Chapter 23

Language Reference

Contents

Overview

This chapter describes all operators, statements, functions, and subroutines that can be used in SAS/IML software. This chapter is divided into the following sections:

- The first section list all statements, functions, and subroutines available in SAS/IML software, grouped by functionality.
- The second section contains operator descriptions, ordered alphabetically by the name of the operator.
- The third section contains descriptions of statements, functions, and subroutines ordered alphabetically by name.

Statements, Functions, and Subroutines by Category

Mathematical Functions

ALLCOMB	generates all combinations of n elements taken k at a time
ALLPERM	generates all permutations of n elements
ABS function	computes the absolute value

EXP function	applies the exponential function
INT function	truncates a value
LOG function	computes the natural logarithm
MOD function	computes the modulo (remainder)
SQRT function	computes the square root
RANCOMB	returns random combinations of n elements taken k at a time
RANPERM	returns random permutations of n elements

You can also call any function in Base SAS software, such as those documented in the following sections:

- "Mathematical Functions and Subroutines" on page 1054
- "Probability Functions" on page 1055
- "Quantile Functions" on page 1055
- "Trigonometric and Hyperbolic Functions" on page 1056
- "Truncation Functions" on page 1057

Reduction Functions

MAX function	finds the maximum value of a matrix
MIN function	finds the smallest element of a matrix
PROD function	multiplies all elements
SSQ function	computes 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
COUNTMISS function	returns the number of missing values
COUNTN function	returns the number of nonmissing values
COUNTUNIQUE function	returns the number of unique values
CHOOSE function	conditionally chooses and changes elements
LOC function	finds indices for the 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 Sorting and BY-Group Processing Functions

SORT call	sorts a matrix by specified columns
SORTNDX call	creates a sorted index for a matrix
UNIQUEBY function	finds locations of unique BY groups in a sorted or indexed matrix

Matrix Reshaping Functions

BLOCK function	forms block-diagonal matrices
BTRAN function	computes a block transpose
DIAG function	creates a diagonal matrix
DO function	produces an arithmetic series
FULL function	converts a matrix stored in a sparse format into a full (dense) matrix
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
SHAPECOL function	reshapes and repeats values by columns
SPARSE function	converts a matrix that contains many zeros into a matrix stored in a sparse format
SQRSYM function	converts a symmetric matrix to a square matrix
SQRVECH function	converts a symmetric matrix which is stored columnwise to a square matrix
SYMSQR function	converts a square matrix to a symmetric matrix
T function	transposes a matrix
VECH function	creates a vector from the columns of the lower triangular elements of a matrix
VECDIAG function	creates a vector from a diagonal

Character Manipulation Functions

BYTE function	translates numbers to ordinal characters
CHANGE call	replaces text
CHAR function	produces a character representation of a matrix
CONCAT function	concatenates elementwise strings
CSHAPE function	reshapes and repeats character values
LENGTH function	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 by using blank compression
SUBSTR function	takes substrings of matrix elements

You can also call functions in Base SAS software such as those documented in "Character and Formatting Functions" on page 1045 and "Character String Matching Functions and Subroutines" on page 1049.

Functions for Random Number Generation

NORMAL function	generates a pseudorandom normal deviate
RANDGEN call	generates random numbers from specified distributions
RANDSEED call	initializes seed for subsequent RANDGEN calls
UNIFORM function	generates pseudorandom uniform deviates

You can also call functions in Base SAS software such as those documented in "Random Number Functions and Subroutines" on page 1056.

Statistical Functions

BIN function	divides numeric values into a set of disjoint intervals
BRANKS function	computes bivariate ranks
CORR function	computes correlation statistics
COUNTMISS function	counts the number of missing values
COUNTN function	counts the number of nonmissing values
COUNTUNIQUE function	returns the number of unique values
COV function	computes a sample variance-covariance matrix
CUSUM function	computes cumulative sums
CUPROD function	computes cumulative products
DESIGN function	creates a design matrix
DESIGNF function	creates a full-rank design matrix
GEOMEAN function	computes geometric means
HADAMARD function	creates a Hadamard matrix
HARMEAN function	computes harmonic means
IPF call	performs an iterative proportional fit of a contingency table
LAV call	performs linear least absolute value regression by solving the L_1 norm minimization problem
LMS call	performs robust least median of squares (LMS) regression

LTS call	performs robust least trimmed squares (LTS) regression
MAD function	finds the univariate (scaled) median absolute deviation
MARG call	evaluates marginal totals in a multiway contingency table
MAXQFORM call	computes the subsets of a matrix system that maximize the quadratic form
MCD call	finds the minimum covariance determinant estimator
MEAN function	computes sample means
MVE call	finds the minimum volume ellipsoid estimator
OPSCAL function	rescales qualitative data to be a least squared fit to qualitative data
QNTL call	computes sample quantiles (percentiles)
RANGE function	returns the range of values for a set of matrices.
RANK function	ranks elements of a matrix, breaking ties arbitrarily
RANKTIE function	ranks elements of a matrix
SEQ call	performs discrete sequential tests
SEQSCALE call	performs estimates of scales associated with discrete sequential tests
SEQSHIFT call	performs estimates of means associated with discrete sequential tests
STD function	computes a sample standard deviation
TABULATE call	counts the number of unique values in a vector
SWEEP function	sweeps a matrix
VAR function	computes a sample variance

You can also call functions in Base SAS software such as those documented in "Descriptive Statistics Functions and Subroutines" on page 1050.

Time Series Functions

ARMACOV call	computes an autocovariance sequence for an autoregressive moving average (ARMA) model
ARMALIK call	computes the log likelihood and residuals for an ARMA model
ARMASIM function	simulates an ARMA series
CONVEXIT function	computes convexity of a noncontingent cash flow
COVLAG function	computes autocovariance estimates for a vector time series
DIF function	computes the difference between a value and a lagged value
DURATION function	computes modified duration of a noncontingent cash flow
FARMACOV call	computes the autocovariance function for an autoregressive fractionally integrated moving average (ARFIMA) model of the form ARFIMA(p, d, q)
FARMAFIT call	estimates the parameters of an ARFIMA(p, d, q) model

FARMALIK call	computes the log-likelihood function of an ARFIMA(p, d, q) model
FARMASIM call	generates an ARFIMA(p, d, q) process
FDIF call	computes a fractionally differenced process
FORWARD function	computes forward rates
KALCVF call	computes the one-step prediction $z_{t+1\|t}$ and the filtered estimate $z_{t\|t}$, in addition to 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 $z_{t\|T}$ and its covariance matrix, $P_{t\|T}$, where T is the number of observations in the complete data set
KALDFF call	computes the one-step forecast of state vectors in a state space model (SSM) by using the diffuse Kalman filter. The call estimates the conditional expectation of 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 squares error matrix from the one-step forecast and mean squares error matrix computed by the KALDFF subroutine.
LAG function	computes lagged values
PV function	computes the present value
RATES function	converts interest rates from one base to another
SPOT function	computes spot rates
TSBAYSEA call	performs Bayesian seasonal adjustment modeling
TSDECOMP call	analyzes nonstationary time series by using smoothness priors modeling
TSMLOCAR call	analyzes nonstationary or locally stationary time series by using a method that minimizes Akaike's information criterion (AIC)
TSMLOMAR call	analyzes nonstationary or locally stationary multivariate time series by using a method that minimizes Akaike's information criterion (AIC)
TSMULMAR call	estimates vector autoregressive (VAR) processes by minimizing the AIC
TSPEARS call	analyzes periodic autoregressive (AR) models by minimizing the AIC
TSPRED call	provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are given
TSROOT call	computes AR and moving average (MA) coefficients from the characteristic roots of the model, or computes 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 by minimizing the AIC, and estimates the AR coefficients
VARMACOV call	computes the theoretical cross-covariance matrices for a stationary vector autoregressive moving average (VARMA(p, q)) model
VARMALIK call	computes the log-likelihood function for a 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 a VARMA(p, q) model
YIELD function	computes yield-to-maturity of a cash-flow stream

You can also call functions in Base SAS software such as those documented in "Financial Functions" on page 1053.

Numerical Analysis Functions

BSPLINE function	computes a B-spline basis
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 first-order vector differential equations with initial boundary conditions
ORPOL function	generates orthogonal polynomials on a discrete set of data
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	fits a cubic spline to data
SPLINEC call	fits a cubic spline to data and returns the spline coefficients
SPLINEV function	evaluates a cubic spline at new data points
TPSPLINE call	computes thin-plate smoothing splines
TPSPLNEV call	evaluates the thin-plate smoothing spline at new data points

Linear Algebra functions

APPCORT call	computes a complete orthogonal decomposition
COMPORT call	computes a 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
EIGVAL function	computes eigenvalues
EIGVEC function	computes eigenvectors
GENEIG call	computes eigenvalues and eigenvectors of a generalized eigenproblem
GINV function	computes a generalized inverse
GSORTH call	computes the Gram-Schmidt orthonormalization
HALF function	computes the 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	computes the inverse
INVUPDT function	updates a matrix inverse
ITSOLVER call	solves a sparse general linear system by iteration
LUPDT call	provides updating and downdating for rank-deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses
QR call	computes the QR decomposition of a matrix by Householder transformations
RDODT call	downdates and updates QR and Cholesky decompositions
ROOT function	performs the Cholesky decomposition of a matrix
RUPDT call	updates QR and Cholesky decompositions
RZLIND call	updates QR and Cholesky decompositions
SOLVE function	solves a system of linear equations
SOLVELIN call	solves a sparse symmetric system of linear equations by direct decomposition
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 extended-precision matrix multiplication

Optimization Subroutines

LCP call	solves the linear complementarity problem
LP call	solves the linear programming problem
NLPCG call	performs nonlinear optimization by conjugate gradient method
NLPDD call	performs nonlinear optimization by double-dogleg method
NLPFDD call	approximates derivatives by finite-differences method

NLPFEA call	computes feasible points subject to constraints
NLPHQN call	computes hybrid quasi-Newton least squares
NLPLM call	computes Levenberg-Marquardt least squares
NLPNMS call	performs nonlinear optimization by Nelder-Mead simplex method
NLPNRA call	performs nonlinear optimization by Newton-Raphson method
NLPNRR call	performs nonlinear optimization by Newton-Raphson ridge method
NLPQN call	performs nonlinear optimization by quasi-Newton method
NLPQUA call	performs nonlinear optimization by quadratic method
NLPTR call	performs nonlinear optimization by trust-region method
Nonlinear optimization and related subroutines	lists the nonlinear optimization and related subroutines in SAS/IML software

Set functions

ELEMENT function	finds elements that are contained in a set
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 PROC IML
APPLY function	applies a module to arguments
CALL statement	calls a subroutine or function
DO statement	groups statements as a unit
DO, iterative statement	iteratively executes a DO group
DO UNTIL statement	iteratively executes statements until a condition is satisfied
DO WHILE statement	iteratively executes statements while a condition is satisfied
END statement	ends a DO loop or DO statement
EXECUTE call	executes statements at run time
FINISH statement	denotes the end of a module
FREE statement	frees matrix storage space
GOTO statement	jumps to a new statement
IF-THEN/ELSE statement	conditionally executes statement
LINK statement	jumps to another statement

MATTRIB statement	associates printing attributes with matrices
PAUSE statement	interrupts module execution
PRINT statement	prints matrix values
PURGE statement	removes observations marked for deletion and renumbers records
PUSH call	pushes statements to the beginning of the command input stream
QUEUE call	queues statements at the end of the command input stream
QUIT statement	exits from PROC 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 statement	defines 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

Data Set and File Functions

APPEND statement	adds observations to SAS data set
CLOSE statement	closes a SAS data set
CLOSEFILE statement	closes a file
CONTENTS function	returns the variables in a SAS data set
CREATE statement	creates a new SAS data set
DATASETS function	obtains the names of SAS data sets
DELETE call	deletes a SAS data set
DELETE statement	marks observations in a data set 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
FORCE statement	is an alias for the SAVE statement

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	computes 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 call	places text horizontally on a graph
GVTEXT call	places text vertically on a graph
GWINDOW call	defines the data window
GXAXIS call	draws a horizontal axis
GYAXIS call	draws a vertical axis
PGRAF call	produces scatter plots
ODSGRAPH call	renders a graph by using ODS Statistical Graphics
WINDOW statement	opens a display window

Wavelet Analysis functions

WAVFT call	computes a wavelet transform of one dimensional data
WAVGET call	returns requested information about a wavelet transform
WAVIFT call	inverts a wavelet transform after applying thresholding to the detail coefficients
WAVPRINT call	displays information about a wavelet transform
WAVTHRSH call	applies specified thresholding to the detail coefficients of a wavelet transform

Genetic Algorithm functions

GAEND call	terminates a genetic algorithm and frees memory resources
GAGETMEM call	gets requested members and objective values from the current solution population
GAGETVAL call	gets objective function values for a requested member of current solution population
GAINIT call	initializes the initial solution population
GAREEVAL call	reevaluates the objective function for all solutions in the current population
GASETCRO call	specifies a current crossover operator
GASETMUT call	specifies a current mutation operator
GASETOBJ call	specifies a current objective function
GASETSEL call	specifies a current selection parameters
GASETUP function	sets up a specific genetic algorithm optimization problem

Calling External Modules

MODULEI call	calls an external routine that has no return code
MODULEIC call	calls an external routine that returns a character
MODULEIN call	calls an external routine that returns a numeric value

Calling SAS statements or R Functions

SUBMIT statement	calls SAS procedures, DATA steps, or macros. You can also use the R option to call functions in the R language.
ENDSUBMIT statement	defines a block of submitted statements. All statements between the SUBMIT and ENDSUBMIT statements are sent to the SAS System or R for processing.
EXPORTDATASETTOR call	transfers data from a SAS data set into an R data frame
EXPORTMATRIXTOR call	transfers data from a SAS/IML matrix into an R matrix
IMPORTDATASETFROMR call	transfers data from a matrix or data frame into a SAS data set
IMPORTMATRIXFROMR call	transfers data from a matrix or data frame into a SAS/IML matrix

Operators

This section describes all operators that are available in SAS/IML software. Each section shows how the operator is used, followed by a description of the operator.

Addition Operator: +

matrix1 + *matrix2* ;

matrix + *scalar* ;

matrix + *vector* ;

The addition operator (+) computes a new matrix that contains elements that are the sums of the corresponding elements of *matrix1* and *matrix2*. If *matrix1* and *matrix2* are both $n \times p$ matrices, then the addition operator adds the element in the ith row and jth column of the first matrix to the element in the ith row and jth column of the second matrix, for $i = 1 \ldots n, j = 1 \ldots p$.

For example, the following statements add two matrices and store the result in the matrix **c**, shown in Figure 23.1:

```
a = {1 2,
```

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

Figure 23.1 Sum of Two Matrices

c	
2	3
4	5

You can also use the addition operator to conveniently add a value to each element of a matrix, to each column of a matrix, or to each row of a matrix.

- When you use the *matrix* + *scalar* form, the scalar value is added to each element of the matrix.
- When you use the *matrix* + *vector* form, the vector is added to each row or column of the $n \times p$ matrix.
  - If you add an $n \times 1$ column vector, each row of the vector is added to each row of the matrix.
  - If you add a $1 \times p$ row vector, each column of the vector is added to each column of the matrix.

For example, you can obtain the same result as the previous example with any of the following statements:

```
c = a+1;
c = a+{1 1};
c = a+{1,1};
```

When an element of a matrix contains a missing value, the corresponding element of the sum is also a missing value.

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

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

matrix1 < *matrix2* ;

matrix1 <= *matrix2* ;

matrix1 > *matrix2* ;

matrix1 >= *matrix2* ;

matrix1 = *matrix2* ;

matrix1 ^= *matrix2* ;

Comparison operators compare two matrices element by element and compute 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 the SAS DATA step, the SAS/IML language does not accept the English equivalents GT and LT for the greater than and less than operators.

For example, the following statements assign the matrix **c**, shown in Figure 23.2:

```
a = {1 7 3,
     6 2 4};
b = {0 8 2,
     4 1 3};
c = a>b;
print c;
```

Figure 23.2 Results of a Matrix Comparison

c		
1	0	1
1	1	1

You can also use the comparison operators to conveniently compare all elements of a matrix with a scalar.

- If either argument is a scalar, then an elementwise comparison is performed between each element of the matrix and the scalar.
- You can also compare an $n \times p$ matrix with a row or column vector.
  - If the comparison is with an $n \times 1$ column vector, each row of the vector is compared to each row of the matrix.
  - If the comparison is with a $1 \times p$ row vector, each column of the vector is compared to each column of the matrix.

For example, the following statements assign the matrix **d**, shown in Figure 23.3:

```
d = (a>=4);       /* the parentheses are not necessary */
print d;
```

Figure 23.3 Results of a Comparison with a Scalar

d		
0	1	0
1	0	1

When you are making conditional comparisons, all values of the result must be nonzero for the condition to be evaluated as true, as shown in the following statements:

```
if a>=b then do;
   /* more statements */
end;
```

The previous DO block is executed only if *every* element of **a** is greater than or equal to the corresponding element in **b**. For the **a** and **b** matrices defined in this section, the DO block is not executed. See the descriptions of the ALL and ANY functions.

If a numeric missing value occurs in a matrix, the inequality comparison operators treat it as a value that is less than any valid nonmissing value.

You can compare elements of a character matrix. Character values are compared in ASCII order. In ASCII order, numerals precede uppercase letters, which precede lowercase letters. If the element lengths of two character matrices are different, the shorter elements are padded on the right with blanks for the comparison.

Concatenation Operator, Horizontal: ||

matrix1 || *matrix2* **;**

The horizontal concatenation operator (||) produces a new matrix by horizontally joining *matrix1* and *matrix2*. The matrices 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 following statements produce the matrix **c**, shown in Figure 23.4:

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

Figure 23.4 Result of Horizontal Concatenation

c					
1	1	1	0	0	0
7	7	7	8	8	8

For character operands, the element size in the result matrix is the larger of the two operands. For example, the following statements produce a matrix **f** which has elements of size 2, which are shown in Figure 23.5:

```
d = {A B C,
     D E F};
e = {"GH" "IJ",
     "KL" "MN"};
f = d||e;
print f;
```

Figure 23.5 Result of Horizontal Concatenation of Character Matrices

```
                                f

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

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

Concatenation Operator, Vertical: //

matrix1 // *matrix2* **;**

The vertical concatenation operator (//) produces a new matrix by vertically joining *matrix1* and *matrix2*. The matrices must have the same number of columns, which is also the number of columns in the new matrix. The number of rows in the new matrix is the number of rows in *matrix1* plus the number of rows in *matrix2*.

For example, the following statements produce the matrix **c**, shown in Figure 23.6:

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

Figure 23.6 Result of Vertical Concatenation

```
                                c

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

For character matrices, the element size of the result matrix is the larger of the element sizes of the two operands, as shown in Figure 23.7:

```
d = {"AB" "CD",
     "EF" "GH"};
e = {"I" "J",
     "K" "L",
     "M" "N"};
f = d//e;
print f;
```

Figure 23.7 Result of Vertical Concatenation

```
f

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

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

Direct Product Operator: @

matrix1 **@** *matrix2* **;**

The direct product operator (@) computes a new matrix that is the direct product (also called the *Kronecker product*) of *matrix1* and *matrix2*. For matrices **A** and **B**, the direct product is 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*.

Specifically, if **A** is an $n \times p$ matrix and **B** is a $m \times q$ matrix, then the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the following $nm \times pq$ block matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}B & \cdots & A_{1p}B \\ \vdots & \ddots & \vdots \\ A_{n1}B & \cdots & A_{np}B \end{bmatrix}$$

For example, the following statements compute the matrices **c** and **d**, which are shown in Figure 23.8:

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

Figure 23.8 Results of Direct Product Computation

```
c

0  2  0  4
0  6  0  8
```

Figure 23.8 *continued*

```
                    d

            0       0       2       4
            0       0       6       8
```

Notice that the direct product of two matrices is not commutative.

The direct product is used in several areas of statistics. For example, in complete balanced designs the sums of squares and the covariance matrices can be expressed in terms of direct products (Hocking 1985).

Division Operator: /

matrix1 / *matrix2* ;

matrix / *scalar* ;

matrix / *vector* ;

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

You can also use the division operator to conveniently divide all elements of a matrix, each column of a matrix, or each row of a matrix.

- When you use the *matrix* / *scalar* form, each element of the matrix is divided by the scalar value.
- When you use the *matrix* / *vector* form, each row or column of the $n \times p$ matrix is divided by a corresponding element of the vector.
  - If you divide by an $n \times 1$ column vector, each row of the matrix is divided by the corresponding row of the vector.
  - If you divide by a $1 \times p$ row vector, each column of the matrix is divided by the corresponding column of the vector.

When an element of a matrix contains a missing value, the corresponding element of the quotient is also a missing value.

If a divisor is zero, the operation displays a warning and assigns a missing value for the corresponding element in the result.

The following statements compute the matrices `c` and `d`, shown in Figure 23.9:

```
a = {1 2,
     3 4};
b = {5 6,
     7 8};
```

```
c = a/b;
d = a/4;
print c, d;
```

Figure 23.9 Results of Division

```
                                c

                              0.2 0.3333333
                        0.4285714       0.5

                                d

                             0.25       0.5
                             0.75         1
```

Element Maximum Operator: <>

matrix1 <> *matrix2* ;

matrix <> *scalar* ;

matrix <> *vector* ;

The element maximum operator (<>) compares each element of *matrix1* to the corresponding element of *matrix2*. The two matrices must be conformable. The operator computes a new matrix that contains the larger of the two values that are being compared.

- If either argument is a scalar, then an elementwise comparison is performed between each element of the matrix and the scalar.
- You can also compare a matrix with a row or column vector, in which case the comparison is performed between the vector and each row or column of the $n \times p$ matrix.
  - If you compare with an $n \times 1$ column vector, each row of the matrix is compared with the corresponding row of the vector.
  - If you compare with a $1 \times p$ row vector, each column of the matrix is compared with the corresponding column of the vector.

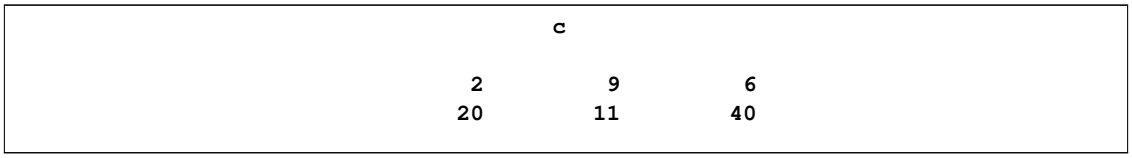
If a numeric missing value occurs in a matrix, the operator treats it as a value that is less than any valid nonmissing value.

The element maximum operator can take as operands two character matrices or a character matrix and a character string. Character values are compared in ASCII order. In ASCII order, numerals precede uppercase letters, which precede lowercase letters. If the element lengths of character 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.

For example, the following statements compute the matrix `c`, shown in Figure 23.10:

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

Figure 23.10 Maximum Elements



c

2	9	6
20	11	40

Element Minimum Operator: ><

matrix1 >< *matrix2* ;

matrix1 >< *scalar* ;

matrix1 >< *vector* ;

The element minimum operator (><) compares each element of *matrix1* with the corresponding element of *matrix2*. The two matrices must be conformable. The operator computes a new matrix that contains the smaller of the two values that are being compared.

- If either argument is a scalar, then an elementwise comparison is performed between each element of the matrix and the scalar.
- You can also compare a matrix with a row or column vector, in which case the comparison is performed between the vector and each row or column of the $n \times p$ matrix.
  - If you compare with an $n \times 1$ column vector, each row of the matrix is compared with the corresponding row of the vector.
  - If you compare with a $1 \times p$ row vector, each column of the matrix is compared with the corresponding column of the vector.

If a numeric missing value occurs in a matrix, the operator treats it as a value that is less than any valid nonmissing value.

The element minimum operator can take as operands two character matrices or a character matrix and a character string. Character values are compared in ASCII order. In ASCII order, numerals precede uppercase letters, which precede lowercase letters. If the element lengths of character 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.

For example, the following statements compute the matrix `c`, shown in Figure 23.10:

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

Figure 23.11 Minimum Elements

```
                                c

                          1         4         2
                         10        10        12
```

Index Creation Operator: :

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, until the last element which is less than or equal to *value2*.

For example, the following statement creates the vector **s** which contains consecutive integers, shown in Figure 23.12:

```
s = 7:10;
print s;
```

Figure 23.12 Increasing Sequence

```
                                s

                     7         8         9        10
```

If *value1* is greater than *value2*, a reverse-order index is created. For example, the following statement creates the vector **r** which contains a decreasing sequence of integers, shown in Figure 23.13:

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

Figure 23.13 Decreasing Sequence

```
                                r

                    10         9         8         7         6
```

Neither *value1* nor *value2* is required to be an integer. Use the DO function if you want an increment other than 1 or −1.

The index creation operator also works on character arguments with a numeric suffix. For example, the following statements create a sequence of values that begin with the prefix "var", shown in Figure 23.14:

```
varList = "var1":"var5";
print varList;
```

Figure 23.14 Sequence of Character Values

```
                                varList

                          var1 var2 var3 var4 var5
```

Sequences of character values are often used to assign names to variables. You can use the string concatenation operator to dynamically determine the length of a sequence, as shown in the following statements:

```
x = {1 2 3 4,
     5 6 7 8,
     7 6 5 4};
numVar = ncol(x);                          /* 4 columns */
varNames = "X1":"X"+strip(char(numVar));   /* "X1":"X4" */
print x[colname=varNames];
```

Figure 23.15 Sequence of Variable Names

```
                                x
                  X1        X2        X3        X4

                   1         2         3         4
                   5         6         7         8
                   7         6         5         4
```

Logical Operators: &, |, ^

matrix1 **&** *matrix2* **;**

matrix **&** *scalar* **;**

matrix **&** *vector* **;**

matrix1 **|** *matrix2* **;**

matrix **|** *scalar* **;**

matrix **|** *vector* **;**

^*matrix* **;**

The logical operators compare two matrices element by element and create a new matrix. For logical comparisons, a missing value is handled as if it is a zero value. That is, in the text that follows in this section, "nonzero" really means "nonzero and nonmissing."

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

An element of the new matrix computed by the AND logical operator (&) is 1 if the corresponding elements of *matrix1* and *matrix2* are both nonzero; otherwise, it is zero.

If either operand is a scalar, the OR and AND operators perform a logical comparison between each element and the scalar value. If either operand is a row or column vector, then the operation is performed by using that vector on each of the rows or columns of the matrix.

The NOT prefix operator (^) examines each element of a matrix and computes a new matrix that contains elements that are ones and zeros. If an element of *matrix* is zero or missing, 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. The results are shown in Figure 23.16:

```
x = {0 1 0 1 . .};
y = {1 1 0 0 1 0};
u = x|y;
v = x&y;
w = ^x;
print u, v, w;
```

Figure 23.16 Results of Logical Comparisons

```
                              u

                1             1             0             1             1             0

                              v

                0             1             0             0             0             0

                              w

                1             0             1             0             1             1
```

Multiplication Operator, Elementwise:

matrix1 # matrix2 ;

matrix # scalar ;

matrix # vector ;

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

For example, the following statements compute the matrix **ab**, shown in Figure 23.17:

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

Figure 23.17 Results of Elementwise Multiplication

```
                                ab

                              4        16
                              0        20
```

In addition to multiplying matrices that have the same dimensions, you can use the elementwise multiplication operator to multiply a matrix and a scalar.

- When either argument is a scalar, each element in *matrix* is multiplied by the scalar value.
- When you use the *matrix # vector* form, each row or column of the $n \times p$ matrix is multiplied by a corresponding element of the vector.
  - If you multiply by an $n \times 1$ column vector, each row of the matrix is multiplied by the corresponding row of the vector.
  - If you multiply by a $1 \times p$ row vector, each column of the matrix is multiplied by the corresponding column of the vector.

For example, a 2×3 matrix can be multiplied on either side by a 2×3, 1×3, 2×1, or 1×1 matrix. The following statements multiply the 2×2 matrix **a** by a column vector and a row vector. The results are shown in Figure 23.18.

