying subintervals with no discontinuities is that the function $f(\cdot)$ is infinitely differentiable in each subinterval.

When $f(\cdot)$ is continuous, the ODE subroutine can compute the integration to the specified precision, even if the function is defined piecewise. Consider the differential equation

$$\frac{dy}{dt} = \begin{cases} t & \text{if } t < 1 \\ t^2 & \text{if } t \geq 1 \end{cases}$$

The following code finds the solution at $t = 2$: Since the function $f(\cdot)$ is continuous, the requirements for error control are satisfied.

```
proc iml;
  /* Define the integrand */
  start fun(t,y);
      if t < 1 then v = t;
      else          v = t*t;
      return(v);
  finish;

  /* Call ODE */
  c   = .5;
  t   = { 0 2};
  h   = { 1E-12 1. 1E-5};
  call ode(r,"FUN",c,t,h) eps=1E-12;

  print r[format=E21.14];
```

This example compares the ODE call to an eigenvalue decomposition for stiff-linear systems. In the problem

$$\frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y} \text{ with } \mathbf{y}(0) = \mathbf{c}$$

where $\mathbf{A}$ is a symmetric constant matrix, the solution can be written in terms of the eigenvalue decomposition, as follows:

$$\mathbf{y}(t) = \mathbf{U}e^{\mathbf{D}t}\mathbf{U}'\mathbf{c}$$

where $\mathbf{U}$ is the matrix of eigenvectors and $\mathbf{D}$ is a diagonal matrix with the eigenvalues on its diagonal.

The following statements produce two solutions, one using the ODE call and the other using the eigenvalue decomposition:

```
proc iml;
   /* Define the integrand */
   start fun(t,x) global(a,count);
      count = count+1;
      v = a*x;
      return(v);
   finish;
```

```
   /* Define the Jacobian */
   start jac(t,x) global(a);
      v = a;
      return(v);
   finish;

   a = {-1000 -1 -2 -3,
           -1 -2  3 -1,
           -2  3 -4 -3,
           -3 -1 -3 -5 };

   /* Call ODE */
   count = 0;
   t = { 0 1 2};
   h = {1E-12  1 1E-5};
   eps = 1E-9;
   c = {1, 0, 0, 0 };
   call ode(z,"FUN",c,t,h)  eps=eps j="JAC";
   print z[format=E21.14];


  print count;


/* Do the eigenvalue decomposition */
start eval(t) global(d,u,c);
   v = u*diag(exp(d*t))*u'*c;
   return(v);
finish;

call eigen(d,u,a);
free z1;
do i = 1 to nrow(t)*ncol(t)-1;
  z1 = z1 || (eval(t[i+1]));
end;
print z1[format=E21.14];
```

The question now is whether this is an $O(1\text{E}{-}9)$ result. Note that for this problem

$$d = \max_{0 \le t \le T}(\|y(t)\|_\infty, 1) = 1$$

with the 1E−6 result, the ODE call should attempt to maintain an accuracy of 1E−9 relative to 1. Therefore, the 1E−6 result should have almost three correct decimal places. At $t = 2$, the first component of $\mathbf{Z}$ is 6.58597048842310E−06, while its more accurate value is 6.58580950203220E−06, showing an $O(1\text{E}{-}10)$ error.

The ODE subroutine may fail for problems with unusual qualitative properties, such as finite escape time in the interval of integration (that is, the solution goes towards infinity at some finite time). In such cases, try testing with different subintervals and different levels of accuracy to gain some qualitative information about the behavior of the solution of the differential equation.

OPSCAL Function

rescales qualitative data to be a least-squares fit to qualitative data

OPSCAL(*mlevel, quanti*<*, qualit*>**)**

The inputs to the OPSCAL function are as follows:

mlevel specifies a scalar that has one of two values. When *mlevel* is 1 the *qualit* matrix is at the nominal measurement level; when *mlevel* is 2 it is at the ordinal measurement level.

quanti specifies an $m \times n$ matrix of quantitative information assumed to be at the interval level of measurement.

qualit specifies an $m \times n$ matrix of qualitative information whose level of measurement is specified by *mlevel*. When *qualit* is omitted, *mlevel* must be 2. When omitted, a temporary *qualit* is constructed that contains the integers from 1 to n in the first row, from $n + 1$ to $2n$ in the second row, from $2n + 1$ to $3n$ in the third row, and so forth, up to the integers $(m - 1)n$ to mn in the last(mth) row. Note that you cannot specify *qualit* as a character matrix.

The result of the OPSCAL function is the optimal scaling transformation of the qualitative (nominal or ordinal) data in *qualit*. The optimal scaling transformation result

- is a least-squares fit to the quantitative data in *quanti*
- preserves the qualitative measurement level of *qualit*

The OPSCAL function performs as a function or call. When used as a call, the first argument of the call is the matrix to contain the result returned.

When *qualit* is at the nominal level of measurement, the optimal scaling transformation result is a least-squares fit to *quanti*, given the restriction that the category structure of *qualit* must be preserved. If element i of *qualit* is in category c, then element i of the optimum scaling transformation result is the mean of all those elements of *quanti* that correspond to elements of *qualit* that are in category c.

For example, consider these statements:

```
quanti={5  4  6  7  4  6  2  4  8  6};
qualit={6  6  2 12  4 10  4 10  8  6};
os=opscal(1,quanti,qualit);
```

The resulting vector **OS** has the following values:

```
          OS        1 row     10 cols    (numeric)

         5      5     6       7       3       5       3
:        5      8      5
```

The optimal scaling transformation result is said to preserve the nominal measurement level of *qualit* because wherever there was a *qualit* category c, there is now a result category label v. The transformation is least squares because the result element v is the mean of appropriate elements of *quanti*. This is Young's (1981) discrete-nominal transformation.

When *qualit* is at the ordinal level of measurement, the optimal scaling transformation result is a least-squares fit to *quanti*, given the restriction that the ordinal structure of *qualit* must be preserved. This is done by determining blocks of elements of *qualit* so that if element i of *qualit* is in block b, then element i of the result is the mean of all those *quanti* elements corresponding to block b elements of *qualit* so that the means are (weakly) in the same order as the elements of *qualit*. For example, consider these statements:

```
quanti={5  4  6  7  4  6  2  4  8  6};
qualit={6  6  2 12  4 10  4 10  8  6};
os=opscal(2,quanti,qualit);
```

The resulting vector **OS** has the following values:

```
          OS      1 row    10 cols  (numeric)

     5        5       4       7      4      6      4
:    6        6       5
```

This transformation preserves the ordinal measurement level of *qualit* because the elements of *qualit* and the result are (weakly) in the same order. It is least-squares because the result elements are the means of appropriate elements of *quanti*. By comparing this result to the nominal one, you see that categories whose means are incorrectly ordered have been merged together to form correctly ordered blocks. This is known as Kruskal's (1964) least-squares monotonic transformation. Consider the following statements:

```
quanti={5  3  6  7  5  7  8  6  7  8};
os=opscal(2,quanti);
```

These statements imply that

```
qualit={ 1  2  3  4  5  6  7  8  9 10} ;
```

which means that the resulting vector has the values

```
         OS       1 row   10 cols    (numeric)

     4     4       6       6        6      7      7
:    7     7       8
```

ORPOL Function

generates orthogonal polynomials

ORPOL(*vector*<, *maxdegree*<, *weights*>>**)**

The inputs to the ORPOL function are as follows:

vector is an $n \times 1$ (or $1 \times n$) vector of values over which the polynomials are to be defined.

maxdegree specifies the maximum degree polynomial to be computed. Note that the number of columns in the computed result is $1+$*maxdegree*, whether *maxdegree* is specified or the default value is used. If *maxdegree* is omitted, IML uses the default value $\min(n, 19)$. If *weights* is specified, *maxdegree must* also be specified.

weights specifies an $n \times 1$ (or $1 \times n$) vector of nonnegative weights to be used in defining orthogonality:

$$\mathbf{P}' * \text{diag}(\textit{weights})\mathbf{P} = \mathbf{I} \, .$$

If you specify *weights*, you *must* also specify *maxdegree*. If *maxdegree* is not specified or is specified incorrectly, the default weights (all weights are 1) are used.

The ORPOL matrix function generates orthogonal polynomials. The result is a column-orthonormal matrix $\mathbf{P}$ with the same number of rows as the vector and with *maxdegree*$+1$ columns:

$$\mathbf{P}'\text{diag}(\textit{weights})\mathbf{P} = \mathbf{I} \, .$$

The result is computed such that $\mathbf{P}[i, j]$ is the value of a polynomial of degree $j - 1$ evaluated at the ith element of the vector.

The maximum number of nonzero orthogonal polynomials (r) that can be computed from the vector and the weights is the number of distinct values in the vector, ignoring any value associated with a zero weight.

The polynomial of maximum degree has degree of $r - 1$. If the value of *maxdegree* exceeds $r - 1$, then columns $r + 1$, $r + 2$,..., *maxdegree*$+1$ of the result are set to 0. In this case,

$$\mathbf{P}'\text{diag}(\textit{weights})\mathbf{P} = \begin{bmatrix} I(r) & 0 \\ 0 & 0 * \mathbf{J}(\textit{maxdegree} + 1 - r) \end{bmatrix}$$

The statement below results in a matrix with 3 orthogonal columns:

```
orpol=orpol(1:5,2);

              ORPOL         5 rows      3 cols    (numeric)

                      0.4472136 -0.632456 0.5345225
                      0.4472136 -0.316228 -0.267261
                      0.4472136         0 -0.534522
                      0.4472136 0.3162278 -0.267261
                      0.4472136 0.6324555 0.5345225
```

See "Acknowledgments" in the front of this book for authorship of the ORPOL function.

ORTVEC Call

provides columnwise orthogonalization by the Gram-Schmidt process and stepwise QR decomposition by the Gram-Schmidt process

CALL ORTVEC(w, r, ρ, *lindep*, v <, q>**);**

The ORTVEC subroutine returns the following values:

w If the Gram-Schmidt process converges (*lindep*=0), w is the $m \times 1$ vector $\mathbf{w}$ orthonormal to the columns of $\mathbf{Q}$, which is assumed to have $n \leq m$ (nearly) orthonormal columns. If the Gram-Schmidt process does not converge (*lindep*=1), w is a vector of missing values. For stepwise QR decomposition, w is the $(n+1)^{\text{th}}$ orthogonal column of the matrix $\mathbf{Q}$. If there is no matrix $\mathbf{Q}$, that is, if the q argument is not specified, w is the normalized value of the vector $\mathbf{v}$,

$$\mathbf{w} = \frac{\mathbf{v}}{\sqrt{\mathbf{v}'\mathbf{v}}}$$

r If the Gram-Schmidt process converges (*lindep*=0), r specifies the $n \times 1$ vector $\mathbf{r}$ of Fourier coefficients. If the Gram-Schmidt process does not converge (*lindep*=1), r is a vector of missing values. If the q argument is not specified, r is a vector with zero dimension. For stepwise QR decomposition, r contains the n upper triangular elements of the $(n+1)^{\text{th}}$ column of $\mathbf{R}$.

ρ If the Gram-Schmidt process converges (*lindep*=0), ρ specifies the distance from $\mathbf{w}$ to the range of $\mathbf{Q}$. Even if the Gram-Schmidt process converges, if ρ is sufficiently small, the vector $\mathbf{v}$ may be linearly dependent on the columns of $\mathbf{Q}$. If the Gram-Schmidt process does not converge (*lindep*=1), ρ is set to 0. For stepwise QR decomposition, ρ contains the diagonal element of the $(n+1)^{\text{th}}$ column of $\mathbf{R}$.

lindep returns a value of 1 if the Gram-Schmidt process does not converge in 10 iterations. In most cases, if *lindep*=1, the input vector $\mathbf{v}$ is linearly dependent on the n columns of the input matrix $\mathbf{Q}$. In that case, ρ is set to 0, and the results w and r contain missing values. If *lindep*=0, the Gram-Schmidt process did converge, and the results w, r, and ρ are computed.

The inputs to the ORTVEC subroutine are as follows:

v specifies an $m \times 1$ vector $\mathbf{v}$ that is to be orthogonalized to the n columns of $\mathbf{Q}$. For stepwise QR decomposition of a matrix, $\mathbf{v}$ is the $(n+1)^{\text{th}}$ matrix column before its orthogonalization.

q specifies an optional $m \times n$ matrix $\mathbf{Q}$ that is assumed to have $n \leq m$ (nearly) orthonormal columns. Thus, the $n \times n$ matrix $\mathbf{Q}'\mathbf{Q}$ should approximate the identity matrix. The column orthonormality assumption is not tested in the ORTVEC call. If it is violated, the results are not predictable. The argument q can be omitted or can have zero rows and columns. For stepwise QR decomposition of a matrix, q contains the first n matrix columns that are already orthogonal.

The relevant formula for the ORTVEC subroutine is

$$\mathbf{v} = \mathbf{Q}\mathbf{r} + \rho\mathbf{w}$$

Assuming that the $m \times n$ matrix $\mathbf{Q}$ has n (nearly) orthonormal columns, the ORTVEC subroutine orthogonalizes the vector $\mathbf{v}$ to the columns of $\mathbf{Q}$. The vector $\mathbf{r}$ is the array of Fourier coefficients, and ρ is the distance from $\mathbf{w}$ to the range of $\mathbf{Q}$.

There are two special cases:

- If $m > n$, ORTVEC normalizes the result $\mathbf{w}$, so that $\mathbf{w}'\mathbf{w} = 1$.
- If $m = n$, the output vector $\mathbf{w}$ is the null vector.

The case $m < n$ is not possible since $\mathbf{Q}$ is assumed to have n (nearly) orthonormal columns.

To initialize a stepwise QR decomposition, ORTVEC can be called to normalize $\mathbf{v}$ only, that is, to compute $\mathbf{w} = \mathbf{v}/\sqrt{\mathbf{v}'\mathbf{v}}$ and $\rho = \sqrt{\mathbf{v}'\mathbf{v}}$ only. There are two ways of using the ORTVEC call for this reason:

- Omit the last argument q, as in `call ortvec(w,r,rho,lindep,v);`.
- Provide a matrix $\mathbf{Q}$ with zero rows and columns, for example, by using the `free q;` command.

In both cases, $\mathbf{r}$ is a column vector with zero rows.

The ORTVEC subroutine is useful for the following applications:

- performing stepwise QR decomposition. Compute $\mathbf{Q}$ and $\mathbf{R}$, so that $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is column orthonormal, $\mathbf{Q}'\mathbf{Q} = \mathbf{I}$, and $\mathbf{R}$ is upper triangular. The j^th step is applied to the j^th column, $\mathbf{v}$, of $\mathbf{A}$, and it computes the j^th column $\mathbf{w}$ of $\mathbf{Q}$ and the j^th column, $(\mathbf{r}\ \rho\ 0)'$, of $\mathbf{R}$.
- computing the $m \times (m - n)$ null space matrix, $\mathbf{Q}_2$, corresponding to an $m \times n$ range space matrix, $\mathbf{Q}_1$ $(m > n)$, by the following stepwise process: set $\mathbf{v} = \mathbf{e}_i$ (where $\mathbf{e}_i$ is the i^th unit vector) and try to make it orthogonal to all column vectors of $\mathbf{Q}_1$ and the already generated $\mathbf{Q}_2$, if the subroutine is successful, append $\mathbf{w}$ to $\mathbf{Q}_2$; otherwise, try $\mathbf{v} = \mathbf{e}_{i+1}$.

The 4×3 matrix $\mathbf{Q}$ contains the unit vectors $\mathbf{e}_1, \mathbf{e}_3$, and $\mathbf{e}_4$. The column vector $\mathbf{v}$ is pairwise linearly independent with the three columns of $\mathbf{Q}$. As expected, the ORTVEC call computes the vector $\mathbf{w}$ as the unit vector $\mathbf{e}_2$ with $\mathbf{u} = (1, 1, 1)$ and $\rho = 1$.

```
proc iml;
   q = { 1  0  0,
         0  0  0,
         0  1  0,
         0  0  1 };
   v = { 1, 1, 1, 1 };
   call ortvec(w,u,rho,lindep,v,q);
   print rho u w;
```

You can perform the QR decomposition of the linearly independent columns of an $m \times n$ matrix $\mathbf{A}$ with the following statements:

```
proc iml;
   a = { . . . enter matrix A here . . . };
   nind = 0;  ndep = 0;  dmax = 0.;
   n = ncol(a);  m = nrow(a);
   free q;
   do j = 1 to n;
      v = a[ ,j];
      call ORTVEC(w,u,rho,lindep,v,q);
      aro = abs(rho);
      if aro > dmax then dmax = aro;
      if aro <= 1.e-10 * dmax then lindep = 1;
      if lindep = 0 then do;
         nind = nind + 1;
         q = q || w;
         if nind = n then r = r || (u // rho);
         else r = r || (u // rho // j(n-nind,1,0.));
      end;
      else do;
         print "Column " j " is linearly dependent.";
         ndep = ndep + 1; ind[ndep] = j;
      end;
   end;
```

Next, process the remaining columns of **A**:

```
do j = 1 to ndep;
   k = ind[ndep-j+1];
   v = a[ ,k];
   call ORTVEC(w,u,rho,lindep,v,q);
   if lindep = 0 then do;
      nind = nind + 1;
      q = q || w;
      if nind = n then r = r || (u // rho);
      else r = r || (u // rho // j(n-nind,1,0.));
   end;
end;
```

Now compute the null space in the last columns of **Q**:

```
do i = 1 to m;
   if nind < m then do;
     v = j(m,1,0.); v[i] = 1.;
     call ORTVEC(w,u,rho,lindep,v,q);
     aro = abs(rho);
     if aro > dmax then dmax = aro;
     if aro <= 1.e-10 * dmax then lindep = 1;
     if lindep = 0 then do;
       nind = nind + 1;
       q = q || w;
     end;
     else print "Unit vector" i "linearly dependent.";
   end;
end;
if nind < m then do;
   print "This is theoretically not possible.";
end;
```

PARSE Statement

parses matrix elements as statements

PARSE *matrices* <**(***matrix-names***)**>**;**

The inputs to the PARSE statement are as follows:

matrices	are character matrices containing IML module statements.
(*matrix-names*)	are character matrices whose elements are the names of character matrices containing IML module statements.

Use the PARSE statement to parse the elements of a character matrix containing IML module statements. For example, the following statement parses the elements (rows) of matrix **A** as lines of code.

```
parse a;
```

You can parse several matrices with one PARSE statement either by listing all of their names in the statement or by first creating a character matrix, say **N**, containing their names as elements and then parsing **N**. Each element of **N** is the name of a matrix containing IML module statements. In this case, enclose **N** in parentheses in the PARSE statement to indicate the indirect references to the elements of **N**.

For example, the statements

```
a={"start mod1;",
   "x={1 2 3};",
   "print x;",
   "finish;"};
parse a;
run mod1;
```

produce the result

```
NOTE: Module MOD1 defined.

                              X
                              1         2         3
```

Alternatively, you can use the following statements to obtain the same result:

```
a={"start mod1;",
   "x={1 2 3};",
   "print x;",
   "finish;"};
c={a};
parse (c);
run mod1;
```

PAUSE Statement

interrupts module execution

PAUSE <*expression*> <*****>**;**

The inputs to the PAUSE statement are as follows:

expression	is a character matrix or quoted literal giving a message to print.
*	suppresses any messages.

The PAUSE statement stops execution of a module, saves the calling chain so that execution can resume later (by a RESUME statement), prints a pause message that you can specify, and puts you in immediate mode so you can enter more statements.

You can specify an operand in the PAUSE statement to supply a message to be printed for the pause prompt. If no operand is specified, the default message,

```
paused in module XXX
```

is printed, where *XXX* is the name of the module containing the pause. If you want to suppress all messages in a PAUSE statement, use an asterisk as the operand:

```
pause *;
```

The PAUSE statement should only be specified in modules. It generates a warning if executed in immediate mode.

When an error occurs while executing inside a module, IML automatically behaves as though a PAUSE statement was issued. PROC IML prints a note saying

```
paused in module
```

and IML puts you in immediate mode within the module environment, where you can correct the error. You can then resume execution from the statement following the one where the error occurred by issuing a RESUME command.

IML supports pause processing of both subroutine and function modules. See also the description of the SHOW statement using the PAUSE option.

PGRAF Call

produces scatter plots

CALL PGRAF($xy <$, *id*$><$, *xlabel*$><$, *ylabel*$><$, *title*$>$**);**

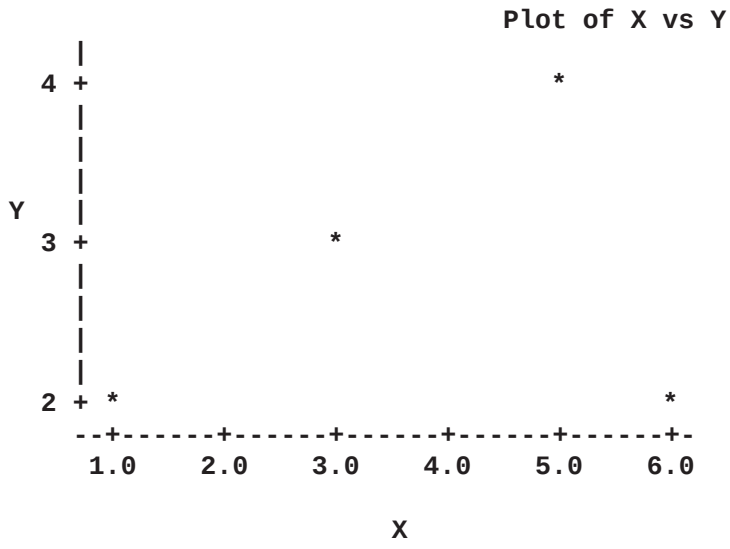
The inputs to the PGRAF subroutine are as follows:

xy	is an $n \times 2$ matrix of (x, y) points.
id	is an $n \times 1$ character matrix of labels for each point. The PGRAF subroutine uses up to 8 characters per point. If *id* is a scalar (1×1), then the same label is used for all of the points. The label is centered over the actual point location. If you do not specify *id*, **x** is the default character for labeling the points.
xlabel	is a character scalar or quoted literal that labels the x axis (centered below the x axis).
ylabel	is a character scalar or quoted literal that labels the y axis (printed vertically to the left of the y axis).
title	is a character scalar or quoted literal printed above the graph.

The PGRAF subroutine produces a scatter plot suitable for display on a line printer or similar device.

The statements below specify a plotting symbol, axis labels, and a title to produce the plot shown.

```
xy={1 2, 3 3, 5 4, 6 2};
call pgraf(xy,'*','X','Y','Plot of X vs Y');
```



POLYROOT Function

finds zeros of a real polynomial

POLYROOT(*vector*)

where *vector* is an $n \times 1$ (or $1 \times n$) vector containing the coefficients of an $(n-1)$ degree polynomial with the coefficients arranged in order of decreasing powers. The POLYROOT function returns the array r, which is an $(n-1) \times 2$ matrix containing the roots of the polynomial. The first column of r contains the real part of the complex roots and the second column contains the imaginary part. If a root is real, the imaginary part will be 0.

The POLYROOT function finds the real and complex roots of a polynomial with real coefficients.

The POLYROOT function uses an algorithm proposed by Jenkins and Traub (1970) to find the roots of the polynomial. The algorithm is not guaranteed to find all roots of the polynomial. An appropriate warning message is issued when one or more roots cannot be found. The POLYROOT algorithm produces roots within the precision allowed by the hardware. If r is given as a root of the polynomial $P(x)$, then $1 + P(R) = 1$ based on the roundoff error of the computer that is employed.

For example, to find the roots of the polynomial

$$P(x) = 0.2567x^4 + 0.1570x^3 + 0.0821x^2 - 0.3357x + 1$$

use the following IML code to produce the result shown.

```
p={0.2567 0.1570 0.0821 -0.3357 1};
r=polyroot(p);
```

```
          R             4 rows      2 cols    (numeric)

                      0.8383029 0.8514519
                      0.8383029 -0.851452
                      -1.144107 1.1914525
                      -1.144107 -1.191452
```

The polynomial has two conjugate pairs of roots that, within machine precision, are given by $r = 0.8383029 \pm 0.8514519i$ and $r = -1.144107 \pm 1.1914525i$.

PRINT Statement

prints matrix values

PRINT <*matrices*> <*(expression)*> <*"message"*>
<*pointer-controls*> <**[***options***]**>**;**

The inputs to the PRINT statement are as follows:

matrices	are the names of matrices.
(*expression*)	is an expression in parentheses that is evaluated. The result of the evaluation is printed. The evaluation of a subscripted matrix used as an expression results in printing the submatrix.
"*message*"	is a message in quotes.
pointer-controls	control the pointer for printing. For example, using a comma (,) skips a single line and using a slash (/) skips to a new page.
[*options*	] are described below.

The PRINT statement prints the specified matrices or message. The options below can appear in the PRINT statement. They are specified in brackets after the matrix name to which they apply.

COLNAME=*matrix*
specifies the name of a character matrix whose first *ncol* elements are to be used for the column labels of the matrix to be printed, where *ncol* is the number of columns in the matrix. (You can also use the RESET *autoname* statement to automatically label columns as COL1, COL2, and so on.)

FORMAT=*format*
specifies a valid SAS or user-defined format to use in printing the values of the matrix, for example,

```
print x[format=5.3];
```

ROWNAME=*matrix*

specifies the name of a character matrix whose first *nrow* elements are to be used for the row labels of the matrix to be printed, where *nrow* is the number of rows in the matrix and where the scan to find the first *nrow* elements goes across row 1, then across row 2, and so forth through row n. (You can also use the RESET *autoname* statement to automatically label rows as ROW1, ROW2, and so on.)

```
reset autoname;
```

For example, you can use the statement below to print a matrix called **X** in format 12.2 with columns labeled AMOUNT and NET PAY, and rows labeled DIV A and DIV B:

```
x={45.125 50.500,
   75.375 90.825};
r={"DIV A"  "DIV B"};
c={"AMOUNT" "NET PAY"};

print x[rowname=r colname=c format=12.2];
```

The output is

```
X            AMOUNT      NET PAY

DIV A         45.13        50.50
DIV B         75.38        90.83
```

To permanently associate the above options with a matrix name, refer to the description of the MATTRIB statement.

If there is not enough room to print all the matrices across the page, then one or more matrices are printed out in the next group. If there is not enough room to print all the columns of a matrix across the page, then the columns are folded, with the continuation lines identified by a colon(:).

The spacing between adjacent matrices can be controlled by the SPACES= option of the RESET statement. The FW= option of the RESET statement can be used to control the number of print positions used to print each numeric element. For more print-related options, see the description of the RESET statement. The example below shows how to print part of a matrix:

```
y=1:10;
   /* prints first five elements of y*/
print (y[1:5]) [format=5.1];
```

PRODUCT Function

multiplies matrices of polynomials

PRODUCT(a, b <, *dim*>**)**

The inputs to the PRODUCT function are as follows:

a is an $m \times (ns)$ numeric matrix. The first $m \times n$ submatrix contains the constant terms of the polynomials, the second $m \times n$ submatrix contains the first order terms, and so on.

b is an $n \times (pt)$ matrix. The first $n \times p$ submatrix contains the constant terms of the polynomials, the second $n \times p$ submatrix contains the first order terms, and so on.

dim is a 1×1 matrix, with value $p > 0$. The value of this matrix is used to set p above. If omitted, the value of p is set to 1.

The PRODUCT function multiplies matrices of polynomials. The value returned is the $m \times (p(s + t - 1))$ matrix of the polynomial products. The first $m \times p$ submatrix contains the constant terms, the second $m \times p$ submatrix contains the first order terms, and so on.

Note: The PRODUCT function can be used to multiply the matrix operators employed in a multivariate time-series model of the form

$$\Phi_1(B)\Phi_2(B)\mathbf{Y}_t = \Theta_1(B)\Theta_2(B)\epsilon_t$$

where $\Phi_1(B)$, $\Phi_2(B)$, $\Theta_1(B)$, and $\Theta_2(B)$ are matrix polynomial operators whose first matrix coefficients are identity matrices. Often $\Phi_2(B)$ and $\Theta_2(B)$ represent seasonal components that are isolated in the modeling process but multiplied with the other operators when forming predictors or estimating parameters. The RATIO function is often employed in a time series context as well.

For example, the statement

```
r=product({1 2 3 4,
           5 6 7 8},
          {1 2 3,
           4 5 6}, 1);
```

produces the result

```
                R            2 rows      4 cols    (numeric)

                             9        31        41        33
                            29        79       105        69
```

PURGE Statement

removes observations marked for deletion and renumbers records

PURGE;

The PURGE data processing statement is used to remove observations marked for deletion and to renumber the remaining observations. This closes the gaps created by deleted records. Execution of this statement may be time consuming because it involves rewriting the entire data set.

CAUTION: Any indexes associated with the data set are lost after a purge.

IML does not do an automatic purge for you at quit time.

In the example that follows, a data set named A is created. Then, you begin an IML session and edit A. You delete the fifth observation, list the data set, and issue a PURGE statement to delete the fifth observation and renumber the remaining observations.

```
data a;
   do i=1 to 10;
      output;
   end;
run;

proc iml;
   edit a;
   delete point 5;
   list all;
   purge;
   list all;
```

PUSH Call

pushes SAS statements into the command input stream

CALL PUSH(*argument1*<*, argument2,. . ., argument15*>**);**

where *argument* is a character matrix or quoted literal containing valid SAS statements.

The PUSH subroutine pushes character arguments containing valid SAS statements (usually SAS/IML statements or global statements) to the input command stream. You can specify up to 15 arguments. Any statements pushed to the input command queue get executed when the module is put in a hold state. This is usually induced by one of the following:

- an execution error within a module

- an interrupt
- a pause command

The string pushed is read before any other lines of input. If you call the PUSH subroutine several times, the strings pushed each time are ahead of the less recently pushed strings. If you would rather place the lines after others in the input stream, then use the QUEUE command instead.

The strings you push do not appear on the log.

CAUTION: Do not push too much code at one time.

Pushing too much code at one time, or getting into infinite loops of pushing, causes problems that may result in exiting the SAS system.

For details, see Chapter 15, "Using SAS/IML Software to Generate IML Statements."