```
c = {10, 100};          /* column vector */
r = {10 100};           /* row vector    */
ac = a#c;
ar = a#r;
print ac, ar;
```

Figure 23.18 Elementwise Multiplication with Vectors

```
                                ac

                             10        20
                            300       400

                                ar

                             10       200
                             30       400
```

Elementwise multiplication is also known as the Schur or Hadamard product. Elementwise multiplication (which uses the # operator) should not be confused with matrix multiplication (which uses the * operator).

When an element of a matrix contains a missing value, the corresponding element of the product is also a missing value.

Multiplication Operator, Matrix: *

*matrix1 * matrix2* ;

The matrix multiplication operator (*) computes 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. That is, if $\boldsymbol{A}$ is an $n \times p$ matrix and $\boldsymbol{B}$ is a $p \times m$ matrix, then the product $A * B$ is an $n \times m$ matrix. The ij th element of the product is the sum $\sum_{k=1}^{p} A_{ik} B_{kj}$.

The matrix multiplication operator does not support missing values.

The following statements multiply matrices. The results are shown in Figure 23.19.

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

Figure 23.19 Result of Matrix Multiplication

```
c

7        10

d

    5
   11
```

Power Operator, Elementwise:

matrix1 ## matrix2 ;

matrix ## scalar ;

matrix ## vector ;

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.

The elementwise power operator enables either operand to be a scalar or a row or column vector.

- If either operand is scalar, the operation applies the power operator to each element and the scalar value.
- When you use the *matrix* / *vector* form, each row or column of the $n \times p$ matrix is raised to a power given by a corresponding element of the vector.

When an element of either matrix contains a missing value, the corresponding element of the result is also a missing value.

For example, the following statements raise each element of a matrix to a power, as shown in Figure 23.20:

```
a = {1 2 3};
b = a##3;
c = a##0.5;
print b, c;
```

Figure 23.20 Result of Raising Each Element to a Power

```
                                b

                1               8              27

                                c

                1 1.4142136 1.7320508
```

Power Operator, Matrix: **

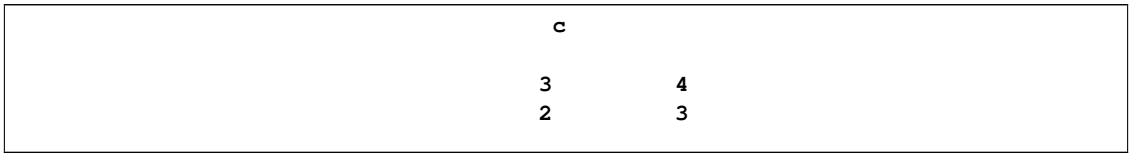
matrix ** *scalar* ;

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

For example, the following statements compute a matrix that is the result of multiplying a matrix by itself. The result is shown in Figure 23.21.

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

Figure 23.21 Result of Raising a Matrix to a Power



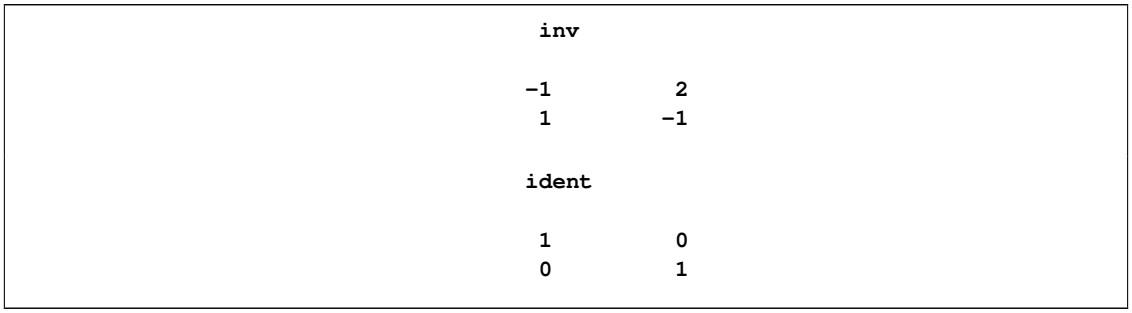
c

3	4
2	3

Note that the expression **a**(-1)** is shorthand for matrix inversion, as shown by the following statements:

```
inv = a**(-1);              /* shorthand for matrix inversion */
ident = inv * a;
print inv, ident;
```

Figure 23.22 Matrix Inversion by Using the Power Operator



inv

-1	2
1	-1

ident

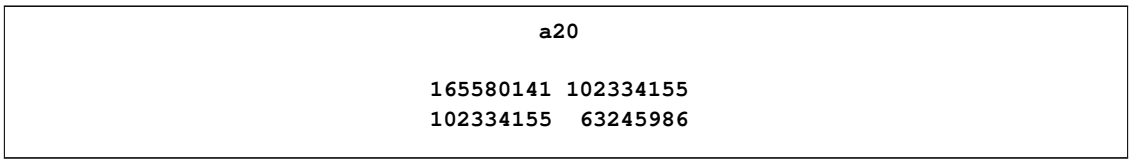
1	0
0	1

The matrix power operator does not support missing values.

Raising a matrix to a large power can cause numerical precision problems. If the matrix is symmetric, it is preferable to operate on its eigenvalues (see the EIGEN call) rather than to use the matrix power operator directly on the matrix, as shown in the following example:

```
b = {2 1,
     1 1};
call eigen(lambda, E, b);   /* recall that b = E*diag(lambda)*E` */
power = 20;
d = lambda##power;
a20 = E*diag(d)*E`;         /* a**20 since E`*E = Identity */
print a20;
```

Figure 23.23 Matrix Powers by Using Eigenvalues



a20

165580141	102334155
102334155	63245986

Sign Reversal Operator: –

–matrix ;

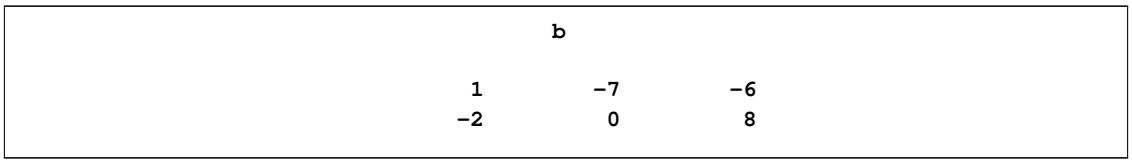
The sign reversal operator (–) computes a new matrix that contains elements that are formed by reversing the sign of each element in *matrix*. The sign reversal operator is also called the *unary minus* operator.

When an element of the matrix contains a missing value, the corresponding element of the result also contains a missing value.

The following statements reverse the signs of each element of a matrix, as shown in Figure 23.24:

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

Figure 23.24 The Result of a Sign Reversal Operator



Subscripts: [ ]

matrix[rows, columns] ;

matrix[elements] ;

Subscripts are used with matrices to select submatrices, where *rows* and *columns* are expressions that evaluate to scalars or vectors. If these expressions are numeric, they must contain valid subscript values of rows and columns in the argument matrix.

For example, the following statements select elements from the second row of the matrix **x**:

```
x = {1 2 3,
     4 5 6,
     7 8 9};
a = 3;
y = x[2, a];
b = 1:3;
z = x[2, b];
w = x[{4 6}];
print y, z, w;
```

Figure 23.25 Submatrices Formed by Specifying Indices

```
                                y

                                    6

                                z

                    4           5           6

                                w

                                    4
                                    6
```

The output is shown in Figure 23.25. The matrix **y** contains the element of **x** from the second row and the third column. The matrix **z** contains the entire second row of **x**. The matrix **w** contains the fourth and sixth elements of **x**. Because SAS/IML software store matrices in row-major order, **w** contains the first and third elements from the second row of **x**.

If a row or column expression is a character matrix, then it refers to columns or rows in the argument matrix that are assigned corresponding labels by a MATTRIB statement or READ statement. For example, the following statements select elements from the second row of **x**, and from the first and third columns:

```
x = {1 2 3,
     4 5 6,
     7 8 9};
c = "col1":"col3";
r = "row1":"row3";
mattrib x colname=c rowname=r;
a = {"col1" "col3"};
m = x["row2", a];
print m;
```

Figure 23.26 Submatrices Formed by Specifying Column Names

```
                                m

                    4           6
```

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, as shown in the following statements:

```
x[1, {1 3}] = .;
x[{1 2}, 2] = {0, 1};
x[7] = -1;
print x;
```

Figure 23.27 Result of Assigning Submatrices of an Existing Matrix

	x		
	col1	col2	col3
row1	.	0	.
row2	4	1	6
row3	-1	8	9

See the section "Using Matrix Expressions" on page 50 for further information about matrix subscripts.

Subtraction Operator: –

matrix1 – *matrix2* ;

matrix – *scalar* ;

matrix – *vector* ;

The subtraction operator (−) computes a new matrix that contains 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 scalar from a matrix or subtract a vector from a matrix.

- When either argument is a scalar, the subtraction is performed between the scalar and each element of the matrix argument. For example, when you use the *matrix* – *scalar* form, the scalar value is subtracted from each element of the matrix.
- When you use the *matrix* – *vector* form, the vector is subtracted from each row or column of the $n \times p$ matrix.
  - If you subtract an $n \times 1$ column vector, each row of the vector is subtracted from each row of the matrix.
  - If you subtract a $1 \times p$ row vector, each column of the vector is subtracted from each column of the matrix.

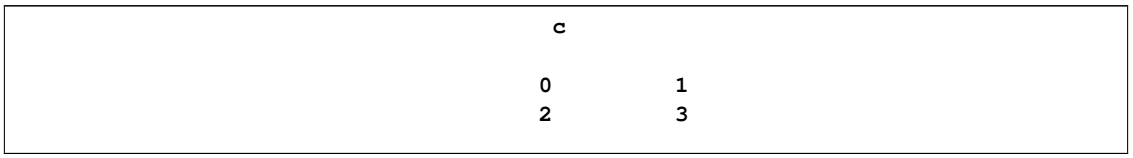
When an element of the matrix contains a missing value, the corresponding element of the result also contains a missing value.

For example, the following statements subtract two matrices and store the result in the matrix `c`, shown in Figure 23.28:

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

```
print c;
```

Figure 23.28 Difference of Two Matrices



```
                                c

                            0           1
                            2           3
```

Transpose Operator: `

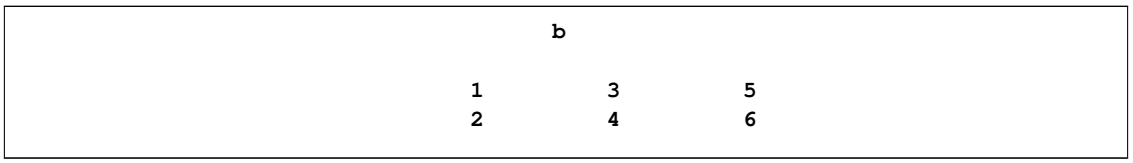
matrix `;

The transpose operator, denoted by the backquote character (`), exchanges the rows and columns of *matrix*, producing the transpose of *matrix*. If v is the value in the ith row and jth column of *matrix*, then the transpose of *matrix* contains v in the jth row and ith column. If *matrix* contains n rows and p columns, the transpose has p rows and n columns.

For example, the following statements transpose the matrix **a**, shown in Figure 23.29:

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

Figure 23.29 Transpose of a Matrix



```
                              b

                        1           3           5
                        2           4           6
```

You can also transpose a matrix with the T function.

Statements, Functions, and Subroutines

This section presents descriptions of all statements, functions, and subroutines that are available in SAS/IML software.

ABORT Statement

ABORT ;

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

The ABORT statement is the run-time equivalent of the QUIT statement. That is, you can use the ABORT statement as part of logical statements such as IF-THEN/ELSE statements, as shown in the following statements:

```
proc iml;
do i = 1 to 10;
   if i>2 then
      abort;
   print i;
end;
/* SAS/IML statements after this line are never executed. */
```

Figure 23.30 Result of Aborting a Computation

```
                                  i

                                  1

                                  i

                                  2
```

ABS Function

ABS(*matrix***);**

The ABS function returns the absolute value of every element of the argument matrix, as shown in the following statements:

```
x = -2:2;
a = abs(x);
print a;
```

Figure 23.31 Absolute Values

```
                                  a

                  2         1         0         1         2
```

ALL Function

ALL(*matrix***);**

The ALL function returns a value of 1 if all elements in *matrix* are nonzero. If any element of *matrix* is zero or missing, the ALL function returns a value of 0.

You can use the ALL function to express the results of a comparison operator as a single 1 or 0. For example, the following statement compares elements in two matrices:

```
a = { 1 2, 3 4};
b = {-1 0, 0 1};
if all(a>b) then
   msg = "a[i,j] > b[i,j] for all i,j";
else
   msg = "for some element, a[i,j] is not greater than b[i,j]";
print msg;
```

Figure 23.32 Result of Comparing All Elements

msg
a[i,j] > b[i,j] for all i,j

In the preceding statements, the comparison operation **a>b** creates a matrix of zeros and ones. The ALL function returns a value of 1 because 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, so in fact the previous IF-THEN statement is equivalent to the following:

```
if a>b then       /* implicit ALL */
   msg = "a[i,j] > b[i,j] for all i,j";
```

ALLCOMB Function

ALLCOMB(*n, k***);**

ALLCOMB(*n, comb, < , idx>* **);**

The ALLCOMB function generates all combinations of *k* elements taken from a set of *n* numerical indices. The combinations are produced in the same order and using the same algorithm (Nijenhuis and Wilf 1978) as the ALLCOMBI function in Base SAS software. In particular, the function returns indices in the range 1–*n*, and each combination is in sorted order.

By default, the ALLCOMB function returns a matrix with $\binom{n}{k}$ rows and *k* columns. Each row of the returned matrix represents a single combination. The following statements generate all combinations of two elements from the set $\{1, 2, 3, 4\}$:

```
n = 4; /* used throughout this example */
k = 2; /* used throughout this example */
c = allcomb(n, k);
print c;
```

Figure 23.33 All Pairwise Combinations of Four Items

c	
1	2
2	3
1	3
3	4
2	4
1	4

The second argument can be a scalar or a vector. If it is a vector, it must contain a valid combination of the set $\{1, 2, \ldots, n\}$. (To be valid, the *comb* elements must be in increasing order.) The number of elements in the vector determines the value of k. For example, the following statements generate all combinations of length two from a set with four elements, beginning with the third combination that is shown in Figure 23.33:

```
d = allcomb(4, {1 3});
```

To obtain all combinations in order, initialize the *comb* argument to `1:k` or to the zero vector with k elements.

The optional third argument, *idx*, controls the number of rows in the output of the function. If you specify *idx*, then the sequence is initialized with the *comb* argument and the first row of the output is the combination that occurs *after* the *comb* argument. For example, the following statements generate five pairwise combinations, beginning *after* the third combination shown in Figure 23.33:

```
e = allcomb(n, {1 3}, 1:5);
```

The *idx* argument must consist of consecutive integers; you cannot use it to randomly access combinations that are out of sequence. The *idx* argument is often used to generate one or more combinations in a loop so that you do not need to allocate a huge matrix that contains all of the combinations at once. The following statements illustrate this usage. Notice that you should initialize the *comb* argument to the zero vector if you want the first result to be the combination `1:k`.

```
ncomb = comb(n, k);
comb = j(1, k, 0);
do i=1 to ncomb;
   comb = allcomb(n, comb, i);
   /* do something with the i_th combination */
end;
```

If you want the combinations in lexicographic order, generate the combinations and then use the SORT subroutine, as follows:

```
c = allcomb(n, k);
call sort(c, 1:k);
```

ALLPERM Function

ALLPERM(*n*);

ALLPERM(*set*, < , *idx* >);

The ALLPERM function generates all permutations of a set with n elements. The permutations are produced in the same order and using the same algorithm (Trotter 1962) as the ALLPERM function in Base SAS software.

By default, the ALLPERM function returns a matrix with $n!$ rows and n columns. Each row of the returned matrix represents a single permutation. The following statements generate all permutations of the set $\{1, 2, 3\}$:

```
n = 3;
p = allperm(n);
print p;
```

Figure 23.34 All Permutations of Three Items

```
                                p

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

The first argument can be a scalar or a vector. If it is a vector, the number of elements in the vector determines the value of n. The ALLPERM function can compute permutations of arbitrary numeric or character matrices. For example, the following statements compute permutations of an unsorted character vector:

```
a = allperm({C B A});
print a;
```

Figure 23.35 All Permutations of a Character Vector

```
                                a

                        C B A
                        C A B
                        A C B
                        A B C
                        B A C
                        B C A
```

The optional second argument, *idx*, can be used to control the number of rows in the output of the function. The argument must consist of consecutive integers; you cannot use it to randomly access permutations that

are out of sequence. The second argument is often used to generate one or more permutations in a loop so that you do not need to allocate a huge matrix that contains all of the permutations at once. The following statements illustrate this usage:

```
perm = 1:n;
do i=1 to fact(n);
   perm = allperm(perm, i);
   /* do something with the i_th permutation */
end;
```

If you want the permutations in lexicographic order, generate the permutations and then use the SORT subroutine, as follows:

```
p = allperm(n);
call sort(p, 1:n);
```

ANY Function

ANY(*matrix***);**

The ANY function returns a value of 1 if any of the elements in *matrix* are nonzero. If all the elements of *matrix* are zero or missing, the ANY function returns a value of 0.

You can use the ANY function to compare elements in two matrices, as shown in the following statements:

```
a = {1 2, 3 4};
b = {3 2, 1 0};
if any(a=b) then
   msg = "for some element, a[i,j] equals b[i,j]";
else
   msg = "a ^= b";
print msg;
```

Figure 23.36 Result of Comparing Elements

msg
for some element, a[i,j] equals b[i,j]

In the preceding statements, the IF-THEN expression is true if at least one element in **a** is the same as the corresponding element in **b**. You can use the ALL function to compare all of the elements in two matrices.

APPCORT Call

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

If $\mathbf{A}$ is rank-deficient, then the least squares problem $\min_x \|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2$ has infinitely many solutions (Golub and Van Loan 1989, p. 241). However, there is a unique solution which has the smallest Euclidean norm. The APPCORT subroutine computes the minimum Euclidean-norm solution of the (rank-deficient) least squares problem by applyng a complete orthogonal decomposition by Householder transformations to the vector $\boldsymbol{b}$.

The input arguments to the APPCORT subroutine are as follows:

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 a $m \times p$ matrix, $\mathbf{B}$.

sing is an optional scalar that specifies 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 Euclidean-norm solution of the rank-deficient least squares problem $\|\mathbf{A}\boldsymbol{x} - \boldsymbol{b}|_2^2$.

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

See the section "COMPORT Call" on page 611 for information about complete orthogonal decomposition.

The following example uses the APPCORT call to solve a rank-deficient least squares problem:

```
/* compute solution for rank-deficient least squares problem:
      min |Ax-b|^2
   The range of A is a line; b is a point not on the line. */
A = {1  2,
     2  4,
    -1 -2};
b = {1, 3, -2};
call appcort(x,lindep,A,b);
print x;
```

Figure 23.37 Solution to a Rank-Deficient Least Squares Problem

```
x

0.3
0.6
```

The argument *b* can also be a matrix. If *b* is an identity matrix, then you can use the APPCORT subroutine to form a generalized inverse, as shown in the following example:

```
/* A has only four linearly independent columns */
A = {1 0 1 0 0,
     1 0 0 1 0,
     1 0 0 0 1,
     0 1 1 0 0,
     0 1 0 1 0,
     0 1 0 0 1 };

/* compute Moore-Penrose generalized inverse */
b = i(nrow(A));                    /* identity matrix */
call appcort(Ainv, lindep, A, b);
print Ainv;

/* verify generalized inverse conditions (Golub & Van Loan, p. 243) */
eps = 1e-12;
if any(A*Ainv*A-A > eps) |
   any(Ainv*A*Ainv-Ainv > eps) |
   any((A*Ainv)`-A*Ainv > eps) |
   any((Ainv*A)`-Ainv*A > eps) then
   msg = "Pseudoinverse conditions not satisfied";
else
   msg = "Pseudoinverse conditions satisfied";
print msg;
```

Figure 23.38 Generalized Inverse

```
                                Ainv

    0.2666667 0.2666667 0.2666667 -0.066667 -0.066667 -0.066667
    -0.066667 -0.066667 -0.066667 0.2666667 0.2666667 0.2666667
          0.4      -0.1      -0.1       0.4      -0.1      -0.1
         -0.1       0.4      -0.1      -0.1       0.4      -0.1
         -0.1      -0.1       0.4      -0.1      -0.1       0.4

                                msg

                  Pseudoinverse conditions satisfied
```

APPEND Statement

APPEND < **VAR** *operand*> **;**

APPEND < **FROM** *matrix*> < *[***ROWNAME***=row-name]*> **;**

The APPEND statement adds observations to the end of a SAS data set.

The arguments to the APPEND statement are as follows:

operand is specified as one of the following:

- a literal matrix that contains variable names
- a character matrix that contains variable names
- an expression in parentheses that yields variable names
- one of the following keywords:

  ALL for all variables

  CHAR for all character variables

  NUM for all numeric variables

matrix is the name of a matrix that contains data to append.

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

You can 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 *matrix*. The FROM clause and the VAR clause cannot be specified together.

The APPEND statement is usually used without any arguments. A common practice is to specify the data in the CREATE statement, as shown in the following example:

```
proc iml;
x = {1,2,3,4};          /* 4 x 1 vector */
y = {4 3,2 1};          /* 2 x 2 matrix */
z = {2,3,4};            /* 3 x 1 vector */
c = {A,B,C,D};          /* 4 x 1 character vector */

create Temp1 var {x y}; /* Temp1 contains two variables */
append;                 /* appends data from x and y */
close Temp1;
quit;

proc print data=Temp1 noobs;
run;
```

The values in the Temp1 data set are shown in Figure 23.39. Notice that the 2×2 matrix **y** is written to the data set in row-major order.

Figure 23.39 Data Set That Contains One Variable for Each Specified Matrix

X	Y
1	4
2	3
3	2
4	1

If you omit the VAR (and FROM) clause in the CREATE statement, then the new data set contains a variable for each SAS/IML matrix that is in scope. You can use the VAR clause in the APPEND statement to write specific variables. Variables that are not explicitly specified receive missing values, as shown in the

following statements:

```
proc iml;
x = {1,2,3,4};          /* 4 x 1 vector */
y = {4 3,2 1};          /* 2 x 2 matrix */
z = {2,3,4};            /* 3 x 1 vector */
c = {A,B,C,D};          /* 4 x 1 character vector */

create Temp2;           /* Temp2 contains a variable for each matrix */
append var {c x z};     /* y gets missing values */
close Temp2;
quit;

proc print data=Temp2 noobs;
run;
```

The values in the Temp2 data set are shown in Figure 23.40. The data set contains four observations because that is the number of elements in the matrix with the greatest number of elements. Elements are appended in row-major order. Notice that the variable z contains a missing value at the end because the variable was created from a SAS/IML matrix that contained fewer than four elements.

Figure 23.40 Data Set That Contains One Variable for Each Matrix

c	x	y	z
A	1	.	2
B	2	.	3
C	3	.	4
D	4	.	.

As shown in the previous example, 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= option in the FROM clause specifies the name of a character matrix to contain row titles. Use this option in conjunction with the identical option in the FROM clause of the CREATE statement, as shown in the following statements:

```
proc iml;
VarName = {"x" "y"};
w = {3 96,
     4 90,
     2 100,
     4 92};                 /* data matrix */
cov = cov(w);               /* sample covariance matrix of data */

create Temp3 from cov[rowname=VarName colname=VarName];
append from cov[rowname=VarName];
close Temp3;
quit;

proc print data=Temp3 noobs;
run;
```

The values in the Temp3 data set are shown in Figure 23.41. The matrix **cov** contains the data that is saved

to the Temp3 data set. The character vector **VarName** contains the names of the variables for the Temp3 data set. (If you use the FROM clause in the CREATE statement, but do not specify the COLNAME= option, then the variables are named COL1, COL2, and so on.) The ROWNAME= option enables you to specify a single character variable when you are creating a data set from a numerical matrix. This is useful for specifying variable names in a correlation or covariance matrix, but can also be used more generally to specify a row label for each observation.

Figure 23.41 Data Set That Contains Row Labels

Var Name	x	y
x	0.91667	−4.1667
y	−4.16667	19.6667

If you do not specify the ROWNAME= option in the CREATE statement, then you do not need to specify the ROWNAME= option in the APPEND statement, as shown in the following example:

```
create Temp3 from cov[colname=VarName];
append from cov;
close Temp3;
```

You can also use the APPEND statement with the EDIT statement. See the documentation for the EDIT statement for examples.

APPLY Function

APPLY(*modname, argument1* < *, argument2, . . . , argument14* > **);**

The APPLY function applies a user-defined module to each element of the argument matrix or matrices and returns a matrix of results.

The arguments to the APPLY statement are as follows:

modname specifies the name of an existing function module. You can specify the module name as a literal string or as matrix that contains the module name.

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

The first argument to APPLY is the name of a function module. The module must take scalar arguments and must already be defined before the APPLY function is executed. The subsequent arguments to the APPLY function are the arguments passed to the module. They all must have the same dimension.

If the function module takes n scalar arguments, *argument1* through *argumentn* should be passed to APPLY where $1 \leq n \leq 14$. The APPLY function calls the module one time for each element in its input arguments. 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 in addition to 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};
s = apply("ABC", a);
print s;
```

The result is shown in Figure 23.42.

Figure 23.42 Result of a Module Applied to Each Argument in a Matrix

s		
106	107	108
109	110	111

The module can also alter the contents of the arguments. In the following example, the statements define the module ABSDIFF and call the APPLY function:

```
/* compute abs(x-y); permute elements of x and y so that x[i] >= y[i] */
start AbsDiff(x, y);
   if x<y then do;  /* swap x and y */
      t = x;
      x = y;
      y = t;
   end;
   return( x-y );
finish;

a = {-1 0 1};
b = {-2 0 2};
mod = "AbsDiff";
r = apply(mod, a, b);
print a, b, r;
```

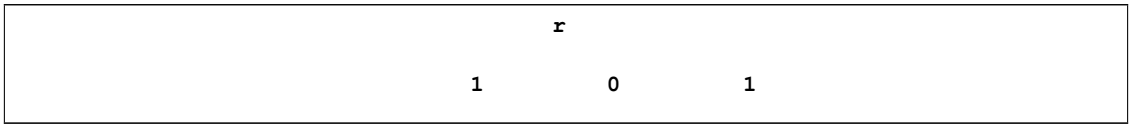
Notice that the third element of the **a** and **b** arguments are exchanged, as shown in Figure 23.43.

Figure 23.43 Result of a Module Applied to Each Argument in a Matrix

a		
-1	0	2

b		
-2	0	1

Figure 23.43 *continued*



ARMACOV Call

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

The ARMACOV subroutine computes an autocovariance sequence for an autoregressive moving average (ARMA) model. The input arguments to the ARMACOV subroutine are as follows:

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

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

num refers to a scalar that contains 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 that contains 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 that contains 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 that contains 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) 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.

Consider the model

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

To compute the autocovariance function at lags zero through four for this model, use the following statements:

```
/* 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 show in Figure 23.44.

Figure 23.44 Result of the ARMACOV Subroutine

```
                                auto

3.2533333 2.4266667 1.2133333 0.6066667 0.3033333

          cross                convol

           2.04        0.8       1.64        0.8
```

ARMALIK Call

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

The ARMALIK subroutine computes the log likelihood and residuals for an autoregressive moving average

(ARMA) model. The input arguments to the ARMALIK subroutine are as follows:

x	is an $n \times 1$ or $1 \times n$ matrix that contains values of the time series (assuming mean zero).
phi	is a $1 \times (p+1)$ matrix that contains the autoregressive parameter values. The first element is assumed to have the value 1.
theta	is a $1 \times (q+1)$ matrix that contains 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 that contains 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 that contains 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 that contains the scale factors used to standardize the residuals. The actual residuals from the one-step-ahead predictions that use 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. Consider the following model:

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

To compute the log likelihood for this model, use the following statements:

```
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;
```

Figure 23.45 Results from an ARMALIK Call

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

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

The ARMASIM function simulates a univariate series from a autoregressive moving average (ARMA) model.

The arguments to the ARMASIM function are as follows:

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

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

mu is a scalar that contains the overall mean of the series.

sigma is a scalar that contains the standard deviation of the innovation series.

n is a scalar that contains n, the length of the series. The value of n must be greater than 0.

seed is a scalar that contains the random number seed. At the first execution of the function, the seed variable is used as follows:

- If *seed* > 0, the input seed is used for generating the series.
- If *seed* = 0, the system clock is used to generate the seed.
- If *seed* < 0, the value –*seed* is used for generating the series.

If the seed is not supplied, the system clock is used to generate the seed.

On subsequent calls to the function, the seed variable is used as follows:

- If *seed* > 0, the seed remains unchanged.
- In other cases, after each execution of the function, the current seed is updated 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 by using an

expanded covariance matrix, and then the remaining values are generated by using the following recursion form of the model:

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

The random number generator RANNOR is used to generate the noise component of the model. Note that the following statement returns n standard normal pseudorandom deviates:

```
y = armasim(1, 1, 0, 1, n, seed);
```

For example, consider the following model:

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

To generate a time series of length 10 from this model, use the following statements to produce the result shown in Figure 23.46:

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

Figure 23.46 Simulated Time Series

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

BIN Function

BIN(*x, cutpoints* **<** *, closed* **>);**

The BIN function divides numeric values into a set of disjoint intervals called bins. The BIN function returns a matrix that is the same shape as *x* and that indicates which elements of *x* are contained in each bin. The arguments are as follows:

x specifies a numerical vector or matrix.

cutpoints specifies the intervals into which to bin the data. This argument can have a vector or a scalar value. A vector defines the endpoints of the intervals; a scalar value specifies the number of evenly spaced intervals into which the range of the data is divided.

closed is an optional argument that specifies whether the bins are open on the right or left sides. The following values are valid:

`"Left"` specifies that the bins are closed on the left and open on the right. The last interval is closed on both sides. This is the default value.

`"Right"` specifies that the intervals are open on the left and closed on the right. The first interval is closed on both sides.

If *cutpoints* is a vector, then it must be ordered so that the first element is the smallest and the last element is the largest. The ordered values define the intervals that are used to bin the values. For example, the following statements bin *x* into the intervals $I_1 = [0, 1)$, $I_2 = [1, 1.8)$, $I_3 = [1.8, 2)$, and $I_4 = [2, 4]$, and return the bin numbers for each element of *x*:

```
x = {0, 0.5, 1, 1.5, 2, 2.5, 3, 0.5, 1.5, 3, 3, 1};
cutpoints = {0 1 1.8 2 4};
b = bin(x, cutpoints);
print x b;
```

Figure 23.47 Bins for Each Observation

x	b
0	1
0.5	1
1	2
1.5	2
2	4
2.5	4
3	4
0.5	1
1.5	2
3	4
3	4
1	2

You can use the special missing values `.M` and `.I` to specify unbounded intervals. A missing value of `.M` in the first element is interpreted as $-\infty$, and a missing value of `.I` in the last element is interpreted as $+\infty$. For example, the following statements are all valid specifications of the *cutpoints* argument:

```
c = {.M -2 -1 0 1 2};
c = {.M -2 -1 0 1 2 .I};
c = {-2 -1 0 1 2 .I};
```

If *cutpoints* is a positive integer, n, then the interval $\min(x), \max(x)$ is divided into n intervals of width $\Delta = (\max(x) - \min(x))/n$ and the data are binned into these intervals. For example, the following statements bin the elements of *x* into one of three intervals $[0, 1)$, $[1, 2)$, or $[2, 3]$:

```
bin = bin(x, 3);
```

```
print x bin;
```

Figure 23.48 Bins That Are Associated with Each Value

x	bin
0	1
0.5	1
1	2
1.5	2
2	3
2.5	3
3	3
0.5	1
1.5	2
3	3
3	3
1	2

Notice in Figure 23.48 that the value 3 is placed into the third interval because the last interval is closed on the right.

The BIN function returns missing values for data values that are not contained in any bin. Missing values are also returned for missing values in the data.

You can use the BIN function in conjunction with the TABULATE function to count the number of observations in each interval. The following statements sample from the standard normal distribution and count the number of observations in a set of evenly spaced intervals:

```
z = rannor(j(1000, 1, 1));
set = do(-3.5, 3.5, 1);
b = bin(z, set);
call tabulate(levels, count, b);

/* label counts by the center of each interval */
intervals = char(do(-3, 3, 1), 2);
print count[colname=intervals];
```

Figure 23.49 Bins Counts for Evenly Spaced Intervals

count						
-3	-2	-1	0	1	2	3
6	65	241	385	235	59	9

BLOCK Function

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

The BLOCK function forms a block-diagonal matrix. The blocks are defined by the arguments to the

function. Up to 15 matrices can be specified. The matrices are combined diagonally to form a new matrix.

For example, if **A**, **B**, and **C** are any matrices, then the block matrix formed from these matrices has the following form:

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

The following statements produce a block-diagonal matrix composed of three blocks, shown in Figure 23.50:

```
a = 1;
b = {2 2,
     3 3};
c = {4 4 4,
     5 5 5};
d = block(a, b, c);
print d;
```

Figure 23.50 Block Matrix

d

1	0	0	0	0	0
0	2	2	0	0	0
0	3	3	0	0	0
0	0	0	4	4	4
0	0	0	5	5	5

BRANKS Function

BRANKS(*matrix*);

The BRANKS function computes 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

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

The results of the BRANKS function can be used to compute rank-based correlation coefficients such as the Spearman rank-order correlation and Hoeffding's D statistic.

The following statements compute the bivariate ranks of two columns of data:

```
z = { 1 2,
      2 1,
      3 3,
      3 5,
      4 4,
      5 4,
      5 4,
      4 5 };

b = branks(z);
print b;
```

Figure 23.51 Tied Ranks and Bivariate Ranks

```
                b

          1       2       1
          2       1       1
        3.5       3       3
        3.5     7.5     3.5
        5.5       5       4
        7.5       5    4.75
        7.5       5    4.75
        5.5     7.5       5
```

BSPLINE Function

BSPLINE(*x, d, k* **<** *, i* **>);**

The BSPLINE function computes a B-spline basis. The arguments to the BSPLINE function are as follows:

x is an $m \times 1$ or $1 \times m$ numeric vector.

d is a nonnegative numeric scalar value that specifies the degree of the B-spline. The order of a B-spline is one greater than the degree.

k is a numeric vector of size n that contains the B-spline knots or a scalar that denotes the number of interior knots. When $n > 1$, the elements of the knot vector must be nondecreasing, $k_{j-1} \leq k_j$ for $j = 2, \ldots, n$.

i is an optional argument that specifies the number of interior knots when $n = 1$ and k contains a missing value. In this case the BSPLINE function constructs a vector of knots as follows: If $x_{(1)}$ and $x_{(m)}$ are the smallest and largest value in the x vector, then interior knots are placed at

$$x_{(1)} + j(x_{(m)} - x_{(1)})/(k+1), \quad j = 1, \ldots, k$$

In addition, d exterior knots are placed under $x_{(1)}$ and max(d,1) exterior knots are placed over $x_{(m)}$. The exterior knots are evenly spaced and start at $x_{(1)}-$ 1E−12 and $x_{(m)}+$ 1E−12. In this case the BSPLINE function returns a matrix with m rows and $i + d + 1$ columns.

The BSPLINE function computes B-splines of degree d. Suppose that $B_j^d(x)$ denotes the jth B-spline of degree d in the knot sequence $k_1, \ldots, k_n$. DeBoor (1981) defines the splines based on the following relationships:

$$B_j^0(x) = \begin{cases} 1 & k_j \le x < k_{j+1} \\ 0 & \text{otherwise} \end{cases}$$

and for $d > 0$

$$\begin{aligned} B_j^d(x) &= w_j^d(x) B_j^{d-1}(x) + (1 - w_{j+1}^d(x)) B_{j+1}^{d-1}(x) \\ w_j^d(x) &= \frac{x - k_j}{k_{j+d} - k_j} \end{aligned}$$

Note that DeBoor (1981) expresses B-splines in terms of order rather than degree; in his notation $B_{j,d} = B_j^{d-1}$. B-splines have many interesting properties, including the following:

- $\sum_j B_j^d = 1$
- The sequence B_j^d is positive on $d + 1$ knots and zero elsewhere.
- The B-spline B_j^d is a piecewise polynomial of at most $d + 1$ pieces.
- If $k_j = k_{j+d}$, then $B_j^{d-1} = 0$.

See DeBoor (1981) for more details. The BSPLINE function defines B-splines of degree 0 as nonzero if $k_j < x \le k_{j+1}$.

A typical knot vector for calculating B-splines consists of d exterior knots smaller than the smallest data value, and max$\{d, 1\}$ exterior knots larger than the largest data value. The remaining knots are the interior knots.

For example, the following statements creates a B-spline basis with three interior knots. The BSPLINE function returns a matrix with $3 + d + 1 = 7$ columns, shown in Figure 23.53.

```
x     = {2.5 3 4.5 5.1};     /* data range is [2.5, 5.1] */
knots = {0 1 2 3 4 5 6 7 8}; /* three interior knots at x=3, 4, 5 */
bsp = bspline(x, 3, knots);
print bsp[format=best7.];
```

Figure 23.52 B-Spline Basis

bsp

0.02083	0.47917	0.47917	0.02083	0	0	0
0	0.16667	0.66667	0.16667	0	0	0
0	0	0.02083	0.47917	0.47917	0.02083	0
0	0	0	0.1215	0.65717	0.22117	0.00017

If you pass an *x* vector of data values, you can also rely on the BSPLINE function to compute a knot vector for you. For example, the following statements compute B-splines of degree 2 based on four equally spaced interior knots:

```
n = 15;
x = ranuni(J(n, 1, 45));
bsp2 = bspline(x, 2, ., 4);
print bsp2[format=8.3];
```

The resulting matrix is shown in Figure 23.53.

Figure 23.53 B-Spline Basis with Four Interior Knots

bsp2

0.000	0.104	0.748	0.147	0.000	0.000	0.000
0.000	0.000	0.000	0.286	0.684	0.030	0.000
0.000	0.000	0.000	0.000	0.000	0.517	0.483
0.000	0.000	0.000	0.217	0.725	0.058	0.000
0.000	0.000	0.239	0.713	0.048	0.000	0.000
0.000	0.000	0.000	0.446	0.553	0.002	0.000
0.000	0.000	0.394	0.600	0.006	0.000	0.000
0.000	0.000	0.000	0.000	0.064	0.729	0.207
0.000	0.389	0.604	0.007	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.500	0.500
0.000	0.000	0.000	0.000	0.210	0.728	0.062
0.000	0.000	0.014	0.639	0.347	0.000	0.000
0.000	0.001	0.546	0.453	0.000	0.000	0.000
0.500	0.500	0.000	0.000	0.000	0.000	0.000
0.304	0.672	0.024	0.000	0.000	0.000	0.000

BTRAN Function

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

The BTRAN function computes the block transpose of a partitioned matrix. The arguments to the BTRAN function are as follows:

x is an $(in) \times (jm)$ 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 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 following statements compute the block transpose of a matrix:

```
a11 = {1 1,                    /* a 3 x 2 matrix */
       1 1,
       1 1};
a12 = 1 + a11;
a13 = 2 + a11;
a21 = 3 + a11;
a22 = 4 + a11;
a23 = 5 + a11;

x = (a11 || a12 || a13) //     /* a partitioned matrix  */
    (a21 || a22 || a23);       /* each submatrix is a 3 x 2 block */

z = btran(x, 3, 2);            /* transpose the blocks */
print z;
```

Figure 23.54 Block Transpose of a Partitioned Matrix

z

1	1	4	4
1	1	4	4
1	1	4	4
2	2	5	5
2	2	5	5
2	2	5	5
3	3	6	6
3	3	6	6
3	3	6	6

BYTE Function

BYTE(*matrix***);**

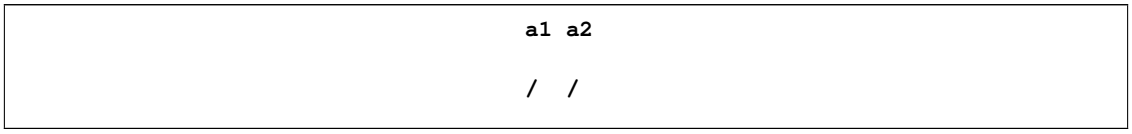
The BYTE function returns values in a computer's character set. The input to the function is a numeric matrix, each element of which specifies the position of a character in the computer's character set. These numeric elements should generally be in the range 0 to 255. The BYTE function returns a character matrix with the same shape as the numeric argument.

For example, in the ASCII character set, the following two statements are equivalent:

```
a1 = byte(47);
```

```
a2 = "/";       /* the slash character */
print a1 a2;
```

Figure 23.55 Specifying the Slash Character

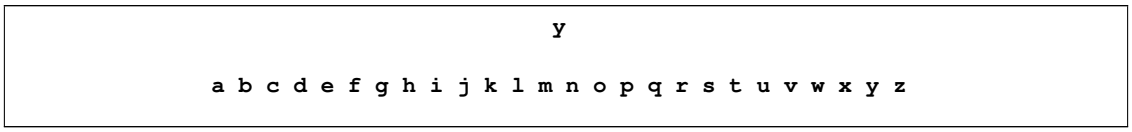


```
a1 a2

/  /
```

The lowercase English letters can be generated with the following statement, shown in Figure 23.56:

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

Figure 23.56 Lowercase English Letters



```
y

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
```

The BYTE function simplifies the use of special characters and control sequences that cannot be entered directly into SAS/IML programs by using the keyboard. Consult the character set tables for your computer to determine the printable and control characters that are available and their ordinal positions.

CALL Statement

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

The CALL statement enables you to call a built-in or user-defined subroutine.

The arguments to the CALL statement are as follows:

name is the name of a built-in subroutine or a user-defined module.

arguments are arguments to the module or subroutine.

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

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

This resolution order is important only if you have defined a module with the same name as a built-in subroutine.

See also the section on the RUN statement.

CHANGE Call

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

The CHANGE subroutine searches for and replaces text in a character matrix. The arguments to the CHANGE call are as follows:

matrix	is a character matrix.
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, consider the following statements:

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

The result of these statements is shown in Figure 23.57.

Figure 23.57 New String

```
                                a

                It was a dark and stormy day.
```

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

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

CHAR Function

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

The CHAR function produces a character representation of a numeric matrix. Essentially, the CHAR function is equivalent to applying a *w.d* format to each element of a numeric matrix.

The arguments 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 the same dimensions as the argument matrix, and with elements that are character representations of the corresponding numeric elements.

If the *w* argument is not supplied, the system default field width is used. If the *d* argument is not supplied, the best representation is used. See also the description of the NUM function, which converts a character matrix into a numeric matrix.

For example, the following statements produce the output shown in Figure 23.58:

```
a = {-1.1 0 3.1415 4};
reset print;       /* display values and type of matrices */
m = char(a, 4, 1);
```

Figure 23.58 Character Matrix

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

                            -1.1  0.0  3.1  4.0
```

CHOOSE Function

CHOOSE(*condition, result-for-true, result-for-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.

The arguments 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.

Each argument must be conformable with the others (or be a scalar value).

For example, suppose that you want to choose between x and y according to whether $x\#y$ is odd or even, respectively. You can use the following statements to execute this task, as shown in Figure 23.59:

```
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;
```

Figure 23.59 Result of the CHOOSE Function

x	y	r
1	101	1
2	205	205
3	133	3
4	806	806
5	500	500

As another example, the following statements replace all missing values in the matrix **z** with zeros, as shown in Figure 23.60:

```
z = {1 2 ., 100 . -90, . 5 8};
newZ = choose(z=., 0, z);
print z, newZ;
```

Figure 23.60 Replacement of Missing Values

z		
1	2	.
100	.	-90
.	5	8

newZ		
1	2	0
100	0	-90
0	5	8

CLOSE Statement

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

The CLOSE statement is used to close one or more SAS data sets opened with the USE, EDIT, or CREATE statement.

The optional argument specifies the name of one or more SAS data sets. The data sets can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). For more information about specifying SAS data sets, refer to Chapter 7, "Working with SAS Data Sets."

You can use the SHOW DATASETS statement to find the names of open data sets.

SAS/IML software automatically closes all open data sets when a QUIT statement is executed.

The following statements provide examples of using the CLOSE statement:

```
use Sashelp.Class;
read all var _NUM_ into x[colname=VarName];
```

```
corr = corr(x);
create ClassCorr from corr[rowname=VarName colname=VarName];
append from corr[rowname=VarName];

show datasets;
close Sashelp.Class ClassCorr;
```

Figure 23.61 Open Data Sets

```
LIBNAME  MEMNAME                          OPEN MODE   STATUS
-------- -------------------------------- ---------   --------
SASHELP  CLASS                            Input
WORK     CLASSCORR                        Update      Current Input/Output
```

It is good programming practice to close data sets when you are finished using them.

CLOSEFILE Statement

CLOSEFILE *files* **;**

The CLOSEFILE statement is used to close files opened by the INFILE or FILE statement.

The statement arguments specify the name of one or more file specifications. You can specify names (for defined filenames), literals, or expressions in parentheses (for pathnames). Each file specification should be the same as when the file was opened.

To find out what files are open, use the SHOW FILES statement. For further information, see Chapter 8. See also the description of the SAVE statement.

SAS/IML software automatically closes all files when a QUIT statement is executed.

The following example opens and closes an external file named *MyData.txt* that resides in the current directory. (If you run PROC IML through a SAS Display Manager Session (DMS), you can change the current directory by selecting **Tools ►Options ►Change Current Folder** from the main menu.)

```
filename MyFile 'MyData.txt';
infile MyFile;
show files;
closefile MyFile;
```

Figure 23.62 Open External File

```
FILE NAME                 MODE           EOF    OPTIONS
---------                 ------         -----  --------
FILENAME:MYFILE           Current Input , no eof, lrecl=512 STOPOVER
```

Alternatively, you can specify the full path of the file, as shown in the following statements:

```
src = "C:\My Data\MyData.txt";
infile (src);
show files;
closefile (src);
```

Figure 23.63 Open File Specified by a Full Path

```
FILE NAME                  MODE            EOF    OPTIONS
---------                  ------          -----  --------
C:\My Data\MyData.txt      Current Input , no eof, lrecl=512 STOPOVER
```

COMPORT Call

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

The COMPORT subroutine provides the complete orthogonal decomposition by Householder transformations of a matrix **A**.

The subroutine returns the following values:

q is a matrix. 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 to 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 of the $(m - n) \times n$ zero matrix to the result r.

p is an $n \times n$ matrix that is the product $\mathbf{P\Pi}$ of a permutation matrix $\mathbf{\Pi}$ with an orthogonal matrix $\mathbf{P}$. The permutation matrix is determined by the vector *piv*.

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 $[\mathbf{A}_{piv[1]} \ldots \mathbf{A}_{piv[n]}]$. 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* is $n - r$.

The input arguments 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 $\mathbf{P}$,

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

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 that specifies a singularity criterion.

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

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

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

   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 $\boldsymbol{\Pi}$ 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 $\text{rank}(\mathbf{L}_{12}) = \text{rank}(\mathbf{L}_1) = r$, $\mathbf{L}_1 \in \mathcal{R}^{r\times r}$ 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 $\mathbf{P}_1, \ldots, \mathbf{P}_r$,

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

   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

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

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

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

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

   (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 by 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 $\mathbf{A}^{-}$ by using the singular value decomposition of $\mathbf{A}$. Using complete orthogonal decomposition to compute $\mathbf{A}^{-}$ usually requires far fewer floating-point operations. However, it can be slightly more sensitive to rounding errors, which can disturb the detection of the true rank of $\mathbf{A}$, than the singular value decomposition.

The following example demonstrates some uses of the COMPORT subroutine:

```
/* Only four linearly independent columns */
A = {1 0 1 0 0,
     1 0 0 1 0,
     1 0 0 0 1,
     0 1 1 0 0,
     0 1 0 1 0,
     0 1 0 0 1 };
m = nrow(A);
n = ncol(A);

call comport(q,r,p,piv,lindep,A);
fullR = r // j(m-n, n, 0);
perm = i(n);
perm[piv,] = i(n);

/* recover A from factorization */
A2 = q*fullR*p`*perm`;
reset fuzz;
print A2;

/* compute Moore-Penrose generalized inverse */
rankA = n - lindep;
subR = R[1:rankA, 1:rankA];
fullRinv = j(n, n, 0);
fullRinv[1:rankA, 1:rankA] = inv(subR);
Ainv = perm*p*fullRinv*q[,1:n]`;
print Ainv;

/* verify generalized inverse */
eps = 1e-12;
if any(A*Ainv*A-A > eps) |
   any(Ainv*A*Ainv-Ainv > eps) |
   any((A*Ainv)`-A*Ainv > eps) |
   any((Ainv*A)`-Ainv*A > eps) then
   msg = "Pseudoinverse conditions not satisfied";
else
   msg = "Pseudoinverse conditions satisfied";
print msg;
```

Figure 23.64 Results from a Complete Orthogonal Factorization

```
                                A2

                  1         0         1         0         0
                  1         0         0         1         0
                  1         0         0         0         1
                  0         1         1         0         0
                  0         1         0         1         0
                  0         1         0         0         1

                               Ainv

      0.2666667 0.2666667 0.2666667 -0.066667 -0.066667 -0.066667
      -0.066667 -0.066667 -0.066667 0.2666667 0.2666667 0.2666667
            0.4      -0.1      -0.1       0.4      -0.1      -0.1
           -0.1       0.4      -0.1      -0.1       0.4      -0.1
           -0.1      -0.1       0.4      -0.1      -0.1       0.4

                               msg

                  Pseudoinverse conditions satisfied
```

CONCAT Function

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

The CONCAT function produces a character matrix that contains elements that are the concatenations of corresponding elements 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 have the same dimensions. 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, suppose you specify the following matrices:

```
b = {"AB" "C ",
     "DE" "FG"};
c = {"H " "IJ",
     " K" "LM"};
```

The following statement produces a new 2×2 character matrix, **a**:

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

Figure 23.65 Elementwise Concatenation of Strings

```
                                a

                           ABH  C IJ
                           DE K FGLM
```

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

CONTENTS Function

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

The CONTENTS function returns a column vector that contains the variable names for a SAS data set. The vector contains *n* rows, 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.

You can specify the SAS data set with a one-level name (for example, A) or with a libref and a SAS data set name (for example, Sashelp.Class). If you specify a one-level name, SAS/IML software uses the default SAS data library (as specified in the DEFLIB= option in the RESET statement.) If no arguments are specified, the current open input data set is used.

The following statements use the CONTENTS function to obtain the names of variables in SAS data sets:

```
x = 1:5;
create temp from x;
append from x;
tempVars = contents();            /* use current open input data set */
close temp;

classVars = contents("sashelp", "class"); /* contents of data set in */
                                          /* SASHELP library         */
print tempVars classVars;
```

Figure 23.66 Names of Variables in SAS Data Sets

```
                          tempVars classVars

                          COL1     Name
                          COL2     Sex
                          COL3     Age
                          COL4     Height
                          COL5     Weight
```

See also the description of the SHOW CONTENTS statement.

CONVEXIT Function

CONVEXIT(*times, flows, ytm***);**

The CONVEXIT function computes and returns a scalar that contains the convexity of a noncontingent cash flow. The arguments to the CONVEXIT function are as follows:

times	is an n-dimensional column vector of times. Elements should be nonnegative.
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:

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

Under certain assumptions, the convexity of cash flows that are not yield-sensitive 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, and the kth cash flow is $c(k)$ t_k periods from the present.

The following statements compute the convexity of a noncontingent cash flow.

```
timesn = T(do(1, 100, 1));
flows = repeat(10, 100);
ytm = 0.1;
convexit = convexit(timesn, flows, ytm);
print convexit;
```

Figure 23.67 Convexity of a Noncontingent Cash Flow

convexit
199.26229

CORR Function

CORR(*x* **<** *, method***>** **<** *, excludemiss***>** **);**

The CORR function computes a sample correlation matrix for data. The arguments are as follows:

x specifies an $n \times p$ numerical matrix of data. The CORR function computes a $p \times p$ correlation matrix of the data.

method specifies the method used to compute the correlation matrix. The following strings are valid:

- `"Pearson"` specifies the computation of Pearson product-moment correlations. The correlations range from -1 to 1. This is the default value.
- `"Hoeffding"` specifies the computation of Hoeffding's D statistics, scaled to range between -0.5 and 1.
- `"Kendall"` specifies the computation of Kendall's tau-b coefficients based on the number of concordant and discordant pairs of observations. Kendall's tau-b ranges from -1 to 1.
- `"Spearman"` specifies the computation of Spearman correlation coefficients based on the ranks of the variables. The correlations range from -1 to 1.

excludemiss specifies how missing values are handled. The following values are valid:

- `"listwise"` specifies that observations with missing values are excluded from the analysis. This is the default value.
- `"pairwise"` specifies that all nonmissing pairs of values for each pair of variables are included in the statistical computations.

The *method* and *excludemiss* arguments are not case-sensitive. The first four characters are used to determine the value. For example, `"LIST"` and `"listwise"` specify the same option.

The CORR function computes a sample correlation matrix for data, as shown in the following example:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
     7 2 13};
corr = corr(x);
spearman = corr(x, "spearman");
print corr, spearman;
```

Figure 23.68 Correlation Matrices

```
                                corr

                        1 0.1091089  0.265165
                0.1091089         1 -0.289319
                 0.265165 -0.289319         1

                           spearman

                        1 0.3441236 0.2236068
                0.3441236         1 -0.410391
                0.2236068 -0.410391         1
```

The CORR function behaves similarly to the CORR procedure. In particular, the documentation for the CORR procedure in the *Base SAS Procedures Guide: Statistical Procedures* includes details about the various correlation statistics.

The CORR function also handles missing values in the same way as the CORR procedure. In particular, be aware that specifying *excludemiss*="pairwise" might result in a correlation matrix that is not nonnegative definite.

You can use the ROWNAME= and COLNAME= options in the MATTRIB statement or the PRINT statement to associate names of variables to rows and columns of the correlation matrix. For example, if the names of the variables in the previous example are X1, X2, and X3, then the following statements associate those names with the matrix returned by the CORR function:

```
varnames = {"X1" "X2" "X3"};
mattrib corr     rowname=varnames colname=varnames
        spearman rowname=varnames colname=varnames;
print corr, spearman;
```

Figure 23.69 Correlation Matrices with Named Rows and Columns

```
                                corr
                          X1         X2         X3

                  X1         1  0.1091089   0.265165
                  X2 0.1091089          1  -0.289319
                  X3  0.265165 -0.289319          1

                             spearman
                          X1         X2         X3

                  X1         1  0.3441236  0.2236068
                  X2 0.3441236          1  -0.410391
                  X3 0.2236068 -0.410391          1
```

Prior to SAS/IML 9.22, there was a module named CORR in the IMLMLIB library. This module has been removed.

COUNTMISS Function

COUNTMISS(*x* < , *method*> **);**

The COUNTMISS function counts the number of missing values in a matrix. The arguments are as follows:

x specifies an $n \times p$ numerical or character matrix. The COUNTMISS function counts the number of missing values in this matrix.

method specifies the method used to count the missing values. This argument is optional. The following are valid values:

"all"	specifies that all missing values are counted. This is the default value. The function returns a 1×1 matrix.
"row"	specifies that the function return an $n \times 1$ matrix whose ith element is the number of missing values in the ith row of *x*.
"col"	specifies that the function return a $1 \times p$ matrix whose jth element is the number of missing values in the jth row of *x*.

The *method* argument is not case-sensitive. The first three characters are used to determine the value.

For example, the following statements count missing values for the matrix **x**:

```
x = {1 2 3,
     . 0 2,
     1 . .,
     1 0 . };
totalMiss = countmiss(x);
rowMiss = countmiss(x, "ROW");
colMiss = countmiss(x, "COL");
print totalMiss, rowMiss, colMiss;
```

Figure 23.70 Counts of Missing Values

```
totalMiss

        4

 rowMiss

        0
        1
        2
        1

 colMiss

1       1       2
```

COUNTN Function

COUNTN(*x* < , *method*> **);**

The COUNTN function counts the number of nonmissing values in a matrix. The arguments are as follows:

x	specifies an $n \times p$ numerical or character matrix. The COUNTN function counts the number of nonmissing values in this matrix.
method	specifies the method used to count the nonmissing values. This argument is optional. The following are valid values:

"all"	specifies that all nonmissing values are counted. This is the default value. The function returns a 1×1 matrix.
"row"	specifies that the function return an $n \times 1$ matrix whose ith element is the number of nonmissing values in the ith row of *x*.
"col"	specifies that the function return a $1 \times p$ matrix whose jth element is the number of nonmissing values in the jth row of *x*.

The *method* argument is not case-sensitive. The first three characters are used to determine the value.

For example, the following statements count nonmissing values for a matrix **x**:

```
x = {1 2 3,
     . 0 2,
     1 . .,
     1 0 . };
totalN = countn(x);
rowN = countn(x, "ROW");
colN = countn(x, "COL");
print totalN, rowN, colN;
```

Figure 23.71 Counts of Nonmissing Values

```
totalN

      8

rowN

      3
      2
      1
      2

colN

3     3     2
```

COUNTUNIQUE Function

COUNTUNIQUE(*x* < , *method*> **);**

The COUNTUNIQUE function counts the number of unique values in a matrix. The arguments are as follows:

x	specifies an $n \times p$ numerical or character matrix. The COUNTUNIQUE function counts the number of unique values in this matrix.
method	specifies the method used to count the missing values. This argument is optional. The following are valid values:

"all"	specifies that the function counts all unique values in the matrix. This is the default value. The function returns a 1×1 matrix.
"row"	specifies that the function counts the unique values in each row. The function returns an $n \times 1$ matrix whose ith element is the number of unique values in the ith row of *x*.
"col"	specifies that the function counts the unique values in each column. The function returns a $1 \times p$ matrix whose jth element is the number of unique values in the jth column of *x*.

The *method* argument is not case-sensitive. The first three characters are used to determine the value.

For example, the following statements count unique values for the matrix **x**:

```
x={1 2 3,
   1 1 2,
   1 1 1,
   1 0 0};
allUnique = countunique(x);
rowUnique = countunique(x, "ROW");
colUnique = countunique(x, "COL");
print allUnique, rowUnique, colUnique;
```

Figure 23.72 Counts of Unique Values

```
                                allUnique

                                        4

                                rowUnique

                                        3
                                        2
                                        1
                                        2

                                colUnique

                              1         3         4
```

COV Function

COV(*x* < , *excludemiss* > **);**

The COV function computes a sample variance-covariance matrix for data. The arguments are as follows:

x	specifies an $n \times p$ numerical matrix of data. The COV function computes a $p \times p$ variance-covariance matrix of the data.
excludemiss	specifies how missing values are handled. The following values are valid:

"listwise"	specifies that observations with missing values are excluded from the analysis. This is the default value.
"pairwise"	specifies that all nonmissing pairs of values for each pair of variables are included in the statistical computations.

The *excludemiss* argument is not case-sensitive. The first four characters are used to determine the value. For example, "LIST" and "listwise" specify the same option.