An example using the PUSH subroutine is shown below:

```
start;
   code='reset pagesize=25;';
   call push(code,'resume;');
   pause;
      /* show that pagesize was set to 25 during   */
      /* a PAUSE state of a module                 */
   show options;
finish;
run;
```

PUT Statement

writes data to an external file

PUT <*operand*> <*record-directives*> <*positionals*> <*format*>**;**

The inputs to the PUT statement are as follows:

operand specifies the value you want to output to the current position in the record. The *operand* can be either a variable name, a literal value, or an expression in parentheses. The *operand* can be followed immediately by an output format specification.

record-directives start new records. There are three types:

holding @ at the end of a PUT statement, instructs IML to put a hold on the current record so that IML can write more to the record with later PUT statements. Otherwise, IML automatically begins the next record for the next PUT statement.

/ writes out the current record and begins forming a new record.

> *operand* specifies that the next record written will start at the indicated byte position in the file (for RECFM=N files only). The *operand* is a literal number, a variable name, or an expression in parentheses, for example,

```
put >3 x 3.2;
```

positionals specify the column on the record to which the PUT statement should go. There are two types of positionals:

@ *operand* specifies to go to the indicated column, where *operand* is a literal number, a variable name, or an expression in parentheses. For example, @30 means to go to column 30.

+ *operand* specifies that the indicated number of columns are to be skipped, where *operand* is a literal number, a variable name, or an expression in parentheses.

format specifies a valid SAS or user-defined output format. These are of the form $w.d$ or $\$w.$ for standard numeric and character formats, respectively, where w is the width of the field and d is the decimal parameter, if any. They can also be a named format of the form *NAME*$w.d$, where *NAME* is the name of the format. If the width is unspecified, then a default width is used; this is 9 for numeric variables.

The PUT statement writes to the file specified in the previously executed FILE statement, putting the values from IML variables. The statement is described in detail in Chapter 7, "File Access."

The PUT statement is a sequence of positionals and record directives, variables, and formats. An example using the PUT statement is shown below:

```
/* output variable A in column 1 using SAS format 6.4. */
/* Skip 3 columns and output X using format 8.4        */

put @1 a 6.4 +3 x 8.4;
```

PV Function

calculates the present value of a vector of cash-flows and returns a scalar

PV($times, flows, freq, rates$)

The PV function returns a scalar containing the present value of the cash-flows based on the specified frequency and rates.

times is an n x 1 column vector of times.
Elements should be non-negative.

flows is an n x 1 column vector of cash-flows.

freq is a scalar which represents the base of the rates
to be used for discounting the cash-flows.
If positive, it represents discrete compounding
as the reciprocal of the number of compoundings.
If zero, it represents continuous compounding.
If -1, it represents per-period discount factors.
No other negative values are allowed.

rates is an n x 1 column vector of rates
to be used for discounting the cash-flows.
Elements should be positive.

A general present value relationship can be written as

$$P = \sum_{k=1}^{K} c(k)D(t_k)$$

where P is the present value of the asset, $\{c(k)\}k = 1,..K$ is the sequence of cash-flows from the asset, t_k is the time to the k-th cash-flow in periods from the present, and $D(t)$ is the discount function for time t.
With per-unit-time-period discount factors d_t:

$$D(t) = d_t^t$$

With continuous compunding:

$$D(t) = e^{-r_t t}$$

With discrete compunding:

$$D(t) = (1 + fr)^{-(t/f)}$$

where $f > 0$ is the frequency, the reciprocal of the number of compoundings per unit time period.

Example

```
proc iml;
   timesn=do(1,100,1);
   timesn=T(timesn);
   flows=repeat(10,100);
   freq={0};
```

```
      do while(freq<50);
      freq=freq+.25;
      end;
      rate=repeat(.10,100);
      pv=pv(timesn,flows,freq,rate);
   print pv;
   quit;
```

```
PV
266.4717
```

QR Call

produces the QR decomposition of a matrix by Householder transformations

CALL QR(*q, r, piv, lindep, a <, ord><, b>***);**

The QR subroutine returns the following values:

q specifies an orthogonal matrix $\mathbf{Q}$ that is the product of the Householder transformations applied to the $m \times n$ matrix $\mathbf{A}$, if the b argument is not specified. In this case, the $\min(m, n)$ Householder transformations are applied, and q is an $m \times m$ matrix. If the b argument is specified, q is the $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$ that has the transposed Householder transformations $\mathbf{Q}'$ applied on the p columns of the argument matrix $\mathbf{B}$.

r specifies a $\min(m, n) \times n$ upper triangular matrix $\mathbf{R}$ that is the upper part of the $m \times n$ upper triangular matrix $\widetilde{\mathbf{R}}$ of the QR decomposition of the matrix $\mathbf{A}$. The matrix $\widetilde{\mathbf{R}}$ of the QR decomposition can be obtained by vertical concatenation (using the IML operator //) of the $(m - \min(m, n)) \times n$ zero matrix to the result matrix $\mathbf{R}$.

piv specifies an $n \times 1$ vector of permutations of the columns of $\mathbf{A}$; that is, on return, the QR decomposition is computed, not of $\mathbf{A}$, but of the permuted matrix whose columns are $[\mathbf{A}_{piv[1]} \cdots \mathbf{A}_{piv[n]}]$. The vector *piv* corresponds to an $n \times n$ permutation matrix $\mathbf{\Pi}$.

lindep is the number of linearly dependent columns in matrix $\mathbf{A}$ detected by applying the $\min(m, n)$ Householder transformations in the order specified by the argument vector *piv*.

The inputs to the QR subroutine are as follows:

a specifies an $m \times n$ matrix $\mathbf{A}$ that is to be decomposed into the product of the orthogonal matrix $\mathbf{Q}$ and the upper triangular matrix $\widetilde{\mathbf{R}}$.

ord specifies an optional $n \times 1$ vector that specifies the order of Householder transformations applied to matrix $\mathbf{A}$, as follows:

ord$[j] > 0$ Column j of $\mathbf{A}$ is an *initial column*, meaning it has to be processed at the start in increasing order of *ord*$[j]$.

ord$[j] = 0$ Column j of $\mathbf{A}$ is allowed to be permuted in order of decreasing residual Euclidean norm (pivoting).

ord$[j] < 0$ Column j of $\mathbf{A}$ is a *final column*, meaning it has to be processed at the end in decreasing order of *ord*$[j]$.

The default is *ord*$[j] = j$, in which case the Householder transformations are done in the same order that the columns are stored in matrix $\mathbf{A}$ (without pivoting).

b specifies an optional $m \times p$ matrix $\mathbf{B}$ that is to be multiplied by the transposed $m \times m$ matrix $\mathbf{Q}'$. If b is specified, the result q contains the $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$. If b is not specified, the result q contains the $m \times m$ matrix $\mathbf{Q}$.

The QR subroutine decomposes an $m \times n$ matrix $\mathbf{A}$ into the product of an $m \times m$ orthogonal matrix $\mathbf{Q}$ and an $m \times n$ upper triangular matrix $\widetilde{\mathbf{R}}$, so that

$$\mathbf{A\Pi} = \mathbf{Q}\widetilde{\mathbf{R}}, \quad \mathbf{Q}'\mathbf{Q} = \mathbf{Q}\mathbf{Q}' = \mathbf{I}_m$$

by means of $\min(m, n)$ Householder transformations.

The $m \times m$ orthogonal matrix $\mathbf{Q}$ is computed only if the last argument b is not specified, as follows:

```
call qr(q,r,piv,lindep,a,ord);
```

In many applications, the number of rows, m, is very large. In these cases, the explicit computation of the $m \times m$ matrix $\mathbf{Q}$ can require too much memory or time.

In the usual case where $m > n$,

$$\mathbf{A} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

$$\widetilde{\mathbf{R}} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}$$

$$\mathbf{Q} = \left[\mathbf{Q}_1 \vdots \mathbf{Q}_2\right], \qquad \widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

where $\mathbf{R}$ is the result returned by the QR subroutine.

The n columns of matrix $\mathbf{Q}_1$ provide an orthonormal basis for the n columns of $\mathbf{A}$ and are called the *range space* of $\mathbf{A}$. Since the $m-n$ columns of $\mathbf{Q}_2$ are orthogonal to

the n columns of $\mathbf{A}$, $\mathbf{Q}_2'\mathbf{A} = \mathbf{0}$, they provide an orthonormal basis for the orthogonal complement of the columns of $\mathbf{A}$ and are called the *null space* of $\mathbf{A}$.

In the case where $m < n$,

$$\mathbf{A} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\widetilde{\mathbf{R}} = \mathbf{R} = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}$$

Specifying the argument *ord* as an n vector lets you specify a special order of the columns in matrix $\mathbf{A}$ on which the Householder transformations are applied. When you specify the *ord* argument, the columns of $\mathbf{A}$ can be divided into the following groups:

- *ord*$[j] > 0$: Column j of $\mathbf{A}$ is an *initial column*, meaning it has to be processed at the start in increasing order of *ord*$[j]$. This specification defines the first nl columns of $\mathbf{A}$ that are to be processed.
- *ord*$[j] = 0$: Column j of $\mathbf{A}$ is a *pivot column*, meaning it is to be processed in order of decreasing residual Euclidean norms. The *pivot columns* of $\mathbf{A}$ are processed after the nl *initial columns* and before the nu *final columns*.
- *ord*$[j] < 0$: Column j of $\mathbf{A}$ is a *final column*, meaning it has to be processed at the end in decreasing order of *ord*$[j]$. This specification defines the last nu columns of $\mathbf{A}$ that are to be processed. If $n > m$, some of these columns will not be processed at all.

There are two special cases:

- If you do not specify the *ord* argument, the default values *ord*$[j] = j$ are used. In this case, Householder transformations are done in the same order in which the columns are stored in $\mathbf{A}$ (without pivoting).
- If you set all components of *ord* to zero, the Householder transformations are done in order of decreasing Euclidean norms of the columns of $\mathbf{A}$.

The resulting $n \times 1$ vector *piv* specifies the permutation of the columns of $\mathbf{A}$ on which the Householder transformations are applied; that is, on return, the QR decomposition is computed, not of $\mathbf{A}$, but of the matrix with columns that are permuted in the order $\mathbf{A}_{piv[1]}, \ldots, \mathbf{A}_{piv[n]}$.

To check the QR decomposition, use the following statements to compute the three residual sum of squares, represented by the variables SS0, SS1, and SS2, which should be close to zero:

```
m = nrow(a); n = ncol(a);
call qr(q,r,piv,lindep,a,ord);
```

```
ss0 = ssq(a[ ,piv] - q[,1:n] * r);
ss1 = ssq(q * q' - i(m));
ss2 = ssq(q' * q - i(m));
```

If the QR subroutine detects linearly dependent columns while processing matrix $\mathbf{A}$, the column order given in the result vector *piv* can differ from an explicitly specified order in the argument vector *ord*. If a column of $\mathbf{A}$ is found to be linearly dependent on columns already processed, this column is swapped to the end of matrix $\mathbf{A}$. The order of columns in the result matrix $\mathbf{R}$ corresponds to the order of columns processed in $\mathbf{A}$. The swapping of a linearly dependent column of $\mathbf{A}$ to the end of the matrix corresponds to the swapping of the same column in $\mathbf{R}$ and leads to a zero row at the end of the upper triangular matrix $\mathbf{R}$.

The scalar result *lindep* counts the number of linearly dependent columns that are detected in constructing the first $\min(m, n)$ Householder transformations in the order specified by the argument vector *ord*. The test of linear dependence depends on the size of the singularity criterion used; currently it is specified as $1\mathrm{E}{-8}$.

Solving the linear system $\mathbf{Rx} = \mathbf{Q}'\mathbf{b}$ with an upper triangular matrix $\mathbf{R}$ whose columns are permuted corresponding to the result vector *piv* leads to a solution $\mathbf{x}$ with permuted components. You can reorder the components of $\mathbf{x}$ by using the index vector *piv* at the left-hand side of an expression, as follows:

```
call qr(qtb,r,piv,lindep,a,ord,b);
x[piv] = inv(r) * qtb[1:n,1:p];
```

The following example solves the full rank linear least-squares problem. Specify the argument b as an $m \times p$ matrix $\mathbf{B}$, as follows:

```
call qr(q,r,piv,lindep,a,ord,b);
```

When you specify the b argument, the QR call computes the matrix $\mathbf{Q}'\mathbf{B}$ (instead of $\mathbf{Q}$) as the result q. Now you can compute the p least-squares solutions $\mathbf{x}_k$ of an overdetermined linear system with an $m \times n, m > n$ coefficient matrix $\mathbf{A}$, rank($\mathbf{A}$) = n, and p right-hand sides $\mathbf{b}_k$ stored as the columns of the $m \times p$ matrix $\mathbf{B}$:

$$\min_{\mathbf{x}_k} \|\mathbf{A}\mathbf{x}_k - \mathbf{b}_k\|^2, \qquad k = 1, \ldots, p$$

where $\|\cdot\|$ is the Euclidean vector norm. This is accomplished by solving the p upper triangular systems with back-substitution:

$$\mathbf{x}_k = \mathbf{\Pi}'\mathbf{R}^{-1}\mathbf{Q}_1'\mathbf{b}_k, \qquad k = 1, \ldots, p$$

For most applications, m, the number of rows of $\mathbf{A}$, is much larger than n, the number of columns of $\mathbf{A}$, or p, the number of right-hand sides. In these cases, you are advised not to compute the large $m \times m$ matrix $\mathbf{Q}$ (which can consume too much memory and time) if you can solve your problem by computing only the smaller $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$ implicitly. For an example, use the first five columns of the 6×6 Hilbert matrix $\mathbf{A}$.

```
proc iml;
   a= {  36      -630      3360     -7560      7560     -2772,
       -630     14700    -88200    211680   -220500     83160,
       3360    -88200    564480  -1411200   1512000   -582120,
      -7560    211680  -1411200   3628800  -3969000   1552320,
       7560   -220500   1512000  -3969000   4410000  -1746360,
      -2772     83160   -582120   1552320  -1746360    698544 };
   b= { 463, -13860, 97020, -258720, 291060, -116424};
   n = 5; aa = a[,1:n];
   call qr(qtb,r,piv,lindep,aa,,b);
   if lindep=0 then x=inv(r)*qtb[1:n];
   print x;
```

Note that you are using only the first n rows, $\mathbf{Q}_1'\mathbf{B}$, of QTB. The IF-THEN statement of the preceding code may be replaced by the more efficient TRISOLV function, as follows:

```
   if lindep=0 then x=TRISOLV(1,r,qtb[1:n],piv);
   print x;
```

Both cases produce the following output:

```
                X
                1
              0.5
        0.3333333
             0.25
              0.2
```

For information on solving rank-deficient linear least-squares problems, see the RZLIND call.

QUAD Call

performs numerical integration of scalar functions in one dimension over infinite, connected semi-infinite, and connected finite intervals

CALL QUAD(*r*, *"fun"*, *points* <, **EPS=***eps*><, **PEAK=***peak*>
<, **SCALE=***scale*><, **MSG=***msg*><, **CYCLES=***cycles*>**);**

The QUAD subroutine returns the following value:

r is a numeric vector containing the results of the integration. The size of r is equal to the number of subintervals defined by the argument *points*. Should the numerical integration fail on a particular subinterval, the corresponding element of r is set to missing.

The inputs to the QUAD are as follows:

"fun" specifies the name of an IML module used to evaluate the integrand.

points specifies a sorted vector that provides the limits of integration over connected subintervals. The simplest form of the vector provides the limits of the integration on one interval. The first element of *points* should contain the left limit. The second element should be the right limit. A missing value of `.M` in the left limit is interpreted as $-\infty$, and a missing value of `.P` is interpreted as $+\infty$. For more advanced usage of the QUAD call, *points* can contain more than two elements. The elements of the vector must be sorted in an ascending order. Each two consecutive elements in *points* defines a subinterval, and the subroutine reports the integration over each specified subinterval. The use of subintervals is important because the presence of internal points of discontinuity in the integrand will hinder the algorithm.

eps is an optional scalar specifying the desired relative accuracy. It has a default value of $1E-7$. You can specify *eps* with the keyword EPS.

peak is an optional scalar that is the approximate location of a maximum of the integrand. By default, it has a location of 0 for infinite intervals, a location that is one unit away from the finite boundary for semi-infinite intervals, and a centered location for bounded intervals. You can specify *peak* with the keyword PEAK.

scale is an optional scalar that is the approximate estimate of any scale in the integrand along the independent variable (see the examples). It has a default value of 1. You can specify *scale* with the keyword SCALE.

msg is an optional character scalar that restricts the number of messages produced by the QUAD subroutine. If *msg* = "NO" then it does not produce any warning messages. You can specify *msg* with the keyword MSG.

cycles is an optional integer indicating the number of refinements allowed to achieve the required accuracy. It has a default value of 8. You can specify *cycles* with the keyword CYCLES.

If the dimensions of any optional argument are 0×0, the QUAD call uses its default value.

The QUAD subroutine quad is a numerical integrator based on adaptive Romberg-type integration techniques. Refer to Rice (1973), Sikorsky (1982), Sikorsky and Stenger (1984), and Stenger (1973a, 1973b, 1978). Many adaptive numerical integration methods (Ralston and Rabinowitz 1978) start at one end of the interval and proceed towards the other end, working on subintervals while locally maintaining a certain prescribed precision. This is not the case with the QUAD call. The QUAD call is an adaptive global-type integrator that produces a quick, rough estimate of the integration result and then refines the estimate until achieving the prescribed accuracy. This gives the subroutine an advantage over Gauss-Hermite and Gauss-Laguerre

quadratures (Ralston and Rabinowitz 1978, Squire 1987), particularly for infinite and semi-infinite intervals, because those methods perform only a single evaluation.

Consider the integration

$$\int_0^\infty e^{-t}\,dt$$

The following statements evaluate this integral:

```
proc iml;
  /* Define the integrand */
  start fun(t);
      v = exp(-t);
      return(v);
  finish;

  /* Call QUAD */
  a   = { 0  .P };
  call quad(z,"fun",a);
  print z[format=E21.14];
```

The integration is carried out over the interval $(0, \infty)$, as specified by the variable A. Note that the missing value in the second element of A is interpreted as ∞. The values of *eps*=1E−7, *peak*=1, *scale*=1, and *cycles*=8 are used by default.

The following code performs the integration over two subintervals, as specified by the variable A:

```
proc iml;
  /* Define the integrand */
  start fun(t);
      v = exp(-t);
      return(v);
  finish;

  /* Call QUAD */
  a   = { 0  3 .P };
  call quad(z,"fun",a);
  print z[format=E21.14];
```

Note that the elements of A are in ascending order. The integration is carried out over $(0, 3)$ and $(3, \infty)$, and the corresponding results are shown in the output. The values of *eps*=1E−7, *peak*=1, *scale*=1, and *cycles*=8 are used by default. To obtain the results of integration over $(0, \infty)$, use the SUM function on the elements of the vector **Z**, as follows:

```
    b = sum(z);
    print b[format=E21.14];
```

The purpose of the *peak* and *scale* options is to enable you to avoid analytically changing the variable of the integration in order to produce a well-conditioned integrand that permits the numerical evaluation of the integration.

Consider the integration

$$\int_0^\infty e^{-10000t}\,dt$$

The following statements evaluate this integral:

```
proc iml;
  /* Define the integrand */
  start fun(t);
      v = exp(-10000*t);
      return(v);
  finish;

  /* Call QUAD */
  a   = { 0  .P };
  /* Either syntax can be used                */
  /*   call quad(z,"fun",a,1E-10,0.0001); or */
  call quad(z,"fun",a) eps=1E-10 peak=0.0001 ;
  print z[format=E21.14];
```

Only one interval exists. The integration is carried out over $(0, \infty)$. The default values of *scale*=1 and *cycles*=8 are used.

If you do not specify a *peak* value, the integration cannot be evaluated to the desired accuracy, a message is printed to the LOG, and a missing value is returned. Note that *peak* can still be set to $1E-7$ and the integration will be successful. The evaluation of the integrand at *peak* must be non-zero for the computation to continue. You should adjust the value of *peak* to get a nonzero evaluation at *peak* before trying to adjust *scale*. Reducing *scale* decreases the initial step size and may lead to an increase in the number of function evaluations per step at a linear rate.

Consider the integration

$$\int_0^\infty e^{-100000(t-3)^2}\,dt$$

The following statements evaluate this integral:

```
proc iml;
  /* Define the integrand */
  start fun(t);
      v = exp(-100000*(t-3)*(t-3));
      return(v);
  finish;
  /* Call QUAD */
  a   = { .M  .P };
  call quad(z,"fun",a) eps=1E-10 peak=3 scale=0.001 ;
  print z[format=E21.14];
```

Only one interval exists. The integration is carried out over $(-\infty, \infty)$. The default value of *cycles*=8 has been used.

If you use the default value of *scale*, the integral cannot be evaluated to the desired accuracy, and a missing value is returned. The variables *scale* and *cycles* can be used to allow an increase in the number of possible function evaluations; the number of possible function evaluations will increase linearly with the reciprocal of *scale*, but it will potentially increase in an exponential manner when *cycles* is increased. Increasing the number of function evaluations increases execution time.

When you perform double integration, you must separate the variables between the iterated integrals. There should be a clear distinction between the variables of the one-dimensional integration at hand and the parameters to be passed to the integrand. Posting the correct limits of integration is also an important issue. For example, consider the binormal probability, given by

$$\text{probbnrm}(a, b, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{a} \int_{-\infty}^{b} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}\right) \, dy \, dx$$

The inner integral is

$$g(x, b, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{b} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}\right) \, dy$$

with parameters x and ρ, and the limits of integration are from $-\infty$ to b. The outer integral is then

$$\text{probbnrm}(a, b, \rho) = \int_{-\infty}^{a} g(x, b, \rho) \, dx$$

with the limits from $-\infty$ to a.

You can write the equation in the form of a function with the parameters a, b, ρ as arguments. The following statements provide an example of this technique:

```
start norpdf2(t) global(yv,rho,omrho2,count);

/*-------------------------------------------------------*/
/* This function is the density function and requires  */
/* the variable T (passed in the argument)             */
/* and a list of parameters, YV, RHO, OMRHO2, COUNT    */
/* (defined in the GLOBAL clause)                      */
/*-------------------------------------------------------*/

   count = count+1;
   q=(t#t-2#rho#t#yv+yv#yv)/omrho2;
   p=exp(-q/2);
   return(p);
finish;
```

```
start marginal(v) global(yy,yv,eps);
/*-----------------------------------------------*/
/* The inner integral                             */
/* The limits of integration from .M to YY        */
/* YV is passed as a parameter to the inner integral*/
/*-----------------------------------------------*/
   interval = .M || yy;
   if ( v < -12 ) then return(0);
   yv = v;
   call quad(pm,"NORPDF2",interval) eps=eps;
   return(pm);
finish;

start norcdf2(a,b,rrho) global(yy,rho,omrho2,eps);
/*-----------------------------------------------*/
/* Post some global parameters                   */
/*    YY, RHO, OMRHO2                            */
/* EPS will be set from IML                      */
/* RHO and B cannot be arguments in the GLOBAL    */
/* list at the same time                         */
/*-----------------------------------------------*/
   rho = rrho;
   yy = b;
   omrho2 = 1-rho#rho;
   /*-----------------------------------------------*/
   /* The outer integral                          */
   /* The limits of integration                   */
   /*-----------------------------------------------*/
   interval= .M || a;


   /*-----------------------------------------------*/
   /*Note that EPS the keyword = EPS the variable  */
   /*-----------------------------------------------*/
   call quad(p,"MARGINAL",interval) eps=eps;

   /*--------------------------*/
   /*  PER will be reset here  */
   /*--------------------------*/
   per = 1/(8#atan(1)#sqrt(omrho2)) * p;
   return(per);
finish;

/*---------------------------------*/
/*First set up the global constants */
/*---------------------------------*/
count = 0;
eps = 1E-11;

/*---------------------------------*/
/* Do the work and print the results  */
/*---------------------------------*/
p =  norcdf2(2,1,0.1);
```

```
print p[format=E21.14];
print count;
```

The variable COUNT contains the number of times the NORPDF2 module is called. Note that the value computed by the NORCDF2 module is very close to that returned by the PROBBNRM function, as computed by the following statements:

```
/*-----------------------------------------------*/
/* Calculate the value with the PROBBNRM function */
/*-----------------------------------------------*/
pp = probbnrm(2,1,0.1);
print pp[format=E21.14];
```

Note the following:

- The iterated inner integral cannot have a left endpoint of $-\infty$. For large values of v, the inner integral does not contribute to the answer but still needs to be calculated to the required relative accuracy. Therefore, either cut off the function (when $v \leq -12$), as in the MARGINAL module in the preceding code, or have the intervals start from a reasonable cutoff value. In addition, the QUAD call stops if the integrands appear to be identically 0 (probably caused by underflow) over the interval of integration.
- This method of integration (iterated, one-dimensional integrals) is extremely conservative and requires unnecessary function evaluations. In this example, the QUAD call for the inner integration lacks information about the final value that the QUAD call for the outer integration is trying to refine. The lack of communication between the two QUAD routines can cause useless computations to be performed in the inner integration.

  To illustrate this idea, let the relative error be 1E−11 and let the answer delivered by the outer integral be 0.8, as in this example. Any computation of the inner execution of the QUAD call that yields 0.8E−11 or less will not contribute to the final answer of the QUAD call for the outer integral. However, the inner integral lacks this information, and for a given value of the parameter yv, it attempts to compute an answer with much more precision than is necessary. The lack of communication between the two QUAD subroutines prevents the introduction of better cut-offs. Although this method can be inefficient, the final calculations are accurate.

QUEUE Call

queues SAS statements into the command input stream

CALL QUEUE(*argument1*<*, argument2,...*, *argument15*>**);**

where *argument* is a character matrix or quoted literal containing valid SAS statements.

The QUEUE subroutine places character arguments containing valid SAS statements (usually SAS/IML statements or global statements) at the end of the input command stream. You can specify up to 15 arguments. The string queued is read after other lines of input already on the queue. If you want to push the lines in front of other lines already in the queue, use the PUSH subroutine instead. Any statements queued to the input command queue get executed when the module is put in a hold state. This is usually induced by one of the following:

- an execution error within a module
- an interrupt
- a pause command.

The strings you queue do not appear on the log.

CAUTION: Do not queue too much code at one time.

Queuing too much code at one time, or getting into infinite loops of queuing, causes problems that may result in exiting the SAS system.

For more examples, consult Chapter 15, “Using SAS/IML Software to Generate IML Statements.”

An example using the QUEUE subroutine follows:

```
start mod(x);
   code='x=0;';
   call queue (code,'resume;');
   pause;
finish;
x=1;
run mod(x);
print(x);
```

produces

```
                                X

                                0
```

QUIT Statement

exits from IML

QUIT;

Use the QUIT statement to exit IML. If a DATA or PROC statement is encountered, QUIT is implied. The QUIT statement is executed immediately; therefore, you cannot use QUIT as an executable statement, that is, as part of a module or conditional clause. (See the description of the ABORT statement.)

PROC IML closes all open data sets and files when a QUIT statement is encountered. Workspace and symbol spaces are freed up. If you need to use any matrix values or any module definitions in a later session, you must store them in a storage library before you quit.

RANK Function

ranks elements of a matrix

RANK(*matrix***)**

where *matrix* is a numeric matrix or literal.

The RANK function creates a new matrix containing elements that are the ranks of the corresponding elements of *matrix*. The ranks of tied values are assigned arbitrarily rather than averaged. (See the description of the RANKTIE function.)

For example, the statements

```
x={2 2 1 0 5};
y=rank(x);
```

produce the vector

```
                Y

                3         4         2         1         5
```

The RANK function can be used to sort a vector **x**:

```
b=x;
x[,rank(x)]=b;

    X
    0         1         2         2         5
```

The RANK function can also be used to find anti-ranks of **x**:

```
r=rank(x);
i=r;
i[,r]=1:ncol(x);

      I
      4         3         1         2         5
```

IML does not have a function that directly computes the rank of a matrix. You can use the following technique to compute the rank of matrix A:

```
rank=round(trace(ginv(a)*a));
```

RANKTIE Function

ranks matrix elements using tie-averaging

RANKTIE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The RANKTIE function creates a new matrix containing elements that are the ranks of the corresponding elements of *matrix*. The ranks of tied values are averaged.

For example, the statements

```
x={2 2 1 0 5};
y=ranktie(x);
```

produce the vector

```
    Y
3.5       3.5         2         1         5
```

The RANKTIE function differs from the RANK function in that RANKTIE averages the ranks of tied values, whereas RANK breaks ties arbitrarily.

RATES Function

calculates a column vector of interest rates converted from one base to another

RATES($rates, oldfreq, newfreq$)

the RATES function returns an n x 1 vector of interest rates converted from one base to another.

rates is an n x 1 column vector of rates.
Elements should be positive.

oldfreq is a scalar which represents the old base.
If positive, it represents discrete compounding as the reciprocal of the number of compoundings.
If zero, it represents continuous compounding.
If -1, it represents discount factors.
No other negative values are allowed

newfreq is a scalar which represents the new base.
If positive, it represents discrete compounding as the reciprocal of the number of compoundings.
If zero, it represents continuous compounding.
If -1, it represents per-period discount factors.
No other negative values are allowed

Let $D(t)$ be the discount function, which is the present value of a unit amount to be received t periods from now. The discount function can be expressed in three different ways:
with per-unit-time-period discount factors d_t:

$$D(t) = d_t^t$$

with continuous compunding:

$$D(t) = e^{-r_t t}$$

with discrete compunding:

$$D(t) = (1 + fr)^{-(t/f)}$$

where $f > 0$ is the frequency, the reciprocal of the number of compoundings per unit time period. The $RATES$ function converts between these three representations.

Example proc iml;
rates=do(.1,.3,.1);
oldfreq=0;
newfreq=0;
rates2=T(rates);
rates=rates(rates2,oldfreq,newfreq);
print rates;
quit;

RATES
0.1
0.2
0.3

RATIO Function

divides matrix polynomials

returns a matrix containing the terms of $\Phi(B)^{-1}\Theta(B)$ considered as a matrix of rational functions in B that have been expanded as power series

RATIO(*ar, ma, terms<, dim>***)**

The inputs to the RATIO function are as follows:

ar is an $n \times (ns)$ matrix representing a matrix polynomial generating function, $\Phi(B)$, in the variable B. The first $n \times n$ submatrix represents the constant term and must be nonsingular, the second $n \times n$ submatrix represents the first order coefficients, and so on.

ma is an $n \times (mt)$ matrix representing a matrix polynomial generating function, $\Theta(B)$, in the variable B. The first $n \times m$ submatrix represents the constant term, the second $n \times m$ submatrix represents the first order term, and so on.

terms is a scalar containing the number of terms to be computed, denoted by r in the discussion below. This value must be positive.

dim is a scalar containing the value of m above. The default value is 1.