The COV function computes a sample variance-covariance matrix for data, as the following example shows:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
     7 2 13};
cov = cov(x);
print cov;
```

Figure 23.73 Variance-Covariance Matrix

cov		
0.5	0.25	0.75
0.25	10.5	-3.75
0.75	-3.75	16

The COV function handles missing values in the same way as the CORR procedure. For additional details, see the documentation for the CORR procedure (especially the NOMISS option) in the *Base SAS Procedures Guide: Statistical Procedures*.

It might be useful to use the ROWNAME= and COLNAME= options in the MATTRIB statement or the PRINT statement to associate names of variables to rows and columns of the correlation matrix, as shown in the example for the CORR function.

COVLAG Function

COVLAG(*x, k***);**

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 arguments 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 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 following statements produce a lagged crossproduct matrix:

```
x = T(do(-9, 9, 2));
cov = covlag(x, 4);
print cov;
```

Figure 23.74 Lagged Crossproduct Matrix

cov			
33	23.1	13.6	4.9

CREATE Statement

CREATE *SAS-data-set* <**VAR** *operand*> **;**

CREATE *SAS-data-set* **FROM** *matrix-name* <*[COLNAME=column-name ROWNAME=row-name]*> **;**

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.

The arguments to the CREATE statement are as follows:

SAS-data-set is the name of a SAS data set. It can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A).

operand specifies a set of existing SAS/IML matrices that contain data. The names of the matrices become the names of the data set variables.

matrix-name specifies a matrix that contains the data. Each column of the matrix produces a variable in the data set.

column-name is a character matrix or quoted literal that contains names of the data set variables.

row-name is a character matrix or quoted literal that contains text to associate with each observation in the data set.

You can specify a set of variables to use with the VAR clause. The *operand* argument can be specified in one of the following ways:

- a literal that contains variable names
- the name of a matrix that contains variable names
- an expression in parentheses that yields variable names
- one of the keywords in the following list:

ALL specifies that all variables in scope should be written to the data set.

CHAR specifies that all character variables in scope should be written to the data set.

NUM specifies that all numeric variables in scope should be written to the data set.

The following example demonstrates ways that you can use the VAR clause:

```
x1 = T(1:5);
x2 = T(5:1);
y = {-1,0,1,0,1};
z = {a,b,c,d,e};
create temp var {x1 y z};    /* a literal matrix of names      */
append;
close temp;

varNames = {"x1" "y" "z"};
create temp var varNames;    /* a matrix that contains names  */
append;
close temp;
free varNames;

create temp var ("x1":"x2"); /* an expression                  */
append;
close temp;

create temp var _all_;       /* all variables in scope         */
append;
close temp;
```

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 that contains labels. The operand must be a character matrix. The length of the resulting data set variable is the length of a matrix element of the operand. The same ROWNAME= matrix should be used in any subsequent APPEND statements for this data set.

The variable types and lengths correspond to the 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. If you do not specify the name of a variable, then all variables in scope are used.

To add observations to your data set, you must use the APPEND statement.

For example, the following statements create a new SAS data named Population that contains two numeric and two character variables:

```
State   = {"NC",       "NC",    "FL",      "FL"};
County  = {"Chatham", "Wake", "Orange", "Seminole"};
Pop2000 = {49329,      627846,  896344,  365196};
Pop2009 = {64772,      897214, 1086480,  413204};
create Population var {"State" "County" "Pop2000" "Pop2009"};
append;
close Population;
```

The data come from vectors with the same names. You must initialize the character variables (State and County) prior to calling the CREATE statement. The State variable has length 2 and the County variable has length 8. The Pop2000 and Pop2009 variables are numeric.

In the next example, you use the FROM clause with the COLNAME= option 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, as shown in the following statements:

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

If you associate a format with a matrix by using the MATTRIB statement, then the CREATE statement assigns that format to the corresponding variable in the data set, as shown in the following example:

```
proc iml;
date = { '20MAR2010'd, '20MAR2011'd, '20MAR2012'd,
         '20MAR2013'd, '20MAR2014'd, '20MAR2015'd };
mattrib date format=WORDDATE.;

/* time of equinox, GMT (Greenwich Mean Time) */
time = { '17:32't,     '23:21't,     '05:14't,
         '11:02't,     '16:57't,     '22:45't };
mattrib time format=TIMEAMPM.;

create MarchEquinox var {"Date" "Time"};
append;
close MarchEquinox;

proc print data=MarchEquinox;
run;
```

Figure 23.75 Data Set That Contains Formats

Obs	Date	Time
1	March 20, 2010	5:32:00 PM
2	March 20, 2011	11:21:00 PM
3	March 20, 2012	5:14:00 AM
4	March 20, 2013	11:02:00 AM
5	March 20, 2014	4:57:00 PM
6	March 20, 2015	10:45:00 PM

CSHAPE Function

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

The CSHAPE function changes the shape of a character matrix by redefining the matrix dimensions.

The arguments 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 an optional padding character.

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 characters in the source matrix are cycled through again. If a padding character was specified, the remaining characters are all the padding character.

If one of the size arguments (*nrow*, *ncol*, or *size*) is zero, the CSHAPE 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.

For example, the following statement produces a 2×2 matrix:

```
a = cshape("abcd", 2, 2, 1);
print a;
```

Figure 23.76 Reshaped Character Matrix

```
                                    a

                                  a b
                                  c d
```

The following statement rearranges the 12 characters in the input matrix into a 2 × 2 matrix with three characters in each element:

```
m = {"ab" "cd",
     "ef" "gh",
     "ij" "kl"};
b = cshape(m, 2, 2, 3);
print b;
```

Figure 23.77 Reshaped Character Matrix

```
                                    b

                                abc def
                                ghi jkl
```

The following statement uses the *size* argument to specify the length of the result matrix. Notice that the characters in the *matrix* argument are reused in order to form a 2 × 2 matrix with three characters in each element.

```
c = cshape("abcde", 2, 2, 3);
print c;
```

Figure 23.78 Reusing Characters

```
                                    c

                                abc dea
                                bcd eab
```

The next example is similar, except that the optional *padchar* argument is used to specify what character to use after the characters in the *matrix* argument are each used once:

```
d = cshape("abcde", 2, 2, 3, "*");
print d;
```

Figure 23.79 Using a Pad Character

```
                                    d

                                abc de*
                                *** ***
```

See also the description of the SHAPE function, which is used with numeric data.

CUSUM Function

CUSUM(*matrix***);**

The CUSUM function computes cumulative sums. The argument to this function 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 adding the nonmissing elements of the argument in row-major order.

For example, the following statements compute cumulative sums:

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

Figure 23.80 Cumulative Sums

```
                                a

                1               3               7              12

                                b

                                5              11
                               14              18
```

CUPROD Function

CUPROD(*matrix***);**

The CUPROD function computes cumulative products. The argument to this function is a numeric matrix or literal.

The CUPROD function returns a matrix of the same dimension as the argument matrix. The result contains the cumulative products obtained by multiplying the nonmissing elements of the argument in row-major order.

For example, the following statements compute cumulative products:

```
a = cuprod({1 2 4 5});
b = cuprod({5 6, . 4});
print a, b;
```

Figure 23.81 Output from the CUPROD Function

```
                              a

                1             2             8            40

                              b

                              5            30
                             30           120
```

CVEXHULL Function

CVEXHULL(*matrix***);**

The CVEXHULL function finds a convex hull of a set of planar points.

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

The CVEXHULL function returns an $n \times 1$ matrix of indices. The indices of points in the convex hull in counterclockwise 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 by using the LOC function. For example, the following statements find the index vector for the convex hull and print the associated points:

```
points = {0  2, 0.5 2, 1 2, 0.5 1, 0 0, 0.5 0, 1  0,
          2 -1,   2 0, 2 1,   3 0, 4 1,   4 0, 4 -1,
          5  2,   5 1, 5 0,   6 0 };
indices = cvexhull( points );
hullIndices = indices[loc(indices>0)];
convexHull = points[hullIndices, ];
print convexHull;
```

Figure 23.82 Convex Hull of a Planar Set of Points

```
                          convexHull

                             0            2
                             0            0
                             2           -1
                             4           -1
                             6            0
                             5            2
```

DATASETS Function

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

The DATASETS function returns a character matrix that contains 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, SAS/IML software uses the default libname. (See the DEFLIB= option in the description of the RESET statement.)

For more information about specifying a SAS data library, see Chapter 7.

Recall that SAS distributes sample data sets in the Sashelp library. The following statements list the names of the first few data sets in the library:

```
lib = "sashelp";
a = datasets(lib);
First5 = a[1:5];
print First5;
```

Figure 23.83 Several Data Sets in the Sashelp Library

```
First5

ACCEPTS
ADOMSG
ADSMSG
AFMSG
AIR
```

DELETE Call

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

The DELETE call deletes one or more SAS data sets. The arguments to the DELETE subroutine are as follows:

libname is a character matrix or quoted literal that contains the name of one or more SAS data libraries.

member-names is a character matrix or quoted literal that contains the names of one or more data sets.

The DELETE subroutine deletes SAS data sets in a specified library. If you omit the *libname* argument, the default SAS data library is used. (See the DEFLIB= option in the description of the RESET statement.)

The following statements use the DATA step to create several data sets and then delete them by using the DELETE subroutine in SAS/IML software:

```
data a b c d e;            /* create data sets in WORK */
   x=1;
run;
```

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

reset deflib=work;          /* sets default libname to WORK */
call delete(b);             /* deletes WORK.B */

members = {"c" "d"};
call delete(members);       /* deletes WORK.C and WORK.D */

ds = datasets("work");      /* returns all data sets in WORK */
call delete("work",ds[1]); /* deletes first data set */
```

DELETE Statement

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

The DELETE statement marks observations (also called records) in the current output data set for deletion. To actually delete the records and renumber the remaining observations, use the PURGE statement.

The arguments 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.

You can specify *range* by using a keyword or by using the POINT operand to specify an observation number. 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 *value* specifies observations by number, where *value* is one of the following:

Value	Example
a single record number	`delete point 5`
a literal that contains several record numbers	`delete point {2 5 10}`
the name of a matrix that contains record numbers	`p=5; delete point p;`
an expression in parentheses	`delete point (p+1);`

The default value for *range* is **CURRENT**. If the current data set has an index in use (see the INDEX statement, the **POINT** keyword 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* **;**

The arguments to the WHERE clause are as follows:

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 like a given string

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

The *operand* argument in a WHERE comparison can be a matrix. For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

=, ?, =:, =*

For the remaining operators, the WHERE clause succeeds provided that *all* the elements in the matrix satisfy the condition.

Logical expressions can be specified within the WHERE clause by using the AND (&) and OR (|) operators. If *clause* is a valid WHERE expression, then you can combine expressions as follows:

Both conditions satisfied	*clause1* & *clause2*
Either condition satisfied	*clause1* | *clause2*

The following statements show examples of using the DELETE statement:

```
proc iml;
/* create sample data set */
sex = { M,  M,  M,  F,  F,  F};
age = {34, 28, 38, 32, 24, 18};
create MyData var {"Sex" "Age"};
append;
close MyData;

/* delete observations in data set */
edit MyData;
delete;                    /* marks the current obs  */
```

```
delete point 3;              /* marks obs 3            */
delete all where(age<21);    /* marks obs where age<21 */
purge;                       /* deletes all marked obs */
close MyData;

proc print data=MyData;
run;
```

Figure 23.84 Observations That Remain after Deletion

Obs	Sex	Age
1	M	28
2	F	32
3	F	24

Notice that observations marked for deletion by using the DELETE statement are not physically removed from the data set until a PURGE statement is executed.

DESIGN Function

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

The DESIGN function creates a design matrix of zeros and ones from the column vector specified by *column-vector*. Each unique value of the column vector generates a column of the design matrix. The columns are arranged in the sort order of the original values. If x_i is the ith sorted value in the column vector, **x**, then the ith column of the design matrix contains ones in rows for which **x** has the value x_i, and contains zeros elsewhere.

For example, the following statements produce a design matrix for a column vector that contains three unique values. The first column corresponds to the 'A' level, the second column corresponds to the 'B' level, and the third column corresponds to the 'C' level.

```
x = {C, A, B, B, A, A};
m = design(x);

cols = unique(x);
print m[colname=cols];
```

Figure 23.85 Design Matrix for a Vector with Three Unique Values

m		
A	B	C
0	0	1
1	0	0
0	1	0
0	1	0
1	0	0
1	0	0

The design matrix that is returned by the DESIGN function corresponds to the GLM parameterization of classification variables as documented in the section "Parameterization of Model Effects" in the *SAS/STAT User's Guide*. See also the documentation for the DESIGNF function.

DESIGNF Function

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

The DESIGNF function creates a design matrix of zeros and ones from the column vector specified by *column-vector*. The DESIGNF function is similar to the DESIGN function. The difference is that the matrix returned by the DESIGNF function is one column smaller than the matrix returned by the DESIGN function. The result of the DESIGNF function is obtained by subtracting the last column of the DESIGN function matrix from the other columns.

For example, the following statements produce a design matrix for a column vector that contains three unique values:

```
x = {C, A, B, B, A, A};
m = designf(x);

cols = unique(x);
print m[colname=cols];
```

Figure 23.86 Design Matrix for Vector with Three Unique Values

m	
A	B
-1	-1
1	0
0	1
0	1
1	0
1	0

The matrix that is returned by the DESIGNF function can be used to produce full-rank designs. The ma-

trix corresponds to the EFFECT parameterization of classification variables as documented in the section "Parameterization of Model Effects" in the *SAS/STAT User's Guide*.

DET Function

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

The DET function computes the determinant of a square matrix. The determinant, the product of the eigenvalues, is a scalar numeric value. If the determinant of a matrix is zero, then the matrix is singular. A singular matrix does not have an inverse.

The DET function performs an LU decomposition and collects the product of the diagonals (Forsythe, Malcom, and Moler 1967). For a matrix with n rows, the DET function allocates a temporary n^2 array in order to compute the determinant.

The following statements compute the determinant of a matrix:

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

Figure 23.87 Determinant of a Matrix

d
2

The DET function uses a criterion to determine whether the input matrix is singular. See the INV function for details.

DIAG Function

DIAG(*matrix***);**

The DIAG function creates a diagonal matrix. The *matrix* argument can be either a numeric square matrix or a vector.

If *matrix* 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 *matrix* 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 following statements produce a diagonal matrix by extracting the diagonal elements of a square matrix:

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

Figure 23.88 Diagonal Matrix Obtained from a Full Matrix

c	
4	0
0	1

The following statements produce a diagonal matrix by using the elements of a vector:

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

Figure 23.89 Diagonal Matrix Obtained from a Vector

d		
1	0	0
0	2	0
0	0	3

DIF Function

DIF(*x* < , *lags* > **);**

The DIF function computes the differences between data values and one or more lagged (shifted) values for time series data. The arguments are as follows:

x	specifies a $n \times 1$ numerical matrix of time series data.
lags	specifies integer lags. The *lags* argument can be an integer matrix with d elements. If so, the DIF function returns an $n \times d$ matrix where the ith column represents the difference between the time series and the lagged data for the ith lag.

The values of the *lags* argument are usually positive integers. A positive lag shifts the time series data backwards in time. A lag of 0 represents the original time series. A negative value for the *lags* argument shifts the time series data forward in time; this is sometimes called a *lead* effect.

For example, the following statements compute the difference between the time series and the lagged data:

```
x = {1,3,4,7,9};
dif = dif(x, {0 1 3});
print dif;
```

Figure 23.90 Differences between Data and Lagged Data

dif		
0	.	.
0	2	.
0	1	.
0	3	6
0	2	6

DISPLAY Statement

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

The DISPLAY statement displays fields in windows. The arguments to the DISPLAY statement are as follows:

group-spec specifies a group. It can be specified as either a compound name of the form *window-name.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 causes the speaker in your workstation to beep when the window is displayed.

The DISPLAY statement directs PROC IML to gather data into fields defined in the window 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 16.

The DISPLAY statement is described in detail in Chapter 16, "Window and Display Features." Following are several examples that demonstrate the use of 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

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

The DO function creates a row vector that contains a sequence of evenly spaced numbers.

The arguments to the DO function are as follows:

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

The DO function creates a row vector that contains 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 (:).

The following statements show examples of using the DO function:

```
i = do(3, 18, 3);
k = do(3,  0, -1);
print i, k;
```

Figure 23.91 Aritmetic Sequences

```
                              i

          3         6         9        12        15        18

                              k

                    3         2         1         0
```

DO Statement

DO ;

statements **;**

END ;

The DO statement specifies that the statements that follow the DO statement be 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:

```
x=0;
y=1;
if x<y then
   do;
      z1 = abs(x+y);
      z2 = abs(x-y);
   end;
```

The statements between the DO and END statements (called the DO group) are executed only if **x < y**. That is, they are executed only if all elements of **x** are less than the corresponding elements of **y**. If any element of **x** is not less than the corresponding element of **y**, the statements in the DO group are skipped and the statement that follows the END statement is executed.

It is good practice to indent the statements in a DO group as shown in the preceding example. However, the DO and END statements do not need to be on separate lines. A popular indenting style is to write the DO statement on the same line as the THEN or ELSE clause, as shown in following statements:

```
if x<y then do;
   z1 = abs(x+y);
   z2 = abs(x-y);
end;
else do;
   z1 = abs(x-y);
   z2 = abs(x+y);
end;
```

DO groups can be nested. There is no limit imposed on the number of nested DO groups. The following statements show an example of nested DO groups:

```
if x<y then do;
   if z1>2 then do;
      z = z1 - z2;
      w = x # y;
   end;
end;
```

DO Statement, Iterative

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

The iterative DO statement executes a group of statements several times.

The arguments to the DO statement are as follows:

variable	is the name of a variable that indexes the loop. This variable is sometimes called an *index variable* or a *looping variable*.
start	is the starting value for *variable*.
stop	is the stopping value for *variable*.
increment	is an increment value.

The *start*, *stop*, and *increment* values should be scalars or expressions that yield scalars.

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

The index variable starts with the *start* value and is incremented by the *increment* value after each iteration. The iterations continue provided that the index variable is less than or equal to the *stop* value. If a negative increment is used, then iterations continue provided that the index variable is greater than or equal to the *stop* value. The *start*, *stop*, and *increment* expressions are evaluated only once before the looping starts.

For example, the following statements print the value of `i` three times, as shown in Figure 23.92:

```
do i = 1 to 5 by 2;
   print "The value of i is:" i;
end;
```

Figure 23.92 Aritmetic Sequences

```
                                        i

                The value of i is:      1

                                        i

                The value of i is:      3

                                        i

                The value of i is:      5
```

DO DATA Statement

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

The DO DATA statement repeats a loop until an end-of-file condition occurs.

The arguments to the DO DATA statement are as follows:

variable	is the name of a variable that indexes 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, the program immediately jumps from the group and starts executing the statement that follows 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.

The following statements read data from an external file. The example reads the first 100 lines or until the end of file, whichever occurs first.

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

If you are reading data from a SAS data set, then you can use the following statements:

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

DO Statement with an UNTIL Clause

DO UNTIL *(expression)* **;**

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

The DO UNTIL statement conditionally executes statements iteratively.

The arguments 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 that indexes 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 enables you to conditionally execute 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 following example, the body of the loop executes until the value of X exceeds 100:

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

Figure 23.93 Result of a DO-UNTIL Statement

x
101

DO Statement with a WHILE Clause

DO WHILE *(expression)* **;**

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

The DO WHILE statement executes statements iteratively while a condition is true.

The arguments 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 that indexes 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 enables you to conditionally execute 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 incrementing:

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

Figure 23.94 Result of a DO-WHILE Statement

x
100

The next example increments the starting value by 2:

```
y = 1;
do i = 1 to 100 by 2 while(y<200);
   y = y # i;
end;
print i y;
```

Figure 23.95 Result of an Iterative DO-WHILE Statement

i	y
11	945

DURATION Function

DURATION(*times, flows, ytm***);**

The DURATION function returns a scalar value that represents the modified duration of a noncontingent cash flow. The arguments are as follows:

times	is an n-dimensional column vector of times. The ith time corresponds to the time (often in years) until the ith cash flow occurs. Elements should be nonnegative.
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, and the kth cash flow is $c(k)$, t_k periods from the present. This measure is referred to as *modified duration* to differentiate it from the *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 (that is, timing) of all the cash flows and is hence an "average time-to-maturity" of the bond.

For example, consider the following statements:

```
times = 1;
flow = 10;
ytm = 0.1;
duration = duration(times, flow, ytm);
print duration;
```

Figure 23.96 Duration of a Cash Flow

duration
0.9090909

ECHELON Function

ECHELON(*matrix***);**

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

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

Figure 23.97 Result of the ECHELON Function

e		
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. See the HERMITE function for details about the Hermite normal form.

EDIT Statement

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

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.

The arguments to the EDIT statement are as follows:

SAS-data-set	can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). For more information about specifying SAS data sets, see 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.

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 that contains variable names

- the name of a matrix that contains variable names
- an expression in parentheses that yields variable names
- one of the keywords described in the following list:

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

The following examples demonstrate each possible way you can use the VAR clause:

```
var {x1 x5 x9};          /* a literal matrix of names         */
var x;                   /* a matrix that contains the names  */
var("x1":"x9");          /* 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)* **;**

The arguments to the WHERE clause are as follows:

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 like 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 by 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 by 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 7 for more information about editing SAS data sets.

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 which the Age variable is greater than 30, use the following statements:

```
proc iml;
/* create sample data set */
sex = { M,  M,  M,  F,  F,  F};
age = {34, 28, 38, 32, 24, 18};
create MyData var {"Sex" "Age"};
append;
close MyData;

edit MyData where (Age>30);
list all;
close MyData;
```

Figure 23.98 Result of the LIST Statement

```
   OBS Sex       Age
------ --- ---------
     1 M     34.0000
     3 M     38.0000
     4 F     32.0000
```

To edit the data set Work.MyData and obtain the number of observations in the data set, use the following statements:

```
edit Work.MyData nobs NumObs;
close Work.MyData;
```

```
print NumObs;
```

Figure 23.99 Number of Observations in a Data Set

NumObs
6

Another example of using the EDIT statement is presented in the documentation for the DELETE statement.

EIGEN Call

CALL EIGEN(*evals, evecs, A***) < VECL=***vl* **>;**

The EIGEN subroutine computes eigenvalues and eigenvectors an arbitrary square numeric matrix.

The *A* argument is the input argument to the EIGEN subroutine. The EIGEN call returns the following values:

evals	names a matrix to contain the eigenvalues of *A*.
evecs	names a matrix to contain the right eigenvectors of *A*.
vl	is an optional $n \times n$ matrix that contains the left eigenvectors of *A* in the same manner that *evecs* contains the right eigenvectors.

The EIGEN subroutine computes *evals*, a matrix that contains the eigenvalues of *A*. If *A* is symmetric, *evals* is the $n \times 1$ vector that contains the n real eigenvalues of *A*. If *A* is not symmetric (as determined by the criteria in the symmetry test described later), *evals* is an $n \times 2$ matrix. The first column of *evals* contains the real parts, $\mathrm{Re}(\lambda)$, and the second column contains the imaginary parts, $\mathrm{Im}(\lambda)$. Each row represents one eigenvalue, $\mathrm{Re}(\lambda) + i\,\mathrm{Im}(\lambda)$.

If *A* is symmetric, the eigenvalues are arranged in descending order. Otherwise, the eigenvalues are sorted first by their real parts, then by the magnitude of their imaginary parts. Complex conjugate eigenvalues, $\mathrm{Re}(\lambda) \pm i\,\mathrm{Im}(\lambda)$, 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 *evecs*, a matrix that contains the orthonormal column eigenvectors that correspond to *evals*. If *A* is symmetric, then the first column of *evecs* is the eigenvector that corresponds to the largest eigenvalue, and so forth. If *A* is not symmetric, then *evecs* is an $n \times n$ matrix that contains the right eigenvectors of *A*. If the eigenvalue in row i of *evals* is real, then column i of *evecs* contains the corresponding real eigenvector. If rows i and $i + 1$ of *evals* contain complex conjugate eigenvalues $\mathrm{Re}(\lambda) \pm i\,\mathrm{Im}(\lambda)$, then columns i and $i + 1$ of *evecs* contain the real part, $\mathbf{u}$, and imaginary part, $\mathbf{v}$, of the two corresponding eigenvectors $\mathbf{u} \pm i\,\mathbf{v}$.

The following paragraphs present some properties of eigenvalues and eigenvectors. Let $\mathbf{A}$ be a general $n \times n$ matrix. The eigenvalues of $\mathbf{A}$ 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 **A** is defined by

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

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

An eigenvector is a nonzero vector, **x**, that satisfies $\mathbf{Ax} = \lambda\mathbf{x}$ for $\lambda \in \lambda(\mathbf{A})$. Right eigenvectors satisfy $\mathbf{Ax} = \lambda\mathbf{x}$, and left eigenvectors satisfy $\mathbf{x}^H\mathbf{A} = \lambda\mathbf{x}^H$, where $\mathbf{x}^H$ is the complex conjugate transpose of **x**. Taking the conjugate transpose of both sides shows that left eigenvectors also satisfy $\mathbf{A}'\mathbf{x} = \bar{\lambda}\mathbf{x}$.

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

- The eigenvalues of an unsymmetric matrix **A** can be complex. If **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 **A**.
- The right and left eigenvectors that correspond to a real eigenvalue of **A** are real. The right and left eigenvectors that correspond to conjugate complex eigenvalues of **A** are also conjugate complex.
- The left eigenvectors of **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 **A**:

1. Select the entry of **A** with the largest magnitude:

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

2. Multiply the value of a_{max} by 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 **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 **A** is a *symmetric* matrix and **M** and **E** are the eigenvalues and eigenvectors, respectively, of **A**, then the matrices have the following properties:

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

These properties imply the following:

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

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

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 by using the following statements:

```
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**, as shown in the following statement:

```
r = einv*H*V - V*diag(L);
```

The values in **r** should be of the order of rounding error.

The following statements compute the eigenvalues and left and right eigenvectors of a nonsymmetric matrix with four real and four complex eigenvalues:

```
A = {-1  2  0       0       0       0       0  0,
     -2 -1  0       0       0       0       0  0,
      0  0  0.2379  0.5145  0.1201  0.1275  0  0,
      0  0  0.1943  0.4954  0.1230  0.1873  0  0,
      0  0  0.1827  0.4955  0.1350  0.1868  0  0,
      0  0  0.1084  0.4218  0.1045  0.3653  0  0,
      0  0  0       0       0       0       2  2,
      0  0  0       0       0       0      -2  0 };
call eigen(val, rvec, A) vecl="lvec";
print val;
```

The sorted eigenvalues of the **A** matrix are shown in Figure 23.100.

Figure 23.100 Complex Eigenvalues of a Nonsymmetric Matrix

val
1 1.7320508
1 −1.732051
1 0
0.2087788 0
0.0222025 0
0.0026187 0
−1 2
−1 −2

You can verify the correctness of the left and right eigenvector computation by using the following statements:

```
/* verify that the right eigenvectors are correct */
vec = rvec;
do j = 1 to ncol(vec);
  /* if eigenvalue is real */
  if val[j,2] = 0. then do;
```

```
    v = A * vec[,j] - val[j,1] * vec[,j];
    if any( abs(v) > 1e-12 ) then
      badVectors = badVectors || j;
    end;
  /* if eigenvalue is complex with positive imaginary part */
  else if val[j,2] > 0. then do;
    /* the real part */
    rp = val[j,1] * vec[,j] - val[j,2] * vec[,j+1];
    v = A * vec[,j] - rp;
    /* the imaginary part */
    ip = val[j,1] * vec[,j+1] + val[j,2] * vec[,j];
    u = A * vec[,j+1] - ip;
    if any( abs(u) > 1e-12 ) | any( abs(v) > 1e-12 ) then
      badVectors = badVectors || j || j+1;
    end;
  end;
if ncol( badVectors ) > 0 then
  print "Incorrect right eigenvectors:" badVectors;
else print "All right eigenvectors are correct";
```

Similar statements can be written to verify the left eigenvectors. The statements use the fact that the left eigenvectors of $\mathbf{A}$ are the same as the complex conjugate right eigenvectors of $\mathbf{A}'$:

```
/* verify that the left eigenvectors are correct */
vec = lvec;
do j = 1 to ncol(vec);
  /* if eigenvalue is real */
  if val[j,2] = 0. then do;
    v = A` * vec[,j] - val[j,1] * vec[,j];
    if any( abs(v) > 1e-12 ) then
      badVectors = badVectors || j;
    end;
  /* if eigenvalue is complex with positive imaginary part */
  else if val[j,2] > 0. then do;
    /* Note the use of complex conjugation */
    /* the real part */
    rp = val[j,1] * vec[,j] + val[j,2] * vec[,j+1];
    v = A` * vec[,j] - rp;
    /* the imaginary part */
    ip = val[j,1] * vec[,j+1] - val[j,2] * vec[,j];
    u = A` * vec[,j+1] - ip;
    if any( abs(u) > 1e-12 ) | any( abs(v) > 1e-12 ) then
      badVectors = badVectors || j || j+1;
    end;
  end;
if ncol( badVectors ) > 0 then
  print "Incorrect left eigenvectors:" badVectors;
else print "All left eigenvectors are correct";
```

The EIGEN call performs most of its computations in the memory allocated for returning the eigenvectors.

EIGVAL Function

EIGVAL(*A*);

The EIGVAL function computes the eigenvalues of a square numeric matrix, *A*. The EIGVAL function returns a column vector that contains the eigenvalues of *A*. See the description of the EIGEN subroutine for more details.