The RATIO function multiplies a matrix of polynomials by the inverse of another matrix of polynomials. It is useful for expressing univariate and multivariate ARMA models in pure moving-average or pure autoregressive forms.

Note that the order of the first two arguments is reversed from the corresponding PROC MATRIX function.

The value returned is an $n \times (mr)$ matrix containing the terms of $\Phi(B)^{-1}\Theta(B)$considered as a matrix of rational functions in B that have been expanded as power series.

Note: The RATIO function can be used to consolidate the matrix operators employed in a multivariate time-series model of the form

$$\Phi(B)\mathbf{Y}_t = \Theta(B)\epsilon_t$$

where $\Phi(B)$ and $\Theta(B)$ are matrix polynomial operators whose first matrix coefficients are identity matrices. The RATIO function can be used to compute a truncated form of $\Psi(B) = \Phi(B)^{-1}\Theta(B)$ for the equivalent infinite order model

$$\mathbf{Y}_t = \Psi(B)\epsilon_t \ .$$

The RATIO function can also be employed for simple scalar polynomial division, giving a truncated form of $\theta(x)/\phi(x)$ for two scalar polynomials $\theta(x)$ and $\phi(x)$.

The cumulative sum of the elements of a column vector $\mathbf{x}$ can be obtained using

```
ratio({ 1 -1} ,x,ncol(x));
```

Consider the following example for multivariate ARMA(1,1):

```
ar={1 0 -.5  2,
    0 1   3 -.8};
ma={1 0 .9  .7,
    0 1   2 -.4};
psi=ratio(ar,ma,4,2);
```

The matrix produced in

```
          PSI
            1    0   1.4  -1.3   2.7  -1.45  11.35
:    -9.165

            0    1    -1   0.4   -5   4.22  -12.1
:     7.726
```

RDODT and RUPDT Calls

downdate and update QR and Cholesky decompositions

CALL RDODT(*def, rup, bup, sup, r, z <, b, y <, ssq>>***);**
CALL RUPDT(*rup, bup, sup, r, z <, b, y <, ssq>>***);**

The RDODT and RUPDT subroutines return the values:

def is only used for downdating, and it specifies whether the downdating of matrix $\mathbf{R}$ by using the q rows in argument z has been successful. The result *def*=2 means that the downdating of $\mathbf{R}$ by at least one row of $\mathbf{Z}$ leads to a singular matrix and cannot be completed successfully (since the result of downdating is not unique). In that case, the results *rup*, *bup*, and *sup* contain missing values only. The result *def*=1 means that the residual sum of squares, *ssq*, could not be downdated successfully and the result *sup* contains missing values only. The result *def*=0 means that the downdating of $\mathbf{R}$ by $\mathbf{Z}$ was completed successfully.

rup is the $n \times n$ upper triangular matrix $\mathbf{R}$ that has been updated or downdated by using the q rows in $\mathbf{Z}$.

bup is the $n \times p$ matrix $\mathbf{B}$ of right-hand sides that has been updated or downdated by using the q rows in argument y. If the argument b is not specified, *bup* is not computed.

sup is a p vector of square roots of residual sum of squares that is updated or downdated by using the q rows of argument y. If *ssq* is not specified, *sup* is not computed.

The inputs to the RDODT and RUPDT subroutines are as follows:

r specifies an $n \times n$ upper triangular matrix $\mathbf{R}$ to be updated or downdated by the q rows in $\mathbf{Z}$. Only the upper triangle of $\mathbf{R}$ is used; the lower triangle can contain any information.

z specifies a $q \times n$ matrix $\mathbf{Z}$ used rowwise to update or downdate the matrix $\mathbf{R}$.

b specifies an optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with $\mathbf{R}$. If b is specified, the argument y must also be specified.

y specifies an optional $q \times p$ matrix $\mathbf{Y}$ used rowwise to update or downdate the right-hand side matrix $\mathbf{B}$. If b is specified, the argument y must also be specified.

ssq is an optional p vector that, if b is specified, specifies the square root of the error sum of squares that should be updated or downdated simultaneously with $\mathbf{R}$ and $\mathbf{B}$.

The upper triangular matrix $\mathbf{R}$ of the QR decomposition of an $m \times n$ matrix $\mathbf{A}$,

$$\mathbf{A} = \mathbf{QR}, \text{ where } \mathbf{Q}'\mathbf{Q} = \mathbf{QQ}' = \mathbf{I}_m$$

is recomputed efficiently in two cases:

- *update*: An n vector $\mathbf{z}$ is added to matrix $\mathbf{A}$.
- *downdate*: An n vector $\mathbf{z}$ is deleted from matrix $\mathbf{A}$.

Computing the whole QR decomposition of matrix $\mathbf{A}$ by Householder transformations requires $4mn^2 - 4n^3/3$ floating point operations, whereas updating or downdating the QR decomposition (by Givens rotations) of one row vector $\mathbf{z}$ requires only $2n^2$ floating point operations.

If the QR decomposition is used to solve the full rank linear least-squares problem

$$\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|^2 = \textit{ssq}$$

by solving the nonsingular upper triangular system

$$\mathbf{x} = \mathbf{R}^{-1}\mathbf{Q}'\mathbf{b}$$

then the RUPDT and RDODT subroutines can be used to update or downdate the p-transformed right-hand sides $\mathbf{Q}'\mathbf{B}$ and the residual sum-of-squares p vector *ssq* provided that for each n vector $\mathbf{z}$ added to or deleted from $\mathbf{A}$ there is also a p vector $\mathbf{y}$ added to or deleted from the $m \times p$ right-hand-side matrix $\mathbf{B}$.

If the arguments z and y of the subroutines RUPDT and RDODT contain $q > 1$ row vectors for which $\mathbf{R}$ (and $\mathbf{Q}'\mathbf{B}$, and eventually *ssq*) is to be updated or downdated, the process is performed stepwise by processing the rows $\mathbf{z}_k$ (and $\mathbf{y}_k$), $k = 1, \ldots, q$, in the order in which they are stored.

The QR decomposition of an $m \times n$ matrix $\mathbf{A}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$,

$$\mathbf{A} = \mathbf{QR}, \text{ where } \mathbf{Q}'\mathbf{Q} = \mathbf{QQ}' = \mathbf{I}_m$$

corresponds to the Cholesky factorization

$$\mathbf{C} = \mathbf{R}'\mathbf{R}, \text{ where } \mathbf{C} = \mathbf{A}'\mathbf{A}$$

of the positive definite $n \times n$ crossproduct matrix $\mathbf{C} = \mathbf{A}'\mathbf{A}$. In the case where $m \geq n$ and $\text{rank}(\mathbf{A}) = n$, the upper triangular matrix $\mathbf{R}$ computed by the QR decomposition (with positive diagonal elements) is the same as the one computed by Cholesky factorization except for numerical error,

$$\mathbf{A}'\mathbf{A} = (\mathbf{QR})'(\mathbf{QR}) = \mathbf{R}'\mathbf{R}$$

Adding a row vector $\mathbf{z}$ to matrix $\mathbf{A}$ corresponds to the rank-1 modification of the crossproduct matrix $\mathbf{C}$

$$\widetilde{\mathbf{C}} = \mathbf{C} + \mathbf{z}'\mathbf{z}, \text{ where } \widetilde{\mathbf{C}} = \widetilde{\mathbf{A}}'\widetilde{\mathbf{A}}$$

and the $(m+1) \times n$ matrix $\widetilde{\mathbf{A}}$ contains all rows of $\mathbf{A}$ with the row $\mathbf{z}$ added.

Deleting a row vector $\mathbf{z}$ from matrix $\mathbf{A}$ corresponds to the rank-1 modification

$$\mathbf{C}^* = \mathbf{C} - \mathbf{z}'\mathbf{z}, \text{ where } \mathbf{C}^* = \mathbf{A}^{*\prime}\mathbf{A}^*$$

and the $(m-1) \times n$ matrix $\mathbf{A}^*$ contains all rows of $\mathbf{A}$ with the row $\mathbf{z}$ deleted. Thus, you can also use the subroutines RUPDT and RDODT to update or downdate the Cholesky factor $\mathbf{R}$ of a positive definite crossproduct matrix $\mathbf{C}$ of $\mathbf{A}$.

The process of downdating an upper triangular matrix $\mathbf{R}$ (and eventually a residual sum-of-squares vector *ssq*) is not always successful. First of all, the downdated matrix $\mathbf{R}$ could be rank deficient. Even if the downdated matrix $\mathbf{R}$ is of full rank, the process of downdating can be ill conditioned and does not work well if the downdated matrix is close (by rounding errors) to a rank-deficient one. In these cases, the downdated matrix $\mathbf{R}$ is not unique and cannot be computed by subroutine RDODT. If $\mathbf{R}$ cannot be computed, *def* returns 2, and the results *rup*, *bup*, and *sup* return missing values.

The downdating of the residual sum-of-squares vector *ssq* can be a problem, too. In practice, the downdate formula

$$ssq_{\text{new}} = \sqrt{ssq_{\text{old}} - ssq_{\text{dod}}}$$

cannot always be computed because, due to rounding errors, the radicand can be negative. In this case, the result vector *sup* returns missing values, and *def* returns 1.

You can use various methods to compute the p columns $\mathbf{x}_k$ of the $n \times p$ matrix $\mathbf{X}$ that minimize the p linear least-squares problems with an $m \times n$ coefficient matrix $\mathbf{A}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$, and p right-hand-side vectors $\mathbf{b}_k$ (stored columnwise in the $m \times p$ matrix $\mathbf{B}$). The first of the following methods solves the *normal equations* and cannot be applied to the example with the 6×5 Hilbert matrix since too much rounding error is introduced. Therefore, use the following simple example:

```
proc iml;
   a = { 1 3 ,
         2 2 ,
         3 1 };
   b = { 1, 1, 1};
   m = nrow(a);
   n = ncol(a);
   p = 1;
```

- Cholesky Decomposition of Crossproduct Matrix:

```
aa = a‘ * a; ab = a‘ * b;
r  = root(aa);
x  = trisolv(2,r,ab);
x  = trisolv(1,r,x);
```

- QR Decomposition by Householder Transformations:

```
call qr(qtb,r,piv,lindep,a, ,b);
x = trisolv(1,r[,piv],qtb[1:n,]);
```

- Stepwise Update by Givens Rotations:

```
r = j(n,n,0.); qtb = j(n,p,0.); ssq = j(1,p,0.);
do i = 1 to m;
   z = a[i,];
   y = b[i,];
   call rupdt(rup,bup,sup,r,z,qtb,y,ssq);
   r   = rup;
   qtb = bup;
   ssq = sup;
end;
x = trisolv(1,r,qtb);
```

  Or equivalently:

```
r   = j(n,n,0.);
qtb = j(n,p,0.);
ssq = j(1,p,0.);
call rupdt(rup,bup,sup,r,a,qtb,b,ssq);
x   = trisolv(1,rup,bup);
```

- Singular Value Decomposition:

```
call svd(u,d,v,a);
d = diag(1 / d);
x = v * d * u‘ * b;
```

For the preceding 3×2 example matrix $\mathbf{A}$, each method obtains the unique LS estimator:

```
ss = ssq(a * x - b);
print ss x;
```

To compute the (transposed) matrix $\mathbf{Q}$, you can use the following specification:

```
r = shape(0,n,n);
y = i(m);
qt = shape(0,n,m);
call rupdt(rup,qtup,sup,r,a,qt,y);
```

READ Statement

reads observations from a data set

READ <*range*> <**VAR** *operand*> <**WHERE(***expression***)**>
<**INTO** *name* <**[ROWNAME=***row-name*

COLNAME=*column-name***]**>> **;**

The inputs to the READ function are as follows:

range	specifies a range of observations.
operand	selects a set of variables.
expression	is evaluated for being true or false.
name	is the name of the target matrix.
row-name	is a character matrix or quoted literal giving descriptive row labels.
column-name	is a character matrix or quoted literal giving descriptive column labels.

The clauses and options are explained below.

Use the READ statement to read variables or records from the current SAS data set into column matrices of the VAR clause or into the single matrix of the INTO clause. When the INTO clause is used, each variable in the VAR clause becomes a column of the target matrix, and all variables in the VAR clause must be of the same type. If you specify no VAR clause, the default variables for the INTO clause are all numeric variables. Read all character variables into a target matrix by using VAR _CHAR_.

You can specify a *range* of observations with a keyword or by record number using the POINT option. You can use any of the following keywords to specify a range:

ALL	all observations
CURRENT	the current observation
NEXT <*number*>	the next observation or the next *number* of observations
AFTER	all observations after the current one
POINT *operand*	observations specified by number, where *operand* can be one of the following.

Operand	Example
a single record number	`point 5`
a literal giving several record numbers	`point {2 5 10}`
the name of a matrix containing record numbers	`point p`
an expression in parentheses	`point (p+1)`

If the current data set has an index in use, the POINT option is invalid.

You can specify a set of variables to use with the VAR clause. The *operand* in the VAR clause can be one of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of keywords described below:

  ALL for all variables
  CHAR for all character variables
  NUM for all numeric variables.

Examples showing each possible way you can use the VAR clause follow.

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the *range* specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable is a variable in the SAS data set.

comparison-op is one of the following comparison operators:

< less than
<= less than or equal to
= equal to
> greater than

>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

You can specify ROWNAME= and COLNAME= matrices as part of the INTO clause. The COLNAME= matrix specifies the name of a new character matrix to be created. This COLNAME= matrix is created in addition to the target matrix of the INTO clause and contains variable names from the input data set corresponding to columns of the target matrix. The COLNAME= matrix has dimension $1 \times nvar$, where *nvar* is the number of variables contributing to the target matrix.

The ROWNAME= option specifies the name of a character variable in the input data set. The values of this variable are put in a character matrix with the same name as the variable. This matrix has the dimension $nobs \times 1$, where *nobs* is the number of observations in the range of the READ statement. The *range*, VAR, WHERE, and INTO clauses are all optional and can be specified in any order.

Row names created via a READ statement are permanently associated with the INTO matrix. You do not need to use a MATTRIB statement to get this association.

For example, to read all observations from the data set variables NAME and AGE, use a READ statement with the VAR clause and the keyword ALL for the *range* operand. This creates two IML variables with the same names as the data set variables.

```
read all var{name age};
```

To read all variables for the 23rd observation only, use the statement

```
read point 23;
```

To read the data set variables NAME and ADDR for all observations with a STATE value of **NJ**, use the statement

```
read all var{name addr} where(state="NJ");
```

See Chapter 6, "Working with SAS Data Sets," for further information.

REMOVE Function

discards elements from a matrix

REMOVE(*matrix, indices***)**

The inputs to the REMOVE function are as follows:

matrix is a numeric or character matrix or literal.

indices refers to a matrix containing the indices of elements that are removed from *matrix*.

The REMOVE function returns as a row vector elements of the first argument, with elements corresponding to the indices in the second argument discarded and the gaps removed. The first argument is indexed in row-major order, as in subscripting, and the indices must be in the range 1 to the number of elements in the first argument. Non-integer indices are truncated to their integer part. You can repeat the indices, and you can give them in any order. If all elements are removed, the result is a null matrix (zero rows and zero columns).

Thus, the statement

```
a=remove({ 5 6, 7 8} , 3);
```

removes the third element, producing the result shown:

```
A
5   6   8
```

The statement

```
a=remove({ 5 6 7 8} , { 3 2 3 1} );
```

causes all but the fourth element to be removed, giving the result shown:

```
A
8
```

REMOVE Statement

removes matrices from storage

REMOVE <**MODULE=(***module-list***)** <*matrix-list*>>**;**

The inputs to the REMOVE statement are as follows:

module-list specifies a module or modules to remove from storage.

matrix-list specifies a matrix or matrices to remove from storage.

The REMOVE statement removes matrices or modules or both from the current library storage. For example, the statement below removes the three modules A, B, and C and the matrix X:

```
remove module=(A B C) X;
```

The special operand _ALL_ can be used to remove all matrices or all modules or both. For example, the following statement removes everything:

```
remove _all_ module=_all_;
```

See Chapter 14, "Storage Features," and also the descriptions of the LOAD, STORE, RESET, and SHOW statements for related information.

RENAME Call

renames a SAS data set

CALL RENAME(<*libname,*> *member-name, new-name***);**

The inputs to the RENAME subroutine are as follows:

libname is a character matrix or quoted literal containing the name of the SAS data library.

member-name is a character matrix or quoted literal containing the current name of the data set.

new-name is a character matrix or quoted literal containing the new data set name.

The RENAME subroutine renames a SAS data set in the specified library. All of the arguments can directly be specified in quotes, although quotes are not required. If a one-word data set name is specified, the libname specified by the RESET *deflib* statement is used. Examples of valid statements follow:

```
call rename('a','b');
call rename(a,b);
call rename(work,a,b);
```

REPEAT Function

creates a new matrix of repeated values

REPEAT(*matrix, nrow, ncol***)**

The inputs to the REPEAT function are as follows:

matrix	is a numeric matrix or literal.
nrow	gives the number of times *matrix* is repeated across rows.
ncol	gives the number of times *matrix* is repeated across columns.

The REPEAT function creates a new matrix by repeating the values of the argument matrix *nrow***ncol* times, *ncol* times across the rows, and *nrow* times down the columns. The *matrix* argument can be numeric or character. For example, the following statements result in the matrix $\mathbf{Y}$, repeating the $\mathbf{X}$ matrix twice down and three times across:

```
x={ 1 2 ,
  3 4} ;
y=repeat(x,2,3);
```

```
  Y
  1        2        1        2        1        2
  3        4        3        4        3        4
  1        2        1        2        1        2
  3        4        3        4        3        4
```

REPLACE Statement

replaces values in observations and updates observations

REPLACE <*range*> <**VAR** *operand*> <**WHERE(***expression***)**>**;**

The inputs to the REPLACE statement are as follows:

range	specifies a range of observations.
operand	selects a set of variables.
expression	is evaluated for being true or false.

The REPLACE statement replaces the values of observations in a SAS data set with current values of IML matrices with the same name. Use the *range*, VAR, and WHERE arguments to limit replacement to specific variables and observations. Replacement matrices should be the same type as the data set variables. The REPLACE

statement uses matrix elements in row order replacing the value in the *i*th observation with the *i*th matrix element. If there are more observations in *range* than matrix elements, the REPLACE statement continues to use the last matrix element.

For example, the statements below cause all occurrences of **ILL** to be replaced by **IL** for the variable STATE:

```
state="IL";
replace all var{state} where(state="ILL");
```

You can specify a *range* of observations with a keyword or by record number using the POINT option. You can use any of the following keywords to specify a range:

ALL	all observations
CURRENT	the current observation
NEXT <*number*>	the next observation or the next *number* of observations
AFTER	all observations after the current one
POINT *operand*	observations by number, where *operand* can be one of the following:

Operand	Example
a single record number	`point 5`
a literal giving several record numbers	`point {2 5 10}`
the name of a matrix containing record numbers	`point p`
an expression in parentheses	`point (p+1)`

If the current data set has an index in use, the POINT option is invalid.

You can specify a set of variables to use with the VAR clause. The *variables* argument can have the following values:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of the keywords described below:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

Examples showing each possible way you can use the VAR clause follow.

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the range specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable is a variable in the SAS data set.

comparison-op is any one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data setvariables, and the expression on the right-hand side refers to matrix values.

The code statement below replaces all variables in the current observation:

```
replace;
```

RESET Statement

sets processing options

RESET <*options*>**;**

where the *options* are described below.

The RESET statement sets processing options. The options described below are currently implemented options. Note that the prefix NO turns off the feature where indicated. For options that take operands, the operand should be a literal, a name of a matrix containing the value, or an expression in parentheses. The SHOW *options* statement displays the current settings of all of the options.

AUTONAME
NOAUTONAME
specifies whether rows are automatically labeled ROW1, ROW2, and so on, and columns are labeled COL1, COL2, and so on, when a matrix is printed. Row-name and column-name attributes specified in the PRINT statement or associated via the MATTRIB statement override the default labels. The AUTONAME option causes the SPACES option to be reset to 4. The default is NOAUTONAME.

CENTER
NOCENTER
specifies whether output from the PRINT statement is centered on the page. The default is CENTER.

CLIP
NOCLIP
specifies whether SAS/IML graphs are automatically clipped outside the viewport; that is, any data falling outside the current viewport is not displayed. NOCLIP is the default.

DEFLIB=*operand*
specifies the default libname for SAS data sets when no other libname is given. This defaults to USER if a USER libname is set up, or WORK if not. The libname operand can be specified with or without quotes.

DETAILS
NODETAILS
specifies whether additional information is printed from a variety of operations, such as when files are opened and closed. The default is NODETAILS.

FLOW
NOFLOW
specifies whether operations are shown as executed. It is used for debugging only. The default is NOFLOW.

FUZZ <**=***number*>
NOFUZZ
specifies whether very small numbers are printed as zero rather than in scientific notation. If the absolute value of the number is less than the value specified in *number*, it will be printed as 0. The *number* argument is optional, and the default value varies across hosts but is typically around 1E−12. The default is NOFUZZ.

FW=*number*
sets the field width for printing numeric values. The default field width is 9.

LINESIZE=*n*
specifies the linesize for printing. The default value is usually 78.

LOG
NOLOG
specifies whether output is routed to the log file rather than to the print file. On the log, the results are interleaved with the statements and messages. The NOLOG option routes output to the OUTPUT window in display manager and to the listing file in batch modes. The default is NOLOG.

NAME
NONAME
specifies whether the matrix name or label is printed with the value for the PRINT statement. The default is NAME.

PAGESIZE=*n*
specifies the pagesize for printing. The default value is usually 21.

PRINT
NOPRINT
specifies whether the final results from assignment statements are printed automatically. NOPRINT is the default.

PRINTALL
NOPRINTALL
specifies whether the intermediate and final results are printed automatically. The default is NOPRINTALL.

SPACES=*n*
specifies the number of spaces between adjacent matrices printed across the page. The default value is 1, except when AUTONAME is on. Then, the default value is 4.

STORAGE=<*libname.*>*memname***;**
specifies the file to be the current library storage for STORE and LOAD statements. The default library storage is SASUSER.IMLSTOR. The *libname* argument is optional and defaults to SASUSER. It can be specified with or without quotes.

RESUME Statement

resumes execution

RESUME;

The RESUME statement enables you to continue execution from the line in the module where the most recent PAUSE statement was executed. PROC IML issues an automatic pause when an error occurs inside a module. If a module was paused due to an error, the RESUME statement resumes execution immediately after the statement that caused the error. The SHOW *pause* statement displays the current state of all paused modules.

RETURN Statement

returns to caller

RETURN<**(***operand***)**>**;**

where *operand* is the value of the function returned. Use *operand* only in function modules.

The RETURN statement causes IML to return to the calling point in a program. If a LINK statement has been issued, IML returns to the statement following the LINK. If no LINK statement was issued, the RETURN statement exits a module. If not in a module, execution is stopped (as with a STOP statement), and IML looks for more statements to parse.

The RETURN statement with an *operand* is used in function modules that return a value. The *operand* can be a variable name or an expression. It is evaluated, and the value is returned.

See the description of the LINK statement. Also, see Chapter 5, "Programming Statements," for details.

If you use a LINK statement, you need a RETURN statement at the place where you want to go back to the statement after LINK.

If you are writing a function, use a RETURN to return the value of the function. An example is shown below.

```
start sum1(a,b);
   sum=a+b;
   return(sum);
finish;
```

ROOT Function

performs the Cholesky decomposition of a matrix

ROOT(*matrix***)**

where *matrix* is a symmetric positive-definite matrix.

The ROOT function performs the Cholesky decomposition of a matrix (for example, $\mathbf{A}$) such that

$$\mathbf{U}'\mathbf{U} = \mathbf{A}$$

where $\mathbf{U}$ is upper triangular. The matrix $\mathbf{A}$ must be symmetric and positive definite.

For example, the statements

```
xpx={4 15, 15 85};
a=root(xpx);
```

produce the result shown below:

```
                A             2 rows      2 cols    (numeric)
                                    2         7.5
                                    0 5.3619026
```

ROWCAT Function

concatenates rows without using blank compression

ROWCAT(*matrix*<, *rows*<, *columns*>>**);**

The inputs to the ROWCAT function are as follows:

matrix	is a character matrix or quoted literal.
rows	select the rows of *matrix*.
columns	select the columns of *matrix*.

The ROWCAT function takes a character matrix or submatrix as its argument and creates a new matrix with one column whose elements are the concatenation of all row elements into a single string. If the argument has n rows and m columns, the result willhave n rows and 1 column. The element length of the result will be m times the element length of the argument. The optional rows and columns arguments may be used to select which rows and columns are concatenated.

For example, the statements

```
b={"ABC"  "D  "  "EF ",
   " GH"  " I "  " JK"};
a=rowcat(b);
```

produce the 2×1 matrix:

```
   A        2 rows      1 col     (character, size 9)

                      ABCD  EF
                       GH I  JK
```

Quotes (") are needed only if you want to embed blanks or special characters or to maintain uppercase and lowercase distinctions.

The form

ROWCAT(*matrix, rows, columns***)**

returns the same result as

ROWCAT(*matrix***[***rows, columns***])**

The form

ROWCAT(*matrix, rows***)**

returns the same result as

ROWCAT(*matrix***[***rows,***])**

ROWCATC Function

concatenates rows using blank compression

ROWCATC(*matrix*<, *rows*<, *columns*>>**);**

The inputs to the ROWCATC function are as follows:

matrix	is a character matrix or quoted literal.
rows	select the rows of *matrix*.
columns	select the columns of *matrix*.

The ROWCATC function works the same way as the ROWCAT function except that blanks in element strings are moved to the end of the concatenation. For example, the statements

```
b={"ABC"  "D  "  "EF ",
   " GH"  " I "  " JK"};
a=rowcatc(b);
```

produce the matrix **A** as shown:

```
A        2 rows      1 col     (character, size 9)

                    ABCDEF
                    GHIJK
```

Quotes (") are needed only if you want to embed blanks or special characters or to maintain uppercase and lowercase distinctions.

RUN Statement

executes statements in a module

RUN <*name*> <**(***arguments***)**>**;**

The inputs to the RUN statement are as follows:

name is the name of a user-defined module or an IML built-in subroutine.

arguments are arguments to the subroutine. Arguments can be both local and global.

The RUN statement executes a user-defined module or invokes PROC IML's built-in subroutines.

The resolution order for the RUN statement is

1. A user-defined module
2. An IML built-in function or subroutine

This resolution order need only be considered if you have defined a module that has the same name as an IML built-in subroutine. If a RUN statement cannot be resolved at resolution time, a warning is produced. If the RUN statement is still unresolved when executed and a storage library is open at the time, IML attempts to load a module from that storage. If no module is found, then the program is interrupted and an error message is generated. By default, the RUN statement tries to run the module named MAIN.

You will usually want to supply both a name and arguments, as follows.

```
run myf1(a,b,c);
```

See Chapter 5, "Programming Statements," for further details.

RUPDT Call

update QR and Cholesky decompositions

CALL RUPDT(*rup, bup, sup, r, z <, b, y <, ssq>>***);**

See the entry for the RDODT subroutine for details.

RZLIND Call

computes rank deficient linear least-squares solutions, complete orthogonal factorization, and Moore-Penrose inverses

CALL RZLIND(*lindep, rup, bup, r <, sing><, b>***);**

The RZLIND subroutine returns the following values:

lindep is a scalar giving the number of linear dependencies that are recognized in $\mathbf{R}$ (number of zeroed rows in `rup[n,n]`).

rup is the updated $n \times n$ upper triangular matrix $\mathbf{R}$ containing zero rows corresponding to zero recognized diagonal elements in the original $\mathbf{R}$.

bup is the $n \times p$ matrix $\mathbf{B}$ of right-hand sides that is updated simultaneously with $\mathbf{R}$. If b is not specified, *bup* is not accessible.

The inputs to the RZLIND subroutine are as follows:

r specifies the $n \times n$ upper triangular matrix $\mathbf{R}$. Only the upper triangle of r is used; the lower triangle may contain any information.

sing is an optional scalar specifying a relative singularity criterion for the diagonal elements of $\mathbf{R}$. The diagonal element r_{ii} is considered zero if $r_{ii} \leq sing\|\mathbf{r}_i\|$, where $\|\mathbf{r}_i\|$ is the Euclidean norm of column $\mathbf{r}_i$ of $\mathbf{R}$. If the value provided for *sing* is not positive, the default value *sing*$=1000\epsilon$ is used, where ϵ is the relative machine precision.

b specifies the optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with $\mathbf{R}$.

The singularity test used in the RZLIND subroutine is a relative test using the Euclidean norms of the columns $\mathbf{r}_i$ of $\mathbf{R}$. The diagonal element r_{ii} is considered as nearly zero (and the i^{th} row is zeroed out) if the following test is true:

$$r_{ii} \leq sing\|\mathbf{r}_i\|, \quad \text{where } \|\mathbf{r}_i\| = \sqrt{\mathbf{r}_i'\mathbf{r}_i}$$

Providing an argument *sing*≤ 0 is the same as omitting the argument *sing* in the RZLIND call. In this case, the default is *sing*$= 1000\epsilon$, where ϵ is the relative machine precision. If $\mathbf{R}$ is computed by the QR decomposition $\mathbf{A} = \mathbf{QR}$, then the Euclidean norm of column i of $\mathbf{R}$ is the same (except for rounding errors) as the Euclidean norm of column i of $\mathbf{A}$.

Consider the following possible application of the RZLIND subroutine. Assume that you want to compute the upper triangular Cholesky factor $\mathbf{R}$ of the $n \times n$ positive semidefinite matrix $\mathbf{A}'\mathbf{A}$,

$$\mathbf{A}'\mathbf{A} = \mathbf{R}'\mathbf{R} \text{ where } \mathbf{A} \in \mathcal{R}^{m \times n},\ \mathrm{rank}(\mathbf{A}) = r,\ r \leq n \leq m$$

The Cholesky factor $\mathbf{R}$ of a positive definite matrix $\mathbf{A}'\mathbf{A}$ is unique (with the exception of the sign of its rows). However, the Cholesky factor of a positive semidefinite (singular) matrix $\mathbf{A}'\mathbf{A}$ can have many different forms.