The following statements compute Example 7.1.1 from Golub and Van Loan (1989):

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

val = eigval(A);
print val;
```

Figure 23.101 Eigenvalues

val	
75	100
75	−100
25	0

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

If a matrix is symmetric, it has real eigenvalues and real eigenvectors. The following statements produce the eigenvalues of a crossproducts matrix:

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

Figure 23.102 Real Eigenvalues of a Symmetric Matrix

rval
33.401219
0.5987805

EIGVEC Function

EIGVEC(*A*);

The EIGVEC function computes the (right) eigenvectors of a square numeric matrix, *A*. You can obtain the left eigenvectors by first transposing *A*. See the description of the EIGEN subroutine for more details.

The following example calculates the eigenvectors of a symmetric matrix:

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

Figure 23.103 Eigenvectors of a Symmetric Matrix

eval
0.3220062 0.9467376
0.9467376 −0.322006

ELEMENT Function

ELEMENT(*x, y***);**

The ELEMENT function returns a matrix that is the same shape as *x*. The return value indicates which elements of *x* are elements of *y*. In particular, if **A = element(x, y)**, then

$$A_i = \begin{cases} 1 & \text{if } x_i \in y \\ 0 & \text{otherwise} \end{cases}$$

The arguments are as follows:

x specifies a matrix of elements to test for membership.

y specifies a set.

If the intersection between *x* and *y* is empty, then the ELEMENT function returns a zero matrix. If *x* is a proper subset of *y*, then the ELEMENT function returns a matrix of ones. In general, the ELEMENT function returns 1 for elements in the intersection of *x* and *y*, as shown in the following statements:

```
x = {0, 0.5, 1, 1.5, 2, 2.5, 3, 0.5, 1.5, 3, 3, 1};
set = {0 1 3}`;
b = element(x, set);

n = sum(b);         /* number of elements of X that are in SET */
idx = t(loc(b));    /* indices of elements of X that are in SET */
values = x[idx];    /* values of elements of X that are in SET */
print n idx values;
```

Figure 23.104 Elements That Belong to a Set

n	idx	values
6	1	0
	3	1
	7	3
	10	3
	11	3
	12	1

END Statement

END ;

The END statement ends a DO loop or DO statement. See the DO statement for details.

ENDSUBMIT Statement

ENDSUBMIT ;

You can use the ENDSUBMIT statement in conjunction with the SUBMIT statement to submit SAS statements for processing from within a SAS/IML program. All statements between the SUBMIT and the ENDSUBMIT statements are referred to as a *SUBMIT block*. The SUBMIT block is processed by the SAS language processor.

If you use the R option in the SUBMIT statement, you can submit statements to the R statistical software for processing.

The ENDSUBMIT statement must appear on a line by itself.

See Chapter 11, "Calling Functions in the R Language," and the documentation for the SUBMIT statement for details and examples.

EXECUTE Call

CALL EXECUTE(*statements***);**

The EXECUTE subroutine immediately executes SAS statements. These can be SAS/IML statements or global SAS statements such as the TITLE statement. The arguments to the EXECUTE subroutine are character matrices or quoted literals that contains valid SAS statements. You can specify up to 15 arguments.

The EXECUTE subroutine pushes character arguments to the input command stream, executes them, and then returns to the calling module. The subroutine should be called from a module rather than from the

immediate environment because it uses the RESUME statement that works only from modules). The strings you push do not appear in the log.

Following are examples of valid EXECUTE subroutines:

```
/* define a module that writes data to a specified data set */
start WriteData(DSName, x);
  CreateStmt = "create " + DSName + " from x;"; /* build CREATE stmt */
  call execute(CreateStmt);                     /* run CREATE stmt */
  append from x;
  CloseStmt = "close " + DSName + ";";          /* build CLOSE stmt */
  call execute(CloseStmt);                      /* run CLOSE stmt */
finish;

y = {1 2 3, 4 5 6, 7 8 0};
run WriteData("MyData", y);                     /* call the module */

use MyData; list all; close MyData;             /* verify contents */
```

Figure 23.105 Results of Executing SAS/IML Statements

OBS	COL1	COL2	COL3
1	1.0000	2.0000	3.0000
2	4.0000	5.0000	6.0000
3	7.0000	8.0000	0

For more details about the EXECUTE subroutine, see Chapter 18, "Using SAS/IML Software to Generate SAS/IML Statements."

EXP Function

EXP(*matrix***);**

The EXP function applies the exponential function to every element of the argument matrix. The exponential is the natural number *e* raised to the indicated power. For example, the following statements compute the exponentials of several numbers:

```
b = {1 2 3 4};
a = exp(b);
print a;
```

Figure 23.106 Exponential of Several Numbers

a			
2.7182818	7.3890561	20.085537	54.59815

If you want to compute the exponential of a matrix, you can call the EXPMATRIX module in IMLMLIB.

EXPORTDATASETTOR Call

CALL EXPORTDATASETTOR(*SAS-data-set, RDataFrame***);**

You can use the EXPORTDATASETTOR subroutine to transfer data from a SAS data set to an R data frame. It is easier to read the subroutine name when it is written in mixed case: ExportDataSetToR.

The arguments to the subroutine are as follows:

SAS-data-set is a literal string or a character matrix that specifies the two-level name of a SAS data set (for example, Sashelp.Class).

RDataFrame is a literal string or a character matrix that specifies the name of an R data frame.

You can call the subroutine provided that the following statements are true:

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See "The RLANG System Option" on page 190.)

The following statements copy data from the Sashelp.Class data set into an R data frame called **class**:

```
proc iml;
call ExportDataSetToR("Sashelp.Class", "class");

submit / R;
names( class )
endsubmit;
```

To demonstrate that the data were successfully transferred, the **names** function in the R language is used to print the names of the variables in the R data frame. The output is shown in Figure 23.107.

Figure 23.107 Output from R

```
[1] "Name"    "Sex"     "Age"     "Height" "Weight"
```

You can transfer data from an R data frame into a SAS data set by using the IMPORTDATASETFROMR call. See Chapter 11, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

EXPORTMATRIXTOR Call

CALL EXPORTMATRIXTOR(*IMLMatrix, RMatrix***);**

You can use the EXPORTMATRIXTOR subroutine to transfer data from a SAS data set to an R data frame. It is easier to read the subroutine name when it is written in mixed case: ExportMatrixToR.

The arguments to the subroutine are as follows:

IMLMatrix is a SAS/IML matrix that contains the data you want to transfer.

RMatrix is a literal string or a character matrix that specifies the name of an R matrix to contain a copy of the data.

You can call the subroutine provided that the following statements are true:

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See "The RLANG System Option" on page 190.)

The following statements define a SAS/IML matrix and copy the data from the matrix to an R matrix called **m**:

```
proc iml;
a = {1 2 3, 4 . 6};
call ExportMatrixToR(a, "m");

submit / R;
print(m)
endsubmit;
```

To demonstrate that the data were successfully transferred, the **print** function in the R language is used to print the values of the **m** matrix. The output is shown in Figure 23.108. Note that the SAS missing value in the SAS/IML matrix was automatically converted to the R missing value (**NA**).

Figure 23.108 Output from R

```
     A1 A2 A3
[1,]  1  2  3
[2,]  4 NA  6
```

You can transfer data from an R matrix frame into a SAS/IML matrix by using the IMPORTMATRIX-FROMR call. See Chapter 11, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

The names of the variables in the R data frame are the same as in the SAS data set.

FARMACOV Call

CALL FARMACOV(*cov, d* < , *phi*> < , *theta*> < , *sigma*> < , *p*> < , *q*> < , *lag*> **);**

The FARMACOV subroutine computes the autocovariance function for an autoregressive fractionally integrated moving average (ARFIMA) model of the form ARFIMA(p, d, q).

The input arguments to the FARMACOV subroutine are as follows:

d specifies a fractional differencing order. The value of d must be in the open interval $(-0.5, 0.5)$ excluding zero. This input is required.

phi specifies an m_p-dimensional vector that contains the autoregressive coefficients, where m_p is the number of the elements in the subset of the AR order. The default is zero. All the roots of $\phi(B) = 0$ should be greater than one in absolute value, where $\phi(B)$ is the finite-order matrix polynomial in the backshift operator B, such that $B^j y_t = y_{t-j}$.

theta specifies an m_q-dimensional vector that contains the moving average coefficients, where m_q is the number of the elements in the subset of the MA order. The default is zero.

p specifies the subset of the AR order. The quantity m_p is defined as the number of elements of *phi*.

If you do not specify p, the default subset is $p = \{1, 2, \ldots, m_p\}$.

For example, consider *phi*=0.5.

If you specify p=1 (the default), the FARMACOV subroutine computes the theoretical autocovariance function of an ARFIMA$(1, d, 0)$ process as $y_t = 0.5\ y_{t-1} + \epsilon_t$.

If you specify p=2, the FARMACOV subroutine computes the autocovariance function of an ARFIMA$(2, d, 0)$ process as $y_t = 0.5\ y_{t-2} + \epsilon_t$.

q specifies the subset of the MA order. The quantity m_q is defined as the number of elements of *theta*.

If you do not specify q, The default subset is $q = \{1, 2, \ldots, m_q\}$.

The usage of q is the same as that of p.

lag specifies the length of lags, which must be a positive number. The default is *lag*$= 12$.

The FARMACOV subroutine returns the following value:

cov is a *lag*+1 vector that contains the autocovariance function of an ARFIMA(p, d, q) process.

As an example, consider the following ARFIMA$(1, 0.3, 1)$ process:

$$(1 - 0.5B)(1 - B)^{0.3} y_t = (1 + 0.1B)\epsilon_t$$

In this process, $\epsilon_t \sim NID(0, 1.2)$. The following statements compute the autocovariance of this process:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
sigma = 1.2;
```

```
call farmacov(cov, d, phi, theta, sigma) lag=5;
print cov;
```

Figure 23.109 Autocovariance of an ARFIMA Process

cov
4.2493033
3.5806774
2.9152846
2.4381017
2.1068697
1.8743199

For $d \in (-0.5, 0.5)\backslash\{0\}$, the series y_t represented as $(1-B)^d y_t = \epsilon_t$ is a stationary and invertible ARFIMA$(0, d, 0)$ process with the autocovariance function

$$\gamma_k = \mathrm{E}(y_t y_{t-k}) = \frac{(-1)^k \Gamma(-2d+1)}{\Gamma(k-d+1)\Gamma(-k-d+1)}$$

and the autocorrelation function

$$\rho_k = \frac{\gamma_k}{\gamma_0} = \frac{\Gamma(-d+1)\Gamma(k+d)}{\Gamma(d)\Gamma(k-d+1)} \sim \frac{\Gamma(-d+1)}{\Gamma(d)} k^{2d-1}, \quad k \to \infty$$

Notice that ρ_k decays hyperbolically as the lag increases, rather than showing the exponential decay of the autocorrelation function of a stationary ARMA(p, q) process.

For $d \in (0.5, 0.5)\backslash\{0\}$, the series y_t is a stationary and invertible ARFIMA(p, d, q) process represented as

$$\phi(B)(1-B)^d y_t = \theta(B)\epsilon_t$$

where $\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p$ and $\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q$ and ϵ_t is a white noise process; all the roots of the characteristic AR and MA polynomial lie outside the unit circle.

Let $x_t = \theta(B)^{-1}\phi(B) y_t$, so that x_t follows an ARFIMA$(0, d, 0)$ process; let $z_t = (1-B)^d y_t$, so that z_t follows an ARMA(p, q) process; let γ_k^x be the autocovariance function of $\{x_t\}$ and γ_k^z be the autocovariance function of $\{z_t\}$.

Then the autocovariance function of $\{y_t\}$ is as follows:

$$\gamma_k = \sum_{j=-\infty}^{j=\infty} \gamma_j^z \gamma_{k-j}^x$$

The explicit form of the autocovariance function of $\{y_t\}$ is given by Sowell (1992).

FARMAFIT Call

CALL FARMAFIT(*d, phi, theta, sigma, series* **<** *, p* **>** **<** *, q* **>** **<** *, opt* **>** **);**

The FARMAFIT subroutine estimates the parameters of an ARFIMA(p, d, q) model.

The input arguments to the FARMAFIT subroutine are as follows:

series specifies a time series (assuming mean zero).

p specifies the set or subset of the AR order. If you do not specify *p*, the default is $p = 0$.

If you specify *p*=3, the FARMAFIT subroutine estimates the coefficient of the lagged variable y_{t-3}.

If you specify *p*={1, 2, 3}, the FARMAFIT subroutine estimates the coefficients of lagged variables y_{t-1}, y_{t-2}, and y_{t-3}.

q specifies the subset of the MA order. If you do not specify *q*, the default value is 0.

If you specify *q*=2, the FARMAFIT subroutine estimates the coefficient of the lagged variable ϵ_{t-2}.

If you specify *q*={1, 2}, the FARMAFIT subroutine estimates the coefficients of lagged variables ϵ_{t-1} and ϵ_{t-2}.

opt specifies the method of computing the log-likelihood function.

- 0 requests the conditional sum of squares function. This is the default.
- 1 requests the exact log-likelihood function. This option requires that the time series be stationary and invertible.

The FARMAFIT subroutine returns the following values:

d is a scalar that contains a fractional differencing order.

phi is a vector that contains the autoregressive coefficients.

theta is a vector that contains the moving average coefficients.

sigma is a scalar that contains a variance of the innovation series.

As an example, consider the following ARFIMA(1, 0.3, 1) model:

$$(1 - 0.5B)(1 - B)^{0.3} y_t = (1 + 0.1B)\epsilon_t$$

In this model, $\epsilon_t \sim NID(0, 1)$. The following statements estimate the parameters of this model:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
call farmasim(yt, d, phi, theta) seed=1234;
call farmafit(d, ar, ma, sigma, yt) p=1 q=1;
print d ar ma sigma;
```

Figure 23.110 Parameter Estimates for a ARFIMA Model

d	ar	ma	sigma
0.3950157	0.5676217	-0.012339	1.2992989

The FARMAFIT subroutine estimates the parameters d, $\phi(B)$, $\theta(B)$, and σ_ϵ^2 of an ARFIMA(p, d, q) model. The log-likelihood function is solved by iterative numerical procedures such as the quasi-Newton optimization. The starting value d is obtained by the approach of Geweke and Porter-Hudak (1983); the starting values of the AR and MA parameters are obtained from the least squares estimates.

FARMALIK Call

CALL FARMALIK(*lnl, series, d* < *, phi*> < *, theta*> < *, sigma*> < *, p*> < *, q*> < *, opt*> **);**

The FARMALIK subroutine evaluates the log-likelihood function of an ARFIMA(p, d, q) model for a given time series.

The input arguments to the FARMALIK subroutine are as follows:

series specifies a time series (assuming mean zero).

d specifies a fractional differencing order. This argument is required; the value of d should be in the open interval $(-1, 1)$ excluding zero.

phi specifies an m_p-dimensional vector that contains the autoregressive coefficients, where m_p is the number of the elements in the subset of the AR order. The default is zero.

theta specifies an m_q-dimensional vector that contains the moving average coefficients, where m_q is the number of the elements in the subset of the MA order. The default is zero.

sigma specifies a variance of the innovation series. The default is one.

p specifies the subset of the AR order. See the FARMACOV subroutine for additional details.

q specifies the subset of the MA order. See the FARMACOV subroutine for additional details.

opt specifies the method of computing the log-likelihood function. The following are valid values:

- 0 requests the conditional sum of squares function. This is the default.
- 1 requests the exact log-likelihood function. This option requires that the time series be stationary and invertible.

The FARMALIK subroutine returns the following value:

lnl is a three-dimensional vector. If *opt*= 0 is specified, the conditional sum of squares function is evaluated and the result returns in `lnl[1]`. Otherwise, `lnl[1]` contains the log-likelihood function of the model; `lnl[2]` contains the sum of the log determinant of the innovation variance; and `lnl[3]` contains the weighted sum of squares of residuals. The log-likelihood function is computed as $-0.5\times$ (`lnl[2]+lnl[3]`).

As an example, consider the following ARFIMA$(1, 0.3, 1)$ model:

$$(1 - 0.5B)(1 - B)^{0.3} y_t = (1 + 0.1B)\epsilon_t$$

In this model, $\epsilon_t \sim NID(0, 1.2)$. The following statements compute the log-likelihood function of this model:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
sigma = 1.2;
call farmasim(yt, d, phi, theta, sigma) seed=1234;
call farmalik(lnl, yt, d, phi, theta, sigma);
print (lnl[1])[label="Conditional Sum of Squares"];
```

Figure 23.111 Log-Likelihood for an ARFIMA Model

Conditional Sum of Squares
-16.67587

The FARMALIK subroutine computes a log-likelihood function of the ARFIMA(p, d, q) model. The exact log-likelihood function was proposed by Sowell (1992); the conditional sum of squares function was proposed by Chung (1996).

The exact log-likelihood function only considers a stationary and invertible ARFIMA(p, d, q) process with $d \in (-0.5, 0.5) \backslash \{0\}$ represented as

$$\phi(B)(1-B)^d y_t = \theta(B)\epsilon_t$$

where $\epsilon_t \sim NID(0, \sigma^2)$.

Let $Y_T = [y_1, y_2, \ldots, y_T]'$ and the log-likelihood function is as follows without a constant term:

$$\ell = -\frac{1}{2}(\log|\Sigma| + Y_T'\Sigma^{-1}Y_T)$$

where $\Sigma = \left[\gamma_{i-j}\right]$ for $i, j = 1, 2, \ldots, T$.

The conditional sum of squares function does not require the normality assumption. The initial observations $y_0, y_{-1}, \ldots$ and $\epsilon_0, \epsilon_{-1}, \ldots$ are set to zero.

Let y_t be an ARFIMA(p, d, q) process represented as

$$\phi(B)(1-B)^d y_t = \theta(B)\epsilon_t$$

Then the conditional sum of squares function is

$$\ell = -\frac{T}{2}\log\left(\frac{1}{T}\sum_{t=1}^{T}\epsilon_t^2\right)$$

FARMASIM Call

CALL FARMASIM(*series, d <, phi> <, theta> <, mu> <, sigma> <, n> <, p> <, q> <, initial> <, seed>* **);**

The FARMASIM subroutine generates an ARFIMA(p, d, q) process. The input arguments to the FARMASIM subroutine are as follows:

d specifies a fractional differencing order. This argument is required; the value of d should be in the open interval $(-1, 1)$ excluding zero.

phi specifies an m_p-dimensional vector that contains the autoregressive coefficients, where m_p is the number of the elements in the subset of the AR order. The default is zero.

theta specifies an m_q-dimensional vector that contains the moving average coefficients, where m_q is the number of the elements in the subset of the MA order. The default is zero.

mu specifies a mean value. The default is zero.

sigma specifies a variance of the innovation series. The default is one.

n specifies the length of the series. The value of n should be greater than or equal to the AR order. The default is $n = 100$ is used.

p specifies the subset of the AR order. See the FARMACOV subroutine for additional details.

q specifies the subset of the MA order. See the FARMACOV subroutine for additional details.

initial specifies the initial values of random variables. The initial value is used for the nonstationary process. If *initial*$= a_0$, then $y_{-p+1}, \ldots, y_0$ take the same value a_0. If the *initial* option is not specified, the initial values are set to zero.

seed is a scalar that contains the random number seed. At the first execution of the subroutine, the seed variable is used as follows:

If *seed* > 0, the input seed is used for generating the series.

If *seed* = 0, the system clock is used to generate the seed.

If *seed* < 0, the value $(-1)\times$(*seed*) is used for generating the series.

If the seed is not supplied, the system clock is used to generate the seed.

On subsequent calls of the subroutine in the DO-loop-like environment, the seed variable is used as follows: If seed > 0, the seed remains unchanged. In other cases, after each execution of the subroutine, the current seed is updated internally.

The FARMASIM subroutine returns the following value:

series is an n vector that contains the generated ARFIMA(p, d, q) process.

As an example, consider the following ARFIMA$(1, 0.3, 1)$ process:

$$(1 - 0.5B)(1 - B)^{0.3}(y_t - 10) = (1 + 0.1B)\epsilon_t$$

In this process, $\epsilon_t \sim NID(0, 1.2)$. The following statements generate this process:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
mu    = 10;
sigma = 1.2;
call farmasim(yt, d, phi, theta, mu, sigma, 10) seed=1234;
print yt;
```

Figure 23.112 Data Simulated from a ARFIMA Process

yt
12.17358
13.954495
15.817231
15.94882
12.25926
13.641022
13.399623
11.930759
10.049435
9.1445036

The FARMASIM subroutine generates a time series of length n from an ARFIMA(p, d, q) model. If the process is stationary and invertible, the initial values $y_{-p+1}, \ldots, y_0$ are produced by using covariance matrices obtained from FARMACOV. If the process is nonstationary, the time series is recursively generated by using the user-defined initial value or the zero initial value.

To generate an ARFIMA(p, d, q) process with $d \in [0.5, 1)$, x_t is first generated for $d' \in (-0.5, 0)$, where $d' = d - 1$ and then y_t is generated by $y_t = y_{t-1} + x_t$.

To generate an ARFIMA(p, d, q) process with $d \in (-1, -0.5]$, a two-step approximation based on a truncation of the expansion $(1 - B)^d$ is used; the first step is to generate an ARFIMA$(0, d, 0)$ process $x_t = (1 - B)^{-d}\epsilon_t$, with truncated moving average weights; the second step is to generate $y_t = \phi(B)^{-1}\theta(B)x_t$.

FDIF Call

CALL FDIF(*out, series, d***);**

The FDIF subroutine computes a fractionally differenced process. The input arguments to the FDIF subroutine are as follows:

series specifies a time series with n length.

d specifies a fractional differencing order. This argument is required; the value of d should be in the open interval $(-1, 1)$ excluding zero.

The FDIF subroutine returns the following value:

out is an n vector that contains the fractionally differenced process.

As an example, consider an ARFIMA$(1, 0.3, 1)$ process

$$(1 - 0.5B)(1 - B)^{0.3} y_t = (1 + 0.1B)\epsilon_t$$

Let $z_t = (1 - B)^{0.3} y_t$; that is, z_t follows an ARMA(1,1). The following statements compute the filtered series z_t:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
call farmasim(yt, d, phi, theta) n=10 seed=1234;
call fdif(zt, yt, d);
print zt;
```

Figure 23.113 A Fractionally Differenced Process

zt
.
3.0146839
4.0190575
3.3402864
-0.41881
1.6149336
1.1998534
-0.137789
-1.475051
-1.670366

FFT Function

FFT(*x*);

The FFT function computes the finite Fourier transform. The argument *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. This is 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).

As an example, suppose you measure a signal at constant time intervals. You believe the signal consists of a small number of Fourier components (that is, sines and cosines) corrupted by noise. The following examples uses the FFT function to transform the signal into the frequency domain. The example then prints the frequencies with the largest amplitudes in the signal. According to this analysis, the signal is primarily composed of a constant signal, a signal with frequency 4 (for example, $A\cos(4t) + B\sin(4t)$), a signal with frequency 1, and a signal with frequency 3. The amplitudes of the remaining Fourier components, are all substantially smaller.

```
Signal = {
 1.96  1.45  0.86  0.46  0.39  0.54 -1.65  0.60  0.43  0.20
-1.15  1.10  0.42  3.22  2.02  3.41  3.46  3.51  4.33  4.38
 3.92  4.35  2.60  3.95  4.72  4.84  1.62  0.97  0.96  1.10
 2.53  1.09  2.84  2.51  2.38  2.40  2.76  3.42  3.78  4.08
 3.84  5.62  4.33  6.66  5.27  3.14  3.82  5.74  3.45  1.07
 0.31  2.07  0.49 -1.85  0.61  0.35 -0.89 -0.92  0.33  2.31
 1.13  2.28  3.73  3.78  2.63  4.15  5.27  3.62  5.99  3.79
 4.00  3.18  3.03  3.52  2.08  1.70 -1.50 -1.35 -0.34 -1.52
-2.37 -2.84 -1.68 -2.22 -2.49 -3.28 -2.12 -0.81  0.84  1.91
 2.10  2.24  1.24  3.24  2.89  3.14  4.21  2.65  4.67  3.87
 }`;

z = fft(Signal);
Amplitude = z[,1]##2 + z[,2]##2;

/* find index into Amplitude so that idx[1] is the largest
   value, idx[2] is the second largest value, etc. */
call sortndx(idx,Amplitude,1,1);

/* print the 10 most dominant frequencies */
Amplitude = Amplitude[idx[1:10],];
print (idx[1:10]-1)[label="Freqs"] Amplitude[format=10.2];
```

Figure 23.114 Frequencies and Amplitudes of a Signal

Freqs	Amplitude
0	38757.80
4	13678.28
1	4077.99
3	2726.76
26	324.23
44	269.48
12	224.09
20	217.35
11	202.30
23	201.05

Based on these results, you might choose to filter the signal to keep only the most dominant Fourier com-

ponents. One way to accomplish this is to eliminate any frequencies with small amplitudes. When the truncated frequencies are transformed back by using IFFT, you obtain a filtered version of the original signal. The following statements perform these tasks:

```
/* based on amplitudes, keep only first few dominant frequencies */
NumFreqs = 4;
FreqsToDrop = idx[(NumFreqs+1):nrow(idx)];
z[FreqsToDrop,] = 0;

FilteredSignal = ifft(z) / nrow(Signal);
```

FILE Statement

FILE *filename* <**RECFM**=*N*> <**LRECL**=*operand*> **;**

The FILE statement opens an external file for output.

The arguments to the FILE statement are as follows:

filename	is a name (for defined filenames), a quoted literal, or an expression in parentheses (for pathnames).
RECFM=N	specifies that the file 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 8.

The *filename* argument is either a predefined filename or a quoted string or character expression in parentheses referring to the pathname. You can refer to an input or output file two ways: by a pathname or by a filename. The pathname 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 pathname by a previously issued FILENAME statement. However, two special filenames are recognized by the SAS/IML language: LOG and PRINT. These refer to the standard output streams for all SAS sessions. To specify a pathname, enclose it in quotes or specify an expression in parentheses that yields the pathname.

When the pathname is specified, the operand is limited to 64 characters.

Note that RECFM=U is equivalent to RECFM=N. If an output file is subsequently read by a SAS DATA step, RECFM=N must be specified in the DATA step to guarantee that the file is read properly.

Following are several valid uses of FILE statement:

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

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

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

The FIND statement finds the observation numbers 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.

The arguments 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.

You can use any of the following keywords to specify the *range* of observations:

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 *value*	observations specified by number, where *value* is one of the following:

Value	Example
A single record number	`point 5`
A literal that contains several record numbers	`point {2 5 10}`
The name of a matrix that contains record numbers	`point p`
An expression in parentheses	`point (p+1)`

If the current data set has an index in use (see the INDEX statement), 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)* **;**

The arguments to the WHERE clause are as follows:

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 like 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 by 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 by 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;
```

The column vectors **p** and **p2** contain the observation numbers that satisfy the WHERE clause in the given range. The default range is all observations.

FINISH Statement

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

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 6 for further information about defining modules.

Some examples follow:

```
start myAdd(a,b);
   return (a+b);
finish;

start mySubtract(a,b);
   return (a-b);
finish mySubtract;

r = myAdd(5, 3);
s = mySubtract(5, 3);
print r s;
```

Figure 23.115 Results of Calling Modules

r	s
8	2

FORCE Statement

The FORCE statement is an alias for the SAVE statement.

FORWARD Function

FORWARD(*times, spot_rates***);**

The FORWARD function computes a column vector of (per-period) forward rates, given vectors of spot rates and times. The arguments to the function are as follows:

times	is an $n \times 1$ column vector of times in consistent units. Elements should be nonnegative.
spot_rates	is an $n \times 1$ column vector of corresponding (per-period) spot rates. Elements should be positive.

The FORWARD function transforms the given spot rates as

$$f_1 = s_1$$

$$f_i = \left(\frac{(1+s_i)^{t_i}}{(1+s_{t_{i-1}})^{t_{i-1}}} \right)^{\frac{1}{t_i - t_{i-1}}} - 1; \quad i = 2, \ldots, n$$

For example, the following statements compute forward rates:

```
time = T(do(1, 5, 1));
spot = T(do(0.05, 0.09, 0.01));
forward = forward(time, spot);
print forward;
```

Figure 23.116 Forward Rates

forward
0.05
0.0700952
0.0902839
0.1105642
0.1309345

FREE Statement

FREE *matrices* **;**

FREE / < *keep-matrices* > **;**

The FREE statement releases memory associated with matrices. The matrices specified in the FREE statement lose their values; the memory bececomes available for other uses. After the FREE statement executes, the matrix does not have a value and the NROW and NCOL functions return 0. 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, the following statement frees the matrices **a**, **b**, and **c**:

```
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, the following statement frees all matrices except **d** and **e**:

```
free / d e;
```

For more information, see the discussion of workspace storage in Chapter 22.

FULL Function

FULL(*x* < , *nrow*> < , *ncol*> **);**

The FULL function converts a matrix stored in a sparse format into a matrix stored in a dense format. See the SPARSE function for a description of how sparse matrices are stored.

The arguments are as follows:

x	specifies a $k \times 3$ numerical matrix that contains a sparse representation of an $n \times p$ matrix.
nrow	specifies the number of rows in the dense matrix. If this argument is not specified, then the number of rows is determined by the maximum value of the second column of *x*.
ncol	specifies the number of columns in the dense matrix. If this argument is not specified, then the number of columns is determined by the maximum value of the third column of *x*.

The matrix returned by the FULL function is an $n \times p$ matrix with k nonzero values determined by the *x* matrix, as shown in the following example:

```
s = {3   1  1,
     1.1 2  1,
     4   2  2,
     1   3  2,
     10  3  3,
     3.2 4  2,
     3   4  4 };
x = full(s);
print x;
```

Figure 23.117 Matrix Converted from Sparse to Dense Storage

```
                                x

                    3           0           0           0
                  1.1           4           0           0
                    0           1          10           0
                    0         3.2           0           3
```

In the previous example, the **s** matrix specifies a lower triangular matrix. However, the **s** matrix might represent a symmetric matrix rather than a lower triangular matrix, but only the lower triangular entries were stored. (For example, the **s** matrix might have been created by the SPARSE function by using the "SYM" option; see the SPARSE function documentation.) If that is the case, you can use the following statement to recover the symmetric matrix representation of **s**:

```
xSym = (x+x`)- diag(x);
print xSym;
```

Figure 23.118 Symmetric Matrix Converted from Sparse Symmetric Storage

xSym			
3	1.1	0	0
1.1	4	1	3.2
0	1	10	0
0	3.2	0	3

By default, the size of the matrix returned by the FULL function is determined by the maximum row and column entry in the first argument. You can override this behavior by specifying values for the number of rows and columns returned by the FULL function, as shown in the following statements:

```
z = full(s, 5, 6);
print z;
```

Figure 23.119 Matrix with Zeros in Last Row or Column

z					
3	0	0	0	0	0
1.1	4	0	0	0	0
0	1	10	0	0	0
0	3.2	0	3	0	0
0	0	0	0	0	0

GAEND Call

CALL GAEND(*id***);**

The GAEND subroutine ends a genetic algorithm optimization and frees memory resources. The arguments to the GAEND call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

The GAEND call ends the genetic algorithm calculations associated with *id* and frees up all associated memory.

See the GASETUP function for an example.

GAGETMEM Call

CALL GAGETMEM(*members, values, id< , index>***);**

The GAGETMEM subroutine gets members of the current solution population for a genetic algorithm optimization.

The GAGETMEM call returns the following values as output arguments:

members names a matrix that contains the members of the current solution population specified by the *index* parameter.

values names a matrix that contains objective function values, with a value at each row that corresponds to the solution in *members*.

The input arguments to the GAGETMEM call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

index is a matrix of indices of the requested solution population members. If *index* is not specified, the entire population is returned.

The GAGETMEM call is used to retrieve members of the solution population and their objective function values. If the *elite* parameter of the GASETSEL call is nonzero, then the first *elite* members of the population have the most optimal objective function values of the population, and those *elite* members are sorted in ascending order of objective function value for a minimization problem and in descending order for a maximization problem.

If a single member is requested, that member is returned as-is in *members*. If more than one member is requested in a GAGETMEM call, each row of *members* has one solution, shaped into a row vector. If solutions are not of fixed length, then the number of columns of *members* equals the number of elements of the largest solution and rows that represent solutions with fewer elements have the extra elements filled in with missing values.

See the GASETUP function for an example.

GAGETVAL Call

CALL GAGETVAL(*values, id*< *, index*>**);**

The GAGETVAL subroutine gets objective function values for members of the population in a genetic algorithm optimization. The GAGETVAL call returns the following output argument:

values names a matrix that contains objective function values for solutions in the current population that are specified by *index*. If *index* is not present, then values for all solutions in the population are returned. Each row in *values* corresponds to one solution.

The input arguments to the GAGETVAL call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

index is a matrix of indices of the requested objective function values. If *index* is not specified, then all objective function values are returned.

The GAGETVAL call is used to retrieve objective function values of the current solution population. If the *elite* parameter of the GASETSEL call is nonzero, then the first *elite* members of the population have the most optimal objective function values of the population, and those *elite* members are sorted in ascending order of objective function value for a minimization problem or in descending order for a maximization problem.

See the GASETUP function for an example.

GAINIT Call

CALL GAINIT(*id, popsize* < *, bounds*> < *, modname*> **);**

The GAINIT subroutine creates and initializes a solution population for a genetic algorithm optimization. The input arguments to the GAINIT call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

popsize is the number of solution matrices to create and initialize.

bounds is an optional parameter matrix that specifies the lower and upper bounds for each element of a solution matrix. It is used only for integer and real fixed-length vector problem encoding.

modname is the name of a user-written module to be called from GAINIT when it generates the initial members of the solution population.

The GAINIT call creates the members and computes the objective values for an initial solution population for a genetic algorithm optimization. If the problem encoding is specified as a sequence in the corresponding GASETUP function call and no *modname* parameter is specified, then GAINIT creates an initial population of vectors of randomly ordered integer values ranging from 1 to the *size* parameter of the GASETUP function call. Otherwise, you control how the population is created and initialized with the *bounds* and *modname* parameters.

If real or integer fixed-length vector encoding is specified in the corresponding GASETUP function call, then the *bounds* parameter can be supplied as a $2 \times n$ matrix, where the dimension n equals the *size* parameter of the GASETUP function call: the first row gives the lower bounds of the corresponding vector elements and the second row gives the upper bounds. The solutions that result from all crossover and mutation operators are checked to ensure they are within the upper and lower bounds, and any solution components that violate the bounds are reset to the bound. However, if user-written modules are provided for these operators, the modules are expected to do the bounds checking internally. If no *modname* parameter is specified, the initial population is generated by random variation of the solution components between the lower and upper bounds.

For all problem encodings, if the *modname* parameter is specified, it is expected to be the name of a user-written subroutine module with one parameter. The module should generate and return an individual solution in that parameter. The GAINIT call invokes that module *popsize* times, once for each member

of the initial solution population. The *modname* parameter is required if the *encoding* parameter of the corresponding GASETUP function call was 0 or if the *bounds* parameter is not specified for real or integer fixed-length vector encoding.

See the GASETUP function for an example.

GAREEVAL Call

CALL GAREEVAL(*id*);

The GAREEVAL subroutine reevaluates the objective function values for a solution population of a genetic algorithm optimization. The input arguments to the GAREEVAL call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

The GAREEVAL call computes the objective values for a solution population of a genetic algorithm optimization. Since the GAINIT call and the GAREGEN call also evaluate the objective function values, it is usually not necessary to call GAREEVAL. It is provided to handle the situation of a user modifying an objective function independently—for example, adjusting a global variable to relax or tighten a penalty constraint. In such a case, GAREEVAL should be called before the next GAREGEN call.

GAREGEN Call

CALL GAREGEN(*id*);

The GAREGEN subroutine replaces the current solution population by applying selection, crossover, and mutation for a genetic algorithm optimization problem. The input arguments to the GAREGEN call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

The GAREGEN call applies the genetic algorithm to create a new solution population from the current population. As the first step, if the *elite* parameter of the corresponding GASETSEL call is nonzero, the best *elite* members of the current population are copied into the new population, sorted by objective value with the best objective value first. If a crossover operator has been specified in a corresponding GASETCRO call or a default crossover operator is in effect, the remaining members of the population are determined by selecting members of the current population, applying the crossover operator to generate offspring, and mutating the offspring according to the mutation probability and mutation operator. Either the mutation probability and operator are specified in the corresponding GASETMUT call or, if no such call is made, a default value of 0.05 is assigned to the mutation probability and a default mutation operator is assigned based on the problem encoding (see the GASETMUT call). The offspring are then transferred to the new population. If the no-crossover option is specified in the GASETCRO call, then only mutation is applied to

the non-elite members of the current population to form the new population. After the new population is formed, it becomes the current solution population, and the objective function specified in the GASETOBJ call is evaluated for each member.

See the GASETUP function for an example.

GASETCRO Call

CALL GASETCRO(*id, crossprob, type* < , *parm* > **);**

The GASETCRO subroutine sets the crossover operator for a genetic algorithm optimization. The input arguments to the GASETCRO call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

crossprob is the crossover probability, which has a range from zero to one. It specifies the probability that selected members of the current generation undergo crossover to produce new offspring for the next generation.

type specifies the kind of crossover operator to be used. *type* is used in conjunction with *parm* to specify either a user-written module for the crossover operator or one of several other operators, as explained in the following list.

parm is a matrix whose interpretation depends on the value of *type*, as described in the following list.

The following list specifies the valid values of the *type* parameter and the corresponding crossover operators:

–1 specifies that no crossover operator be applied and the new population be generated by applying the mutation operator to the old population, according to the mutation probability.

0 specifies that a user-written module, whose name is passed in the *parm* parameter, be used as the crossover operator. This module should be a subroutine with four parameters. The module should return the new offspring solutions in the first two parameters based on the input parent solutions, which are selected by the genetic algorithm and passed into the module in the last two parameters. The module is called once for each crossover operation within the GAREGEN call to create a new generation of solutions.

1 specifies the simple operator, defined for fixed-length integer and real vector encoding. To apply this operator, a position k within the vector of length n is chosen at random, such that $1 \leq k < n$. Then for parents **p1** and **p2** the offspring are as follows:

```
c1= p1[1,1:k] || p2[1,k+1:n];
c2= p2[1,1:k] || p1[1,k+1:n];
```

For real fixed-length vector encoding, you can specify an additional parameter, a, with the *parm* parameter, where a is a scalar and $0 < a \leq 1$. It modifies the offspring as follows:

```
x2 = a * p2 + (1-a) * p1;
```

```
c1 = p1[1,1:k] || x2[1,k+1:n];

x1 = a * p1 + (1-a) * p2
c2 = p2[1,1:k] || x1[1,k+1:n];
```

Note that for $a = 1$, which is the default value, **x2** and **x1** are the same as **p2** and **p1**. Small values of a reduce the difference between the offspring and parents. For integer encoding, the *parm* parameter is ignored and a is always 1.

2 specifies the two-point operator, defined for fixed-length integer and real vector encoding with length $n \geq 3$. To apply this operator, two positions k_1 and k_2 within the vector are chosen at random, such that $1 \leq k_1 < k_2 < n$. Element values between those positions are swapped between parents. For parents **p1** and **p2** the offspring are as follows:

```
c1 = p1[1,1:k1] || p2[1,k1+1:k2] || p1[1,k2+1:n];
c2 = p2[1,1:k1] || p1[1,k1+1:k2] || p2[1,k2+1:n];
```

For real vector encoding, you can specify an additional parameter, a, in the *parm* field, where $0 < a \leq 1$. It modifies the offspring as follows:

```
x2 = a * p2 + (1-a) * p1;
c1 = p1[1,1:k1] || x2[1,k1+1:k2] || p1[1,k2+1:n];

x1 = a * p1 + (1-a) * p2;
c2 = p2[1,1:k1] || x1[1,k1+1:k2] || p2[1,k2+1:n];
```

Note that for $a = 1$, which is the default value, **x2** and **x1** are the same as **p2** and **p1**. Small values of a reduce the difference between the offspring and parents. For integer encoding, the *parm* parameter is ignored if present and a is always 1.

3 specifies the arithmetic operator, defined for real and integer fixed-length vector encoding. This operator computes offspring of parents **p1** and **p2** as follows:

```
c1 = a * p1 + (1-a) * p2;
c2 = a * p2 + (1-a) * p1;
```

where a is a random number between 0 and 1. For integer encoding, each component is rounded off to the nearest integer. An advantage of this operator is that it always produces feasible offspring for a convex solution space. A disadvantage is that it tends to produce offspring toward the interior of the search region, so that it can be less effective if the optimum lies on or near the search region boundary.

4 specifies the heuristic operator, defined for real fixed-length vector encoding. This operator computes the first offspring from the two parents **p1** and **p2** as follows:

```
c1 = a * (p2 - p1) + p2;
```

where **p2** is the parent with the better objective value and a is a random number between 0 and 1. The second offspring is computed as in the arithmetic operator, as follows:

```
c2 = (1 - a) * p1 + a * p2;
```

This operator is unusual in that it uses the objective value. It has the advantage of directing the search in a promising direction and automatically fine-tuning the search in an area where solutions are clustered. If upper and lower bound constraints are specified in the GAINIT call, the offspring are checked against the bounds and any component outside its bound is set equal to that bound.

5 specifies the partial match operator, defined for sequence encoding. This operator produces offspring by transferring a subsequence from one parent and filling the remaining positions in a way consistent with the position and ordering in the other parent. Start with two parents and randomly chosen cut-points as follows:

```
p1 = {1 2|3 4 5 6|7 8 9};
p2 = {8 7|9 3 4 1|2 5 6};
```

The first step is to cross the selected segments; a missing value (.) indicates a position that is not determined):

```
c1 = {. . 9 3 4 1 . . .};
c2 = {. . 3 4 5 6 . . .};
```

Next, define a mapping according to the two selected segments, as follows:

$$9 \leftrightarrow 3, 3 \leftrightarrow 4, 4 \leftrightarrow 5, 1 \leftrightarrow 6$$

Next, fill in the positions where there is no conflict from the corresponding parent:

```
c1 = {. 2 9 3 4 1 7 8 .};
c2 = {8 7 3 4 5 6 2 . .};
```

Last, fill in the remaining positions from the subsequence mapping. In this case, for the first child $1 \rightarrow 6$ and $9 \rightarrow 3$, and for the second child $5 \rightarrow 4$, $3 \rightarrow 9$, and $6 \rightarrow 1$:

```
c1 = {6 2 9 3 4 1 7 8 5};
c2 = {8 7 3 4 5 6 2 9 1};
```

This operator tends to maintain similarity of both the absolute position and relative ordering of the sequence elements, and is useful for a wide range of sequencing problems.

6 specifies the order operator, defined for sequence encoding. This operator produces offspring by transferring a subsequence of random length and position from one parent and filling the remaining positions according to the order from the other parent. For parents **p1** and **p2**, first choose a subsequence, as follows:

```
p1 = {1 2|3 4 5 6|7 8 9};
p2 = {8 7|9 3 4 1|2 5 6};
c1 = {. . 3 4 5 6 . . .};
c2 = {. . 9 3 4 1 . . .};
```

Starting at the second cut-point, the elements of **p2** are in the following order (cycling back to the beginning):

```
2 5 6 8 7 9 3 4 1
```

After removing 3, 4, 5, and 6, which have already been placed in **c1**, you have the following:

```
2 8 7 9 1
```

Placing these back in order, starting at the second cut-point, yields the following:

```
c1 = {9 1 3 4 5 6 2 8 7};
```

Applying this logic to **c2** yields the following:

```
c2 = {5 6 9 3 4 1 7 8 2};
```

This operator maintains the similarity of the relative order (also called the adjacency) of the sequence elements of the parents. It is especially effective for circular path-oriented optimizations, such as the traveling salesman problem.

7 specifies the cycle operator, defined for sequence encoding. This operator produces offspring such that the position of each element value in the offspring comes from one of the parents. For example, consider the following parents **p1** and **p2**:

```
p1 = {1 2 3 4 5 6 7 8 9};
p2 = {8 7 9 3 4 1 2 5 6};
```

For the first child, pick the first element from the first parent, as follows:

```
c1 = {1 . . . . . . . .};
```

To maintain the condition that the position of each element value must come from one of the parents, the position of the '8' value must come from **p1**, because the '8' position in **p2** is already taken by the '1' in **c1**:

```
c1 = {1 . . . . . . 8 .};
```

Now the position of '5' must come from **p1** and so on until the process returns to the first position:

```
c1 = {1 . 3 4 5 6 . 8 9};
```

At this point, choose the remaining element positions from **p2**:

```
c1 = {1 7 3 4 5 6 2 8 9};
```

For the second child, starting with the first element from the second parent, similar logic produces the following:

```
c2 = {8 2 9 3 4 1 7 5 6};
```

This operator is most useful when the absolute position of the elements is of most importance to the objective value.

A GASETCRO call is required when 0 is specified for the *encoding* parameter in the GASETUP function. But for fixed-length vector and sequence encoding, a default crossover operator is used in the GAREGEN call when no GASETCRO call is made. For sequence encoding, the default is the partial match operator, unless the traveling salesman option was specified in the GASETOBJ call, in which case the order operator

is the default. For integer fixed-length vector encoding, the default is the simple operator. For real fixed-length vector encoding, the default is the heuristic operator.

See the GASETUP function for an example.

GASETMUT Call

CALL GASETMUT(*id, mutprob* < , *type* > < , *parm* > **);**

The GASETMUT subroutine sets the mutation operator for a genetic algorithm optimization. The input arguments to the GASETMUT call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

mutprob is the probability for a given solution to undergo mutation, a number between 0 and 1.

type specifies the kind of mutation operator to be used. *type* is used in conjunction with *parm* to specify either a user-written module for the mutation operator or one of several other operators, as explained in the following list.

parm is a matrix whose interpretation depends on the value of *type*, as described in the following list.

The GASETMUT call enables you to specify the frequency of mutation and the mutation operator to be used in the genetic algorithm optimization problem. If the *type* parameter is not specified, then the GASETMUT call only alters the mutation probability, without resetting the mutation operator, and any operator set by a previous GASETMUT call remains in effect. You can specify the following mutation operators with the *type* parameter:

0 specifies that a user-written module, whose name is passed in the *parm* parameter, be used as the mutation operator. This module should be a subroutine with one parameter, which receives the solution to be mutated. The module is called once for each mutation operation and is expected to modify the input solution according to the desired mutation operation. Any checking of bounds specified in the GAINIT call should be done inside the module; in this case they are not checked by the SAS/IML language.

1 specifies the uniform mutation operator, defined for fixed-length real or integer encoding, with upper and lower bounds specified in the GAINIT call. The *parm* parameter is not used with this option. To apply this operator, a position k is randomly chosen within the solution vector v and $v[k]$ is modified to a random value between the upper and lower bounds for element k. This operator can prove especially useful in early stages of the optimization, since it tends to distribute solutions widely across the search space and avoid premature convergence to a local optimum. However, in later stages of an optimization with real vector encoding when the search needs to be fine-tuned to home in on an optimum, the uniform operator can hinder the optimization.

2 specifies the delta mutation operator, defined for integer and real fixed-length vector encoding. This operator first chooses an element of the solution at random, and then perturbs that element by a fixed amount, *delta*, which is set with the *parm* parameter. *delta* has the same dimension as the solution vectors, and each element *delta*$[k]$ is set to *parm*$[k]$, unless *parm* is a scalar, in which case

all elements are set equal to *parm*. For integer encoding, all *delta*[*k*] are truncated to integers if they are not integers in *parm*. To apply the mutation, a randomly chosen element *k* of the solution vector *v* is modified such that one of the following statements is true:

```
v[k] = v[k] + delta[k]; /* with probability 0.5 */
       or
v[k] = v[k] - delta[k];
```

If bounds are specified for the problem in the GAINIT call, then *v*[*k*] is adjusted as necessary to fit within the bounds. This operator enables you to control the scope of the search with the *parm* matrix. One possible strategy is to start with a larger *delta* value and then reduce it with subsequent GASETMUT calls as the search progresses and begins to converge to an optimum. This operator is also useful if the optimum is known to be on or near a boundary, in which case *delta* can be set large enough to always perturb the solution element to a boundary.

3 specifies the swap operator, which is defined for sequence problem encoding. This operator picks two random locations in the solution vector and swaps their values. It is the default mutation operator for sequence encoding, except for when the traveling salesman option is specified in the GASETOBJ call. You can also specify that multiple swaps be made for each mutation with the *parm* parameter. The number of swaps defaults to 1 if *parm* is not specified, and is equal to *parm* otherwise.

4 specifies the invert operator, defined for sequence encoding. This operator picks two locations at random and then reverses the order of elements between them. This operator is most often applied to the traveling salesman problem. The *parm* parameter is not used with this operator.

Mutation is generally useful in the application of the genetic algorithm to ensure that a diverse population of solutions is sampled to avoid premature convergence to a local optimum. More than one GASETMUT call can be made at any time in the progress of the algorithm. This enables flexible adaptation of the mutation process, either changing the mutation probability or changing the operator itself. You can do this to ensure a wide search at the beginning of the optimization, and then reduce the variation later to narrow the search close to an optimum.

A GASETMUT call is required when an *encoding* parameter of 0 is specified in the GASETUP function. But when no GASETMUT call is made for fixed-length vector and sequence encoding, a default value of 0.05 is set for *mutprob* and a default mutation operator is used in the GAREGEN call. The mutation operator defaults to the uniform operator for fixed-length vector encoding with bounds specified in the GAINIT call, the delta operator with a *parm* value of 1 for fixed-length vector encoding with no bounds specified, the invert operator for sequence encoding when the traveling salesman option is chosen in the GASETOBJ call, and the swap operator for all other sequence encoded problems.

See the GASETUP function for an example.

GASETOBJ Call

CALL GASETOBJ(*id, type* < *, parm* > **);**

The GASETOBJ subroutine sets the objective function for a genetic algorithm optimization. The input arguments to the GASETOBJ call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

type specifies the type of objective function to be used.

parm is a matrix whose interpretation depends on the value of *type*, as described in the following list.

You can specify that a user-written module be used to compute the value of the objective function, or you can specify a standard preset function. This is specified with the *type* and *parm* parameters. The following list specifies the valid values of the *type* parameter:

0 specifies that a user-written function module is to be minimized. The name of the module is supplied in the *parm* parameter. The specified module should take a single parameter that represents a given solution, and return a scalar numeric value for the objective function.

1 specifies that a user-written function module be maximized. The name of the module is supplied in the *parm* parameter. The specified module should take a single parameter that represents a given solution, and return a scalar numeric value for the objective function.

2 specifies an objective function from the traveling salesman problem, which is minimized. This option is valid only if three conditions are met: sequence encoding was specified in the GASETUP function call, the solution vector is to be interpreted as a circular route, and each element represents a location. The *parm* parameter should be a square cost matrix, such that *parm*[*i*, *j*] is the cost of going from location *i* to location *j*. The dimension of the matrix should be the same as the *size* parameter of the corresponding GASETUP function call.

The specified objective function is called once for each solution to evaluate the objective values for the GAREGEN call, GAINIT call, and GAREEVAL call. Also, the objective values for the current solution population are reevaluated if GASETOBJ is called after a GAINIT call.

See the GASETUP function for an example.

GASETSEL Call

CALL GASETSEL(*id, elite, type, parm* **);**

The GASETSEL subroutine sets the selection parameters for a genetic algorithm optimization.

The input arguments to the GASETSEL call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

elite specifies the number of solution population members to carry over unaltered to the next generation in the GAREGEN call. If nonzero, then *elite* members with the best objective function values are carried over without crossover or mutation.

type specifies the selection method to use.

parm is a parameter used to control the selection pressure.

This module sets the selection parameters that are used in the GAREGEN call to select solutions for the crossover operation. You can choose between two variants of the "tournament" selection method in which a group of different solutions is picked at random from the current solution population and the solution from that group with the best objective value is selected. In the first variation, chosen by setting *type* to 0, the most optimal solution is always selected, and the *parm* parameter is used to specify the size of the group, always two or greater. The larger the group size, the greater the selective pressure. In the second variation, chosen by setting *type* to 1, the group size is set to 2 and the best solution is chosen with probability specified by *parm*. If *parm* is 1, the best solution is always picked; a *parm* value of 0.5 is equivalent to pure random selection. The *parm* value must be between 0.5 and 1. When *type* is 0, the selective pressure is greater than when *type* is 1. Higher selective pressure leads to faster convergence of the genetic algorithm, but is more likely to give premature convergence to a local optimum.

In order to ensure that the best solution of the current solution population is always carried over to the next generation, an *elite* value of 1 should be specified. Higher values of *elite* generally lead to faster convergence of the algorithm, but they increase the chances of premature convergence to a local optimum. If GASETSEL is not called, the optimization uses the default values of 1 for *elite*, 1 for *type*, and 2 for *parm*.

See the GASETUP function for an example.

GASETUP Function

GASETUP(*encoding, size* < *, seed* > **);**

The GASETUP function sets up the problem encoding for a genetic algorithm optimization problem. The GASETUP function returns a scalar number that identifies the genetic algorithm optimization problem. This number is used in subsequent calls to carry out the optimization.

The arguments to the GASETUP function are as follows:

encoding is a scalar number used to specify the form or structure of the problem solutions to be optimized. A value of 0 indicates a numeric matrix of arbitrary dimensions, 1 indicates a fixed-length floating-point row vector, 2 indicates a fixed-length integer row vector, and 3 indicates a fixed-length sequence of integers, with alternate solutions distinguished by different sequence ordering.

size is a numeric scalar, whose value is the vector or sequence length, if a fixed-length *encoding* is specified. For arbitrary matrix encoding (*encoding* value of 0), *size* is not used.

seed is an optional initial random number seed to be used for the initialization and the selection process. If *seed* is not specified or its value is 0, an initial seed is derived from the current system time.

GASETUP is the first call that must be made to set up a genetic algorithm optimization problem. It specifies the problem encoding, the size of a population member, and an optional seed that initializes the random number generator used in the selection process. GASETUP returns an identifying number that must be passed to the other modules that specify genetic operators and control the execution of the genetic algorithm. More than one optimization can be active concurrently, and optimization problems with different problem identifiers are completely independent. When a satisfactory solution has been determined, the op-

timization problem should be terminated with a call to GAEND to free up resources associated with the genetic algorithm.

The following example demonstrates the use of several genetic algorithm subroutines:

```
/* Use a genetic algorithm to explore the solution space for the
   "traveling salesman" problem. First, define the objective
   function to minimize:
   Compute the sum of distances between sequence of cities */
start EvalFitness( pop ) global ( dist );
   fitness = j( nrow(pop),1 );
   do i = 1 to nrow(pop);
      city1 = pop[i,1];
      city2 = pop[i,ncol(pop)];
      fitness[i] = dist[ city1, city2 ];
      do j = 1 to ncol(pop)-1;
         city1 = pop[i,j];
         city2 = pop[i,j+1];
         fitness[i] = fitness[i] + dist[city1,city2];
      end;
   end;
   return ( fitness );
finish;

/* Set up parameters for the genetic algorithm */

mutationProb = 0.15;   /* prob that a child will be mutated */
numElite = 2;          /* copy this many to next generation */
numCities = 15;        /* number of cities to visit */
numGenerations = 100;  /* number of generations to evolve */
seed = 54321;          /* random number seed */

/* fix population size; generate random locations for cities */
popSize = max(30,2*numCities);
locations = uniform( j(numCities,2,seed) );

/* compute distances between cities one time */
dist = j( numCities, numCities, 0 );
do i = 1 to numCities;
   do j = 1 to i-1;
      v = locations[i,]-locations[j,];
      dist[i,j] = sqrt( v[##] );
      dist[j,i] = dist[i,j];
   end;
end;

/* run the genetic algorithm */
id = gasetup( 3, numCities, seed);
call gasetobj(id, 0, "EvalFitness" );
call gasetcro(id, 1.0, 6);
call gasetmut(id, mutationProb, 3);
call gasetsel(id, numElite, 1, 0.95);
call gainit(id, popSize );
```

```
do i = 1 to numGenerations;
   if mod(i,20)=0 then do;
      call gagetval( value, id, 1 );
      print "Iteration:" i "Top value:" value;
   end;
   call garegen(id);
end;

/* report final sequence for cities */
call gagetmem(mem, value, id, 1);
print mem, value;
call gaend(id);
```

Figure 23.120 Result of a Genetic Algorithm Optimization

```
                                i                value

                Iteration:     20 Top value: 3.6836569

                                i                value

                Iteration:     40 Top value: 3.5567152

                                i                value

                Iteration:     60 Top value: 3.4562136

                                i                value

                Iteration:     80 Top value: 3.4562136

                                i                value

                Iteration:    100 Top value:  3.437183

                                  mem
                    COL1      COL2      COL3      COL4      COL5

          ROW1         6         4        12         7        13

                                  mem
                    COL6      COL7      COL8      COL9     COL10

          ROW1        15         8         9        11         5

                                  mem
                   COL11     COL12     COL13     COL14     COL15

          ROW1         2        14        10         3         1

                                value

                             3.437183
```

GBLKVP Call

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

The GBLKVP subroutine is a graphical call that defines a blanking viewport.

The arguments to the GBLKVP subroutine are as follows:

viewport is a numeric matrix or literal that defines 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 the 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.

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 they are 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 following 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 the rectangular area, use the *inside* parameter, as follows:

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

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

GBLKVPD Call

CALL GBLKVPD ;

The GBLKVPD subroutine is a graphical call that deletes and releases the current blanking area. It enables 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

CALL GCLOSE ;

The GCLOSE subroutine is a graphical call that 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

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

The GDELETE subroutine is a graphical call that 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, as follows:

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

GDRAW Call

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

The GDRAW subroutine is a graphical call that draws a polyline.

The required arguments to the GDRAW subroutine are as follows:

x is a vector that contains the horizontal coordinates of points used to draw a sequence of lines.

y is a vector that contains the vertical coordinates of points used to draw a sequence of lines.

The optional arguments to the GDRAW subroutine are as follows:

style is a numeric matrix or literal that specifies an index that corresponds 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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 are $n - 1$ lines. The first line is 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 that uses the GDRAW subroutine follows:

```
call gstart;
/* line from (50,50) to (75,75) */
call gdraw({50 75},{50 75});
call gshow;
```

GDRAWL Call

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

The GDRAWL subroutine is a graphical call that draws individual lines.

The required arguments 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.

The optional arguments to the GDRAWL subroutine are as follows:

style is a numeric matrix or literal that specifies an index that corresponds 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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 coordinates of the beginning points of the line segment, and the first two columns of *xy2* have coordinates of the corresponding endpoints. If *xy1* and *xy2* have *n* rows, *n* lines are drawn.

The lines are drawn in the same color and line style. The coordinates in use for this graphics command are world coordinates. An example that uses the GDRAWL call follows:

```
proc iml;
call gstart;
/* three line segments */
xy1 = { 0  0,  25 50, 50 75};
xy2 = {25 25,  50 50, 75 50};
call gdrawl(xy1, xy2);
call gshow;
```

GENEIG Call

CALL GENEIG(*eval, evecs, sym-matrix1, sym-matrix2***);**

The GENEIG subroutine computes eigenvalues and eigenvectors of a generalized eigenproblem.

The input arguments to the GENEIG subroutine are as follows:

sym-matrix1 is a symmetric numeric matrix.

sym-matrix2 is a positive definite symmetric matrix.

The subroutine returns the following output arguments:

evals names a vector in which the eigenvalues are returned.

evecs names a matrix in which the corresponding eigenvectors are returned.

The GENEIG subroutine computes eigenvalues and eigenvectors of the generalized eigenproblem. If **A** and **B** are symmetric and **B** is positive definite, then the vector **M** and the matrix **E** solve the generalized eigenproblem provided that

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

The vector **M** contains the eigenvalues arranged in descending order, and the matrix **E** contains the corresponding eigenvectors in the columns.