In the following example, $\mathbf{A}$ is a 12×8 matrix with linearly dependent columns $\mathbf{a}_1 = \mathbf{a}_2 + \mathbf{a}_3 + \mathbf{a}_4$ and $\mathbf{a}_1 = \mathbf{a}_5 + \mathbf{a}_6 + \mathbf{a}_7$ with $r = 6$, $n = 8$, and $m = 12$.

```
proc iml;
   a = {1 1 0 0 1 0 0,
        1 1 0 0 1 0 0,
        1 1 0 0 0 1 0,
        1 1 0 0 0 0 1,
        1 0 1 0 1 0 0,
        1 0 1 0 0 1 0,
        1 0 1 0 0 1 0,
        1 0 1 0 0 0 1,
        1 0 0 1 1 0 0,
        1 0 0 1 0 1 0,
        1 0 0 1 0 0 1,
        1 0 0 1 0 0 1};
   a = a || uniform(j(12,1,1));
   aa = a' * a;
   m = nrow(a); n = ncol(a);
```

Applying the ROOT function to the coefficient matrix $\mathbf{A}'\mathbf{A}$ of the normal equations,

```
r1 = root(aa);
ss1 = ssq(aa - r1' * r1);
print ss1 r1 [format=best6.];
```

generates an upper triangular matrix $\mathbf{R}_1$ where linearly dependent rows are zeroed out, and you can verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_1'\mathbf{R}_1$.

Applying the QR subroutine with column pivoting on the original matrix $\mathbf{A}$ yields a different result, but you can also verify $\mathbf{A}'\mathbf{A} = \mathbf{R}_2'\mathbf{R}_2$ after pivoting the rows and columns of $\mathbf{A}'\mathbf{A}$:

```
ord = j(n,1,0);
call qr(q,r2,pivqr,lindqr,a,ord);
ss2 = ssq(aa[pivqr,pivqr] - r2' * r2);
print ss2 r2 [format=best6.];
```

Using the RUPDT subroutine for stepwise updating of $\mathbf{R}$ by the m rows of $\mathbf{A}$ will finally result in an upper triangular matrix $\mathbf{R}_3$ with $n - r$ nearly zero diagonal elements. However, other elements in rows with nearly zero diagonal elements can have significant values. The following statements verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_3'\mathbf{R}_3$,

```
r3 = shape(0,n,n);
call rupdt(rup,bup,sup,r3,a);
r3 = rup;
ss3 = ssq(aa - r3' * r3);
print ss3 r3 [format=best6.];
```

The result $\mathbf{R}_3$ of the RUPDT subroutine can be transformed into the result $\mathbf{R}_1$ of the ROOT function by left applications of Givens rotations to zero out the remaining significant elements of rows with *small* diagonal elements. Applying the RZLIND subroutine on the upper triangular result $\mathbf{R}_3$ of the RUPDT subroutine will generate a Cholesky factor $\mathbf{R}_4$ with zero rows corresponding to diagonal elements that are small, giving the same result as the ROOT function (except for the sign of rows) if its singularity criterion recognizes the same linear dependencies.

```
call rzlind(lind,r4,bup,r3);
ss4 = ssq(aa - r4' * r4);
print ss4 r4 [format=best6.];
```

Consider the rank-deficient linear least-squares problem:

$$\min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 \text{ where } \mathbf{A} \in \mathcal{R}^{m \times n},\ \text{rank}(\mathbf{A}) = r,\ r \leq n \leq m$$

For $r = n$, the optimal solution, $\hat{\mathbf{x}}$, is unique; however, for $r < n$, the rank-deficient linear least-squares problem has many optimal solutions, each of which has the same least-squares residual sum of squares:

$$\text{ss} = (\mathbf{A}\hat{\mathbf{x}} - \mathbf{b})'(\mathbf{A}\hat{\mathbf{x}} - \mathbf{b})$$

The solution of the full rank problem, $r = n$, is illustrated in the QR call. The following list shows several solutions to the singular problem. This example uses the 12×8 matrix from the preceding section and generates a new column vector $\mathbf{b}$. The vector $\mathbf{b}$ and the matrix $\mathbf{A}$ are shown in the output.

```
b = uniform(j(12,1,1));
ab = a' * b;
print b a [format=best6.];
```

Each entry in the following list solves the rank-deficient linear least-squares problem. Note that while each method minimizes the residual sum of squares, not all of the given solutions are of minimum Euclidean length.

- Use the singular value decomposition of $\mathbf{A}$, given by $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}'$. Take the reciprocals of significant singular values and set the small values of $\mathbf{D}$ to zero.

```
call svd(u,d,v,a);
t = 1e-12 * d[1];
do i=1 to n;
   if d[i] < t then d[i] = 0.;
   else d[i] = 1. / d[i];
end;
x1 = v * diag(d) * u' * b;
len1 = x1' * x1;
ss1 = ssq(a * x1 - b);
x1 = x1';
print ss1 len1, x1 [format=best6.];
```

The solution $\hat{\mathbf{x}}_1$ obtained by singular value decomposition, $\hat{\mathbf{x}}_1 = \mathbf{V}\mathbf{D}^{-}\mathbf{U}'\mathbf{b}/4$, is of minimum Euclidean length.

- Use QR decomposition with column pivoting:

$$\mathbf{A}\mathbf{\Pi} = \mathbf{Q}\mathbf{R} = \begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \mathbf{Y} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \end{bmatrix}$$

Set the right part $\mathbf{R}_2$ to zero and invert the upper triangular matrix $\mathbf{R}_1$ to obtain a generalized inverse $\mathbf{R}^{-}$ and an optimal solution $\hat{\mathbf{x}}_2$:

$$\mathbf{R}^{-} = \begin{bmatrix} \mathbf{R}_1^{-1} \\ \mathbf{0} \end{bmatrix} \qquad \hat{\mathbf{x}}_2 = \mathbf{\Pi}\mathbf{R}^{-}\mathbf{Y}'\mathbf{b}$$

```
ord = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr = n - lindqr;
r = r2[1:nr,1:nr];
x2 = shape(0,n,1);
x2[pivqr] = trisolv(1,r,qtb[1:nr]) // j(lindqr,1,0.);
len2 = x2' * x2;
ss2 = ssq(a * x2 - b);
x2 = x2';
print ss2 len2, x2 [format=best6.];
```

Note that the residual sum of squares is minimal, but the solution $\hat{\mathbf{x}}_2$ is not of minimum Euclidean length.

- Use the result $\mathbf{R}_1$ of the ROOT function on page 728 to obtain the vector *piv* indicating the zero rows in the upper triangular matrix $\mathbf{R}_1$:

```
r1 = root(aa);
nr = n - lind;
piv = shape(0,n,1);
j1 = 1; j2 = nr + 1;
do i=1 to n;
   if r1[i,i] ^= 0 then do;
      piv[j1] = i; j1 = j1 + 1;
   end;
   else do;
      piv[j2] = i; j2 = j2 + 1;
   end;
end;
```

Now compute $\hat{\mathbf{x}}_3$ by solving the equation $\hat{\mathbf{x}}_3 = \mathbf{R}^{-1}\mathbf{R}^{-\prime}\mathbf{A}'\mathbf{b}$.

```
r  = r1[piv[1:nr],piv[1:nr]];
x  = trisolv(2,r,ab[piv[1:nr]]);
x  = trisolv(1,r,x);
x3 = shape(0,n,1);
x3[piv] = x // j(lind,1,0.);
len3 = x3' * x3;
ss3  = ssq(a * x3 - b);
x3 = x3';
print ss3 len3, x3 [format=best6.];
```

Note that the residual sum of squares is minimal, but the solution $\hat{\mathbf{x}}_3$ is not of minimum Euclidean length.

- Use the result $\mathbf{R}_3$ of the RUPDT call on page 729 and the vector *piv* (obtained in the previous solution), which indicates the zero rows of upper triangular matrices $\mathbf{R}_1$ and $\mathbf{R}_3$. After zeroing out the rows of $\mathbf{R}_3$ belonging to small diagonal pivots, solve the system $\hat{\mathbf{x}}_4 = \mathbf{R}^{-1}\mathbf{Y}'\mathbf{b}$.

```
r3 = shape(0,n,n);
qtb = shape(0,n,1);
call rupdt(rup,bup,sup,r3,a,qtb,b);
r3 = rup; qtb = bup;
call rzlind(lind,r4,bup,r3,,qtb);
qtb = bup[piv[1:nr]];
x  = trisolv(1,r4[piv[1:nr],piv[1:nr]],qtb);
x4 = shape(0,n,1);
x4[piv] = x // j(lind,1,0.);
len4 = x4' * x4;
ss4 = ssq(a * x4 - b);
x4 = x4';
print ss4 len4, x4 [format=best6.];
```

Since the matrices $\mathbf{R}_4$ and $\mathbf{R}_1$ are the same (except for the signs of rows), the solution $\hat{\mathbf{x}}_4$ is the same as $\hat{\mathbf{x}}_3$.

- Use the result $\mathbf{R}_4$ of the RZLIND call in the previous solution, which is the result of the first step of *complete QR decomposition*, and perform the second step of complete QR decomposition. The rows of matrix $\mathbf{R}_4$ can be permuted to the upper trapezoidal form

$$\begin{bmatrix} \widehat{\mathbf{R}} & \mathbf{T} \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

where $\widehat{\mathbf{R}}$ is nonsingular and upper triangular and $\mathbf{T}$ is rectangular. Next, perform the second step of complete QR decomposition with the lower triangular matrix

$$\begin{bmatrix} \widehat{\mathbf{R}}' \\ \mathbf{T}' \end{bmatrix} = \bar{\mathbf{Y}} \begin{bmatrix} \bar{\mathbf{R}} \\ \mathbf{0} \end{bmatrix},$$

which leads to the upper triangular matrix $\bar{\mathbf{R}}$.

```
r = r4[piv[1:nr],]';
call qr(q,r5,piv2,lin2,r);
y  = trisolv(2,r5,qtb);
x5 = q * (y // j(lind,1,0.));
len5 = x5' * x5;
ss5 = ssq(a * x5 - b);
x5 = x5';
print ss5 len5, x5 [format=best6.];
```

The solution $\hat{\mathbf{x}}_5$ obtained by complete QR decomposition has minimum Euclidean length.

- Perform both steps of complete QR decomposition. The first step performs the pivoted QR decomposition of $\mathbf{A}$,

$$\mathbf{A\Pi} = \mathbf{QR} = \mathbf{Y}\begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} = \mathbf{Y}\begin{bmatrix} \widehat{\mathbf{R}}\mathbf{T} \\ \mathbf{0} \end{bmatrix}$$

where $\widehat{\mathbf{R}}$ is nonsingular and upper triangular and $\mathbf{T}$ is rectangular. The second step performs a QR decomposition as described in the previous method. This results in

$$\mathbf{A\Pi} = \mathbf{Y}\begin{bmatrix} \bar{\mathbf{R}}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\bar{\mathbf{Y}}'$$

where $\bar{\mathbf{R}}'$ is lower triangular.

```
ord  = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr   = n - lindqr;
r    = r2[1:nr,]';
call qr(q,r5,piv2,lin2,r);
y    = trisolv(2,r5,qtb[1:nr]);
x6   = shape(0,n,1);
x6[pivqr] = q * (y // j(lindqr,1,0.));
len6 = x6' * x6;
ss6  = ssq(a * x6 - b);
x6   = x6';
print ss6 len6, x6 [format=best6.];
```

The solution $\hat{\mathbf{x}}_6$ obtained by complete QR decomposition has minimum Euclidean length.

- Perform complete QR decomposition with the QR and LUPDT calls:

```
ord  = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr   = n - lindqr;
r    = r2[1:nr,1:nr]'; z = r2[1:nr,nr+1:n]';
call lupdt(lup,bup,sup,r,z);
rd   = trisolv(3,lup,r2[1:nr,]);
rd   = trisolv(4,lup,rd);
x7   = shape(0,n,1);
x7[pivqr] = rd' * qtb[1:nr,];
len7 = x7' * x7;
```

```
ss7  = ssq(a * x7 - b);
x7 = x7';
print ss7 len7, x7 [format=best6.];
```

The solution $\hat{\mathbf{x}}_7$ obtained by complete QR decomposition has minimum Euclidean length.

- Perform complete QR decomposition with the RUPDT, RZLIND, and LUPDT calls:

```
r3 = shape(0,n,n);
qtb = shape(0,n,1);
call rupdt(rup,bup,sup,r3,a,qtb,b);
r3 = rup; qtb = bup;
call rzlind(lind,r4,bup,r3,,qtb);
nr = n - lind; qtb = bup;
r = r4[piv[1:nr],piv[1:nr]]';
z = r4[piv[1:nr],piv[nr+1:n]]';
call lupdt(lup,bup,sup,r,z);
rd = trisolv(3,lup,r4[piv[1:nr],]);
rd = trisolv(4,lup,rd);
x8 = shape(0,n,1);
x8 = rd' * qtb[piv[1:nr],];
len8 = x8' * x8;
ss8 = ssq(a * x8 - b);
x8 = x8';
print ss8 len8, x8 [format=best6.];
```

The solution $\hat{\mathbf{x}}_8$ obtained by complete QR decomposition has minimum Euclidean length. The same result can be obtained with the APPCORT or COMPORT call.

You can use various methods to compute the Moore-Penrose inverse $\mathbf{A}^-$ of a rectangular matrix $\mathbf{A}$ using orthogonal methods. The entries in the following list find the Moore-Penrose inverse of the matrix $\mathbf{A}$ shown on page 729.

- Use the GINV operator. The GINV operator in IML uses the singular decomposition $\mathbf{A} = \mathbf{UDV}'$. The result $\mathbf{A}^- = \mathbf{VD}^-\mathbf{U}'$ should be identical to the result given by the next solution.

```
ga = ginv(a);
t1 = a * ga; t2 = t1';
t3 = ga * a; t4 = t3';
ss1 = ssq(t1 - t2) + ssq(t3 - t4) +
      ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss1, ga [format=best6.];
```

- Use singular value decomposition. The singular decomposition $\mathbf{A} = \mathbf{UDV}'$ with $\mathbf{U}'\mathbf{U} = \mathbf{I}_m$, $\mathbf{D} = \mathrm{diag}(d_i)$, and $\mathbf{V}'\mathbf{V} = \mathbf{VV}' = \mathbf{I}_n$, can be used to compute $\mathbf{A}^- = \mathbf{VD}^\dagger\mathbf{U}'$, with $\mathbf{D}^\dagger = \mathrm{diag}(d_i^\dagger)$ and

$$d_i^\dagger = \begin{cases} 0 & \text{where } d_i \leq \epsilon \\ 1/d_i & \text{otherwise} \end{cases}$$

The result $\mathbf{A}^-$ should be the same as that given by the GINV operator if the singularity criterion ϵ is selected correspondingly. Since you cannot specify the criterion ϵ for the GINV operator, the singular value decomposition approach can be important for applications where the GINV operator uses an unsuitable ϵ criterion. The slight discrepancy between the values of SS1 and SS2 is due to rounding that occurs in the statement that computes the matrix GA.

```
call svd(u,d,v,a);
do i=1 to n;
   if d[i] <= 1e-10 * d[1] then d[i] = 0.;
   else d[i] = 1. / d[i];
end;
ga = v * diag(d) * u';
t1 = a * ga; t2 = t1';
t3 = ga * a; t4 = t3';
ss2 = ssq(t1 - t2) + ssq(t3 - t4) +
      ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss2;
```

- Use complete QR decomposition. The complete QR decomposition

$$\mathbf{A} = \mathbf{Y} \begin{bmatrix} \bar{\mathbf{R}}' & 0 \\ 0 & 0 \end{bmatrix} \bar{\mathbf{Y}}' \mathbf{\Pi}'$$

where $\bar{\mathbf{R}}'$ is lower triangular, yields the Moore-Penrose inverse

$$\mathbf{A}^- = \mathbf{\Pi} \bar{\mathbf{Y}} \begin{bmatrix} \bar{\mathbf{R}}^{-\prime} & 0 \\ 0 & 0 \end{bmatrix} \mathbf{Y}'$$

```
ord = j(n,1,0);
call qr(q1,r2,pivqr,lindqr,a,ord);
nr = n - lindqr;
q1 = q1[,1:nr]; r = r2[1:nr,]';
call qr(q2,r5,piv2,lin2,r);
tt = trisolv(4,r5',q1');
ga = shape(0,n,m);
ga[pivqr,] = q2 * (tt // shape(0,n-nr,m));
t1 = a * ga; t2 = t1';
t3 = ga * a; t4 = t3';
ss3 = ssq(t1 - t2) + ssq(t3 - t4) +
      ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss3;
```

- Use complete QR decomposition with QR and LUPDT:

```
ord = j(n,1,0);
call qr(q,r2,pivqr,lindqr,a,ord);
nr = n - lindqr;
r = r2[1:nr,1:nr]'; z = r2[1:nr,nr+1:n]';
call lupdt(lup,bup,sup,r,z);
rd = trisolv(3,lup,r2[1:nr,]);
rd = trisolv(4,lup,rd);
ga = shape(0,n,m);
ga[pivqr,] = rd' * q[,1:nr]';
```

```
t1 = a * ga; t2 = t1';
t3 = ga * a; t4 = t3';
ss4 = ssq(t1 - t2) + ssq(t3 - t4) +
      ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss4;
```

- Use complete QR decomposition with RUPDT and LUPDT:

```
r3 = shape(0,n,n);
y = i(m); qtb = shape(0,n,m);
call rupdt(rup,bup,sup,r3,a,qtb,y);
r3 = rup; qtb = bup;
call rzlind(lind,r4,bup,r3,,qtb);
nr = n - lind; qtb = bup;
r = r4[piv[1:nr],piv[1:nr]]';
z = r4[piv[1:nr],piv[nr+1:n]]';
call lupdt(lup,bup,sup,r,z);
rd = trisolv(3,lup,r4[piv[1:nr],]);
rd = trisolv(4,lup,rd);
ga = shape(0,n,m);
ga = rd' * qtb[piv[1:nr],];
t1 = a * ga; t2 = t1';
t3 = ga * a; t4 = t3';
ss5 = ssq(t1 - t2) + ssq(t3 - t4) +
      ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss5;
```

SAVE Statement

saves data

SAVE;

The SAVE statement forces out any data residing in output buffers for all active output data sets and files to ensure that the data are written to disk. This is equivalent to closing and then reopening the files.

SEQ, SEQSCALE, and SEQSHIFT Calls

perform discrete sequential tests

CALL SEQ(*prob, domain* <, <**TSCALE=***tscale*><, <**EPS=***eps*>
<, <**DEN=***den*>>>>**);**

CALL SEQSCALE(*prob, gscale, domain, level*<, <**IGUESS=***iguess*>
<, <**TSCALE=***tscale*><, <**EPS=***eps*><, <**DEN=***den*>>>>>**);**

CALL SEQSHIFT(*prob, shift, domain, plevel*<, <**IGUESS=***iguess*>
<, <**TSCALE=***tscale*><, <**EPS=***eps*><, <**DEN=***den*>>>>>**);**

The SEQSHIFT subroutine returns the following values:

prob is an $(m+1) \times n$ matrix. The $[i, j]$ entry in the array contains the probability at the $[i, j]$ entry of the argument *domain*. Also, the probability at infinity at every level j is returned in the last entry ($[m+1, j]$) of column j. Upon a successful completion of any routine, this variable is always returned.

gscale is a numeric variable that returns from the routine SEQSCALE and contains the scaling of the current geometry defined by *domain* that would yield a given significance level *level*.

shift is a numeric variable that returns from the routine SEQSHIFT and contains the shift of current geometry defined by *domain* that would yield a given power level *plevel*.

The inputs to the SEQSHIFT subroutine are as follows:

domain specifies an $m \times n$ matrix containing the boundary points separating the intervals of continuation/stopping of the sequential test. Each column k contains the boundary points at level k sorted in an ascending order, with `.M` and `.P` representing $-\infty$ and $+\infty$, respectively. They must start on the first row, and any remaining entries must be filled with a missing value. Elements that follow the missing value in any column will be ignored. The number of columns n is equal to the number of stages present in the sequential test. The row dimension m must be even, and it is equal to the maximum number of boundary points in a level. In fact, *domain* is the tabular form of the finite boundary points. Entries in *domain* with absolute values that exceed a standardized value of 8 at any level will be internally reset to a standardized value of 8 or -8, depending on the sign of the entry. This is reflected in the results returned for the probabilities and the densities.

tscale specifies an optional $n-1$ vector that describes the time intervals between two consecutive stages. In the absence of *tscale*, these time intervals will be internally set to 1. The IML keyword for *tscale* is TSCALE.

eps specifies an optional numeric parameter for controlling the absolute precision of the computation. In the absence of *eps*, the precision will be internally set to 1E−7. The IML keyword for *eps* is EPS.

den specifies an optional character string to describe the name of an $m \times n$ matrix. The $[i, j]$ entry in the matrix returns the density of the distribution at the $[i, j]$ entry of the matrix specified by the *domain* argument. The IML keyword for *den* is DEN.

iguess specifies an optional numeric parameter that contains an initial guess for the variable *gscale* in the SEQSCALE subroutine or for the variable *mean* in the SEQSHIFT subroutine. In general, very good estimates for these initial guesses can be provided by an iterative process, and these estimates become extremely valuable near convergence. The IML keyword for *iguess* is IGUESS.

level specifies a numeric parameter in the SEQSCALE subroutine that contains the required significance level to be achieved through scaling the *domain* (see the description of SEQSCALE).

plevel specifies a numeric parameter in the SEQSHIFT subroutine that provides the required power level to be achieved through shifting the *domain* (see the description of SEQSHIFT).

SEQ Call

To compute the probability from a sequential test, you must specify a matrix containing the boundaries. With the optional additional information concerning the time intervals and the target accuracy, or their default values, the SEQ subroutine returns the matrix that contains the probability and optionally returns the density from a sequential test evaluated at each given point of the boundary. Let C_j denote the continuation set at each level j. C_j is defined to be the union at the j^{th} level of all the intervals bounded from below by the points with even indices $0, 2, 4, \ldots$ and from above by the points with odd indices $1, 3, \ldots$.

The SEQ call computes, with $\mu = 0$, the densities

$$f_j(s, \mu) = \int_{C_{j-1}} \phi(s - y, \mu, t_{j-1}) f_{j-1}(y, \mu)\, dy, \text{ for } j = 2, 3, \ldots$$

with

$$f_1(s, \mu) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{(s-\mu)^2}{2}\right]$$

and

$$\phi(s, \mu, t) = \frac{1}{\sqrt{2\pi t}} \exp\left[-\frac{(s-\mu)^2}{2t}\right]$$

with the associated probability at any point a at level j to be

$$P_j(a,\mu) = \int_{C_{j-1}} \Phi(a-y,\mu,t_j) f_{j-1}(y,\mu)\, dy, \text{ for } j = 2,3,\ldots$$

with

$$\Phi(b,\mu,t) = \int_{-\infty}^{b} \phi(s,\mu,t)\, ds$$

The notation τ denotes the vector of time intervals $t_1,\ldots,t_{n-1}$, and $P_j(g,\mu,\tau)$ denotes the probability of continuation at the jth level for a given domain g, a given mean μ, and a given time vector τ. The variance at the jth level can be calculated from τ.

$$\begin{aligned} \sigma_1^2 &= 1 \\ \sigma_{j+1}^2 &= \sigma_j^2 + \tau_j, \text{ for } j = 1,2,\ldots \end{aligned}$$

It is important to understand the limitations that are imposed internally on the domain by the numerical method. Any element g_{ij} will always be limited within a symmetric interval with standardized values not to exceed 8. That is,

$$g_{ij} = \max[\min(g_{ij}, 8\sigma_j), -8\sigma_j]$$

SEQSCALE Call

Given a domain g, an optional time vector τ, and a probability level p_s, the SEQSCALE subroutine finds the amount of scaling s that would solve the problem

$$P_n(gs, 0) = p_s$$

The result for the amount of scaling s is returned as the second argument of the SEQSCALE subroutine, *scale*. Note that because of the complexity of the problem, the SEQSCALE subroutine will not attempt to scale a domain with multiple intervals of continuation.

For a significance level of α, set $p_s = 1 - \alpha$.

SEQSHIFT Call

Given a geometry g, an optional time vector τ, and a power level $1-\beta$, the SEQSHIFT subroutine finds the mean μ that solves $\mu \geq 0$ such that $P_n(g,\mu) = \beta$.

Actually, a simple transformation of the variables in the sequential problem yields the following result:

$$P_j(g^{\mu}, 0) = P_j(g,\mu), \text{ for } j = 1,2,\ldots,n$$

where g^{μ} is given by $g_{ij}^{\mu} = g_{ij} - \mu j$.

Many options are available with the NLP family of optimization routines, which are described in Chapter 4, "Nonlinear Optimization Subroutines."

Consider the following continuation intervals:

$$
\begin{aligned}
C_1 &= \{-6, 2\} \\
C_2 &= \{-6, 3\} \\
C_3 &= \{-6, 4, 5, 6\} \\
C_4 &= \{-6, 4\}
\end{aligned}
$$

The following IML program computes the probability from the sequential test at each boundary point specified in the geometry.

```
proc iml;
   /* function to insert in m the geometry column a at level k*/
   start table(m,a,k);
      if ncol(m) = 0 & nrow(m) = 0  then    m = j(nrow(a),k,.);
      if nrow(m) < nrow(a) then  m = m// j(nrow(a)-nrow(m),ncol(m),.);
      if ncol(m) < k  then  m = m || j(nrow(m),k-ncol(m),.);
      m[1:nrow(a),k] = a;
   finish;

   call table(m,{-6,2},1);
   call table(m,{-6,3},2);
   call table(m,{-6,4,5,6},3);
   call table(m,{-6,4},4);
   call seq(prob,m) eps = 1.e-8 den="density";
   print m;
   print prob;
   print density;
```

The following output displays the values returned for m, *prob* and *den*, respectively.

The probability at the level $k = 3$ at the point $x = 6$ is *prob*$[4, 3] = 0.96651$, while the density at the same point is *density*$[4, 3] = 0.0000524$.

Consider the continuation intervals

$$
\begin{aligned}
C_1 &= \{-20, 2\} \\
C_2 &= \{-20, 20\} \\
C_3 &= \{-3, 3\}
\end{aligned}
$$

Note that the continuation at level 2 can be effectively considered infinite, and it does not numerically affect the results of the computation at level 3. The following IML program verifies this by using the *tscale* parameter to compute this problem.

```
proc iml;
   reset nocenter;
/* function to insert in m the geometry column a at level k*/
start table(m,a,k);
```

```
   if ncol(m) = 0 & nrow(m) = 0  then    m = j(nrow(a),k,.);
   if nrow(m) < nrow(a) then  m = m// j(nrow(a)-nrow(m),ncol(m),.);
   if ncol(m) < k  then  m = m || j(nrow(m),k-ncol(m),.);
   m[1:nrow(a),k] = a;
finish;

call table(m,{-20,2},1);
call table(m,{-20,20},2);
call table(m,{-3,3},3);

/************************************/
/* TSCALE has the default value of 1  */
/************************************/
call seq(prob1,m) eps = 1.e-8 den="density";
print m[format=f5.]  prob1[format=e12.5];

call table(mm,{-20,2},1);
call table(mm,{-3,3},2);
   /* We can show a 2-step separation between the levels */
   /* while dropping the intermediate level at 2        */
tscale = { 2 };
call seq(prob2,mm) eps = 1.e-8 den="density" TSCALE=tscale;
print mm[format=f5.]  prob2[format=e12.5];
```

The values returned for the variables m and *prob1* as well as mm and *prob2* are shown in the output.

Some internal limitations are imposed on the geometry. Consider the three-level case with geometry m in the preceding code. Since the *tscale* variable is not specified, it is set to its default value, $(1, 1)$. The variance at the j^{th} level is $\sigma_j^2 = j$ for $j = 1, 2, 3$. The first level has a lower boundary point of -20, as represented by the value of $m[1, 1]$. Since the absolute standardized value is larger than 8, this point is replaced internally by the value -8. Hence, the densities and the probabilities reported for the first level at this point are not for the given value -20; instead, they are for the internal value of -8. For practical purposes, this limitation is not severe since the absolute error introduced is of the order of 10^{-16}.

The computations performed by the first call of the SEQ subroutine can be simplified since the second level is large enough to be considered infinite. The matrix MM contains the first and third columns of the matrix M. However, in order to specify the two-step separation between the levels, you must specify *tscale*=2.

This example verifies some of the results published in Table 3 of Pocock (1982). That is, the following IML program verifies for the given domain that the significance level is 0.05 and that the power is $1 - \beta$ under the alternative hypothesis:

```
proc iml;
   /*****************************************/
   /* first check whether the numbers yield */
   /* 0.95 for the alpha level              */
   /*****************************************/
```

```
bm      ={-3.663  -2.884 -2.573 -2.375 -2.037,
          -2.988  -2.537 -2.407 -2.346 -2.156,
          -2.598  -2.390 -2.390 -2.390 -2.310,
          -2.446  -2.404 -2.404 -2.404 -2.396};

bplevel = { 0.5 0.25 0.1  0.05};
level   = 0.95; /* this the required alpha value  */
sigma   = diag(sqrt(1:5)); /* global sigma matrix */

do i = 1 to 4;
   m      = bm[i,];
   plevel = bplevel[i];
   geom   = (m//(-m))*sigma;

   /***************************/
   /* Try the null hypothesis */
   /***************************/

   call seq(prob,geom) eps = 1.e-10;
   palpha  = (prob[2,]-prob[1,])[5];

   /*********************************/
   /* Try the alternative hypothesis */
   /*********************************/

   call seqshift(prob,mean,geom,plevel);
   beta  = (prob[2,] -prob[1,])[5];
   p     = prob[3,]-prob[2,]+prob[1,];

   /*********************************/
   /* Number of patients per group   */
   /*********************************/

   tn    = 4*mean##2;
   maxn  = 5*tn;

   /************************************/
   /* compute the average sample number */
   /************************************/

   asn   = tn *( 5 - (4:0) * p');
   summary = summary // ( palpha  || level || beta ||
             plevel || tn  || maxn ||asn);
end;
print summary[format=10.5];
```

Note that the variables *eps* and *tscale* have been internally set to their default values. The following values are returned for the matrix SUMMARY:

These values compare well with the values shown in Table 3 of Pocock (1982). Differences are of the order of 10^{-5}.

This example shows how to verify the results in Table 1 of Wang and Tsiatis (1987). For a given δ, the following program finds Γ that yields a symmetric continuation

interval given by

$$-\Gamma j^{\delta} \leq C_j \leq \Gamma j^{\delta}$$

with a given significance level of α:

```
proc iml;
   start func(delta,k) global(level);
      m       = ((1:k))##delta;
      mm      = (-m//m);
      /*******************************/
      /* meet the significance level */
      /* by scaling                  */
      /*******************************/
      call seqscale(prob,scale,mm,level);
      return(scale);
   finish;

   /*********************************/
   /* alpha levels of 0.05 and 0.01 */
   /*********************************/

   blevel        = {0.95 0.99};
   do i = 1 to 2;
      level      = blevel[i];
      free summary;
      do delta   = 0 to .7 by .1;
         free row;
         do k=2 to 5;
            x    = func(delta,k);
            row  = row || x;
         end;
         summary = summary //row;
      end;
      print summary[format=10.5];
   end;
```

The value of SUMMARY for the 0.95 level is as follows.

The value for SUMMARY for the 0.99 level is as follows.

Note that since *eps* and *tscale* are not specified, they are internally set to their default values.

This example verifies the results in Table 2 of Pocock (1977). The following program finds Γ that yields a symmetric continuation interval given by

$$-\Gamma\sqrt{j} \leq C_j \leq \Gamma\sqrt{j}$$

for five groups. The overall significance level is α (the probability $palpha = 1 - \alpha$), and the power is $1 - \beta$.

```
proc iml;
   %let nl = 5;
   start func(plevel) global(level,scale,mean,palpha,beta,tn,asn);
      m       = sqrt((1: &nl));
      mm      =  -m //m;
      /*****************************/
      /* meet the significance level */
      /* by scaling                  */
      /*****************************/

      call seqscale(prob,scale,mm,level);
      palpha  = (prob[2,]-prob[1,])[&nl];
      mm      = mm *scale;

      /*****************************/
      /* meet the power condition    */
      /*****************************/

      call seqshift(prob,mean,mm,plevel);
      return(mean);
   finish;

   /****************/
   /* alpha = 0.95 */
   /****************/

   level   = 9.50000E-01;
   bplevel = { 0.5 .25 .1 0.05 0.01};
   free summary;
   do i = 1 to 5;
      summary = summary || func(bplevel[i]);
   end;
   print summary[format=10.5];
```

The value returned for SUMMARY are shown in the following table, and the entries agree with Table 2 of Pocock (1977).

This example illustrates how to find the optimal boundary of the δ-class of Wang and Tsiatis (1987). The δ-class boundary has the form

$$-\Gamma j^{\delta} \leq C_j \leq \Gamma j^{\delta}$$

The δ-class boundary is optimal if it minimizes the average sample number while satisfying the required significance level α and the required power $1 - \beta$. You can use the following program to verify some of the results published in Tables 2 and 3 of Wang and Tsiatis (1987):

```
proc iml;
   %let nl=5;
   start func(delta) global(level,plevel,mean,
                            scale,alpha,beta,tn,asn);
```

```
      m       = ((1: &nl))##delta;
      mm      = (-m//m);

      /*****************************/
      /* meet the significance level */
      /*****************************/

      call seqscale(prob,scale,mm,level);
      alpha   = (prob[2,]-prob[1,])[&nl];
      mm      = mm *scale;

      /*****************************/
      /* meet the power condition    */
      /*****************************/

      call seqshift(prob,mean,mm,plevel);
      beta    = (prob[2,]-prob[1,])[&nl];

      /***********************************/
      /* compute the average sample number */
      /***********************************/

      p       = prob[3,]-prob[2,]+prob[1];
      tn      = 4*mean##2; /* number per group */
      asn     = tn *( &nl - p *(%eval(&nl-1):0)‘);
      return(asn);
   finish;


/*********************************************/
/* set up the global variables needed by func */
/*********************************************/

level   = 0.95;
plevel  = 0.01;

/*****************************************/
/* set up the controlling options of the */
/* optimization routine                  */
/*****************************************/

opt     = {0 2 0 1 6};
tc      = repeat(.,1,12);
tc[1]   = 100;
tc[7]   = 1.e-4;
par     = { 1.e-13 . 1.e-10 . . .} || . || epsd;
```

```
/*****************************/
/* provide the initial guess */
/* and let nlpdd do the work */
/*****************************/

delta   = 0.5;
call nlpdd(rc,rx,"func",delta) opt=opt tc=tc par=par;
```

The following output displays the results.

```
                  Optimization Start
                 Parameter Estimates
                                          Gradient
                                         Objective
        N Parameter         Estimate        Function

        1 X1               -1.500000        -8.09752

           Value of Objective Function = 35.232023082



           Double Dogleg Optimization
  Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
           Without Parameter Scaling
         Gradient Computed by Finite Differences
       Number of Parameter Estimates 1

         Parameter Estimates                   2
         Functions (Observations)              2

                  Optimization Start

  Active Constraints              0  Criterion = 35.232
  Max Abs Gradient Element     8.098  Radius = 1.000


                 Function    Active      Objective
 Iter    Restart    Calls    Constraints    Function

    1       0          3         0          34.8914
    2*      0          4         0          34.8774
    3*      0          5         0          34.8774


 Iter    difcrit      maxgrad     lambda       slope

    1      0.3406         1.644    49.273     -0.830
    2*     0.0140        0.0440         0    -0.0144
    3*    0.00001       0.00013         0      -1E-5
```

```
                Optimization Results

Iterations                  3  Function Calls             6
Gradient Calls              5  Active Constraints         0
Criterion           34.877417  Max Grad Element  0.000126832
Slope          -0.0000100034  Radius                     1

   NOTE:   FCONV convergence criterion satisfied.

           Optimization Results
           Parameter Estimates

       N Parameter      Estimate      Gradient

       1 X1             0.586554     -0.0001268

      Value of Objective Function = 34.877416815
```

The optimal function value of 34.88 agrees with the entry in Table 2 of Wang and Tsiatis (1987) for five groups, $\alpha = 0.05$, and $1 - \beta = 0.99$. Note that the variables *eps* and *tscale* are internally set to their default values. For more information on the NLPDD subroutine, see the section "NLPDD Call" on page 635. For details on the *opt*, *tc*, and *par* arguments in the NLPDD call, see the "Options Vector" section on page 319, the "Termination Criteria" section on page 325, and the section "Control Parameters Vector" on page 332, respectively.

You can replicate other values in Table 2 of Wang and Tsiatis (1987) by changing the values of the variables NL and PLEVEL. You can obtain values from Table 3 by changing the value of the variable LEVEL to 0.99 and specifying NL and PLEVEL accordingly.

This example illustrates how to find the boundaries that minimize ASN given the required significance level and the required power. It replicates some of the results published in Table 3 of Pocock (1982). The IML program computes the domain that

- minimizes the ASN
- yields a given significance level of 0.05
- yields a given power $1 - \beta$ under the alternative hypothesis

The last two nonlinear conditions on the optimization process can be incorporated as a penalty applied on the error in these nonlinear conditions. The following IML program does the computations for a power of 0.9.

```
proc iml ;
   %let nl=5;
   start func(m) global(level,plevel,sigma,epss,
                  geometry,stgeom,gscale,mean,alpha,beta,tn,asn);
      m       =  abs(m);
```

```
   mm      = ( -m  // m)*sigma;
   /*****************************/
   /* meet the significance level */
   /*****************************/

   call seqscale(prob,gscale,mm,level) iguess=gscale eps=epss;
   stgeom  = gscale*m;
   geometry= mm*gscale;

   alpha   = (prob[2,]-prob[1,])[&nl];

   /*****************************/
   /* meet the power condition    */
   /*****************************/

   call seqshift(prob,mean,geometry,plevel) iguess=mean eps=epss;
   beta    = (prob[2,]-prob[1,])[&nl];
   p       = prob[3,] - prob[2,]+prob[1,];

   /***********************************/
   /* compute the average sample number */
   /***********************************/

   tn      = 4*mean##2; /* number per group */
   asn     = tn *( &nl - p *(%eval(&nl-1):0)‘);
   return(asn);
finish;

/*********************************************/
/* set up the global variables needed by func */
/*********************************************/
epss    = 1.e-8;
epso    = 1.e-5;
level   = 9.50000E-01;
plevel  = 0.05;
sigma   = diag(sqrt(1:5));

/*************************************/
/* set up the controlling options of the */
/* optimization routine                  */
/*************************************/

opt     = {0 2 0 1 6};
tc      = repeat(.,1,12);
tc[1]   = 100;
tc[7]   = 1.e-4;
par     = { 1.e-13 . 1.e-10 . . .} || . || epso;

/***********************************/
/* provide the constraint matrix    */
/* we need monotonically increasing */
/* significance levels              */
/***********************************/
```

```
con     = { .  .  .  .  .    .  . ,
                .  .  .  .  .    .  . ,
               1 -1  .  .  .    1  0 ,
               .  1 -1  .  .    1  0 ,
               .  .  1 -1  .    1  0 ,
               .  .  .  1 -1    1  0 };

   /*****************************/
   /* provide the initial guess */
   /* and let nlp do the work   */
   /*****************************/

   m       = { 1  1  1  1  1  };
   call nlpdd(rc,rx,"func",m) opt=opt blc = con tc=tc par=par;
   print stgeom;
```

Note that while *eps* has been set to *eps*=10^{-8}, *tscale* has been internally set to its default value. You may choose to run the IML program with and without the specification of the keyword IGUESS to see the effect on the execution time.

Note the following about the optimization process:

- Different levels of precision are imposed on different modules. In this example, *epss*, which is used as the precision for the sequential tests, is 1E−8. The absolute and relative function criteria for the objective function are set to *par*[7]=1E−5 and *tc*[7]=1E−4, respectively. Since finite differences are used to compute the first and second derivatives, the sequential test should be more precise than the optimization routine. Otherwise, the finite difference estimation is worthless. Optimally, if the precision of the function evaluation is $O(\epsilon)$, the first- and second-order derivatives should be estimated with perturbations $O(\epsilon^{\frac{1}{2}})$ and $O(\epsilon^{\frac{1}{3}})$, respectively. For example, if all three precision levels are set to 1E−5, the optimization process does not work properly.
- Line search techniques that do not depend on the computation of the derivative are preferable.
- The amount of printed information from the optimization routines is controlled by *opt*[2] and can be set to any value between 0 and 3, with larger numbers representing more printed output.

SEQSCALE Call

perform discrete sequential tests

CALL SEQSCALE(*prob, gscale, domain, level*<, <**IGUESS=***iguess*>
<, <**TSCALE=***tscale*>><, <**EPS=***eps*>><, <**DEN=***den*>>>>>**);**

See the entry for the SEQ subroutine for details.

SEQSHIFT Call

perform discrete sequential tests

CALL SEQSHIFT(*prob, shift, domain, plevel*<, <**IGUESS=***iguess*>
<, <**TSCALE=***tscale*><, <**EPS=***eps*><, <**DEN=***den*>>>>>**);**

See the entry for the SEQ subroutine for details.

SETDIF Function

compares elements of two matrices

SETDIF(*matrix1, matrix2***)**

The inputs to the SETDIF function are as follows:

matrix1 is a reference matrix. Elements of *matrix1* not found in *matrix2* are returned in a vector. It can be either numeric or character.

matrix2 is the comparison matrix. Elements of *matrix1* not found in *matrix2* are returned in a vector. It can be either numeric or character, depending on the type of *matrix1*.

The SETDIF function returns as a row vector the sorted set (without duplicates) of all element values present in *matrix1* but not in *matrix2*. If the resulting set is empty, the SETDIF function returns a null matrix (with zero rows and zero columns). The argument matrices and result can be either both character or both numeric. For character matrices, the element length of the result is the same as the element length of the *matrix1*. Shorter elements in the second argument are padded on the right with blanks for comparison purposes.

For example, the statements

```
a={1 2 4 5};
b={3 4};
c=setdif(a,b);
```

produce the result

```
C            1 row       3 cols    (numeric)

               1         2         5
```

SETIN Statement

makes a data set current for input

SETIN *SAS-data-set* <**NOBS** *name*> <**POINT** *operand*>**;**

The inputs to the SETIN statement are as follows:

SAS-data-set can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on data sets in *SAS Language Reference: Concepts*.

name is the name of a variable to contain the number of observations in the data set.

operand specifies the current observation.

The SETIN statement chooses the specified data set from among the data sets already opened for input by the EDIT or USE statement. This data set becomes the current input data set for subsequent data management statements. The NOBS option is not required. If specified, the NOBS option returns the number of observations in the data set in the scalar variable *name*. The POINT option makes the specified observation the current one. It positions the data set to a particular observation. The SHOW *datasets* command lists data sets already opened for input.

In the example that follows, if the data set WORK.A has 20 observations, the variable SIZE is set to 20. Also, the current observation is set to 10.

```
setin work.a nobs size point 10;
list;                /* lists observation 10 */
```

SETOUT Statement

makes a data set current for output

SETOUT *SAS-data-set* <**NOBS** *name*> <**POINT** *operand*>**;**

The inputs to the SETOUT statement are as follows:

SAS-data-set can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on SAS data sets in *SAS Language Reference: Concepts*.

name is the name of a variable to contain the number of observations in the data set.

operand specifies the observation to be made the current observation.

The SETOUT statement chooses the specified data set from among those data sets already opened for output by the EDIT or CREATE statement. This data set becomes the current output data set for subsequent data management statements. If specified, the NOBS option returns the number of observations currently in the data set in the scalar variable *name*. The POINT option makes the specified observation the current one.

In the example that follows, the data set WORK.A is made the current output data set and the fifth observation is made the current observation. The number of observations in WORK.A is returned in the variable SIZE.

```
setout work.a nobs size point 5;
```

SHAPE Function

reshapes and repeats values

SHAPE(*matrix*<*, nrow*<*, ncol*<*, pad-value*>>>**)**

The inputs to the SHAPE function are as follows:

matrix	is a numeric or character matrix or literal.
nrow	gives the number of rows of the new matrix.
ncol	gives the number of columns of the new matrix.
pad-value	is a fill value.

The SHAPE function shapes a new matrix from a matrix with different dimensions; *nrow* specifies the number of rows, and *ncol* specifies the number of columns in the new matrix. The operator works for both numeric and character operands. The three ways of using the function are outlined below:

- If only *nrow* is specified, the number of columns is determined as the number of elements in the object matrix divided by *nrow*. The number of elements must be exactly divisible; otherwise, a conformability error is diagnosed.
- If both *nrow* and *ncol* are specified, but not *pad-value*, the result is obtained moving along the rows until the desired number of elements is obtained. The operation cycles back to the beginning of the object matrix to get more elements, if needed.
- If *pad-value* is specified, the operation moves the elements of the object matrix first and then fills in any extra positions in the result with the *pad-value*.

If *nrow* or *ncol* is specified as 0, the number of rows or columns, respectively, becomes the number of values divided by *ncol* or *nrow*.

For example, the statement

```
r=shape(12,3,4);
```

produces the result shown:

```
R             3 rows      4 cols    (numeric)

           12          12          12          12
           12          12          12          12
           12          12          12          12
```

The next statement

```
r=shape(77,1,5);
```

produces the result matrix by moving along the rows until the desired number of elements is obtained, cycling back as necessary:

```
R          1 row       5 cols    (numeric)

77          77          77          77          77
```

The statement below

```
r=shape({1 2, 3 4, 5 6},2);
```

has *nrow* specified and converts the 3×2 matrix into a 2×3 matrix.

```
R             2 rows      3 cols    (numeric)

              1           2           3
              4           5           6
```

The statement

```
r=shape({99 31},3,3);
```

demonstrates the cycling back and repetition of elements in row-major order until the number of elements desired is obtained.

```
R             3 rows      3 cols    (numeric)

             99          31          99
             31          99          31
             99          31          99
```

SHOW Statement

prints system information

SHOW *operands*;

where *operands* are any of the valid operands to the SHOW statement. These are given below.

The SHOW statement prints system information. The following *operands* are available:

ALL	shows all the information included by OPTIONS, SPACE, DATASETS, FILES, and MODULES.
ALLNAMES	behaves like NAMES, but also shows names without values.
CONTENTS	shows the names and attributes of the variables in the current SAS data set.
DATASETS	shows all open SAS data sets.
FILES	shows all open files.
MEMORY	returns the size of the largest chunk of main memory available.
MODULES	shows all modules that exist in the current IML environment. A module already referenced but not yet defined is listed as undefined.
name	shows attributes of the specified matrix. If the name of a matrix is one of the SHOW keywords, then both the information for the keyword and the matrix are shown.
NAMES	shows attributes of all matrices having values. Attributes include number of rows, number of columns, data type, and size.
OPTIONS	shows current settings of all IML options (see the RESET statement).
PAUSE	shows the status of all paused modules that are pending resume.
SPACE	shows the workspace and symbolspace size and their current usage.
STORAGE	shows the modules and matrices in the current IML library storage.
WINDOWS	shows all active windows opened by WINDOW statements.

An example of a valid statement follows:

```
show all;
```

SOLVE Function

solves a system of linear equations

SOLVE(A, B**)**

The inputs to the SOLVE function are as follows:

A is an $n \times n$ nonsingular matrix.

B is an $n \times p$ matrix.

The SOLVE function solves the set of linear equations $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$. $\mathbf{A}$ must be square and nonsingular.

$\mathbf{X} = \text{SOLVE}(\mathbf{A}, \mathbf{B})$ is equivalent to using the INV function as $\mathbf{X} = \text{INV}(\mathbf{A}) * \mathbf{B}$. However, the SOLVE function is recommended over the INV function because it is more efficient and more accurate. An example follows:

```
x=solve(a,b);
```

The solution method used is discussed in Forsythe, Malcolm, and Moler (1967).

The SOLVE function (as well as the DET and INV functions) uses the following criterion to decide whether the input matrix, $\mathbf{A} = [a_{ij}]_{i,j=1,\ldots,n}$, is singular:

$$\textit{sing} = 100 \times \textit{MACHEPS} \times \max_{1 \le i,j \le n} |a_{ij}|$$

where *MACHEPS* is the relative machine precision.

All matrix elements less than or equal to *sing* are now considered rounding errors of the largest matrix elements, so they are taken to be zero. For example, if a diagonal or triangular coefficient matrix has a diagonal value less than or equal to *sing*, the matrix is considered singular by the DET, INV, and SOLVE functions.

Previously, a much smaller singularity criterion was used, which caused algebraic operations to be performed on values that were essentially floating point error. This occasionally yielded numerically unstable results. The new criterion is much more conservative, and it generates far fewer erroneous results. In some cases, you may need to scale the data to avoid singular matrices. If you think the new criterion is too strong,

- try the GINV function to compute the generalized inverse
- examine the size of the singular values returned by the SVD function. The SVD function can be used to compute a generalized inverse with a user-specified singularity criterion.

SORT Statement

sorts a SAS data set

SORT <**DATA=**>*SAS-data-set* <**OUT=***SAS-data-set*>
BY <**DESCENDING**> *variables*;

where you can use the following clauses with the SORT statement:

DATA=*SAS-data-set*	names the SAS data set to be sorted. It can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on SAS data sets in *SAS Language Reference: Concepts*. Note that the DATA= portion of the specification is optional.
OUT=*SAS-data-set*	specifies a name for the output data set. If this clause is omitted, the DATA= data set is sorted and the sorted version replaces the original data set.
BY *variables*	specifies the variables to be sorted. A BY clause *must* be used with the SORT statement.
DESCENDING	specifies the variables are to be sorted in descending order.

The SORT statement sorts the observations in a SAS data set by one or more variables, stores the resulting sorted observations in a new SAS data set, or replaces the original. As opposed to all other IML data processing statements, it is *mandatory* that the data set to be sorted be closed prior to the execution of the SORT statement.

The SORT statement first arranges the observations in the order of the first variable in the BY clause; then it sorts the observations with a given value of the first variable by the second variable, and so forth. Every variable in the BY clause can be preceded by the keyword DESCENDING to denote that the variable that follows is to be sorted in descending order. Note that the SORT statement in IML always retains the same relative positions of the observations with identical BY variable values.

For example, the IML statement

```
sort class out=sclass by descending age height;
```

sorts the SAS data set CLASS by the variables AGE and HEIGHT, where AGE is sorted in descending order, and all observations with the same AGE value are sorted by HEIGHT in ascending order. The output data set SCLASS contains the sorted observations. When a data set is sorted in place (without the OUT= clause) any indexes associated with the data set become invalid and are automatically deleted.

Note that all the clauses of the SORT statement must be specified in the order given above.

SOUND Call

produces a tone

CALL SOUND(*freq*<, *dur*>**);**

The inputs to the SOUND subroutine are as follows:

freq is a numeric matrix or literal giving the frequency in hertz.

dur is a numeric matrix or literal giving the duration in seconds. Note that the *dur* argument differs from that in the DATA step.

The SOUND subroutine generates a tone using *freq* for frequency (in hertz) and *dur* for duration (in seconds). Matrices may be specified for frequency and duration to produce multiple tones, but if both arguments are nonscalar, then the number of elements must match. The duration argument is optional and defaults to 0.25 (one quarter second).

For example, the following statements produce tones from an ascending musical scale, all with a duration of 0.2 seconds:

```
notes=400#(2##do(0, 1, 1/12));
call sound(notes,0.2);
```

SPLINE and SPLINEC Calls

provide cubic spline fits

CALL SPLINE(*fitted, data*<, *smooth*><, *delta*><, *nout*>
<, *type*><, *slope*>**);**

CALL SPLINEC(*fitted, coeff, endval, data*<, *smooth*><, *delta*>
<, *nout*><, *type*><, *slope*>**);**

The SPLINEC subroutine returns the following values:

fitted is an $n \times 2$ matrix of fitted values.

coeff is an $n \times 5$ (or $n \times 9$) matrix of spline coefficients. The matrix contains the cubic polynomial coefficients for the spline for each interval. Column 1 is the left endpoint of the x-interval for the regular (nonparametric) spline or the left endpoint of the parameter for the parametric spline. Columns $2 - 5$ are the constant, linear, quadratic, and cubic coefficients, respectively, for the x-component. If a parametric spline is used, then columns $6 - 9$ are the constant, linear, quadratic, and cubic coefficients, respectively, for the y-component. The coefficients for

each interval are with respect to the variable $x - x_i$ where x_i is the left endpoint of the interval and x is the point of interest. The matrix *coeff* can be processed to yield the integral or the derivative of the spline. This, in turn, can be used with the SPLINEV function to evaluate the resulting curves. The SPLINEC call returns *coeff*.

endval is a 1×2 matrix of endpoint values. Slopes of the two ends of the curve are reported as angles expressed in degrees. The SPLINEC call returns the *endval* argument.

The inputs to the SPLINEC subroutine are as follows:

data specifies a $n \times 2$ (or $n \times 3$) matrix of (x, y) points on which the spline is to be fit. The optional third column is used to specify a weight for each data point. If *smooth* > 0, the weight column is used in calculations. A weight ≤ 0 causes the data point to be ignored in calculations.

delta is an optional scalar specifying the resolution constant. If *delta* is specified, the fitted points are spaced by the amount *delta* on the scale of the first column of *data* if a regular spline is used or on the scale of the curve length if a parametric spline is used. If both *nout* and *delta* are specified, *nout* is used and *delta* is ignored.

nout is an optional scalar specifying the number of fitted points to be computed. The default is *nout*=200. If *nout* is specified, then *nout* equally spaced points are returned. The *nout* argument overrides the *delta* argument.

slope is an optional 1×2 matrix of endpoint slopes given as angles in degrees. If a parametric spline is used, the angle values are used modulo 360. If a nonparametric spline is used, the tangent of the angles is used to set the slopes (that is, the effective angles range from -90 to 90 degrees).

smooth is an optional scalar specifying the degree of smoothing to be used. If *smooth* is omitted or set equal to 0, then a cubic interpolating spline is fit to the data. If *smooth* > 0, then a cubic spline is used. Larger values of *smooth* generate more smoothing.

type is an optional 1×1 (or 1×2) character matrix or quoted literal giving the type of spline to be used. The first element of *type* should be one of the following:

- `periodic`, which requests periodic endpoints
- `zero`, which sets second derivatives at endpoints to 0

The *type* argument controls the endpoint constraints unless the *slope* argument is specified. If `periodic` is specified, the response values at the beginning and end of column 2 of *data* must be the same unless the smoothing spline is being used. If the values are not the same, an error message is printed and no spline is fit. The default value is `zero`. The second element of *type* should be one of the following.

- `nonparametric`, which requests a nonparametric spline
- `parametric`, which requests a parametric spline

If `parametric` is specified, a parameter sequence $\{t_i\}$ is formed as follows: $t_1 = 0$ and

$$t_i = t_{i-1} + \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Splines are then fit to both the first and second columns of *data*. The resulting splined values are paired to form the output. Changing the relative scaling of the first two columns of *data* changes the output because the sequence $\{t_i\}$ assumes Euclidean distance.

Note that if the points are not arranged in ascending order by the first columns of *data*, then a parametric method must be used. An error message results if the nonparametric spline is requested.

Refer to Stoer and Bulirsch (1980), Reinsch (1967), and Pizer (1975) for descriptions of the methods used to fit the spline.

```
proc iml;
   data = { 0 1, 1 2, 2 3, 3 4, 4 5, 5 6, 6 7, 7 8, 8 9, 9 10 };
   call splinec(fitted,coeff,endval,data,,,,'zero',{45 45});
   v =  splinev(coeff,{-1 1 2 3 3.5 4 20});
   print v;
   v =  splinev(coeff,,3);
   print v;
```

This example takes a function defined by discrete data and finds the integral and the first moment of the function.

```
data func;
   input x @@;
   y = x+0.1*sin(x);
   datalines;
0   2   5   7   8  10
;

proc iml;
   use func;
   read all into a;
   call splinec(fit,coeff,endval,a,,,,'zero');

   start fcheck(x) global(coeff,pow);
      /***********************************/
      /* Note that the first column of v  */
      /* contains the points of           */
      /* evaluation and the second column */
      /* contains the evaluation.         */
      /***********************************/
      v = x##pow # splinev(coeff,x //x)[1,2];
      return(v);
   finish;
```

```
start moment(po) global(coeff,pow);
   pow = po;
   call quad(z,'fcheck',coeff[,1]) eps = 1.e-10;
   v1  = sum(z);
   return(v1);
finish;

mass  = moment(0);   /* to compute the mass */
mass  = mass //
       (moment(1)/mass) // /* to compute the mean */
       (moment(2)/mass) ;  /* to compute the meansquare */
print mass;

/*****************************************************/
/* Use Gauss-Legendre integration: this is not     */
/* adaptive, but it is good for moments up to maxng. */
/*****************************************************/

gauss = {
    -9.3246951420315205e-01
    -6.6120938646626448e-01
    -2.3861918608319743e-01
     2.3861918608319713e-01
     6.6120938646626459e-01
     9.3246951420315183e-01,
     1.713244923791701e-01
     3.607615730481388e-01
     4.679139345726905e-01
     4.679139345726904e-01
     3.607615730481389e-01
     1.713244923791707e-01  };
ngauss = ncol(gauss);
maxng  =  2*ngauss-4;
start moment1(pow) global(coeff,gauss,ngauss,maxng);
   if pow < maxng  then do;
      nrow    = nrow(coeff);
      ncol    = ncol(coeff);
      left    = coeff[1:nrow-1,1];
      right   = coeff[2:nrow,1];
      mid     = 0.5*(left+right);
      interv  = 0.5*(right - left);
    /* scale the weights on each interval */
      wgts    = shape(interv*gauss[2,],1);
    /* scale the points on each interval */
      pts     = shape(interv*gauss[1,] + mid * repeat(1,1,ngauss),1) ;
    /* evaluate the function */
      eval    = splinev(coeff,pts)[,2]';
      mat     = sum (wgts#pts##pow#eval);
   end;
   return(mat);
finish;

mass    = moment1(0);       /* to compute the mass */
mass    = mass // (moment1(1)/mass) //  (moment1(2)/mass) ;
print mass;
```

SPLINEV Function

provides cubic spline evaluations

SPLINEV(*coeff*<*, delta*<*, nout*>>**)**

The SPLINEV function returns a two-column matrix containing the points of evaluation in the first column and the corresponding fitted values of the spline in the second column.

The inputs to the SPLINEV function are as follows:

coeff is an $n \times 5$ (or $n \times 9$) matrix of spline coefficients, as returned by the SPLINEC Call. The *coeff* argument should not contain missing values.

delta is an optional vector specifying evaluation points. If *delta* is a scalar, the spline will be evaluated at equally spaced points *delta* apart. If *delta* is a vector arranged in ascending order, the spline will be evaluated at each of these values. Evaluation at a point outside the support of the spline results in a missing value in the output. If you specify the *delta* argument, you cannot specify the *nout* argument.

nout is an optional scalar specifying the number of fitted points desired. The default is *nout*=200. If you specify the *nout* argument, you cannot specify the *delta* argument.

See the section "SPLINE and SPLINEC Calls" on page 756 for details and examples.

SPOT Function

calculates a column vector of spot rates given vectors of forward rates and times

SPOT($times, forward_$rates)

The SPOT function returns an n x 1 vector of spot rates.

times is an n x 1 column vector of times in consistent units. Elements should be non-negative.

forward_$rates$ is an n x 1 column vector of corresponding per-period forward rates. Elements should be positive.

The SPOT function transforms the given spot rates as

$$s_1 = f_1$$

$$s_i = \Pi_{j=1}^{j=i}(1 + f_j)^{t_j - t_{j-1}} - 1.0; \quad i = 2, ...n$$

Example

```
proc iml;
   fwd={.05};
   times={1}; \\
   spot=spot(times,fwd);
print spot;
quit;
```

```
SPOT
 0.05
```

SQRSYM Function

converts a symmetric matrix to a square matrix

SQRSYM(*matrix***)**

where *matrix* is a symmetric numeric matrix.

The SQRSYM function takes a matrix such as those generated by the SYMSQR function and transforms it back into a square matrix. The elements of the argument are unpacked into the lower triangle of the result and reflected across the diagonal into the upper triangle.

For example, the following statement

```
sqr=sqrsym(symsqr({1 2, 3 4}));
```

which is the same as

```
sqr=sqrsym({ 1, 3, 4} );
```

produces the result

```
              SQR          2 rows      2 cols    (numeric)

                                 1           3
                                 3           4
```

SQRT Function

calculates the square root **SQRT(***matrix***)**

where *matrix* is a numeric matrix or literal.