The following example is from Wilkinson and Reinsch (1971):

```
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);
print M, E;
```

Figure 23.121 Solution of a Generalized Eigenproblem

```
                                  M

                           1.4923532
                           1.1092845
                            0.943859
                           0.6636627
                           0.4327872

                                  E

          -0.076387  0.142012   0.19171  -0.08292 -0.134591
           0.017098   0.14242 -0.158991 -0.153148 0.0612947
          -0.066665 0.1209976 0.0748391 0.1186037 0.1579026
           0.086048  0.125531 -0.137469  0.182813 -0.109466
          0.2894334 0.0076922 0.0889779 -0.003562  0.041473
```

GEOMEAN Function

GEOMEAN(*matrix***);**

The GEOMEAN function returns a scalar that contains the geometric mean of the elements of the input matrix. The geometric mean of a set of nonnegative numbers $a_1, a_2, \ldots, a_n$ is the nth root of the product $a_1 \cdot a_2 \cdots a_n$.

The geometric mean is zero if any of the a_i are zero. The geometric mean is not defined for negative numbers. If any of the a_i are missing, they are excluded from the computation.

The geometric mean can be used to compute the average return on an investment. For example, the following data gives the annual returns on U.S. Treasury bonds from 1994 to 2004. The following statements compute the average rate of return during this time. The output, shown in Figure 23.122, shows that the average rate of return was 6.43%.

```
/*          year  return% */
TBonds = { 1994  -8.04,
           1995  23.48,
           1996   1.43,
           1997   9.94,
           1998  14.92,
           1999  -8.25,
           2000  16.66,
           2001   5.57,
           2002  15.12,
           2003   0.38,
           2004   4.49 };

proportion = 1 + TBonds[,2]/100; /* convert to proportion */
aveReturn = geomean( proportion );
print aveReturn;
```

Figure 23.122 Average Rate of Return for an Investment

aveReturn
1.0643334

GGRID Call

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

The GGRID subroutine is a graphical call that draws a grid on a graphical window. The required arguments to the GGRID subroutine are as follows:

x is a vector of points that contains the horizontal coordinates of the grid lines.

y is a vector of points that contains the vertical coordinates of the grid lines.

The optional arguments to the GGRID subroutine are as follows:

style is a numeric matrix or literal that specifies an index that corresponds 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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:

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

/* Places a grid in the lower left corner of the screen, */
/* assuming the default window and viewport              */
call ggrid(x,y);
call gshow;
```

GINCLUDE Call

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

The GINCLUDE subroutine is a graphical call that includes a previously defined graph in the current graph. The segment that is included is named *segment-name* and is in the same catalog as the current graph. The included segment is defined in the current viewport but not in the current window.

The implementation of the GINCLUDE subroutine makes it possible to include other segments in 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        */
/* that contains the segment name          */
segment1={myplot};
call ginclude(segment1);

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

GINV Function

GINV(*matrix***);**

The GINV function computes 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 by using the singular value decomposition (Wilkinson and Reinsch 1971).

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

As an example, consider the following model:

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

Least squares regression for this model can be performed by using the quantity **ginv(x)*y** as the estimate of $\boldsymbol{\beta}$. This solution has minimum $\mathbf{b}'\mathbf{b}$ among all solutions that minimize $\boldsymbol{\epsilon}'\boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} = \mathbf{Y} - \mathbf{X}\mathbf{b}$.

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

The following program demonstrates some common uses of the GINV function:

```
A = {1 0 1 0 0,
     1 0 0 1 0,
     1 0 0 0 1,
     0 1 1 0 0,
     0 1 0 1 0,
     0 1 0 0 1 };

/* find generalized inverse */
Ainv = ginv(A);

/* find LS solution: min  |Ax-b|^2 */
b = { 3, 2, 4, 2, 1, 3 };
x = Ainv*b;

/* form projection matrix onto row space.
   Note P = P` and P*P = P */
P = Ainv*A;

/* find numerical rank of A */
rankA = round(trace(P));

reset fuzz;
print Ainv, rankA, x, P;
```

Figure 23.123 Common Uses of the Generized Inverse

```
                                Ainv

         0.2666667 0.2666667 0.2666667 -0.066667 -0.066667 -0.066667
         -0.066667 -0.066667 -0.066667 0.2666667 0.2666667 0.2666667
               0.4      -0.1      -0.1       0.4      -0.1      -0.1
              -0.1       0.4      -0.1      -0.1       0.4      -0.1
              -0.1      -0.1       0.4      -0.1      -0.1       0.4

                               rankA

                                     4
```

Figure 23.123 *continued*

```
                              x

                                  2
                                  1
                                  1
                                  0
                                  2

                              P

                0.8        -0.2        0.2        0.2        0.2
               -0.2         0.8        0.2        0.2        0.2
                0.2         0.2        0.8       -0.2       -0.2
                0.2         0.2       -0.2        0.8       -0.2
                0.2         0.2       -0.2       -0.2        0.8
```

If $\mathbf{A}$ is an $n \times m$ matrix, then, in addition to the memory allocated for the return matrix, the GINV function temporarily allocates an $n^2 + nm$ array for performing its computation.

GOPEN Call

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

The GOPEN subroutine is a graphical call that starts a new graphics segment.

The arguments to the GOPEN subroutine are as follows:

segment-name	is a character matrix or quoted literal that specifies 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 by 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 named segment twice, the second segment is given an automatically generated name.

The following statement opens a new segment named "cosine", replaces the existing segment of the same name, and attaches a description to the segment:

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

GOTO Statement

GOTO *label* **;**

The GOTO statement causes a program to jump to a new statement in the program. When the GOTO statement is executed, the program jumps immediately to the statement with the given *label* and begin executing statements from that point. A label is a name followed by a colon that precedes an executable statement.

GOTO statements are often clauses of IF-THEN statements. For example, the following statements use a GOTO statement to iterate until a condition is satisfied:

```
start Iterate;
   x = 1;
   TheStart:
   if x > 10 then
      goto TheEnd;
   x = x + 1;
   goto TheStart;

   TheEnd: print x;
finish;

run Iterate;
```

Figure 23.124 Iteration by Using the GOTO Statement

x
11

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

```
x = 1;
do until(x > 10);
   x = x + 1;
end;

print x;
```

Figure 23.125 Avoiding the GOTO Statement

x
11

As good programming practice, you should avoid using a GOTO statement that refers to a label that precedes the GOTO statement; otherwise, an infinite loop is possible. You cannot use a GOTO statement to jump out of a module; use the RETURN statement instead.

GPIE Call

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

The GPIE subroutine is a graphical call that draws pie slices.

The required arguments to the GPIE subroutine are as follows:

x is a scalar value that contains the horizontal coordinates of the center of the pie slices. This argument can also be a vector, in which case it defines centers for multiple pie slices.

y is a scalar value that contains the vertical coordinates of the center of the pie slices. This argument can also be a vector, in which case it defines centers for multiple pie slices.

r is a scalar or vector that contains the radii of the pie slices.

The optional arguments to the GPIE subroutine are as follows:

angle1 is a scalar or vector that contains the start angles. It defaults to 0.

angle2 is a scalar or vector that contains 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

outline is an index that indicates 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 that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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 specified in degrees. The start angle (*angle1*) defaults to 0, and the terminal angle (*angle2*) defaults to 360. The *outline* argument is an index that indicates the side of the slice to draw; it can have the following values:

< 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.

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

```
call gstart;
center = {50 50};
r = 30;
angle1 = {0 90 180 270};
angle2 = {90 180 270 360};
/* draw a pie with 4 slices of equal size */
call gpie(center[1], center[2], r, angle1, angle2);
```

GPIEXY Call

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

The GPIEXY subroutine is a graphical call that converts from polar to world coordinates.

The GPIEXY subroutine returns the following output arguments:

x	names a vector to contain the horizontal coordinates returned by GPIEXY.
y	names a vector to contain the vertical coordinates returned by GPIEXY.

The required input arguments to the GPIEXY subroutine are as follows:

fract-radii	is a vector of fractions of the radius of the reference circle.
angles	is the vector of angle coordinates in degrees.

The optional input arguments to the GPIEXY subroutine are as follows:

center	defines the reference circle.
radius	defines the reference circle.
window	is a numeric matrix or literal that specifies 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 0.33 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 halfway 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 can appear distorted when drawn on some devices. The PROC IML graphics subsystem adjusts computations to compensate for this distortion. Thus, for any given point, the transformation from polar coordinates to world coordinates might need an equivalent adjustment. The GPIEXY subroutine ensures that the same adjustment applied in the GPIE subroutine is applied to the conversion. An example that uses the GPIEXY call follows:

```
call gstart;
center = {50 50};
r = 30;
angle1 = {0 90 180 270};
angle2 = {90 180 270 360};
call gpie(center[1], center[2], r, angle1, angle2);
/* add labels to a pie with 4 slices of equal size */
angle = (angle1+angle2)/2;   /* middle of slice */
call gpiexy(x, y, 1.2, angle, center, r);

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

GPOINT Call

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

The GPOINT subroutine is a graphical call that draws symbols at specified locations.

The required arguments to the GPOINT subroutine are as follows:

x is a vector that contains the horizontal coordinates of points.

y is a vector that contains the vertical coordinates of points.

The optional arguments to the GPOINT subroutine are as follows:

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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

height is a numeric matrix or literal that specifies the character height.

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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

The following example plots the curve $y = 50 + 25\sin(x/10)$ for $0 \leq x \leq 100$:

```
call gstart;
x = 0:100;
y = 50 + 25*sin(x/10);
call gpoint(x, y);
call gshow;
```

The following example uses the GPOINT subroutine to plot symbols at specific locations on the screen:

```
marker = {a b c d e '@' '#' '$' '%' '^' '&' '*' '-' '+' '='};
x = 5*(1:ncol(marker));
y = x;
call gpoint(x, y, marker);
call gshow;
```

See Chapter 15 for further examples that use the GPOINT subroutine.

GPOLY Call

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

The GPOLY subroutine is a graphical call that draws and fills a polygon.

The required arguments to the GPOLY subroutine are as follows:

x is a vector that defines the horizontal coordinates of the corners of the polygon.

y is a vector that defines the vertical coordinates of the corners of the polygon.

The optional inputs to the GPOLY subroutine are as follows:

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

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

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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 that uses the GPOLY subroutine follows:

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

GPORT Call

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

The GPORT subroutine is a graphical call that defines a 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 by using device coordinates (always 0 to 100). Changing the viewport can affect the height of the character fonts; if so, you might 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

CALL GPORTPOP ;

The GPORTPOP subroutine is a graphical call that deletes the top viewport from the stack.

GPORTSTK Call

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

The GPORTSTK subroutine is a graphical call that 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 *viewport* argument is a numeric matrix or literal defined in normalized coordinates of the form

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

This graphics command uses world coordinates. An example of a valid statement follows:

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

GSCALE Call

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

The GSCALE subroutine computes a suitable scale and tick values for labeling axes.

The required arguments to the GSCALE subroutine are as follows:

scale	is a returned vector that contains 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.

The optional arguments to the GSCALE subroutine are as follows:

nicenum	is numeric and provides up to 10 numbers to use for scaling. By default, *nicenum* is the vector {1,2,2.5,5}.
fixed-end	is a character argument that 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 (Lewart 1973) of the *Collected Algorithms* from the Association for Computing Machinery (ACM). 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.

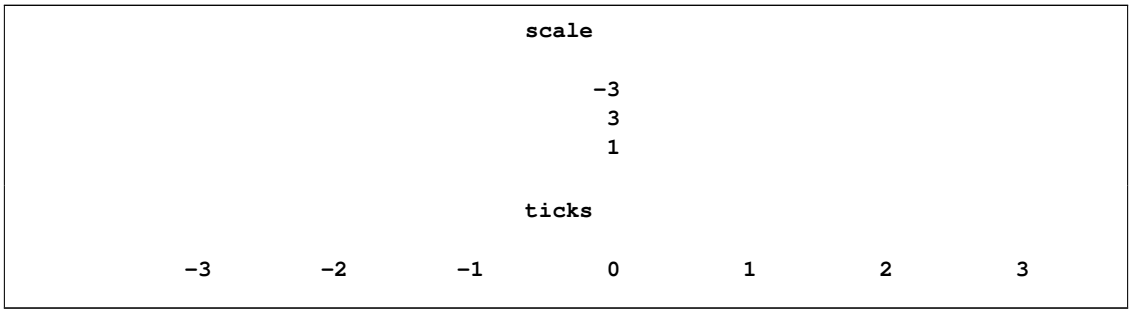
The *nicenum* and *fixed-end* arguments are optional. The *nicenum* argument provides up to 10 numbers, all between 1 and 10 (inclusive of the endpoints), to be used for scaling. The default for *nicenum* is 1, 2, 2.5,

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 that uses the GSCALE subroutine follows:

```
x = normal( j(100,1) );   /* generate standard normal data */
call gscale(scale, x, 5); /* ask for about 5 intervals */
ticks = do(scale[1], scale[2], scale[3]);
print scale, ticks;
```

Figure 23.126 Tick Marks for Standard Normal Data



GSCRIPT Call

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

The GSCRIPT subroutine is a graphical call that writes multiple text strings.

The required arguments to the GSCRIPT subroutine are as follows:

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

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

text is a character vector of text strings.

The optional arguments to the GSCRIPT subroutine are as follows:

angle is the slant of each text string.

rotate is the rotation of individual characters.

height is a real number that specifies 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

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

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

The GSET subroutine is a graphical call that sets attributes for a graphics segment.

The arguments to the GSET subroutine are as follows:

attribute is a graphics attribute. This argument can be a character matrix or quoted literal.

value is the value to which the attribute is set. This 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

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 that corresponds to a valid line style.

thick an integer that specifies line thickness.

To reset the PROC 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 subroutine. Single or double quotes can be used around this argument. For more information about the attributes, see Chapter 15.

Examples of valid statements follow:

```
call gset("pattern", "m1n45");
call gset("font", "simplex");

f = "font";
s = "simplex";
call gset(f, s);
```

For example, the following statement resets *color* to its default:

```
call gset("color");
```

GSHOW Call

CALL GSHOW(< *segment-name* > **);**

The GSHOW subroutine is a graphical call that displays a window. 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);
```

See Chapter 15 for examples that use the GSHOW subroutine.

GSORTH Call

CALL GSORTH(*P, T, lindep, A***);**

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. The GSORTH subroutine implements an algorithm described by Golub (1969).

The GSORTH subroutine has a single input argument:

A is an input $m \times n$ matrix.

The output arguments 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.

Specifically, 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, $\text{rank}(\mathbf{A}) = n$), 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, $\text{rank}(\mathbf{A}) = k < n$) 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 following statements call the GSORTH subroutine and print the output parameters to the call:

```
x = {1 1 3 1 2,
     1 0 1 2 3,
     1 1 3 3 4,
     1 0 1 4 5,
     1 1 3 5 6,
     1 0 1 6 7};
call gsorth(P, T, lindep, x);
reset fuzz;
print P, T, lindep;
```

Figure 23.127 Results of a Gram-Schmidt Orthonormalization

```
                                P

          0.4082483 0.4082483         0      -0.5         0
          0.4082483 -0.408248         0      -0.5         0
          0.4082483 0.4082483         0         0         0
          0.4082483 -0.408248         0         0         0
          0.4082483 0.4082483         0       0.5         0
          0.4082483 -0.408248         0       0.5         0

                                T

          2.4494897 1.2247449 4.8989795 8.5732141 11.022704
                  0 1.2247449 2.4494897 -1.224745 -1.224745
                  0         0         0         0         0
                  0         0         0         4         4
                  0         0         0         0         0

                              lindep

                                   1
```

If *lindep* is 1, you can permute the columns of **P** and rows of **T** so that the zero columns of **P** are rightmost—that is, $\mathbf{P} = (\mathbf{P}_1, \ldots, \mathbf{P}_k, 0, \ldots, 0)$, where k is the column rank of **A** and the equality $\mathbf{A} = \mathbf{P}*\mathbf{T}$ is preserved. The following statements show a permutation of columns:

```
d = loc(vecdiag(T)^=0) || loc(vecdiag(T)=0);
temp = P;
P[,d] = temp;
temp = T;
T[,d] = temp;
print d, P, T;
```

Figure 23.128 Rearranging Columns

```
                              d

                1         2         4         3         5

                              P

          0.4082483 0.4082483      -0.5         0         0
          0.4082483 -0.408248      -0.5         0         0
          0.4082483 0.4082483         0         0         0
          0.4082483 -0.408248         0         0         0
          0.4082483 0.4082483       0.5         0         0
          0.4082483 -0.408248       0.5         0         0

                              T

          2.4494897 1.2247449 8.5732141 4.8989795 11.022704
                  0 1.2247449 -1.224745 2.4494897 -1.224745
                  0         0         0         0         0
                  0         0         4         0         4
                  0         0         0         0         0
```

The GSORTH subroutine is not recommended for the construction of matrices of values of orthogonal polynomials; the ORPOL function should be used for that purpose.

GSTART Call

CALL GSTART(< *catalog* > < , *replace* > **);**

The GSTART subroutine initializes the graphics system the first time it is called. A catalog is opened to capture any graphics segments generated in the session. If you do not specify a catalog, PROC IML uses the temporary catalog Work.Gseg.

The arguments to the GSTART subroutine are as follows:

catalog is a character matrix or quoted literal that specifies the SAS catalog for saving the graphics segments.

replace is a numeric argument.

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 to their defaults unless the GOPTION corresponds to a GSET parameter. The GOPEN subroutine also resets GSET parameters.

An example of using the GSTART subroutine is provided in the documentation for the GPOINT subroutine.

GSTOP Call

CALL GSTOP ;

The GSTOP subroutine deactivates the graphics system. The graphics subsystem is disabled until the GSTART subroutine is called again.

GSTRLEN Call

CALL GSTRLEN(*length, text* < *, height*> < *, font*> < *, window*> **);**

The GSTRLEN subroutine returns the lengths of text strings represented in a given font and for a given character height. The lengths are given in world coordinates. The required arguments to the GSTRLEN subroutine are as follows:

length	is a matrix of lengths specified in world coordinates.
text	is a matrix of text strings.

The optional arguments to the GSTRLEN subroutine are as follows:

height	is a numeric matrix or literal that specifies the character height.
font	is a character matrix or quoted literal that specifies a valid font name.
window	is a numeric matrix or literal that specifies a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

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* is 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 that uses the GSTRLEN subroutine follows:

```
call gstart;
/* centers text at coordinates */
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");
call gshow;
```

GTEXT and GVTEXT Calls

CALL GTEXT(*x, y, text* < *, color*> < *, window*> < *, viewport*> **);**

CALL GVTEXT(*x, y, text* < *, color*> < *, window*> < *, viewport*> **);**

The GTEXT subroutine places text horizontally on a graph; the GVTEXT subroutine places text vertically on a graph.

The required arguments to the GTEXT and GVTEXT subroutines are as follows:

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

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

text is a vector of text strings.

The optional arguments to the GTEXT and GVTEXT subroutines are as follows:

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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

window is a numeric matrix or literal that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

The GTEXT subroutine places text horizontally on 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 use a viewport or if you change the HEIGHT or

ASPECT parameters by using the GSET subroutine or the GOPTIONS statement. The coordinates in use for this graphics command are world coordinates.

Examples of the GTEXT and GVTEXT subroutines follow:

```
call gstart;
call gopen;
call gport({0 0 50 50});
call gset("height", 3); /* set character height */
msg = "GTEXT: This will start in the center of the viewport";
call gtext(50, 50, msg);
msg = "GVTEXT: Vertical string";
call gvtext(0.35, 10, msg, 'red', {0.2 -1, 1.5 6.5}, {0 0, 100 100});
call gshow;
```

GWINDOW Call

CALL GWINDOW(*window***);**

The GWINDOW subroutine sets up the window for scaling data values in subsequent graphics primitives. The argument *window* is a numeric matrix or literal that specifies 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 window remains until the next GWINDOW call or until the segment is closed. The coordinates in use for this graphics command are world coordinates. An example that uses the GWINDOW subroutine follows:

```
x = rannor( j(20,1) );
y = 3 + x + 0.5*rannor( j(20,1) );

call gstart;
/* define window to contain the data range plus 5% margins */
xMargin = 0.05*(max(x) - min(x));
yMargin = 0.05*(max(y) - min(y));
wd = (min(x)-xMargin) || (min(y)-yMargin) ||
     (max(x)+xMargin) || (max(y)+yMargin);
call gwindow(wd);
call gpoint(x, y);
call gshow;
```

GXAXIS and GYAXIS Calls

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 GXAXIS subroutine is a graphical call that draws a horizontal axis. The GYAXIS subroutine draws a vertical axis.

The required arguments 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 that contains the length of the axis, specified in world coordinates.

nincr is a numeric scalar that contains the number of major tick marks on the axis. The first tick mark corresponds to *starting-point*.

The optional arguments to the GXAXIS and GYAXIS subroutines are as follows:

nminor is an integer that specifies 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 that contains a valid color as an element, or a color number (such as 1) that refers to a color in the color list.

fixed-end holds one end of the scale fixed. 'U' fixes the upper end; 'L' fixes the lower end; 'X' allows both ends to vary from the data values. In addition, you can 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 that specifies 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 that specifies a viewport. This is given in normalized coordinates and has the same form as the *window* argument.

The GXAXIS and GYAXIS subroutines 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 that contains the two elements, 10 and 44, the scale obtained should be the same as that obtained by the GXAXIS subroutine. Sometimes, it can be helpful to use the GSCALE subroutine in conjunction with the axis subroutines 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 following statements draw these axes. Each axis is four units long. 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 tick marks are labeled because the *noticklab* option has the value 0.

```
call gstart;
call gport({20 20 80 80});
call gwindow({-2 -2 2 2});
call gxaxis({-2,0}, 4, 5, 2, 0);
call gyaxis({0,-2}, 4, 5, 2, 0);
call gshow;
```

HADAMARD Function

HADAMARD(*n*, < , *i*> **);**

The HADAMARD function returns a Hadamard matrix. The arguments to the HADAMARD function are as follows:

n specifies the order of the Hadamard matrix. You can specify that *n* is 1, 2, or a multiple of 4. Furthermore, *n* must satisfy at least one of the following conditions:

- $n \leq 256$
- $n - 1$ is prime
- $(n/2) - 1$ is prime and $n/2 = 2 \bmod 4$
- $n = 2^p h$ for some positives integers p and h, and h satisfies one of the preceding conditions

When any other *n* is specified, the HADAMARD function returns a zero.

i specifies the row number to return. When *i* is not specified or *i* is negative, the full Hadamard matrix is returned.

The HADAMARD function returns a Hadamard matrix, which is an $n \times n$ matrix that consists entirely of the values 1 and –1. The columns of a Hadamard matrix are all orthogonal. Hadamard matrices are frequently used to make orthogonal array experimental designs for two-level factors. For example, the following statements create a 12×12 Hadamard matrix:

```
h = hadamard(12);
print h[format=2.];
```

The output is shown in Figure 23.129. The first column is an intercept and the next 11 columns form an orthogonal array experimental design for 11 two-level factors in 12 runs, 2^{11}.

Figure 23.129 A Hadamard Matrix

```
                                h

 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 1  1 -1  1 -1 -1 -1  1  1  1 -1  1
 1  1  1 -1  1 -1 -1 -1  1  1  1 -1
 1 -1  1  1 -1  1 -1 -1 -1  1  1  1
 1  1 -1  1  1 -1  1 -1 -1 -1  1  1
 1  1  1 -1  1  1 -1  1 -1 -1 -1  1
 1  1  1  1 -1  1  1 -1  1 -1 -1 -1
 1 -1  1  1  1 -1  1  1 -1  1 -1 -1
 1 -1 -1  1  1  1 -1  1  1 -1  1 -1
 1 -1 -1 -1  1  1  1 -1  1  1 -1  1
 1  1 -1 -1 -1  1  1  1 -1  1  1 -1
 1 -1  1 -1 -1 -1  1  1  1 -1  1  1
```

To request the seventeenth row of a Hadamard matrix of order 448, use the following statement:

```
h17 = hadamard(448, 17);
```

HALF Function

HALF(*matrix*);

The HALF function is an alias for the ROOT function, which computes the Cholesky decomposition of a symmetric positive definite matrix.

HANKEL Function

HANKEL(*matrix*);

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, $\mathbf{R}$, are the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, then the first p rows of $\mathbf{R}$ are the same as $\mathbf{A}$.

The HANKEL function is especially useful in time series applications that involve a set of variables that represent the present and past and a set of variables that represent the present and future. In this situation, the covariance matrix between the sets of variables 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 **R** is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 & \cdots & \mathbf{0} \\ \mathbf{A}_3 & \mathbf{A}_4 & \mathbf{A}_5 & \cdots & \mathbf{0} \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

If

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

and if **R** is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 & \cdots & \mathbf{0} \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

For example, the following statements produce Hankel matrices, as shown in Figure 23.130:

```
r1 = hankel({1 2 3 4 5});
r2 = hankel({1 2 ,
             3 4 ,
             5 6 ,
             7 8});
r3 = hankel({1 2 3 4 ,
             5 6 7 8});
print r1, r2, r3;
```

Figure 23.130 Hankel Matrices

r1

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

r2

1	2	5	6
3	4	7	8
5	6	0	0
7	8	0	0

Figure 23.130 *continued*

```
                                r3

                1          2          3          4
                5          6          7          8
                3          4          0          0
                7          8          0          0
```

HARMEAN Function

HARMEAN(*matrix***);**

The HARMEAN function returns a scalar that contains the harmonic mean of the elements of the input matrix. The input matrix must contain only nonnegative numbers. The harmonic mean of a set of positive numbers $a_1, a_2, \ldots, a_n$ is n divided by the sum of the reciprocals of a_i. That is, $n / \sum a_i^{-1}$.

The harmonic mean is zero if any of the a_i are zero. The harmonic mean is not defined for negative numbers. If any of the a_i are missing, they are excluded from the computation.

The harmonic mean is sometimes used to compute an average sample size in an unbalanced experimental design. For example, the following statements compute an average sample size for five samples:

```
sizes = { 8, 12, 23, 10, 8 }; /* sample sizes */
aveSize = harmean( sizes );
print aveSize;
```

Figure 23.131 Harmonic Mean

```
                              aveSize

                            10.486322
```

HDIR Function

HDIR(*matrix1, matrix2***);**

The HDIR function computes the horizontal direct product of two numeric matrices. This operation is useful in constructing design matrices of interaction effects.

Specifically, 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. The *matrix1* and *matrix2* arguments must have the same number of rows, which is also the same number of rows in the result matrix. The number of columns in the result matrix is equal to the product of the number of columns in *matrix1* and *matrix2*.

For example, the following statements produce the matrix `c`, shown in Figure 23.132:

```
a = {1 2,
     2 4,
     3 6};
b = {0  2,
     1  1,
     0 -1};
c = hdir(a, b);
print c;
```

Figure 23.132 Horizontal Direct Product

c			
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

HERMITE(*matrix*);

The HERMITE function uses elementary row operations to compute the Hermite normal form of a matrix. For square matrices this normal form is upper triangular and idempotent.

If the argument is square and nonsingular, the result is the identity matrix. In general the result satisfies the following four conditions (Graybill 1969):

- 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.

The following statements compute an example from Graybill (1969):

```
a = {3  6  9,
     1  2  5,
     2  4 10};
h = hermite(a);
print h;
```

Figure 23.133 Hermite Matrix

h		
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

HOMOGEN(*matrix*);

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 properties that $\mathbf{A} * \mathbf{X} = \mathbf{0}$ and $\mathbf{X}'\mathbf{X} = \mathbf{I}$. In other words, the columns of $\mathbf{X}$ form an orthonormal basis for the nullspace of A.

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 $\mathbf{X}$.

The following statements compute an example from Wilkinson and Reinsch (1971):

```
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);
print x;
```

Figure 23.134 Solutions to a Homogeneous System

x	
-0.419095	0
0.4405091	0.4185481
-0.052005	0.3487901
0.6760591	0.244153
0.4129773	-0.802217

In addition, you can use the HOMOGEN function to determine the rank of an $m \times n$ matrix $\mathbf{A}$ where $m \geq n$ by counting the number of columns in the matrix $\mathbf{X}$.

If A is an $n \times m$ matrix, then, in addition to the memory allocated for the return matrix, the HOMOGEN function temporarily allocates an $n^2 + nm$ array for performing its computation.

I Function

I(*dim***);**

The I function creates an identity matrix with *dim* rows and columns. The diagonal elements of an identity matrix are ones; all other elements are zeros. The value of *dim* must be an integer greater than or equal to 1. Noninteger operands are truncated to their integer part.

For example, the following statements compute a 3×3 identity matrix:

```
a = I(3);
print a;
```

Figure 23.135 An Identity Matrix

a		
1	0	0
0	1	0
0	0	1

IF-THEN/ELSE Statement

IF *expression* **THEN** *statement1* **;**

ELSE *statement2* **;**

The IF-THEN/ELSE statement conditionally executes statements. The ELSE statement is optional.

The arguments to the IF-THEN/ELSE statement are as follows:

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:

```
a = {0, 5, 1, 10};
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 the matrix **a** is less than 20, the scalar value **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, consider the following statements:

```
a = { 1 2, 3 4};
b = {-1 0, 0 1};
if a>b then do;
   /* statements */
end;
```

This code produces the same result as the following statements:

```
if all(a>b) then do;
   /* statements */
end;
```

IF statements can be nested within the clauses of other IF or ELSE statements. There is no limit on the number of nesting levels. Consider the following example:

```
if a>b then
   if a>abs(b) then do;
      /* statements */
   end;
```

Consider the following statements:

```
if a^=b then do;
   /* statements */
end;
if ^(a=b) then do;
   /* statements */
end;
```

The two IF statements are equivalent. In each case, the THEN clause is executed only when all corresponding elements of **a** and **b** are unequal.