The SQRT function is the scalar function returning the positive square roots of each element of the argument. An example of a valid statement follows.

```
a=sqrt(c);
```

SSQ Function

calculates the sum of squares of all elements

SSQ(*matrix1*<, *matrix2*,..., *matrix15*>**)**

where *matrix* is a numeric matrix or literal.

The SSQ function returns as a single numeric value the (uncorrected) sum of squares for all the elements of all arguments. You can specify as many as 15 numeric argument matrices.

The SSQ function checks for missing arguments and does not include them in the accumulation. If all arguments are missing, the result is 0.

An example of a valid statement follows:

```
a={1 2 3, 4 5 6};
x=ssq(a);
```

START and FINISH Statements

define a module

START <*name*> <**(***arguments***)**> <**GLOBAL(***arguments***)**>**;**
*module statements***;**

FINISH <*name*>**;**

The inputs to the START and FINISH statements are as follows:

name	is the name of a user-defined module.
arguments	are names of variable arguments to the module. Arguments can be either input variables or output (returned) variables. Arguments listed in the GLOBAL clause are treated as global variables. Otherwise, the arguments are local.
module statements	are statements making up the body of the module.

The START statement instructs IML to enter a module-collect mode to collect the statements of a module rather than execute them immediately. The FINISH statement signals the end of a module. Optionally, the FINISH statement can take the module name as its argument. When no *name* argument is given in the START statement, the module name MAIN is used by default. If an error occurs during module compilation, the module is not defined. See Chapter 5, "Programming Statements," for details.

The example below defines a module named MYMOD that has two local variables (A and B) and two global variables (X and Y). The module creates the variable Y from the arguments A, B, and X.

```
start mymod(a,b) global(x,y);
   y=a*x+b;
finish;
```

STOP Statement

stops execution of statements

STOP;

The STOP statement stops the IML program, and no further matrix statements are executed. However, IML continues to execute if more statements are entered. See also the descriptions of the RETURN and ABORT statements.

If IML execution was interrupted by a PAUSE statement or by a break, the STOP statement clears all the paused states and returns to immediate mode.

IML supports STOP processing of both regular and function modules.

STORAGE Function

lists names of matrices and modules in storage

STORAGE();

The STORAGE function returns a matrix of the names of all of the matrices and modules in the current storage library. The result is a character vector with each matrix or module name occupying a row. Matrices are listed before modules. The SHOW *storage* command separately lists all of the modules and matrices in storage.

For example, the following statements reset the current library storage to MYLIB and then print a list of the modules and matrices in storage:

```
reset storage="MYLIB";
```

Then issue the command below to get the resulting matrix:

```
a=storage();
print a;
```

STORE Statement

stores matrices and modules in library storage

STORE <**MODULE=(***module-list***)**> <*matrix-list*>**;**

The inputs to the STORE statement are as follows.

module-list is a list of module names.

matrix-list is a list of matrix names.

The STORE statement stores matrices or modules in the storage library. For example, the following statement stores the modules A, B, and C and the matrix X:

```
store module=(A B C) X;
```

The special operand _ALL_ can be used to store all matrices or all modules. For example, the following statement stores all matrices and modules:

```
store _all_ module=_all_;
```

The storage library can be specified using the RESET *storage* command and defaults to SASUSER.IMLSTOR. The SHOW *storage* command lists the current contents of the storage library. The following statement stores all matrices:

```
store;
```

See Chapter 14, "Storage Features," and also the descriptions of the LOAD, REMOVE, RESET, and SHOW statements for related information.

SUBSTR Function

takes substrings of matrix elements

SUBSTR(*matrix, position*<*, length*>**)**

The inputs to the SUBSTR function are as follows:

matrix is a character matrix or quoted literal.

position is a numeric matrix or scalar giving the starting position.

length is a numeric matrix or scalar giving the length of the substring.

The SUBSTR function takes a character matrix as an argument along with starting positions and lengths and produces a character matrix with the same dimensions as the argument. Elements of the result matrix are substrings of the corresponding argument elements. Each substring is constructed using the starting *position* supplied. If a *length* is supplied, this length is the length of the substring. If no *length* is supplied, the remainder of the argument string is the substring.

The *position* and *length* arguments can be scalars or numeric matrices. If *position* or *length* is a matrix, its dimensions must be the same as the dimensions of the argument matrix or submatrix. If either one is a matrix, its values are applied to the substringing of the corresponding elements of the *matrix*. If *length* is supplied, the element length of the result is MAX(*length*); otherwise, the element length of the result is

$$\mathrm{NLENG}(\mathit{matrix}) - \mathrm{MIN}(\mathit{position}) + 1 \ .$$

The statements

```
B={abc def ghi, jkl mno pqr};
a=substr(b,3,2);
```

return the matrix

```
          A        2 rows      3 cols    (character, size 2)

                             C  F  I
                             L  O  R
```

The element size of the result is 2; the elements are padded with blanks.

SUM Function

sums all elements

SUM(*matrix1*<, *matrix2*,. . ., *matrix15*>**)**

where *matrix* is a numeric matrix or literal.

The SUM function returns as a single numeric value the sum of all the elements in all arguments. There can be as many as 15 argument matrices. The SUM function checks for missing values and does not include them in the accumulation. It returns 0 if all values are missing.

For example, the statements

```
a={2 1, 0 -1};
b=sum(a);
```

return the scalar

```
              B             1 row       1 col     (numeric)

                                        2
```

SUMMARY Statement

computes summary statistics for SAS data sets

SUMMARY <**CLASS** *operand*> <**VAR** *operand*> <**WEIGHT** *operand*> <**STAT** *operand*> <**OPT** *operand*> <**WHERE(***expression***)**>**;**

where the *operands* used by most clauses take either a matrix name, a matrix literal, or an expression yielding a matrix name or value. A discussion of the clauses and *operands* follows.

The SUMMARY statement computes statistics for numeric variables for an entire data set or a subset of observations in the data set. The statistics can be stratified by the use of class variables. The computed statistics are displayed in tabular form and optionally can be saved in matrices. Like most other IML data processing statements, the SUMMARY statement works on the current data set.

The following options are available with the SUMMARY statement:

CLASS *operand*
specifies the variables in the current input SAS data set to be used to group the summaries. The *operand* is a character matrix containing the names of the variables, for example,

```
summary class { age sex} ;
```

Both numeric and character variables can be used as class variables.

VAR *operand*
calculates statistics for a set of numeric variables from the current input data set. The *operand* is a character matrix containing the names of the variables. Also, the special keyword _NUM_ can be used as a VAR operand to specify all numeric variables. If the VAR clause is missing, the SUMMARY statement produces only the number of observations in each class group.

WEIGHT *operand*
specifies a character value containing the name of a numeric variable in the current data set whose values are to be used to weight each observation. Only one variable can be specified.

STAT *operand*
computes the statistics specified. The *operand* is a character matrix containing the names of statistics. For example, to get the mean and standard deviation, specify

```
summary stat{mean std};
```

Below is a list of the keywords that can be specified as the STAT *operand*:

CSS	computes the corrected sum of squares.
MAX	computes the maximum value.
MEAN	computes the mean.
MIN	computes the minimum value.
N	computes the number of observations in the subgroup used in the computation of the various statistics for the corresponding analysis variable.
NMISS	computes the number of observations in the subgroup having missing values for the analysis variable.
STD	computes the standard deviation.

SUM	computes the sum.
SUMWGT	computes the sum of the WEIGHT variable values if WEIGHT is specified; otherwise, IML computes the number of observations used in the computation of statistics.
USS	computes the uncorrected sum of squares.
VAR	computes the variance.

When the STAT clause is omitted, the SUMMARY statement computes these statistics for each variable in the VAR clause:

- MAX
- MEAN
- MIN
- STD.

Note that NOBS, the number of observations in each CLASS group, is always given.

OPT *operand*
sets the PRINT or NOPRINT and SAVE or NOSAVE options. The NOPRINT option suppresses the printing of the results from the SUMMARY statement. The SAVE option requests that the SUMMARY statement save the resultant statistics in matrices. The *operand* is a character matrix containing one or more of the options.

When the SAVE option is set, the SUMMARY statement creates a class vector for each class variable, a statistic matrix for each analysis variable, and a column vector named _NOBS_. The class vectors are named by the corresponding class variable and have an equal number of rows. There are as many rows as there are subgroups defined by the interaction of all class variables. The statistic matrices are named by the corresponding analysis variable. Each column of the statistic matrix corresponds to a statistic requested, and each row corresponds to the statistics of the subgroup defined by the class variables. If no class variable has been specified, each statistic matrix has one row, containing the statistics of the entire population. The _NOBS_ vector contains the number of observations for each subgroup.

The default is PRINT NOSAVE.

WHERE *expression*
conditionally selects observations, within the *range* specification, according to conditions given in *expression*. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable	is a variable in the SAS data set.

comparison-op is one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause, using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

See Chapter 6, "Working with SAS Data Sets," for an example using the SUMMARY statement.

SVD Call

computes the singular value decomposition

CALL SVD(u, q, v, a**);**

In the SVD subroutine:

a	is the input matrix that is decomposed as described below.
u, q, and v	are the returned decomposition matrices.

The SVD subroutine decomposes a real $m \times n$ matrix $\mathbf{A}$ (where m is greater than or equal to n) into the form

$$\mathbf{A} = \mathbf{U}\text{diag}(\mathbf{Q})\mathbf{V}'$$

where

$$\mathbf{U}'\mathbf{U} = \mathbf{V}'\mathbf{V} = \mathbf{V}\mathbf{V}' = \mathbf{I}_n$$

and $\mathbf{Q}$ contains the singular values of $\mathbf{A}$. $\mathbf{U}$ is $m \times n$, $\mathbf{Q}$ is $n \times 1$, and $\mathbf{V}$ is $n \times n$.

When m is greater than or equal to n, $\mathbf{U}$ consists of the orthonormal eigenvectors of $\mathbf{A}\mathbf{A}'$, and $\mathbf{V}$ consists of the orthonormal eigenvectors of $\mathbf{A}'\mathbf{A}$. $\mathbf{Q}$ contains the square roots of the eigenvalues of $\mathbf{A}'\mathbf{A}$ and $\mathbf{A}\mathbf{A}'$, except for some zeros.

If m is less than n, a corresponding decomposition is done where $\mathbf{U}$ and $\mathbf{V}$ switch roles:

$$\mathbf{A} = \mathbf{U}\text{diag}(\mathbf{Q})\mathbf{V}'$$

but

$$\mathbf{U}'\mathbf{U} = \mathbf{U}\mathbf{U}' = \mathbf{V}'\mathbf{V} = \mathbf{I}_w \ .$$

The singular values are sorted in descending order.

For information about the method used in the SVD subroutine, refer to Wilkinson and Reinsch (1971). Consider the following example (Wilkinson and Reinsch 1971, p. 149):

```
a={22  10   2   3   7,
   14   7  10   0   8,
   -1  13  -1 -11   3,
   -3  -2  13  -2   4,
    9   8   1  -2   4,
    9   1  -7   5  -1,
    2  -6   6   5   1,
    4   5   0  -2   2};
call svd(u,q,v,a);
```

The results are

```
  U             8 rows      5 cols    (numeric)

 0.7071068 0.1581139 -0.176777  -0.06701  0.279804
 0.5303301 0.1581139 0.3535534 -0.045208 -0.645372
 0.1767767 -0.790569 0.1767767 0.5368704 -0.060458
         0 0.1581139 0.7071068 0.1086593  0.592536
 0.3535534 -0.158114         0 -0.228736 0.2300372
 0.1767767 0.1581139  -0.53033 0.5116134  0.212316
```

```
         0 0.4743416 0.1767767 0.5867386 -0.102189
 0.1767767 -0.158114         0 -0.187346 0.2049688


  Q             5 rows      1 col     (numeric)

                     35.327043
                            20
                     19.595918
                     1.281E-15
                     3.661E-16


  V             5 rows      5 cols    (numeric)

 0.8006408 0.3162278 -0.288675 0.4190955         0
 0.4803845 -0.632456         0 -0.440509 -0.418548
 0.1601282 0.3162278 0.8660254 0.0520045  -0.34879
         0 0.6324555 -0.288675 -0.676059 -0.244153
 0.3202563         0 0.2886751 -0.412977 0.8022171
```

SWEEP Function

sweeps a matrix

SWEEP(*matrix, index-vector***)**

The inputs to the SWEEP function are as follows:

matrix	is a numeric matrix or literal.
index-vector	is a numeric vector indicating the pivots.

The SWEEP function sweeps *matrix* on the pivots indicated in *index-vector* to produce a new matrix. The values of the index vector must be less than or equal to the number of rows or the number of columns in *matrix*, whichever is smaller.

For example, suppose that $\mathbf{A}$ is partitioned into

$$\begin{bmatrix} \mathbf{R} & \mathbf{S} \\ \mathbf{T} & \mathbf{U} \end{bmatrix}$$

such that $\mathbf{R}$ is $q \times q$ and $\mathbf{U}$ is $(m-q) \times (n-q)$. Let

```
I = [1 2 3 . . . q]
```

Then, the statement

```
s=sweep(A,I);
```

becomes

$$\begin{bmatrix} \mathbf{R}^{-1} & \mathbf{R}^{-1} \\ -\mathbf{T}\mathbf{R}^{-1} & \mathbf{U}-\mathbf{T}\mathbf{R}^{-1} \end{bmatrix} .$$

The index vector can be omitted. In this case, the function sweeps the matrix on all pivots on the main diagonal 1:MIN(*nrow*,*ncol*).

The SWEEP function has sequential and reversibility properties when the submatrix swept is positive definite:

- SWEEP(SWEEP($\mathbf{A}$,1),2)=SWEEP($\mathbf{A}$,{ 1 2 })
- SWEEP(SWEEP($\mathbf{A}$,$\mathbf{I}$),$\mathbf{I}$)=$\mathbf{A}$

See Beaton (1964) for more information about these properties.

To use the SWEEP function for regression, suppose the matrix $\mathbf{A}$ contains

$$\begin{bmatrix} \mathbf{X}'\mathbf{X} & \mathbf{X}'\mathbf{Y} \\ \mathbf{Y}'\mathbf{X} & \mathbf{Y}'\mathbf{Y} \end{bmatrix}$$

where $\mathbf{X}'\mathbf{X}$ is $k \times k$.

Then $\mathbf{B} = \text{SWEEP}(\mathbf{A}, 1 \ldots k)$ contains

$$\begin{bmatrix} (\mathbf{X}'\mathbf{X})^{-1} & (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}' \\ -\mathbf{Y}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} & \mathbf{Y}'(\mathbf{I}-\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}')\mathbf{Y} \end{bmatrix}$$

The partitions of $\mathbf{B}$ form the beta values, SSE, and a matrix proportional to the covariance of the beta values for the least-squares estimates of $\mathbf{B}$ in the linear model

$$\mathbf{Y} = \mathbf{X}\mathbf{B} + \epsilon .$$

If any pivot becomes very close to zero (less than or equal to 1E−12), the row and column for that pivot are zeroed. See Goodnight (1979) for more information.

SYMSQR Function

converts a square matrix to a symmetric matrix

SYMSQR(*matrix***)**

where *matrix* is a square numeric matrix.

The SYMSQR function takes a square numeric matrix (size $n \times n$) and compacts the elements from the lower triangle into a column vector ($n(n+1)/2$ rows). The matrix is not checked for actual symmetry.

Therefore, the statement

```
sym=symsqr({1 2, 3 4});
```

sets

```
                SYM           3 rows      1 col     (numeric)

                                          1
                                          3
                                          4
```

Note that the 2 is lost since it is only present in the upper triangle.

T Function

transposes a matrix

T(*matrix***)**

where *matrix* is a numeric or character matrix or literal.

The T (transpose) function returns the transpose of its argument. It is equivalent to the transpose operator as written with a transpose postfix operator (‘), but since some keyboards do not support the backquote character, this function is provided as an alternate.

For example, the statements

```
x={1 2, 3 4};
y=t(x);
```

result in the matrix

```
                Y             2 rows      2 cols    (numeric)

                                      1         3
                                      2         4
```

TEIGEN Call

computes the eigenvalues and eigenvectors of square matrices

The TEIGEN subroutine is an alias for the EIGEN subroutine.

TEIGVAL Functions

compute eigenvalues of square matrices

The TEIGVAL function is an alias for the EIGVAL function.

TEIGVEC Functions

compute eigenvectors of square matrices

The TEIGVEC function is an alias for the EIGVEC function.

TOEPLITZ Function

generates a Toeplitz or block-Toeplitz matrix

TOEPLITZ(*a*)

where a is either a vector or a numeric matrix.

The TOEPLITZ function generates a Toeplitz matrix from a vector, or a block Toeplitz matrix from a matrix. A block Toeplitz matrix has the property that all matrices on the diagonals are the same. The argument a is an $(np) \times p$ or $p \times (np)$ matrix; the value returned is the $(np) \times (np)$ result.

The TOEPLITZ function uses the first $p \times p$ submatrix, $\mathbf{A}_1$, of the argument matrix as the blocks of the main diagonal. The second $p \times p$ submatrix, $\mathbf{A}_2$, of the argument matrix forms one secondary diagonal, with the transpose $\mathbf{A}_2'$ forming the other. The remaining diagonals are formed accordingly. If the first $p \times p$ submatrix of the argument matrix is symmetric, the result is also symmetric. If $\mathbf{A}$ is $(np) \times p$, the first p columns of the returned matrix, $\mathbf{R}$, will be the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, the first p rows of $\mathbf{R}$ will be the same as $\mathbf{A}$. The TOEPLITZ function is especially useful in time-series applications, where the covariance matrix of a set of variables with its lagged set of variables is often assumed to be a block Toeplitz matrix.

If

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2|\mathbf{A}_3|\cdots|\mathbf{A}_n]$$

and if $\mathbf{R}$ is the matrix formed by the TOEPLITZ function, then

$$\mathbf{R} = \left[\begin{array}{c|c|c|c|c} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2' & \mathbf{A}_1 & \mathbf{A}_2 & \cdots & \mathbf{A}_{n-1} \\ \mathbf{A}_3' & \mathbf{A}_2' & \mathbf{A}_1 & \cdots & \mathbf{A}_{n-2} \\ \vdots & & & & \\ \mathbf{A}_n' & \mathbf{A}_{n-1}' & \mathbf{A}_{n-2}' & \cdots & \mathbf{A}_1 \end{array}\right]$$

If

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

and if $\mathbf{R}$ is the matrix formed by the TOEPLITZ function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2' & \mathbf{A}_3' & \cdots & \mathbf{A}_n' \\ \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_2' & \cdots & \mathbf{A}_{n-1}' \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{A}_{n-1} & \mathbf{A}_{n-2} & \cdots & \mathbf{A}_1 \end{bmatrix}$$

Three examples follow.

```
r=toeplitz(1:5);
```

```
          R              5 rows      5 cols    (numeric)

                1          2          3          4          5
                2          1          2          3          4
                3          2          1          2          3
                4          3          2          1          2
                5          4          3          2          1
```

```
r=toeplitz({1 2 ,
            3 4 ,
            5 6 ,
            7 8});
```

```
          R              4 rows      4 cols    (numeric)

                    1          2          5          7
                    3          4          6          8
                    5          6          1          2
                    7          8          3          4
```

```
r=toeplitz({1 2 3 4 ,
            5 6 7 8});
```

```
          R              4 rows      4 cols    (numeric)

                    1          2          3          4
                    5          6          7          8
                    3          7          1          2
                    4          8          5          6
```

TPSPLINE Call

computes thin-plate smoothing splines

CALL TPSPLINE(*fitted, coeff, adiag, gcv,* x, y <, *lambda*> **);**

The TSPLINE subroutine computes thin-plate smoothing spline (TPSS) fits to approximate smooth multivariate functions that are observed with noise. The generalized cross validation (GCV) function is used to select the smoothing parameter.

The TPSPLINE subroutine returns the following values:

fitted is an $n \times 1$ vector of fitted values of the TPSS fit evaluated at the design points $\mathbf{x}$. The n is the number of observations. The final TPSS fit depends on the optional *lambda*.

coeff is a vector of spline coefficients. The vector contains the coefficients for basis functions in the null space and the representer of evaluation functions at unique design points. (Refer to Wahba 1990 for more detail on reproducing kernel Hilbert space and representer of evaluation functions.) The length of *coeff* vector depends on the number of unique design points and the number of variables in the spline model. In general, let *nuobs* and k be the number of unique rows and the number of columns of $\mathbf{x}$ respectively. The length of *coeff* equals to $k +$ *nuobs* $+ 1$. The *coeff* vector can be used as an input of TPSPLNEV to evaluate the resulting TPSS fit at new data points.

adiag is an $n \times 1$ vector of diagonal elements of the "hat" matrix. See the "Details" section.

gcv If *lambda* is not specified, then *gcv* is the minimum value of the GCV function. If *lambda* is specified, then *gcv* is a vector (or scalar if *lambda* is a scalar) of GCV values evaluated at the *lambda* points. It provides users both with the ability to study the GCV curves by plotting *gcv* against *lambda*, and with the chance to identify a possible local minimum.

The inputs to the TPSPLINE subroutine are as follows:

x is an $n \times k$ matrix of design points on which the TPSS is to be fit. The k is the number of variables in the spline model. The columns of x need to be linearly independent and contain no constant column.

y is the $n \times 1$ vector of observations.

lambda is a optional $q \times 1$ vector containing λ values in $log_{10}(n\lambda)$ scale. This option gives users the power to control how they want the TPSPLINE subroutine to function. If *lambda* is not specified (or *lambda* is specified and $q > 1$) the GCV function is used to choose

the "best" λ and the returning *fitted* values are based on the λ that minimizes the GCV function. If *lambda* is specified and $q = 1$, no minimization of the GCV function is involved and the *fitted*, *coeff* and *adiag* values are all based on the TPSS fit using this particular *lambda*. This gives users the freedom to choose the λ which they think appropriate.

Aside from the values returned, the TPSPLINE subroutine also prints other useful information such as the number of unique observations, the dimensions of the null space, the number of parameters in the model, a GCV estimate of λ, the smoothing penalty, the residual sum of square, the trace of $(I - A(\lambda))$, an estimate of σ^2, and the sum of squares for replication.

Note: No missing values are allowed within the input arguments. Also, you should use caution if you want to specify small *lambda* values. Since the true $\lambda = (10^{\log_{10} lambda})/n$, a very small value for *lambda* can cause λ to be smaller than the magnitude of machine error and usually the returned *gcv* values from such a λ cannot be trusted. Finally, when using TPSPLINE be aware that TPSS is a computationally intensive method. Therefore a large data set (that is, a large number of unique design points) will take a lot of computer memory and time.

For convenience, we illustrate the TPSS method with a two-dimensional independent variable $\mathbf{X} = (\mathbf{x}^1, \mathbf{x}^2)$. More details can be found in Wahba (1990), or in Bates, *et al.* (1987).

Assume that the data is from the model

$$y_i = f(\mathbf{x}_i) + \epsilon_i,$$

where $(\mathbf{x}_i, y_i), i = 1, \ldots, n$ are the observations. The function f is unknown and you assume that it is reasonably smooth. The error terms $\epsilon_i, i = 1, \ldots, n$ are independent zero-mean random variables.

You will measure the smoothness of f by the integral over the entire plane of the square of the partial derivatives of f of total order 2, that is

$$J_2(f) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \left[\frac{\partial^2 f}{\partial {x_1}^2}\right]^2 + 2\left[\frac{\partial^2 f}{\partial x_1 \partial x_2}\right]^2 + \left[\frac{\partial^2 f}{\partial {x_2}^2}\right]^2 \, dx_1 dx_2.$$

Using this as a smoothness penalty, the thin-plate smoothing spline estimate f_λ of f is the minimizer of

$$S_\lambda(f) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(\mathbf{x}_i))^2 + \lambda J_2(f).$$

Duchon (1976) derived that the minimizer f_λ can be represented as

$$f_\lambda(\mathbf{x}) = \sum_{i=1}^{3}\beta_i\phi_i(\mathbf{x}) + \sum_{i=1}^{n}\delta_i E_2(\mathbf{x} - \mathbf{x}_i),$$

where $(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \phi_3(\mathbf{x})) = (1, \mathbf{x}^1, \mathbf{x}^2)$ and $E_2(\backslash) = \frac{1}{2^3\pi}\|\backslash\|^2 ln(\|\backslash\|)$.

Let matrix $\mathbf{K}$ have entries $(\mathbf{K})_{ij} = E_2(\mathbf{x}_i - \mathbf{x}_j)$ and matrix $\mathbf{T}$ have entries $(\mathbf{T})_{ij} = \phi_j(\mathbf{x}_i)$. Then the minimization problem can be rewritten as finding coefficients β and δ to minimize

$$S_\lambda(\beta, \delta) = \frac{1}{n}\|\mathbf{y} - \mathbf{T}\beta - \mathbf{K}\delta\|^2 + \lambda\delta^T\mathbf{K}\delta.$$

The final TPSS fits can be viewed as a type of generalized ridge regression estimator. The λ is called the smoothing parameter, which controls the balance between the goodness of fit and the smoothness of the final estimate. The smoothing parameter can be chosen by minimizing the Generalized Cross Validation function (GCV). If you write

$$\hat{\mathbf{y}} = \mathbf{A}(\lambda)\mathbf{y},$$

and call the $\mathbf{A}(\lambda)$ as the "hat" matrix, the GCV function $V(\lambda)$ is defined as

$$V(\lambda) = \frac{(1/n)\|(\mathbf{I} - \mathbf{A}(\lambda)\mathbf{y}\|^2}{[(1/n)\text{tr}(\mathbf{I} - \mathbf{A}(\lambda))]^2}.$$

The returned values from this function call will provide the $\hat{\mathbf{y}}$ as *fitted*, the (β, δ) as *coeff*, and $diag(A(\lambda))$ as *adiag*.

To evaluate the TPSS fit $f_\lambda(\mathbf{x})$ at new data points, you can use the TPSPLNEV call.

Suppose $\mathbf{X}^{\text{new}}$, a $m \times k$ matrix, contains the m new data points at which you want to evaluate f_λ. Let $(\mathbf{T}^{\text{new}}_{ij}) = \phi_j(\mathbf{x}^{\text{new}}_i)$ and $(\mathbf{K}^{\text{new}}_{ij}) = E_2(\mathbf{x}^{\text{new}}_i - \mathbf{x}_j)$ be the ij^{th} elements of $\mathbf{T}^{\text{new}}$ and $\mathbf{K}^{\text{new}}$ respectively. The prediction at new data points $\mathbf{X}^{\text{new}}$ is

$$\mathbf{y}_{\text{pred}} = T^{\text{new}}\beta + K^{\text{new}}\delta.$$

Therefore, using the coefficient (β, δ) obtained from TPSPLINE call, the $\mathbf{y}_{\text{pred}}$ can be easily evaluated.

TPSPLNEV Call

evaluates the thin-plate smoothing spline at new data points

It can be used only after the TPSPLINE call.

CALL TPSPLNEV(*pred, xpred, x, coeff***);**

The TPSPLNEV subroutine returns the following value:

pred is an $m \times 1$ vector of the predicated values of the TPSS fit evaluated at m new data points.

The inputs to the TPSPLNEV subroutine are as follows:

xpred is an $m \times k$ matrix of data points at which the f_λ is evaluated, where m is the number of new data points and k is the number of variables in the spline model.

x is an $n \times k$ matrix of design points which is used as an input of TPSPLINE call.

coeff is the coefficient vector returned from the TPSPLINE call.

See the previous section on the TPSPLINE call for details about the TSPLNEV subroutine.

As an example, consider the following data set, which consists of two independent variables. The plot of the raw data can be found in the first panel of Figure 17.1.

```
proc iml;
x={ -1.0 -1.0,    -1.0 -1.0,     -.5 -1.0,     -.5 -1.0,
      .0 -1.0,      .0 -1.0,      .5 -1.0,      .5 -1.0,
     1.0 -1.0,     1.0 -1.0,    -1.0 -.5,     -1.0 -.5,
      -.5 -.5,      -.5 -.5,      .0 -.5,       .0 -.5,
       .5 -.5,       .5 -.5,     1.0 -.5,      1.0 -.5,
      -1.0 .0,      -1.0 .0,      -.5 .0,       -.5 .0,
        .0 .0,        .0 .0,       .5 .0,        .5 .0,
       1.0 .0,       1.0 .0,     -1.0 .5,      -1.0 .5,
       -.5 .5,       -.5 .5,       .0 .5,        .0 .5,
        .5 .5,        .5 .5,      1.0 .5,       1.0 .5,
     -1.0 1.0,     -1.0 1.0,     -.5 1.0,      -.5 1.0,
       .0 1.0,       .0 1.0,      .5 1.0,       .5 1.0,
      1.0 1.0,      1.0 1.0 };

y={15.54483570,  15.76312613,   18.67397826,   18.49722167,
   19.66086310,  19.80231311,   18.59838649,   18.51904737,
   15.86842815,  16.03913832,   10.92383867,   11.14066546,
   14.81392847,  14.82830425,   16.56449698,   16.44307297,
   14.90792284,  15.05653924,   10.91956264,   10.94227538,
   9.614920104,  9.646480938,   14.03133439,   14.03122345,
   15.77400253,  16.00412514,   13.99627680,   14.02826553,
   9.557001644,  9.584670472,   11.20625177,   11.08651907,
   14.83723493,  14.99369172,   16.55494349,   16.51294369,
   14.98448603,  14.71816070,   11.14575565,   11.17168689,
   15.82595514,  15.96022497,   18.64014953,   18.56095997,
   19.54375504,  19.80902641,   18.56884576,   18.61010439,
   15.86586951,  15.90136745 };
```

Now generate a sequence of λ from -3.8 to -3.3 so that we can study the GCV function within this range.

```
do j=-3.8 to -3.3 by 0.1;
  lambda=lambda||j;
end;
lambda=t(lambda);
```

Use the following IML statement to do the thin-plate smoothing spline fit and returning the fitted values on those design points.

```
call tpspline(fit,coef,adiag,gcv, x, y,lambda);
```

The output from this call follows.

SUMMARY OF TPSPLINE CALL

Number of observations	50
Number of unique design points	25
Dimension of polynomial Space	3
Number of Parameters	28

GCV Estimate of Lambda	0.00000668
Smoothing Penalty	2558.14323
Residual Sum of Squares	0.24611
Trace of (I-A)	25.40680
Sigma^2 estimate	0.00969
Sum of Squares for Replication	0.24223

After this TPSPLINE call, you obtained the fitted value. The fitted surface is plotted in the second panel of Figure 17.1. Also in Figure 17.1, panel 4, you plot the GCV function values against *lambda*. From panel 2, you see that because of the spare design points, the fitted surface is a little bit rough. In order to study the TPSS fit $f_\lambda(\mathbf{x})$ more closely, you use the following IML statements to generate a more dense grid on $[-1,1] \times [-1,1]$.

```
do i1=-1 to 1 by 0.1;
   do i2=-1 to 1 by 0.1;
      x1=x1||i1;
      x2=x2||i2;
   end;
end;
x1=t(x1);
x2=t(x2);
xpred=x1||x2;
```

Now you can use the function TPSPLNEV to evaluate $f_\lambda(\mathbf{x})$ on this dense grid.

```
call tpsplnev(pred,  xpred, x, coef);
```

The final fitted surface is plotted in Figure 17.1, panel 3.

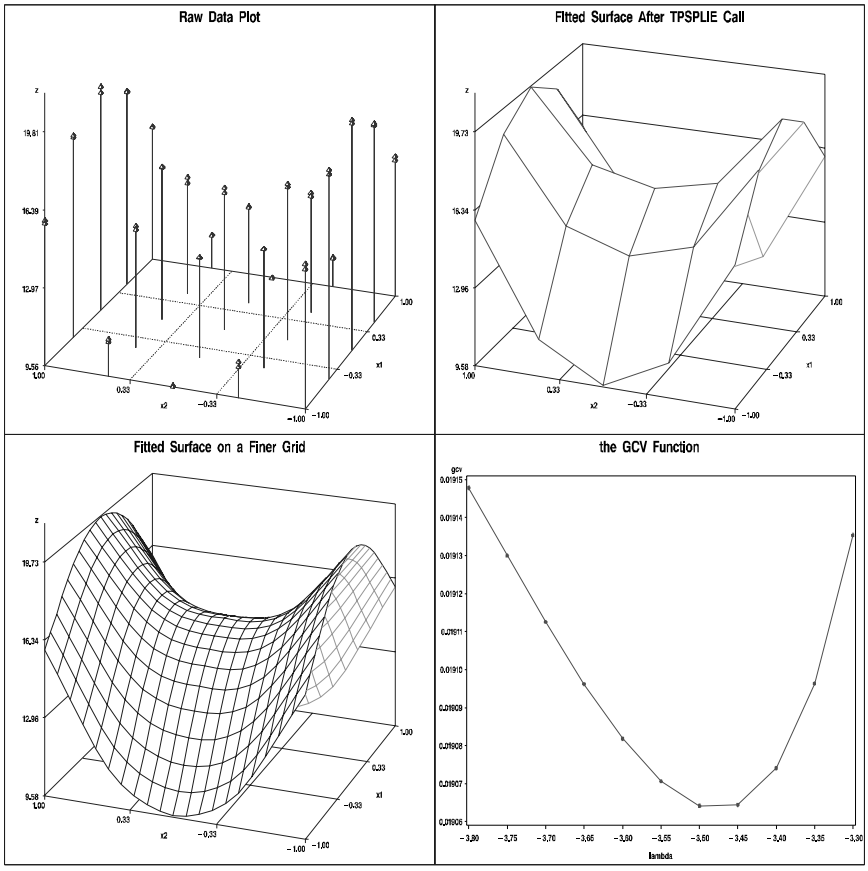


Figure 17.1. Plots of Fitted Surface

TRACE Function

sums diagonal elements

TRACE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The TRACE function produces a single numeric value that is the sum of the diagonal elements of *matrix*. For example, the statement

```
a=trace({5 2, 1 3});
```

produces the result

```
A             1 row       1 col     (numeric)

                          8
```

TRISOLV Function

solves linear systems with triangular matrices

TRISOLV(*code*, r, b <, *piv*>**)**

The TRISOLV function returns the following value:

x is the $n \times p$ matrix $\mathbf{X}$ containing p solutions of the p linear systems specified by *code*, r, and b.

The inputs to the TRISOLV function are as follows:

code specifies which of the following forms of triangular linear system has to be solved:

code=1	solve $\mathbf{Rx} = \mathbf{b}$, $\mathbf{R}$ upper triangular
code=2	solve $\mathbf{R}'\mathbf{x} = \mathbf{b}$, $\mathbf{R}$ upper triangular
code=3	solve $\mathbf{R}'\mathbf{x} = \mathbf{b}$, $\mathbf{R}$ lower triangular
code=4	solve $\mathbf{Rx} = \mathbf{b}$, $\mathbf{R}$ lower triangular

r specifies the $n \times n$ nonsingular upper (*code*=1,2) or lower (*code*=3,4) triangular coefficient matrix $\mathbf{R}$. Only the upper or lower triangle of argument matrix r is used; the other triangle can contain any information.

b specifies the $n \times p$ matrix, $\mathbf{B}$, of p right-hand sides $\mathbf{b}_k$.

piv specifies an optional n vector that relates the order of the columns of matrix $\mathbf{R}$ to the order of the columns of an original coefficient matrix

$\mathbf{A}$ for which matrix $\mathbf{R}$ has been computed as a factor. For example, the vector *piv* can be the result of the QR decomposition of a matrix $\mathbf{A}$ whose columns were permuted in the order $\mathbf{A}_{piv[1]}, \ldots, \mathbf{A}_{piv[n]}$.

For *code*=1 and *code*=3, the solution is obtained by backward elimination. For *code*=2 and *code*=4, the solution is obtained by forward substitution.

If TRISOLV recognizes the upper or lower triangular matrix $\mathbf{R}$ as a singular matrix (that is, one that contains at least one zero diagonal element), it exits with an error message.

See the example in the QR call section.

TSBAYSEA Call

performs Bayesian seasonal adjustment modeling

CALL TSBAYSEA(*trend, season, series, adjust, abic, data*
*<,order, sorder, rigid, npred, opt, cntl, print>***);**

The inputs to the TSBAYSEA subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector.

order specifies the order of trend differencing. The default is *order*=2.

sorder specifies the order of seasonal differencing. The default is *sorder*=1.

rigid specifies the rigidity of the seasonal pattern. The default is *rigid*=1.

npred specifies the length of the forecast beyond the available observations. The default is *npred*=0.

opt specifies the options vector.

- *opt*[1] specifies the number of seasonal periods (*speriod*). By default, *opt*[1]=12.
- *opt*[2] specifies the year when the series starts (*year*). If *opt*[2]=0, there will be no trading day adjustment. By default, *opt*[2]=0.
- *opt*[3] specifies the month when the series starts (*month*). If *opt*[2]=0, this option is ignored. By default, *opt*[3]=1.
- *opt*[4] specifies the upper limit value for outlier determination (*rlim*). Outliers are considered as missing values. If this value is less than or equal to 0, TSBAYSEA assumes that the input data does not contain outliers. The default is *rlim*=0. See the "Missing Values" section on page 269.
- *opt*[5] refers to the number of time periods processed at one time (*span*). The default is *opt*[5]=4.
- *opt*[6] specifies the number of time periods to be shifted (*shift*). By default, *opt*[6]=1.

opt[7] controls the transformation of the original series (*logt*). If *opt*[7]=1, log transformation is requested. No transformation (*opt*[7]=0) is the default.

cntl specifies control values for the TSBAYSEA subroutine. These values will be automatically set. Be careful if you change these values.

cntl[1] controls the adaptivity of the trading day adjustment component (*wtrd*). The default is *cntl*[1]=1.0.

cntl[2] controls the sum of seasonal components within a period (*zersum*). The larger the value of *cntl*[2], the closer to zero this sum is. By default, *cntl*[2]=1.0.

cntl[3] controls the leap year effect (*delta*). The default is *cntl*[3]=7.0.

cntl[4] specifies the prior variance of the initial trend (*alpha*). The default is *cntl*[4]=0.01.

cntl[5] specifies the prior variance of the initial seasonal component (*beta*). The default is *cntl*[5]=0.01. [.03in]

cntl[6] specifies the prior variance of the initial sum of seasonal components (*gamma*). The default is *cntl*[6]=0.01.

print requests the power spectrum and the estimated and forecast values of time series components. If *print*=2, the spectra of irregular, differenced trend and seasonal series are printed, together with estimates and forecast values. If *print*=1, only the estimates and forecast values of time series components are printed.

If *print*=0, printed output is suppressed. The default is *print*=0.

The TSBAYSEA subroutine returns the following values:

trend refers to the estimate and forecast of the trend component.

season refers to the estimate and forecast of the seasonal component.

series refers to the smoothed and forecast values of the time series.

adjust refers to the seasonally adjusted series.

abic refers to the value of ABIC from the final estimates.

Bayesian seasonal adjustments are performed with the TSBAYSEA subroutine. The smoothness of the trend and seasonal components is controlled by the prior distribution. The Akaike Bayesian Information Criterion (ABIC) is defined to compare with alternative models. The basic TSBAYSEA procedure processes the block of data in which the length is SPAN*SPERIOD, while the first block of data consists of length (2*SPAN-1)*SPERIOD. The block of data is shifted successively by SHIFT*SPERIOD.

The TSBAYSEA call decomposes the series y_t into the following form:

$$y_t = T_t + S_t + \epsilon_t$$

where T_t is a trend component, S_t denotes a seasonal component, and ϵ_t is an irregular component. To estimate the seasonal and trend components, some constraints are imposed such that the sum of squares of $\nabla^k T_t$, $\nabla_L^l S_t$, and $\sum_{i=0}^{L-1} S_{t-i}$ is small, where ∇ and ∇_L are difference operators. Then the solution can be obtained by minimizing

$$\sum_{t=1}^{N} \left\{ (y_t - T_t - S_t)^2 + d^2 \left[s^2 (\nabla^k T_t)^2 + (\nabla_L^l S_t)^2 + z^2 (S_t + \cdots + S_{t-L+1})^2 \right] \right\}$$

where d measures the smoothness of the trend and seasonality, s measures the smoothness of the trend, and z is a smoothness constant for the sum of the seasonal variability. The value of d is estimated while the constants, s and z, are chosen *a priori*. The value of s is equal to $\frac{1}{RIGID}$, and the constant z is determined as ZERSUM*RIGID/SPERIOD$^{1/2}$. The larger the constant RIGID, the more rigid the seasonal pattern is. See the section, "Bayesian Constrained Least Squares", for more information.

To analyze the monthly data with rigidity 0.5, you can specify

```
call tsbaysea(trend,season,series,adj,abic) data=z order=2
        sorder=1 rigid=0.5 npred=10 print=2;
```

or

```
call tsbaysea(trend,season,series,adj,abic,z,2,1,0.5,10,,,2);
```

The TREND, SEASON, and SERIES components contain 10-period-ahead forecast values as well as the smoothed estimates. The detailed result is also printed since the PRINT=2 option is specified.

TSDECOMP Call

analyzes nonstationary time series by using smoothness priors modeling

CALL TSDECOMP(*comp, est, aic, data, <,xdata, order, sorder, nar, npred, init, opt, icmp, print>***);**

The inputs to the TSDECOMP subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector.

xdata specifies a $T \times K$ explanatory data matrix.

order specifies the order of trend differencing (0, 1, 2, or 3). The default is 2.

sorder specifies the order of seasonal differencing (0, 1, or 2). The default is 1.

nar specifies the order of the AR process. The default is 0.

npred specifies the length of the forecast beyond the available observations. The default is 0.

init specifies the initial values of parameters. The initial values are specified as variances for trend difference equation, AR process, seasonal difference equation, regression equation, and partial AR coefficients. The corresponding default variance values are 0.005, 0.8, 1E−5, and 1E−5. The default partial AR coefficient values are determined as

$$\psi_i = 0.88 \times (-0.6)^{i-1} \qquad i = 1, 2, \ldots, nar$$

opt specifies the options vector.

opt[1] specifies the mean deletion option. The mean of the original series is subtracted from the series if *opt*[1]=−1. By default, the original series is processed (*opt*[1]=0). When regressors are specified, only the *opt*[1]=0 option is allowed.

opt[2] specifies the trading day adjustment. The default is *opt*[2]=0.

opt[3] specifies the year ($\geq$ 1900) when the series starts. If *opt*[3]=0, there is no trading day adjustment. By default, *opt*[3]=0.

opt[4] specifies the number of seasons within a period (*speriod*). By default, *opt*[4]=12.

opt[5] controls the transformation of the original series. If *opt*[5]=1, log transformation is requested. By default, there is no transformation (*opt*[5]=0).

opt[6] specifies the maximum number of iterations allowed. The default is *opt*[6] = 200.

opt[7] specifies the update technique for the quasi-Newton optimization technique. If *opt*[7]=1 is specified, the dual Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update method is used. If *opt*[7]=2 is specified, the dual Davidon, Fletcher, and Powell (DFP) update method is used. The default is *opt*[7]=1.

opt[8] specifies the line search technique for the quasi-Newton optimization method. The default is *opt*[8] = 2.

- *opt*[8]=1 specifies a line search method that requires the same number of objective function and gradient calls for cubic interpolation and extrapolation.
- *opt*[8]=2 specifies a line search method that requires more objective function calls than gradient calls for cubic interpolation and extrapolation.
- *opt*[8]=3 specifies a line search method that requires the same number of objective function and gradient calls for cubic interpolation and extrapolation.
- *opt*[8]=4 specifies a line search method that requires the same number of objective function and gradient calls for cubic interpolation and stepwise extrapolation.
- *opt*[8]=5 specifies a line search method that is a modified version of *opt*[8]=4.
- *opt*[8]=6 specifies the golden section line search method that uses only function values for linear approximation.

opt[8]=7 specifies the bisection line search method that uses only function values for linear approximation.

opt[8]=8 specifies the Armijo line search method that uses only function values for linear approximation.

opt[9] specifies the upper bound of the variance estimates. If you specify *opt*[9]=*value*, the variances are estimated with the constraint that $\sigma \leq value$. When you specify the *opt*[9]=0 option, the upper bound is not imposed. The default is *opt*[9]=0.

opt[10] specifies the length of data used in backward filtering for the Kalman filter initialization. The default value of *opt*[10] is 100 if the number of observations is greater than 100; otherwise, the default value is the number of observations.

icmp specifies which component is calculated.

icmp=1 requests the estimate and forecast of trend component.

icmp=2 requests the estimate and forecast of seasonal component.

icmp=3 requests the estimate and forecast of AR component.

icmp=4 requests the trading day adjustment component.

icmp=5 requests the regression component.

icmp=6 requests the time-varying regression coefficients.

You can calculate multiple components by specifying a vector. For example, you can specify *icmp*={1 2 3 5}.

print specifies the print option. By default, printed output is suppressed (*print*=0). If you specify *print*=1, the subroutine prints the final estimates. The iteration history is printed if you specify *print*=2.

The TSDECOMP subroutine returns the following values:

comp refers to the estimate and forecast of the trend component.

est refers to the parameter estimates including coefficients of the AR process.

aic refers to the AIC statistic obtained from the final estimates.

The TSDECOMP subroutine analyzes nonstationary time series by using smoothness priors modeling (see the "Smoothness Priors Modeling" section on page 252 for more details). The likelihood function is maximized with respect to hyperparameters. The Kalman filter algorithm is used for filtering, smoothing, and forecasting. The TSDECOMP call decomposes the time series y_t as follows:

$$y_t = T_t + S_t + TD_t + u_t + R_t + \epsilon_t$$

where T_t represents the trend component, S_t denotes the seasonal component, TD_t represents the trading day adjustment component, u_t denotes the autoregressive process component, R_t denotes regression effect components, and ϵ_t represents the irregular term with zero mean and constant variance.

The trend components are constrained as follows:

$$\nabla^k T_t = w_{1t}, \qquad w_{1t} \sim N(0, \tau_1^2)$$

When you specify the ORDER=0 option, the trend component is not estimated. The maximum order of differencing is 3 ($k = 0, \ldots, 3$).

The seasonal components are denoted as a stochastically perturbed equation:

$$\left(1 + \sum_{i=1}^{L-1} \mathbf{B}^i\right)^l S_t = w_{2t}, \qquad w_{2t} \sim N(0, \tau_2^2)$$

When you specify SORDER=0, the seasonal component is not estimated. The maximum value of l is 2 ($l = 0, 1,$ or 2).

The stationary autoregressive (AR) process is denoted as a stochastically perturbed equation:

$$u_t = \sum_{i=1}^{p} \alpha_i u_{t-i} + w_{3t}, \qquad w_{3t} \sim N(0, \tau_3^2)$$

where p is the order of AR process. When NAR=0 is specified, the AR process component is not estimated.

The time-varying regression coefficients are estimated if you include exogenous variables:

$$R_t = \mathbf{X}_t \beta_t$$

where $\mathbf{X}_t$ contains m regressors except the constant term and $\beta_t' = (\beta_{1t}, \ldots, \beta_{mt})$. The time-varying coefficients β_t follow the random walk process:

$$\beta_{jt} = \beta_{jt-1} + v_{jt}, \qquad v_{jt} \sim N(0, \sigma_j^2)$$

where β_{jt} is an element of the coefficient vector β_t.

The trading day adjustment component TD_t is deterministically restricted. See the section, "State Space and Kalman Filter Method", for more information.

You can estimate the time-varying coefficient model as follows:

```
call tsdecomp COMP=beta ORDER=0 SORDER=0 NAR=0
              DATA=y XDATA=x ICMP=6;
```

The output matrix BETA contains time-varying regression coefficients.

TSMLOCAR Call

analyzes nonstationary or locally stationary time series by using the minimum AIC procedure

CALL TSMLOCAR(*arcoef, ev, nar, aic, start, finish, data*
<,maxlag, opt, missing, print>);

The inputs to the TSMLOCAR subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector.

maxlag specifies the maximum lag of the AR process. This value should be less than half the length of locally stationary spans. The default is *maxlag*=10.

opt specifies an options vector.

- *opt*[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=-1. An intercept coefficient is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean value of the input series is 0. The default is *opt*[1]=0.
- *opt*[2] specifies the number (J) of basic spans. By default, *opt*[2]=1.
- *opt*[3] specifies the minimum AIC option. If *opt*[3]=0, the *maximum lag* AR process is estimated. If *opt*[3]=1, the minimum AIC procedure is performed. The default is *opt*[3]=1.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean. *print*] specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the AR estimation result, while the *print*=2 option plots the power spectral density as well as the AR estimates.

The TSMLOCAR subroutine returns the following values:

arcoef refers to an *nar* $\times$ 1 AR coefficient vector of the final model if the intercept estimate is not included. If *opt*[1]=1, the first element of the *arcoef* vector is an intercept estimate.

ev refers to the error variance.

nar is the selected AR order of the final model. If *opt*[3]=0, *nar=maxlag*.

aic refers to the minimum AIC value of the final model.

start refers to the starting position of the input series, which corresponds to the first observation of the final model.

finish refers to the ending position of the input series, which corresponds to the last observation of the final model.

The TSMLOCAR subroutine analyzes nonstationary (or locally stationary) time series by using the minimum AIC procedure. The data of length T is divided into J locally stationary subseries, which consist of $\frac{T}{J}$ observations. See the "Nonstationary Time Series" section on page 256 for details.

TSMLOMAR Call

analyzes nonstationary or locally stationary multivariate time series by using the minimum AIC procedure

CALL TSMLOMAR(*arcoef, ev, nar, aic, start, finish, data*
*<,maxlag, opt, missing, print>***);**

The inputs to the TSMLOMAR subroutine are as follows:

data specifies a $T \times M$ data matrix, where T is the number of observations and M is the number of variables to be analyzed.

maxlag specifies the maximum lag of the vector AR (VAR) process. This value should be less than $\frac{1}{2M}$ of the length of locally stationary spans. The default is *maxlag*=10.

opt specifies an options vector.

- *opt*[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=-1. An intercept coefficient is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean values of input series are zeroes. The default is *opt*[1]=0.
- *opt*[2] specifies the number (J) of basic spans. By default, *opt*[2]=1.
- *opt*[3] specifies the minimum AIC option. If *opt*[3]=0, the *maximum lag* VAR process is estimated. If *opt*[3]=1, a minimum AIC procedure is used. The default is *opt*[3]=1.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the AR estimates, minimum AIC, minimum AIC order, and innovation variance matrix.

The TSMLOMAR subroutine returns the following values.

arcoef refers to an $M \times (M * nar)$ VAR coefficient vector of the final model if the intercept vector is not included. If *opt*[1]=1, the first column of the *arcoef* matrix is an intercept estimate vector.

ev refers to the error variance matrix.

nar is the selected VAR order of the final model. If *opt*[3]=0, *nar=maxlag*.

aic refers to the minimum AIC value of the final model.

start refers to the starting position of the input series *data*, which corresponds to the first observation of the final model.

finish refers to the ending position of the input series *data*, which corresponds to the last observation of the final model.

The TSMLOMAR subroutine analyzes nonstationary (or locally stationary) multivariate time series by using the minimum AIC procedure. The data of length T is divided into J locally stationary subseries. See "Nonstationary Time Series" in the "Nonstationary Time Series" section on page 256 for details.

TSMULMAR Call

estimates VAR processes by using the minimum AIC procedure

CALL TSMULMAR(*arcoef, ev, nar, aic, data*
<*,maxlag, opt, missing, print*>**);**

The inputs to the TSMULMAR subroutine are as follows:

data specifies a $T \times M$ data matrix, where T is the number of observations and M is the number of variables to be analyzed.

maxlag specifies the maximum lag of the VAR process. This value should be less than $\frac{1}{2M}$ of the length of input data. The default is *maxlag*=10.

opt specifies an options vector.

- *opt*[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=-1. An $M \times 1$ intercept vector is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean value of the input data is 0. The default is *opt*[1]=0.
- *opt*[2] specifies the minimum AIC option. If *opt*[2]=0, the *maximum lag* AR process is estimated. If *opt*[2]=1, the minimum AIC procedure is used, while the *opt*[2]=2 option specifies the VAR order selection method based on the AIC. The default is *opt*[2]=1.
- *opt*[3] specifies instantaneous response modeling if *opt*[3]=1. The default is *opt*[3]=0. See the section "Multivariate Time Series Analysis" on page 259 for more information.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the final estimation result, while the *print*=2 option prints intermediate and final results.

The TSMULMAR subroutine returns the following values:

arcoef refers to an $M \times (M * nar)$ AR coefficient matrix if the intercept is not included. If *opt*[1]=1, the first column of the *arcoef* matrix is an intercept vector estimate.

ev refers to the error variance matrix.

nar is the selected VAR order of the minimum AIC procedure. If *opt*[2]=0, *nar*=*maxlag*. *aic*] refers to the minimum AIC value.

The TSMULMAR subroutine estimates the VAR process by using the minimum AIC method. The widely used VAR order selection method is added to the original TIMSAC program, which considers only the possibilities of zero coefficients at the beginning and end of the model. The TSMULMAR subroutine can also estimate the instantaneous response model. See the "Multivariate Time Series Analysis" section on page 259 for details.

TSPEARS Call

analyzes periodic AR models with the minimum AIC procedure

CALL TSPEARS(*arcoef, ev, nar, aic, data*
<*,maxlag, opt, missing, print*>**);**

The inputs to the TSPEARS subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data matrix.

maxlag specifies the maximum lag of the periodic AR process. This value should be less than $\frac{1}{2J}$ of the input series. The default is *maxlag*=10.

opt specifies an options vector.

opt[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=-1. An intercept coefficient is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean values of input series are zeroes. The default is *opt*[1]=0.

opt[2] specifies the number of instants per period. By default, *opt*[2]=1.

opt[3] specifies the minimum AIC option. If *opt*[3]=0, the *maximum lag* AR process is estimated. If *opt*[3]=1, the minimum AIC procedure is used. The default is *opt*[3]=1.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the periodic AR estimates and intermediate process.

The TSPEARS subroutine returns the following values:

arcoef refers to a periodic AR coefficient matrix of the periodic AR model. If *opt*[1]=1, the first column of the *arcoef* matrix is an intercept estimate vector.

ev refers to the error variance.

nar refers to the selected AR order vector of the periodic AR model.

aic refers to the minimum AIC values of the periodic AR model.

The TSPEARS subroutine analyzes the periodic AR model by using the minimum AIC procedure. The data of length T are divided into d periods. There are J instants in one period. See the "Multivariate Time Series Analysis" section on page 259 for details.

TSPRED Call

provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are input

CALL TSPRED(*forecast, impulse, mse, data, coef, nar, nma*
*<,ev, npred, start, constant>***);**

The inputs to the TSPRED subroutine are as follows:

data specifies a $T \times M$ data matrix if the intercept is not included, where T denotes the length of the time series and M is the number of variables to be analyzed. If the univariate time series is analyzed, the input data should be a column vector.

coef refers to the $M(P+Q) \times M$ ARMA coefficient matrix, where P is an AR order and Q is an MA order. If the intercept term is included (*constant*=1), the first row of the coefficient matrix is considered as the intercept term and the coefficient matrix is an $M(P+Q+1) \times M$ matrix. If there are missing values in the *coef* matrix, these are converted to zero.

nar specifies the order of the AR process. If the subset AR process is requested, *nar* should be a row or column vector. The default is *nar*=0.

nma specifies the order of the MA process. If the subset MA process is requested, *nma* should be a vector. The default is *nma*=0.

ev specifies the error variance matrix. If the *ev* matrix is not provided, the prediction error covariance will not be computed.

npred specifies the maximum length of multistep forecasting. The default is *npred*=0.

start specifies the position where the multistep forecast starts. The default is *start*=T.

constant specifies the intercept option. No intercept estimate is included if *constant*=0; otherwise, the intercept estimate is included in the first row of the coefficient matrix. If *constant*=-1, the coefficient matrix is estimated by using mean deleted series. By default, *constant*=0.

The TSPRED subroutine returns the following values:

forecast refers to predicted values.

impulse refers to the impulse response function.

mse refers to the mean square error of s-step-ahead forecast. A scalar missing value is returned if the error variance (*ev*) is not provided.

TSROOT Call

calculates AR and MA coefficients from the characteristic roots of the model or calculates the characteristic roots of the model from the AR and MA coefficients

CALL TSROOT(*matout, matin, nar, nma, <,qcoef, print>***);**

The inputs to the TSROOT subroutine are as follows:

matin refers to the $(nar + nma) \times 2$ characteristic root matrix if the polynomial (ARMA) coefficients are requested (*qcoef*=1), where the first column of the *matin* matrix contains the real part of the root and the second column of the *matin* matrix contains the imaginary part of the root. When the characteristic roots are requested (*qcoef*=0), the first *nar* rows are complex AR coefficients and the last *nma* rows are complex MA coefficients. The default is *qcoef*=0.

nar specifies the order of the AR process. If you specify the subset AR model, the input *nar* should be a row or column vector.

nma specifies the order of the MA process. If you specify the subset MA model, the input *nma* should be a row or column vector.

qcoef requests the ARMA coefficients when the characteristic roots are provided (*qcoef*=1). By default, the characteristic roots of the polynomial are calculated (*qcoef*=0).

print specifies the print option if *print*=1. By default, printed output is suppressed (*print*=0).

The TSROOT subroutine returns the following values

matout refers to the characteristic root matrix if *qcoef*=0; otherwise, the *matout* matrix contains the AR and MA coefficients.

TSTVCAR Call

analyzes time series that are nonstationary in the covariance function

CALL TSTVCAR(*arcoef, variance, est, aic, data*
*<,nar, init, opt, outlier, print>***);**

The inputs to the TSTVCAR subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector.

nar specifies the order of the AR process. The default is *nar*=8.

init specifies the initial values of the parameter estimates. The default is (1E−4, 0.3, 1E−5, 0).

opt specifies an options vector.

- *opt*[1] specifies the mean deletion option. The mean of the original series is subtracted from the series if *opt*[1]=−1. By default, the original series is processed (*opt*[1]=0).
- *opt*[2] specifies the filtering period (*nfilter*). The number of state vectors is determined by $\frac{T}{nfilter}$. The default is *opt*[2]=10.
- *opt*[3] specifies the numerical differentiation method. If *opt*[3]=1, the one-sided (forward) differencing method is used. The two-sided (or central) differencing method is used if *opt*[3]=2. The default is *opt*[3]=1.

outlier specifies the vector of outlier observations. The value should be less than or equal to the maximum number of observations. The default is *outlier*=0.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the final estimates. The iteration history is printed if *print*=2.

The TSTVCAR subroutine returns the following values:

arcoef refers to the time-varying AR coefficients.

variance refers to the time-varying error variances. See the "Smoothness Priors Modeling" section on page 252 for details.

est refers to the parameter estimates.

aic refers to the value of AIC from the final estimates.

Nonstationary time series modeling usually deals with nonstationarity in the mean. The TSTVCAR subroutine analyzes the model that is nonstationary in the covariance. Smoothness priors are imposed on each time-varying AR coefficient and frequency response function. See the "Nonstationary Time Series" section on page 256 for details.

TSUNIMAR Call

determines the order of an AR process with the minimum AIC procedure and estimates the AR coefficients

CALL TSUNIMAR(*arcoef, ev, nar, aic, data*
<,*maxlag, opt, missing, print*>**);**

The inputs to the TSUNIMAR subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector, where T is the number of observations.

maxlag specifies the maximum lag of the AR process. This value should be less than half the number of observations. The default is *maxlag*=10.

opt specifies an options vector.

- *opt*[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=−1. An intercept term is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean value of the input data is 0. The default is *opt*[1]=0.
- *opt*[2] specifies the minimum AIC option. If *opt*[2]=0, the *maximum lag* AR process is estimated. The minimum AIC option, *opt*[2]=1, is the default.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the final estimation result, while the *print*=2 option prints intermediate and final results.

The TSUNIMAR subroutine returns the following values.

arcoef refers to an *nar* $\times$ 1 AR coefficient vector if the intercept is not included. If *opt*[1]=1, the first element of the *arcoef* vector is an intercept estimate.

ev refers to the error variance.

nar refers to the selected AR order by minimum AIC procedure. If *opt*[2]=0, *nar = maximum lag*.

aic refers to the minimum AIC value.

The TSUNIMAR subroutine determines the order of the AR process by using the minimum AIC procedure and estimates the AR coefficients. All AR coefficient estimates up to maximum lag are printed if you specify the print option. See the section, "Least Squares and Householder Transformation", for more information.

TYPE Function

determines the type of a matrix

TYPE(*matrix***)**

where *matrix* is a numeric or character matrix or literal.

The TYPE function returns a single character value; it is **N** if the type of the matrix is numeric; it is **C** if the type of the matrix is character; it is **U** if the matrix does not have a value. Examples of valid statements follow.

The statements

```
a={tom};
r=type(a);
```

set R to **C**. The statements

```
free a;
r=type(a);
```

set R to **U**. The statements

```
a={1 2 3};
r=type(a);
```

set R to **N**.

UNIFORM Function

generates pseudo-random uniform deviates

UNIFORM(*seed***)**

where *seed* is a numeric matrix or literal. The *seed* can be any integer value up to $2^{31} - 1$.

The UNIFORM function returns one or more pseudo-random numbers with a uniform distribution over the interval 0 to 1. The UNIFORM function returns a matrix with the same dimensions as the argument. The first argument on the first call is used for the seed, or if that argument is 0, the system clock is used for the seed. The function is equivalent to the DATA step function RANUNI. An example of a valid statement follows:

```
c=uniform(0);
```

UNION Function

performs unions of sets

UNION(*matrix1*<, *matrix2*,..., *matrix15*>**)**

where *matrix* is a numeric or character matrix or quoted literal.

The UNION function returns as a row vector the sorted set (without duplicates) which is the union of the element values present in its arguments. There can be up to 15 arguments, which can be either all character or all numeric. For character arguments, the element length of the result is the longest element length of the arguments. Shorter character elements are padded on the right with blanks. This function is identical to the UNIQUE function. For example, the statements

```
a={1 2 4 5};
b={3 4};
c=union(a,b);
```

set

```
C            1 row       5 cols    (numeric)

      1         2         3         4         5
```

The UNION function can be used to sort elements of a matrix when there are no duplicates by calling UNION with a single argument.

UNIQUE Function

sorts and removes duplicates

UNIQUE(*matrix1*<*, matrix2,. . ., matrix15*>**)**

where *matrix* is a numeric or character matrix or quoted literal.

The UNIQUE function returns as a row vector the sorted set (without duplicates) of all the element values present in its arguments. The arguments can be either all numeric or all character, and there can be up to 15 arguments specified. This function is identical to the UNION function, the description of which includes an example.

USE Statement

opens a SAS data set for reading

USE *SAS-data-set* <**VAR** *operand*> <**WHERE(***expression***)**>
<**NOBS** *name*>**;**

The inputs to the USE statement are as follows:

SAS-data-set	can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on SAS data sets in *SAS Language Reference: Concepts*.
operand	selects a set of variables.
expression	is evaluated for being true or false.
name	is the name of a variable to contain the number of observations.

If the data set has not already been opened, the USE statement opens the data set for read access. The USE statement also makes the data set the current input data set so that subsequent statements act on it. The USE statement optionally can define selection criteria that are used to control access.

The VAR clause specifies a set of variables to use, where *operand* can be any of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names

- one of the following keywords:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

The following examples show each possible way you can use the VAR clause:

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the *range* specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable is a variable in the SAS data set.

comparison-op is one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

^= ^? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&*clause* (for an AND clause)
clause|*clause* (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

The VAR and WHERE clauses are optional, and you can specify them in any order. If a data set is already open, all the options that the data set was first opened with are still in effect. To override any old options, the new USE statement must explicitly specify the new options. Examples of valid statements follow.

```
use class;
use class var{name sex age};
use class var{name sex age} where(age>10);
```

VALSET Call

performs indirect assignment

CALL VALSET(*char-scalar, argument***);**

The inputs to the VALSET subroutine are as follows:

char-scalar is a character scalar containing the name of a matrix.

argument is a value to which the matrix is set.

The VALSET subroutine expects a single character string argument containing the name of a matrix. It looks up the matrix and moves the value of the second argument to this matrix. For example, the following statements find that the value of the argument **B** is **A** and then assign the value 99 to **A**, the indirect result:

```
b="A";
call valset(b,99);
```

The previous value of the indirect result is freed. The following statement sets **B** to 99, but the value of **A** is unaffected by this statement:

```
b=99;
```

VALUE Function

assigns values by indirect reference

VALUE(*char-scalar***)**

where *char-scalar* is a character scalar containing the name of a matrix.

The VALUE function expects a single character string argument containing the name of a matrix. It looks up the matrix and moves its value to the result. For example, the statements

```
a={1 2 3};
b="A";
c=value(b);
```

find that the value of the argument **B** is **A** and then look up **A** and copy the value 1 2 3 to **C**.

```
C             1 row       3 cols    (numeric)

                1         2         3
```

VARMACOV Call

computes the theoritical auto-cross covariance matrices for stationary VARMA(p, q) model

CALL VARMACOV(*cov, phi, theta, sigma <, p, q, lag>***);**

The inputs to the VARMACOV subroutine are as follows:

phi specifies to a $kp \times k$ matrix containing the vector autoregressive coefficient matrices. All the roots of $|\Phi(B)| = 0$ are greater than one in absolute value.

theta specifies to a $kq \times k$ matrix containing the vector moving-average coefficient matrices. You must specify either *phi* or *theta*.

sigma specifies a $k \times k$ symmetric positive-definite covariance matrix of the innovation series. By default, *sigma* is an identity matrix with dimension k.

p specifies the order of AR. You can also specify the subset of the order of AR. By default, let $phi = \Phi$,

$$p = \frac{\text{the number of row of matrix } \Phi}{\text{the number of column of matrix } \Phi}.$$

For example, consider a 4 dimensional vector time series, if $phi = \Phi$ is 4×4 matrix and $p = 1$, the VARMACOV subroutine computes the theoritical auto-cross covariance matrices of VAR(1) as follows

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \mathbf{m}\epsilon_t.$$

If $phi = \Phi$ is 4×4 matrix and $p = 2$, the VARMACOV subroutine computes the theoritical auto-cross covariance matrices of VAR(2) as follows

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-2} + \mathbf{m}\epsilon_t.$$

If $phi = [\Phi_1' \ \Phi_3']'$ is 8×4 matrix and $p = \{1, 3\}$, the VARMACOV subroutine computes the theoritical auto-cross covariance matrices of VAR(3) as follows

$$\mathbf{y}_t = \Phi_1 \mathbf{y}_{t-1} + \Phi_3 \mathbf{y}_{t-3} + \mathbf{m}\epsilon_t.$$

q specifies the order of MA. You can specify the subset of the order of MA. By default, let $theta = \Theta$,

$$q = \frac{\text{the number of row of matrix } \Theta}{\text{the number of column of matrix } \Theta}.$$

The usage of q is the same as that of p.

lag specifies the length of lags, which must be a positive number. If $lag = h$, the VARMACOV computes the auto-cross covariance matrices from at lag zero to at lag h. By default, *lag* = 12.

The VARMACOV subroutine returns the following value:

cov refers an $(k * lag) \times k$ matrices the theoritical auto-cross covariance VARMA(p, q) series. In case of VMA(q) when $p = 0$, the VARMACOV computes the auto-cross covariance matrices from at lag zero to at lag q.

To compute the theoritical auto-cross covariance matrices of a bivariate ($k = 2$) VARMA(1,1) model

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \mathbf{m}\epsilon_t - \Theta \mathbf{m}\epsilon_{t-1},$$

with $\mathbf{m}\epsilon_t \sim WN(\mathbf{0}, \Sigma)$, where

$$\Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}, \Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix}, \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix},$$

you can specify

```
call varmacov(cov, phi, theta, sigma) lag=5;
```

The VARMACOV subroutine computes theoritical auto-cross covariance matrices for the VARMA(p,q) model when AR coefficient matrices Φ_i, MA coefficient matrices Θ_i, and an inovation covariance matrix Σ are known. Auto-cross covariance matrices $\Gamma(l)$ are

$$\Gamma(l) = \sum_{j=1}^{p} \Gamma(l-j)\Phi_j' - \sum_{j=l}^{q} \Psi(j-l)\Sigma\Theta_j', \quad \text{for } l = 0, \ldots, q$$

$$\Gamma(l) = \sum_{j=1}^{p} \Gamma(l-j)\Phi_j' \quad \text{for } l > q$$

where Ψ_j satisfy

$$\Psi_j = \Phi_1\Psi_{j-1} + \Phi_2\Psi_{j-2} + \cdots + \Phi_p\Psi_{j-p} - \Theta_j$$

with $\Theta_0 = -I$, $\Psi_0 = I$, and $\Psi_j = 0$ for $j < 0$.

VARMASIM Call

generates a VARMA(p,q) series

CALL VARMASIM(*series, phi, theta, mu, sigma, n <, p, q, initial, seed>***);**

The inputs to the VARMASIM subroutine are as follows:

phi specifies to a $kp \times k$ matrix containing the vector autoregressive coefficient matrices.

theta specifies to a $kq \times k$ matrix containing the vector moving-average coefficient matrices. You must specify either *phi* or *theta*.

mu specifies a $k \times 1$ (or $1 \times k$) mean vector of the series. By default, *mu* is a zero vector.

sigma specifies a $k \times k$ covariance matrix of the innovation series. By default, *sigma* is an identity matrix with dimension k.

n specifies the length of the series. By default, $n = 100$.

p specifies the order of VAR. See the VARMACOV subroutine.

q specifies the order of VMA. See the VARMACOV subroutine.

initial specifies the initial values of random variables. If $initial = a_0$, $\mathbf{y}_{-p+1}, \ldots, \mathbf{y}_0$ and $\mathbf{m}\epsilon_{-q+1}, \ldots, \epsilon_0$ take all the same value as $initial = a_0$. If *initial* option is not specified, the initial values are estimated using VARMACOV for stationary vector time series, while the initial values assume as zero values for nonstationary vector time series.

seed specifies the random number seed. See the VNORMAL subroutine.

The VARMASIM subroutine returns the following value:

series refers an $n \times k$ matrices the generated VARMA(p,q) series. When either *initial* option is specified or the zero initial values are used, the returns do not print these initial values.

To generate a bivariate($k = 2$) stationary VARMA(1,1) time series

$$\mathbf{y}_t - \mathbf{m}\mu = \Phi(\mathbf{y}_{t-1} - \mathbf{m}\mu) + \mathbf{m}\epsilon_t - \Theta\mathbf{m}\epsilon_{t-1},$$

with $\mathbf{m}\epsilon_t \sim WN(0, \Sigma)$, where

$$\Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}, \mu = \begin{bmatrix} 10 \\ 20 \end{bmatrix}, \Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix}, \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix},$$

you can specify

```
call varmasim(yt, phi, theta, mu, sigma, 100);
```

To generate a bivariate($k = 2$) nonstationary VARMA(1,1) time series with the same *mu*, *sigma*, and *theta* in previous example and the AR coefficient

$$\Phi = \begin{bmatrix} 1.0 & 0 \\ 0 & 0.3 \end{bmatrix},$$

you can specify

```
call varmasim(yt, phi, theta, mu, sigma, 100) initial=3;
```

VECDIAG Function

creates a vector from a diagonal

VECDIAG(*square-matrix***)**

where *square-matrix* is a square numeric matrix.

The VECDIAG function creates a column vector whose elements are the main diagonal elements of *square-matrix*. For example, the statements

```
a={2 1, 0 -1};
c=vecdiag(a);
```

produce the column vector

```
C              2 rows      1 col     (numeric)

                            2
                           -1
```

VNORMAL Call

generates multivariate normal random series

CALL VNORMAL(*series, mu, sigma, n <, seed>***);**

The inputs to the VNORMAL subroutine are as follows:

mu specifies a $k \times 1$ (or $1 \times k$) mean vector. By default, *mu* is a zero vector.

sigma specifies a $k \times k$ symmetric positive-definite covariance matrix. By default, *sigma* is an identity matrix with dimension k. You must specify either *mu* or *sigma*.

n specifies the length of the series. By default, $n = 100$.

seed specifies the random number seed. If it is not supplied, the system clock is used to generate the seed. If it is negative, then the absolute value is used as the starting seed; otherwise, subsequent calls ignore the value of *seed* and use the last seed generated internally.

The VNORMAL subroutine returns the following value:

series refers an $n \times k$ matrices the generated normal random series.

To generate a bivariate space ($k = 2$) normal random series with mean μ and covariance matrix Σ, where

$$\mu = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \text{ and } \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

you can specify

```
call vnormal(et, mu, sigma, 100);
```

VTSROOT Call

calculates the characteristic roots of the model from AR and MA characteristic functions

CALL VTSROOT(*root, phi, theta<, p, q>***);**

The inputs to the VTSROOT subroutine are as follows:

phi specifies to a $kp \times k$ matrix containing the vector autoregressive coefficient matrices.

theta specifies to a $kq \times k$ matrix containing the vector moving-average coefficient matrices. You must specify either *phi* or *theta*.

p specifies the order of VAR. See the VARMACOV subroutine.

q specifies the order of VMA. See the VARMACOV subroutine.

The VTSROOT subroutine returns the following value:

root refers an $k*(maxar+maxma)\times 5$ matrices, where $maxar$ and $maxma$ are the maximum orders corresponding to AR and MA, respectively. The first $k*maxar$ rows refer the results of the AR part, and the last $k*maxma$ rows refer the results of the MA part. The first column refers the real part of eigenvalues of companion matrices associated with the VAR(p) characteristic function. The second column refers the imaginary part of eigenvalues. The third column refers the modulus of eigenvalues. The forth column refers the degree(radian) of eigenvalues. The fifth column refers the degree(radian$\times 180$) of eigenvalues.

To compute the roots of a bivariate($k=2$) VARMA(1,1) model

$$\mathbf{y}_t = \Phi\mathbf{y}_{t-1} + \mathbf{m}\epsilon_t - \Theta\mathbf{m}\epsilon_{t-1}$$

where

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix}, \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix}$$

you can specify

```
call vtsroot(root, phi, theta);
```

The VTSROOT subroutine computes the eigenvalues of the $kp \times kp$ companion matrices associated with the AR(p) characteristic function, where k is the number of dependent variables. They indicate the stationary condition of the process since the stationary condition on the roots of $|\Phi(B)| = 0$ in the AR(p) is equivalent to the condition in the corresponding AR(1) representation that all eigenvalues of the companion matrix be less than one in absolute value. The stationarity condition is equivalent to the condition in the corresponding AR(1) representation, $\mathbf{Y}_t = \mathbf{m}\Phi\mathbf{Y}_{t-1} + \mathbf{m}\varepsilon_t$, that all eigenvalues of the $kp \times kp$ companion matrix Φ be less than one in absolute value, where $\mathbf{Y}_t = (\mathbf{y}_t', \ldots, \mathbf{y}_{t-p+1}')'$, $\mathbf{m}\varepsilon_t = (\mathbf{m}\epsilon_t', 0', \ldots, 0')'$, and

$$\mathbf{m}\Phi = \begin{bmatrix} \Phi_1 & \Phi_2 & \cdots & \Phi_{p-1} & \Phi_p \\ I & 0 & \cdots & 0 & 0 \\ 0 & I & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & I & 0 \end{bmatrix}$$

Similarly, it can apply for checking the invertible condition of the MA proces.

WINDOW Statement

opens a display window

WINDOW <**CLOSE=**>*window-name* <*window-options*>
<**GROUP=***group-name field-specs*>

< . . .**GROUP=***group-name field-specs*>**;**

where the arguments and options are described below.

The WINDOW statement defines a window on the display and can include a number of fields. The DISPLAY statement actually writes values to the window. The following fields can be specified in the WINDOW statement:

window-name
specifies a name 1 to 8 characters long for the window. This name is displayed in the upper-left border of the window.

CLOSE=*window-name*
closes the window.

window-options
control the size, position, and other attributes of the window. The attributes can also be changed interactively with window commands such as WGROW, WDEF, WSHRINK, and COLOR. A description of the window options follows.

GROUP=*group-name*
starts a repeating sequence of groups of fields defined for the window. The *group-name* specification is a name 1 to 8 characters long used to identify a group of fields in a later DISPLAY statement.

field-specs
are a sequence of field specifications made up of positionals, field operands, formats, and options. These are described in the next section.

The following window options can be specified in the WINDOW statement:

CMNDLINE=*name*
specifies the name of a variable in which the command line entered by the user will be stored.

COLOR=*operand*
specifies the background color for the window. The *operand* is either a quoted character literal, a name, or an operand. The valid values are "WHITE", "BLACK", "GREEN", "MAGENTA", "RED", "YELLOW", "CYAN", "GRAY", and "BLUE". The default value is BLACK.

COLUMNS=*operand*
specifies the starting number of columns for the window. The *operand* is either a literal number, a variable name, or an expression in parentheses. The default value is 78 columns.

ICOLUMN=*operand*
specifies the initial starting column position of the window on the display. The *operand* is either a literal number or a variable name. The default value is column 1.

IROW=*operand*
specifies the initial starting row position of the window on the display. The *operand* is either a literal number or a variable name. The default value is row 1.

MSGLINE=*operand*
specifies the message to be displayed on the standard message line when the window is made active. The *operand* is almost always the name of a variable, but a character literal can be used.

ROWS=*operand*
determines the starting number of rows of the window. The *operand* is either a literal number, the name of a variable containing the number, or an expression in parentheses yielding the number. The default value is 23 rows.

Both the WINDOW and DISPLAY statements allow field specifications, which have the general form:

<positionals> field-operand <format> <field-options>

In the preceding statement,

positionals are directives determining the position on the screen to begin the field. There are four kinds of positionals; any number of positionals are allowed for each field operand.

 # operand specifies the row position; that is, it moves the current position to column 1 of the specified line. The *operand* is either a number, a name, or an expression in parentheses.

/ specifies that the current position move to column 1 of the next row.

@ operand specifies the column position. The *operand* is either a number, a name, or an expression in parentheses. The @ directive should come after the # position if # is specified.

+ operand specifies a skip of columns. The *operand* is either a number, a name, or an expression in parentheses.

field-operand is a character literal in quotes or the name of a variable that specifies what is to go in the field.

format is the format used for display, the value, and the informat applied to entered values. If no format is specified, then the standard numeric or character format is used.

field-options specify the attributes of the field as follows:

PROTECT=YES
P=YES
specifies that the field is protected; that is, you cannot enter values in the field. If the field operand is a literal, it is already protected.

COLOR=*operand*
specifies the color of the field. The *operand* is a literal character value in quotes, a variable name, or an expression in parentheses. The colors available are "WHITE", "BLACK", "GREEN", "MAGENTA", "RED", "YELLOW", "CYAN", "GRAY", and "BLUE". Note that the color specification is different from that of the corresponding DATA step value because it is an operand rather than a name without quotes. The default value is "BLUE".

XMULT Function

performs accurate matrix multiplication

XMULT(*matrix1, matrix2***)**

where *matrix1* and *matrix2* are numeric matrices.

The XMULT function computes the matrix product like the matrix multiplication operator (*) except XMULT uses extended precision to accumulate sums of products. You should use the XMULT function only when you need great accuracy.

XSECT Function

intersects sets

XSECT(*matrix1*<, *matrix2*,. . ., *matrix15*>**)**

where *matrix* is a numeric or character matrix or quoted literal.

The XSECT function returns as a row vector the sorted set (without duplicates) of the element values that are present in all of its arguments. This set is the intersection of the sets of values in its argument matrices. When the intersection is empty, the XSECT function returns a null matrix (zero rows and zero columns). There can be up to 15 arguments, which must all be either character or numeric. For characters, the element length of the result is the same as the shortest of the element lengths of the arguments. For comparison purposes, shorter elements are padded on the right with blanks.

For example, the statements

```
a={1 2 4 5};
b={3 4};
c=xsect(a,b);
```

return the result shown:

```
                C             1 row       1 col     (numeric)

                                          4
```

YIELD Function

calculates yield-to-maturity of a cash-flow stream and returns a scalar

YIELD($times, flows, freq, value$)

The YIELD function returns a scaler containing yield-to-maturity of a cash-flow stream based on frequency and value specified.

times is an n-dimensional column vector of times.
Elements should be non-negative.

flows is an n-dimensional column vector of cash-flows.

freq is a scalar which represents the base of the rates
to be used for discounting the cash-flows.
If positive, it represents discrete compounding
as the reciprocal of the number of compoundings.
If zero, it represents continuous compounding.
No negative values are allowed.

value is a scalar which is the discounted present value of the cash-flows.

The present value relationship can be written as

$$P = \sum_{k=1}^{K} c(k)D(t_k)$$

where P is the present value of the asset, $\{c(k)\}k = 1, ..K$ is the sequence of cash-flows from the asset, t_k is the time to the k-th cash-flow in periods from the present, and $D(t)$ is the discount function for time t.
With continuous compunding:

$$D(t) = e^{-yt}$$

With discrete compunding:

$$D(t) = (1 + fy)^{-(t/f)}$$

where $f > 0$ is the frequency, the reciprocal of the number of compoundings per unit time period and y is the yield-to-maturity. The $YIELD$ function solves for y.

Example proc iml;
timesn=do(1,100,1);
timesn=T(timesn);
flows=repeat(10,100);
freq=0;
do while(freq<50);
freq=freq+.25;
end;
value=682.31027;
yield=yield(timesn,flows,freq,value);
print yield;
quit;

YIELD
0.0100001

References

- Al-Baali, M. and Fletcher, R. (1985), "Variational Methods for Nonlinear Least Squares", *J. Oper. Res. Soc.*, **36**, 405 –421.
- Al-Baali, M. and Fletcher, R. (1986), "An Efficient Line Search for Nonlinear Least Squares", *J. Optimiz. Theory Appl.*, **48**, 359 –377.
- Ansley, C. (1979), "An Algorithm for the Exact Likelihood of a Mixed Autoregressive-Moving Average Process," *Biometrika*, 66, 59 –65.

- Bates, D., Lindstrom, M., Wahba, G. and Yandell, B. (1987), " GCVPACK-Routines for generalized Cross Validation," *Comm. Statist. B-Simulation Comput.*, 16, 263 –297.
- Barrodale, I. and Roberts, F.D.K. (1974), "Algorithm 478: Solution of an overdetermined system of equations in the l1-norm", *Communications ACM*, 17, 319 –320.
- Beale, E.M.L. (1972), "A Derivation of Conjugate Gradients", in *Numerical Methods for Nonlinear Optimization*, ed. F. A. Lootsma (ed.), London: Academic Press.
- Bishop, Y.M., Fienberg, S.E., and Holland, P.W. (1975), *Discrete Multivariate Analysis: Theory and Practice*, Cambridge, MA: MIT Press.
- Box, G.E.P. and Jenkins, G.M. (1976), *Time Series Analysis: Forecasting and Control*, Oakland, CA: Holden-Day.
- Bishop, Y.M., Fienberg, S.E., and Holland, P.W. (1975), *Discrete Multivariate Analysis: Theory and Practice*, Cambridge, MA: MIT Press.
- Charnes, A., Frome, E.L., and Yu, P.L. (1976), "The Equivalence of Generalized Least Squares and Maximum Likelihood Estimation in the Exponential Family," *Journal of the American Statistical Association*, 71, 169 –172.
- Cox, D.R. and Hinkley, D.V. (1974), *Theoretical Statistics*, London: Chapman and Hall.
- De Jong, P. (1991b), "Stable algorithms for the state space model," it Journal of Time Series Analysis, 12, 143 –157.
- Dennis, J.E., Gay, D.M., and Welsch, R.E. (1981), "An Adaptive Nonlinear Least-Squares Algorithm", *ACM Trans. Math. Software*, **7**, 348 –368.
- Dennis, J.E. and Mei, H.H.W. (1979), "Two new unconstrained optimization algorithms which use function and gradient values", *J. Optim. Theory Appl.*, **28**, 453 –482.
- Duchon, J. (1976), "Fonctions-Spline et Esperances Conditionnelles de Champs Guassiens," in *Ann. Sci. Univ. Clermont Ferrand II Math*, 14, 19 –27.
- Eskow, E. and Schnabel, R.B. (1991), "Algorithm 695: Software for a New Modified Cholesky Factorization", *ACM Trans. Math. Software*, 17, 306 –312.
- Fletcher, R. (1987), *Practical Methods of Optimization*, 2nd Ed., Chichester: John Wiley & Sons.
- Fletcher, R. and Xu, C. (1987), "Hybrid Methods for Nonlinear Least Squares", *Journal of Numerical Analysis*, 7, 371 –389.
- Forsythe, G.E., Malcom, M.A., and Moler, C.B. (1967), *Computer Solution of Linear Algebraic Systems*, Chapter 17, Englewood Cliffs, NJ: Prentice-Hall, Inc.

- Gay, D.M. (1983), "Subroutines for Unconstrained Minimization", *ACM Trans. Math. Software*, 9, 503 –524.
- Golub, G.H., and Van Loan, C.F. (1989), *Matrix Computations*, 2nd Ed., Baltimore: J. Hopkins Univerity Press.
- Gonin, R. and Money, A.H. (1989), *Nonlinear L_p-norm Estimation*, New York: M. Dekker, Inc.
- Graybill, F.A. (1969), *Introduction to Matrices with Applications in Statistics*, Belmont, CA: Wadsworth, Inc.
- Gentleman, W.M. and Sande, G. (1966), "Fast Fourier Transforms-for Fun and Profit," *AFIPS Proceedings of the Fall Joint Computer Conference*, 19, 563 –578.
- George, J.A. and Liu, J.W. (1981), *Computer Solution of Large Sparse Positive Definite Systems*, New Jersey: Prentice-Hall.
- Gill, E.P., Murray, W., Saunders, M.A., and Wright, M.H. (1984), "Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints", *ACM Trans. Math. Software*, 10, 282 –298.
- Goodnight, J.H. (1979) "A Tutorial on the SWEEP Operator," *The American Statistician*, 33, 149 –158.
- Grizzle, J.E., Starmer, C.F., and Koch, G.G. (1969), "Analysis of Catagorical Data by Linear Models," *Biometrics*, 25, 489 –504.
- Hadley, G. (1963), *Linear Programming*, Reading, MA: Addison-Wesley Publishing Company, Inc.
- Harvey, A.C. (1989), *Forecasting, Structural Time Series Models and the Kalman Filter*, Cambridge: Cambridge University Press.
- Jenkins, M.A. and Traub, J.F. (1970), " A Three-Stage Algorithm for Real Polynomials Using Quadratic Iteration," *SIAM Journal of Numerical Analysis*, 7, 545 –566.
- Jenrich, R.I. and Moore R.H. (1975), "Maximum Likelihood Estimation by Means of Nonlinear Least Squares," *American Statistical Association, 1975 Proceedings of the Statistical Computing Section*, 57 –65.
- Kastenbaum, M.A. and Lamphiear, D.E. (1959), " Calculation of Chi-Square to Test the No Three-Factor Interaction Hypothesis," *Biometrics*, 15, 107 –122.
- Kaiser, H.F. and Caffrey, J. (1965), "Alpha Factor Analysis," *Psychometrika*, 30, 1 –14.
- Kruskal, J.B. (1964), "Nonmetric Multidimensional Scaling," *Psychometrika*, 29, 1-27, 115 –129.
- Nelder, J.A. and Wedderburn, R.W.M. (1972), "Generalized Linear Models," *Journal of the Royal Statistical Society*, A.3, 370.

- Lee, W. and Gentle, J.E. (1986), "The LAV Procedure", *SUGI Supllememtal Library User's Guide*, Cary: SAS Institute, Chapter 21, pp. 257 –260.
- Lindström, P. and Wedin, P.A. (1984), "A new linesearch algorithm for nonlinear least-squares problems", *Mathematical Programming*, **29**, 268 –296.
- Madsen, K. and Nielsen, H.B. (1993), "A finite smoothing algorithm for linear l1 estimation", *SIAM Journal on Optimization*, **3**, 223 –235.
- McKean, J.W. and Schrader, R.M. (1987), "Least absolute errors analysis of variance", In: Y. Dodge, ed. *Statistical Data Analysis - Based on L_1 Norm and Related Methods*, Amsterdam: North Holland, 297 –305.
- McLeod, I. (1975), "Derivation of the Theoretical Autocovariance Function of Autoregressive-Moving Average Time Series," *Applied Statistics*, 24, 255 –256.
- Monro, D.M. and Branch, J.L. (1976), "Algorithm AS 117. The Chirp Discrete Fourier Transform and General Length," *Applied Statistics*, 26, 351 –361.
- Moré, J.J. (1978), "The Levenberg-Marquardt Algorithm: Implementation and Theory", in *Lecture Notes in Mathematics 630*, ed. G.A. Watson, Springer Verlag, Berlin-Heidelberg-New York, 105 –116.
- Moré, J.J. and Sorensen, D.C. (1983), "Computing a Trust-Region Step", *SIAM J. Sci. Stat. Comput.*, **4**, 553 –572.
- Nussbaumer, H.J. (1982), *Fast Fourier Transform and Convolution Algorithms*, Second Edition, New York: Springer-Verlag.
- Pizer, S.M. (1975), *Numerical Computing and Mathematical Analysis*, Chicago: Science Research Associates, Inc.
- Powell, J.M.D. (1977), "Restart Procedures for the Conjugate Gradient Method", *Mathematical Programming*, **12**, 241 –254.
- Powell, J.M.D. (1978a), "A fast algorithm for nonlinearly constraint optimization calculations", in *Numerical Analysis, Dundee 1977, Lecture Notes in Mathematics 630*, ed. G.A. Watson, Springer Verlag, Berlin, 144 –175.
- Powell, J.M.D. (1982b), "VMCWD: A Fortran subroutine for constrained optimization", *DAMTP 1982/NA4*, Cambridge, England.
- Rao, C.R. and Mitra, S.K. (1971), *Generalized Inverse of Matrices and its Applications*, New York: John Wiley & Sons, Inc.
- Reinsch, Christian H. (1967), "Smoothing by Spline Functions," *Numerische Mahematik*, 10, 177 –183.
- Singleton, R.C. (1969), "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Transactions on Audio and Electoacoustics*, AU-17, 93 –103.

- Stoer, J. and Bulirsch, R. (1980), *Introduction to Numerical Analysis*, New York: Springer-Verlag.
- Wahba, G. (1990), *Spline Models for Observational Data*, Philadelphia: Society for Industrial and Applied Mathematics.
- Wilkinson, J.H. and Reinsch, C. (eds.), (1971), *Linear Algebra, Handbook for Automatic Computation*, Volume 2, New York: Sprringer-Verlag.
- Woodfield, Terry J. (1988), "Simulating Stationary Gaussian ARMA Time Series," *Computer Science and Statistics: Proceedings of the 20th Symposium on the Interface*, 612 –617.
- Young, F.W. (1981), "Quantitative Analysis of Qualitative Data," *Psychometrika*, 46, 357 –388.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/IML User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. 846 pp.

SAS/IML User's Guide, Version 8

ISBN 1–58025–553–1

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999