Evaluation of the following statement requires only one element of **a** and **b** to be unequal in order for the expression to be true:

```
if any(a^=b) then do;
   /* statements */
end;
```

IFFT Function

IFFT(*f*);

The IFFT function computes the inverse finite Fourier transform of a matrix *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.

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

$$x_i = F_{1,1} + 2\sum_{j=2}^{np}\left(F_{j,1}\cos\left(\frac{2\pi}{n}(j-1)(i-1)\right) + F_{j,2}\sin\left(\frac{2\pi}{n}(j-1)(i-1)\right)\right) + q_i$$

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.

For the 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).

The expression 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 in which the FFT and IFFT functions can be used for operations on multidimensional or complex data (Gentleman and Sande 1966; Nussbaumer 1982).

As an example, the convolution of two vectors $\mathbf{x}$ ($n \times 1$) and $\mathbf{y}$ ($m \times 1$) can be accomplished by using the following module:

```
start conv(u,v);
/* w = conv(u,v) convolves vectors u and v.
 * Algebraically, convolution is the same operation as
 * multiplying the polynomials whose coefficients are the
 * elements of u and v. Straight convolution is too slow,
 * so use the FFT.
 *
 * Both of u and v are column vectors.
 */
   m = nrow(u);
   n = nrow(v);

   wn = m + n - 1;
   /* find p so that 2##(p-1) < wn <= 2##p */
   p = ceil( log(wn)/ log(2) );
   nice = 2##p;

   a = fft( u // j(nice-m,1,0) );
   b = fft( v // j(nice-n,1,0) );
   /* complex multiplication of a and b */
   wReal = a[,1]#b[,1] - a[,2]#b[,2];
```

```
   wImag = a[,1]#b[,2] + a[,2]#b[,1];
   w = wReal || wImag;
   z=ifft(w);
   z = z[1:wn,1] / nice;  /* take real part and first wn elements */
   return (z);
finish;

/* example of convolution of two waveforms */
TimeStep = 0.01;
t = T( do(0,8,TimeStep) );

Signal = j(nrow(t),1,5);
Signal[ loc(t>4) ] = -5;

ImpulseResponse = j(nrow(t),1,0);
ImpulseResponse[ loc(t<=2) ] = 3;

/* The time domain for this convolution is [0,16]
   with the same time step.
   For waveforms, rescale amplitude by the time step. */
y = conv(Signal,ImpulseResponse) * TimeStep;
```

Other applications of the FFT and IFFT functions include windowed spectral estimates and the inverse autocorrelation function.

IMPORTDATASETFROMR Call

CALL IMPORTDATASETFROMR(*SAS-data-set, RExpr***);**

You can use the IMPORTDATASETFROMR subroutine to transfer data from an R data frame to a SAS data set. It is easier to read the subroutine name when it is written in mixed case: ImportDataSetFromR.

The arguments to the subroutine are as follows:

SAS-data-set is a literal string or a character matrix that specifies the two-level name of a SAS data set (for example, Work.MyData).

RExpr is a literal string or a character matrix that specifies the name of an R data frame or, in general, an R expression that can be coerced to an R data frame.

You can call the subroutine provided that the following statements are true:

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See "The RLANG System Option" on page 190.)

The following statements create a data frame in R named RData and copy the data into Work.MyData. The SHOW CONTENTS statement is then used to display attributes of the Work.MyData data, which demonstrates that the data were successfully transferred.

```
proc iml;
submit / R;
z = c('a','b','c','d','e')
RData <- data.frame(x=1:5, y=(1:5)^2, z=z)
endsubmit;

call ImportDataSetFromR("Work.MyData", "RData");

use Work.MyData;
show contents;
close Work.MyData;
```

Figure 23.136 Contents of a SAS Data Set Created from R Data

```
DATASET : WORK.MYDATA.DATA

 VARIABLE                        TYPE  SIZE
 -------------------------------  ----  ----
 A                               num      8
 Y                               num      8
 Z                               char     1

Number of Variables   : 3
Number of Observations: 5
```

You can transfer data from a SAS data set into an R data frame by using the EXPORTDATASETTOR call. See Chapter 11, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

The names of the variables in the SAS data set are derived from the names of the variables in the R data frame. The following rules are used to convert an R variable name to a valid SAS variable name:

1. If the name is longer than 32 characters, it is truncated to 32 characters.
2. A SAS variable name must begin with one of the following characters: 'A'–'Z', 'a'–'z', or the underscore (_). Therefore, if the first character is not a valid beginning character, it is replaced by an underscore (_).
3. A SAS variable name can contain only the following characters: 'A'–'Z', 'a'–'z', '0'–'9', or the underscore (_). Therefore, if any of the remaining characters is not valid in a SAS variable name, it is replaced by an underscore.
4. If the resulting name duplicates an existing name in the data set, a number is appended to the name to make it unique. If appending the number causes the length of the name to exceed 32 characters, the name is truncated to make room for the number.

IMPORTMATRIXFROMR Call

CALL IMPORTMATRIXFROMR(*IMLMatrix, RExpr***);**

You can use the IMPORTMATRIXFROMR subroutine to transfer data from an R data frame to a SAS data set. It is easier to read the subroutine name when it is written in mixed case: ImportMatrixFromR.

The arguments to the subroutine are as follows:

IMLMatrix is a SAS/IML matrix to contain the data you want to transfer.

RExpr is a literal string or a character matrix that specifies the name of an R matrix, data frame, or an R expression that can be coerced to an R data frame.

If the *RExpr* argument is a data frame, then the resulting SAS/IML matrix has columns that correspond to variables from the data frame. If the first variable in the data frame is a numeric variable, a numeric matrix is created from all numeric variables in the data frame. If the first variable in the data frame is a character variable, a character matrix is created from all character variables in the data frame.

You can call the subroutine provided that the following statements are true:

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See "The RLANG System Option" on page 190.)

The following statements define an R matrix and copy the data from the matrix to a SAS/IML matrix:

```
proc iml;
submit / R;
m <- matrix( c(1,2,3,4,NA,6), nrow=2, byrow=TRUE)
endsubmit;

call ImportMatrixFromR(a, "m");
print a;
```

To demonstrate that the data were successfully transferred, the PRINT statement is used to print the values of the **a** matrix. The output is shown in Figure 23.137. Note that the R missing value (**NA**) in the R matrix **m** was automatically converted to the SAS missing value in the SAS/IML matrix, **a**.

Figure 23.137 Data from R

a		
1	2	3
4	.	6

You can transfer data from a SAS/IML matrix into an R matrix frame by using the EXPORTMATRIXTOR call. See Chapter 11, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

INDEX Statement

INDEX *variables* | **NONE ;**

The INDEX statement creates an index for the named variables in the current input SAS data set. An index is created for each variable listed, provided that the variable 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 can use this index to retrieve observations from the data. 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 statements copy the Sasuser.Class data set and create indexes for the Name and Sex variables. Current retrieval is set to use the Sex variable, as shown in Figure 23.138.

```
data class;
   set sashelp.class;
run;

proc iml;
use class;
index name sex;
list all;
close class;
```

Figure 23.138 Result of Listing Observations of an Indexed Data Set

OBS	Name	Sex	Age	Height	Weight
2	Alice	F	13.0000	56.5000	84.0000
3	Barbara	F	13.0000	65.3000	98.0000
4	Carol	F	14.0000	62.8000	102.5000
7	Jane	F	12.0000	59.8000	84.5000
8	Janet	F	15.0000	62.5000	112.5000
11	Joyce	F	11.0000	51.3000	50.5000
12	Judy	F	14.0000	64.3000	90.0000
13	Louise	F	12.0000	56.3000	77.0000
14	Mary	F	15.0000	66.5000	112.0000
1	Alfred	M	14.0000	69.0000	112.5000
5	Henry	M	14.0000	63.5000	102.5000
6	James	M	12.0000	57.3000	83.0000
9	Jeffrey	M	13.0000	62.5000	84.0000
10	John	M	12.0000	59.0000	99.5000
15	Philip	M	16.0000	72.0000	150.0000
16	Robert	M	12.0000	64.8000	128.0000
17	Ronald	M	15.0000	67.0000	133.0000
18	Thomas	M	11.0000	57.5000	85.0000
19	William	M	15.0000	66.5000	112.0000

The INDEX NONE statement can be used to set retrieval back to physical order.

When a WHERE clause is being processed, the SAS/IML language 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

INFILE *operand* < *options* > **;**

The INFILE statement opens an external file for input or, if the file is already open, makes it the current input file. Subsequent INPUT statements read from the specified file.

The arguments to the INFILE statement are as follows:

operand	is either a predefined filename or a quoted string that contains in parentheses the filename or character expression that refers to the pathname.
options	are explained in the following list.

The valid values for the *options* argument are as follows:

LENGTH=variable
specifies a variable into which the length of a record is stored.

RECFM=N
specifies that the file 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 (<) in the INPUT statement to get new records rather than use separate input statements or the new line (/) operator.

The following keywords control how a program behaves when an INPUT statement tries to read past the end of a record. The default behavior is STOPOVER.

FLOWOVER
enables 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 follow:

```
filename in1 "student.dat";        /* specify filename IN1   */
infile in1;                        /* infile pathname       */

infile "student.dat";              /* path by quoted literal */

infile "student.dat" missover;     /* use missover option   */
```

See Chapter 8 for further information.

INPUT Statement

INPUT < *variables* > < *informats* > < *record-directives* > < *positionals* > **;**

The INPUT statement reads records from the current input file, placing the values into matrices. The INFILE statement sets up the current input file. See Chapter 8 for details.

The INPUT statement supports the following arguments:

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 named SAS format such as BEST*w.d*. 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 about formats, see *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 8.

Examples of valid INPUT statements follow:

```
input x y;
input @1 name $ @20 sex $ @(20+2) age 3.;

eight=8;
input >9 <eight  number2 ib8.;
```

The following example uses binary input:

```
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 hold the current record so that you can continue to read from the record with later INPUT statements. Otherwise, the next record is used for the next INPUT statement.
	/	advances to the next record.
	> *operand*	specifies that the next record to be read 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.
	< *operand*	specifies that the indicated number of bytes are read 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	specifies a specific column on the record. The *positionals* are the following:	
	@ *operand*	specifies a 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 8.
	+ *operand*	skips the indicated number of columns. The *operand* is a literal number, a variable name, or an expression in parentheses.

INSERT Function

INSERT(*x, y, row*< *, column*>**);**

The INSERT function inserts one matrix inside another.

The arguments 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* has m rows and n columns, *row* can range from 0 to $m + 1$ and *column* can range from 0 to $n + 1$.

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.

The following statements show two examples of the INSERT function. Figure 23.139 shows that the matrix **c** is the result of inserting matrix **b** prior to the second row of matrix **a**. The matrix **d** is the result of inserting matrix **b** after the second column of matrix **a**.

```
a = {1 2, 3 4};
b = {5 6, 7 8};
c = insert(a, b, 2, 0);
d = insert(a, b, 0, 3);
print c, d;
```

Figure 23.139 Inserted Matrices

```
                                c

                              1         2
                              5         6
                              7         8
                              3         4

                                d

                    1         2         5         6
                    3         4         7         8
```

INT Function

INT(*matrix***);**

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, as shown in the following statements:

```
y = 2.8;
b = int(y);
x={12.95  10.9999999999999,
   -30.5  1e-6};
c = int(x);
print b, c;
```

Figure 23.140 Truncated Values

```
                                b

                                2

                                c

                              12        11
                             -30         0
```

In Figure 23.140, notice that the value 11 is returned as the second element of **c**. If a value is within 10^{-12} of an integer, the INT function rounds up.

INV Function

INV(*matrix***);**

The INV function computes the inverse of a square and nonsingular matrix.

For $\mathbf{G} = \text{INV}(\mathbf{A})$ 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 expression **x = inv(a)*b**. However, the SOLVE function is more accurate and efficient for this task.

The following statements compute a matrix inverse and solve a linear system:

```
A = {0 0 1 0 1,
     1 0 0 1 0,
     0 1 1 0 1,
     1 0 0 0 1,
     0 1 0 1 0};

 b = {9, 4, 10, 8, 2};

 /* find inverse and solve linear system */
 Ainv = inv(A);
 x1 = Ainv*b;

 /* solve by using a more efficient algorithm */
 x2 = solve(A,b);
 print x1 x2;
```

Figure 23.141 Solving a Linear System

x1	x2
3	3
1	1
4	4
1	1
5	5

The INV function uses an LU decomposition followed by back substitution to solve for the inverse, as described in Forsythe, Malcom, and Moler (1967).

The INV function (in addition to 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:

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

where MACHEPS is the relative machine precision.

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

The criterion is used by some functions to detect a singular matrix and to abort a computation that cannot be performed on a singular matrix. The typical error message is as follows:

```
ERROR: (execution) Matrix should be non-singular.
```

If you are getting this error message but believe that your matrix is actually nonsingular, you can try one of the following:

- Center and scale the data.
- Use the GINV function to compute the generalized inverse.
- Examine the size of the singular values returned by the SVD call. The SVD call can be used to compute a generalized inverse with a user-specified singularity criterion.

If $\mathbf{A}$ is an $n \times n$ matrix, the INV function allocates an $n \times n$ matrix in order to return the inverse. It also temporarily allocates an n^2 array in order to compute the inverse.

INVUPDT Function

INVUPDT(*matrix, vector*< , *scalar*>**);**

The INVUPDT function updates a matrix inverse.

The arguments to the INVUPDT function are as follows:

matrix	is an $n \times n$ nonsingular matrix. In most applications *matrix* is symmetric positive definite.
vector	is an $n \times 1$ or $1 \times n$ vector.
scalar	is a numeric scalar.

The Sherman-Morrison-Woodbury formula is

$$(\mathbf{A} + \mathbf{U}\mathbf{V}')^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}'\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}'\mathbf{A}^{-1}$$

where $\mathbf{A}$ is an $n \times n$ nonsingular matrix and $\mathbf{U}$ and $\mathbf{V}$ are $n \times k$. The formula shows that a rank k update to $\mathbf{A}$ corresponds to a rank k update of $\mathbf{A}^{-1}$.

The INVUPDT function implements the Sherman-Morrison-Woodbury formula for rank-one updates with $\mathbf{U} = w\mathbf{X}$ and $\mathbf{V} = \mathbf{X}$, where $\mathbf{X}$ is an $n \times 1$ vector and w is a scalar.

If $\mathbf{M} = \mathbf{A}^{-1}$, then you can call the INVUPDT function as follows:

```
R = invupdt(M, X, w);
```

This statement computes the following matrix:

$$\mathbf{R} = \mathbf{M} - w\mathbf{M}\mathbf{X}(\mathbf{I} + w\mathbf{X}'\mathbf{M}\mathbf{X})^{-1}\mathbf{X}'\mathbf{M}$$

The matrix $\mathbf{R}$ is equivalent to $(\mathbf{A} + w\mathbf{X}\mathbf{X}')^{-1}$. If $\mathbf{A}$ is symmetric positive definite, then so is $\mathbf{R}$.

If w is not specified, then it is given a default value of 1.

A common use of the INVUPDT function is in linear regression. If $\mathbf{Z}$ is a design matrix, $\mathbf{M} = (\mathbf{Z}'\mathbf{Z})^{-1}$ is the associated inverse crossproduct matrix, and $\mathbf{v}$ 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 $\mathbf{M}$ by using the following statement:

```
M2 = invupdt(M, v);
```

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.

The following program demonstrates adding or removing observations from a linear fit and updating the inverse crossproduct matrix:

```
X = {0, 1, 1, 1, 2, 2, 3, 4, 4};
Y = {1, 1, 2, 6, 2, 3, 3, 3, 4};

/* find linear fit */
Z = j(nrow(X), 1, 1) || X;           /* design matrix */
M = inv(Z`*Z);

b = M*Z`*Y;                          /* LS estimate */
resid = Y - Z*b;                     /* residuals */
print "Original Fit", b resid;

/* residual for observation (1,6) seems too large.
   Take obs number 4 out of data set and refit. */
v = z[4,];
M = invupdt(M, v, -1);               /* update inverse crossprod */

keepObs = (1:3) || (5:nrow(X));
Z = Z[keepObs, ];
Y = Y[keepObs, ];
b = M*Z`*Y;                          /* new LS estimate */
print "After deleting observation 4", b;

/* Add a new obs (x,y) = (0,2) and refit. */
obs = {0 2};
v = 1 || obs[1];                     /* new row in design matrix */
```

```
M = invupdt(M, v);

Z = Z // v;
Y = Y // obs[2];
b = M*Z`*Y;                          /* new LS estimate */
print "After adding observation (0,2)", b;
```

Figure 23.142 Refitting Linear Regression Models

```
                    Original Fit

                        b     resid

                2.0277778 -1.027778
                    0.375 -1.402778
                          -0.402778
                          3.5972222
                          -0.777778
                          0.2222222
                          -0.152778
                          -0.527778
                          0.4722222

            After deleting observation 4

                        b

                            1
                    0.6470588

           After adding observation (0,2)

                        b

                          1.3
                    0.5470588
```

IPF Call

CALL IPF(*fit, status, dim, table, config* < , *initab*> < , *mod*> **);**

The IPF subroutine performs an iterative proportional fit of a contingency table. This is a standard statistical technique to obtain maximum likelihood estimates for cells under any hierarchical log-linear model. The algorithm is described in Bishop, Fienberg, and Holland (1975).

The arguments to the IPF subroutine are as follows:

fit is a returned matrix. The matrix *fit* contains an array of the estimates of the expected number in each cell under the model specified in *config*. This matrix conforms to *table*, meaning that it has the same dimensions and order of variables.

status is a returned matrix. The *status* argument is a row vector of length 3. *status[1]* is 0 if there is convergence to the desired accuracy, otherwise it is nonzero. *status[2]* is the maximum difference between estimates of the last two iterations of the IPF algorithm. *status[3]* is the number of iterations performed.

dim is an input matrix. If the problem contains v variables, then *dim* is $1 \times v$ row vector. The value *dim[i]* is the number of possible levels for variable i in a contingency table.

table is an input matrix that specifies an array of the number of observations at each level of each variable. Variables are nested across columns and then across rows.

config is an input matrix that specifies which marginal totals to fit. Each column of *config* 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 that specifies initial values for the iterative procedure. If you do not specify values, ones 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 a two-element vector that specifies the stopping criteria. If *mod*= {*MaxDev*, *MaxIter*}, then the procedure iterates either until the maximum difference between estimates of the last two iterations is less than *MaxDev* or until *MaxIter* iterations are completed. Default values are *MaxDev*=0.25 and *MaxIter*=15.

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*.

Adjusting a Table from Marginals

A common use of the IPF subroutine is to adjust the entries of a table in order to fit a new set of marginals while retaining the interaction between cell entries.

Example 1: Adjusting Marital Status by Age Bishop, Fienberg, and Holland (1975) present data from D. Friedlander that shows the distribution of women in England and Wales according to their marital status in 1957. One year later, new official marginal estimates were announced. The problem is to adjust the entries in the 1957 table so as to fit the new marginals while retaining the interaction between cells. This problem can arise when you have internal cells that are known from sampling a population and then get margins based on a complete census.

When you want to adjust an observed table of cell frequencies to a new set of margins, you must set the *initab* parameter to be the table of observed values. The new marginals are specified through the *table* argument. The particular cell values for *table* are not important, since only the marginals are used (the proportionality between cells is determined by *initab*).

There are two easy ways to create a table that contains given margins. Recall that a table of independent variables has an expected cell value $A_{ij} =$ (sum of row i)(sum of col j)/(sum of all cells). Thus you could form a table with these cell entries. Another possibility is to use a "greedy algorithm" to assign as many of the marginals as possible to the first cell, then assign as many of the remaining marginals as possible to

the second cell, and so on until all of the marginals have been distributed. Both of these approaches are encapsulated into modules in the following program:

```
/* Return a table such that cell (i,j) has value
   (sum of row i)(sum of col j)/(sum of all cells) */
start GetIndepTableFromMargins( bottom, side );
   if bottom[+] ^= side[+] then do;
      print "Marginal totals are not equal";
      abort;
   end;
   table = side*bottom/side[+];
   return (table);
finish;

/* Use a "greedy" algorithm to create a table whose
   marginal totals match given marginal totals.
   Margin1 is the vector of frequencies totaled down
           each column. Margin1 means that
           Variable 1 has NOT been summed over.
   Margin2 is the vector of frequencies totaled across
           each row. Margin2 means that Variable 2
           has NOT been summed over.
   After calling, use SHAPE to change the shape of
           the returned argument. */
start GetGreedyTableFromMargins( Margin1, Margin2 );
   /* copy arguments so they are not corrupted */
   m1 = colvec(Margin1); /* colvec is in IMLMLIB */
   m2 = colvec(Margin2);
   if m1[+] ^= m2[+] then do;
      print "Marginal totals are not equal";
      abort;
   end;
   dim1 = nrow(m1);
   dim2 = nrow(m2);
   table = j(1,dim1*dim2,0);
   /* give as much to cell (1,1) as possible,
      then as much as remains to cell (1,2), etc,
      until all the margins have been distributed */
   idx = 1;
   do i2 = 1 to dim2;
      do i1 = 1 to dim1;
         t = min(m1[i1],m2[i2]);
         table[idx] = t;
         idx = idx + 1;
         m1[i1] = m1[i1]-t;
         m2[i2] = m2[i2]-t;
      end;
   end;
   return (table);
finish;

Mod = {0.01 15}; /* tighten stopping criterion */

Columns = {"  Single" " Married" "Widow/Divorced"};
```

```
Rows    = {"15 - 19" "20 - 24" "25 - 29" "30 - 34"
           "35 - 39" "40 - 44" "45 - 49" "50 Or Over"};

/* Marital status has 3 levels. Age has 8 levels */
Dim = {3 8};

/* Use known distribution for start-up values */
IniTab = { 1306  83    0 ,
            619  765    3 ,
            263 1194    9 ,
            173 1372   28 ,
            171 1393   51 ,
            159 1372   81 ,
            208 1350  108 ,
           1116 4100 2329 };

/* New marginal totals for age by marital status */
NewMarital = { 3988 11702 2634 };
NewAge = {1412,1402,1450,1541,1681,1532,1662,7644};

/* Create any table with these marginals */
Table = GetGreedyTableFromMargins(NewMarital, NewAge);
Table = shape(Table, nrow(IniTab), ncol(IniTab));

/* Consider all main effects */
Config = {1 2};

call ipf(Fit, Status, Dim, Table, Config, IniTab, Mod);

if Status[1] = 0 then
   print "Known Distribution (1957)",
      IniTab [colname=Columns rowname=Rows format=8.0],,
      "Adjusted Estimates of Distribution (1958)",
      Fit [colname=Columns rowname=Rows format=8.2];
else
   print "IPF did not converge in "
         (Status[3]) " iterations";
```

The results of this program are shown in Figure 23.143. The same results are obtained if the *table* parameter is formed by using the "independent algorithm."

Figure 23.143 Iterative Proportional Fitting

Known Distribution (1957)

	IniTab		
	Single	Married	Widow/Divorced
15 – 19	1306	83	0
20 – 24	619	765	3
25 – 29	263	1194	9
30 – 34	173	1372	28
35 – 39	171	1393	51
40 – 44	159	1372	81
45 – 49	208	1350	108
50 Or Over	1116	4100	2329

Adjusted Estimates of Distribution (1958)

	Fit		
	Single	Married	Widow/Divorced
15 – 19	1325.27	86.73	0.00
20 – 24	615.56	783.39	3.05
25 – 29	253.94	1187.18	8.88
30 – 34	165.13	1348.55	27.32
35 – 39	173.41	1454.71	52.87
40 – 44	147.21	1308.12	76.67
45 – 49	202.33	1352.28	107.40
50 Or Over	1105.16	4181.04	2357.81

Example 2: Adjusting Votes by Region A similar technique can be used to standardize data from raw counts into percentages. For example, consider data from a 1836 vote in the U.S. House of Representatives on a resolution that the House should adopt a policy of tabling all petitions for the abolition of slavery. Attitudes toward abolition were different among slaveholding states that would later secede from the Union ("the South"), slaveholding states that refused to secede ("the Border States"), and nonslaveholding states ("the North").

The raw votes for the resolution are defined in the following statements. The data are hard to interpret because the margins are not homogeneous.

```
/*          Yea Abstain Nay */
IniTab = {  61    12    60,   /* North */
            17     6     1,   /* Border */
            39    22     7 }; /* South */
```

Standardizing the data by specifying homogeneous margins reveals interactions and symmetry that were not apparent in the raw data. Suppose the margins are specified as follows:

```
NewVotes   = {100 100 100};
NewSection = {100,100,100};
```

In this case, the program for marital status by age can be easily rewritten to adjust the votes into a standardized form. The resulting output is shown in Figure 23.144:

Figure 23.144 Standardizing Counts into Percentages

	Fit		
	Yea	Abstain	Nay
North	20.1	10.2	69.7
Border	47.4	42.8	9.8
South	32.5	47.0	20.5

Generating a Table with Given Marginals The "greedy algorithm" presented in the Marital-Status-By-Age example can be extended in a natural way to the case where you have n one-way marginals and want to form an n-dimensional table. For example, a three-dimensional "greedy algorithm" would allocate the vector *table* as `table=j(dim1*dim2*dim3,1,0);` and have three nested loops as indicated in the following statements. Afterwards, the *table* parameter can be reshaped by using the SHAPE function.

```
do i3 = 1 to dim3;
   do i2 = 1 to dim2;
      do i1 = 1 to dim1;
         t = min(m1[i1],m2[i2],m3[i3]);
         table[idx] = t;
         idx = idx + 1;
         m1[i1] = m1[i1]-t;
         m2[i2] = m2[i2]-t;
         m3[i3] = m3[i3]-t;
      end;
   end;
end;
```

The idea of the "greedy algorithm" can be extended to marginals that are not one-way. For example, the following three-dimensional table is similar to one that appears in Christensen (1997) based on data from M. Rosenberg. The table presents data on a person's self-esteem for people classified according to their religion and their father's educational level.

		Father's Educational Level				
Religion	Self-Esteem	Not HS Grad	HS Grad	Some Coll	Coll Grad	Post Coll
Catholic	High	575	388	100	77	51
	Low	267	153	40	37	19
Jewish	High	117	102	67	87	62
	Low	48	35	18	12	13
Protestant	High	359	233	109	197	90
	Low	159	173	47	82	32

Since the father's education level is nested across columns, it is Variable 1 with levels that correspond to not finishing high school, graduating from high school, attending college, graduating from college, and

attending graduate courses. The variable that varies the quickest across rows is Self-Esteem, so Self-Esteem is Variable 2 with values "High" and "Low." The Religion variable is Variable 3 with values "Catholic," "Jewish," and "Protestant."

The following program encodes this table by using the MARG call to compute a two-way marginal table by summing over the third variable, and a one-way marginal by summing over the first two variables. Then a new table (**NewTable**) is created by applying the greedy algorithm to the two marginals. Finally, the marginals of **NewTable** are computed and compared with those of **table**.

```
dim={5 2 3};

table={
/* Father's Education:
           NotHSGrad HSGrad Col ColGrad PostCol
          Self-
   Relig  Esteem                          */
/* Cath-   Hi */ 575   388  100    77   51,
/* olic    Lo */ 267   153   40    37   19,

/* Jew-    Hi */ 117   102   67    87   62,
/*  ish    Lo */  48    35   18    12   13,

/* Prot-   Hi */ 359   233  109   197   90,
/* estant  Lo */ 159   173   47    82   32
        };

config = { 1 3,
           2 0 };
call marg(locmar, marginal, dim, table, config);
print locmar, marginal, table;

/* Examine marginals: The name indicates the
   variable(s) that are NOT summed over.
   The locmar variable tells where to index
   into the marginal variable. */
Var12_Marg = marginal[1:(locmar[2]-1)];
Var12_Marg = shape(Var12_Marg,dim[2],dim[1]);
Var3_Marg  = marginal[locMar[2]:ncol(marginal)];

NewTable = j(nrow(table),ncol(table),0);
/* give as much to cell (1,1,1) as possible,
   then as much as remains to cell (1,1,2), etc,
   until all the margins have been distributed. */
idx = 1;
do i3 = 1 to dim[3];        /* over Var3 */
   do i2 = 1 to dim[2];     /* over Var2 */
      do i1 = 1 to dim[1];  /* over Var1 */
         /* Note Var12_Marg has Var1 varying across
            the columns */
         t = min(Var12_Marg[i2,i1],Var3_Marg[i3]);
         NewTable[idx] = t;
         idx = idx + 1;
         Var12_Marg[i2,i1] = Var12_Marg[i2,i1]-t;
```

```
         Var3_Marg[i3] = Var3_Marg[i3]-t;
      end;
   end;
end;

call marg(locmar, NewMarginal, dim, table, config);
maxDiff = abs(marginal-NewMarginal)[<>];
if maxDiff=0 then
   print "Marginals are unchanged";
print NewTable;
```

Figure 23.145 Table with Given Marginals

locmar

1	11

marginal

	COL1	COL2	COL3	COL4	COL5	COL6	COL7
ROW1	1051	723	276	361	203	474	361

marginal

	COL8	COL9	COL10	COL11	COL12	COL13
ROW1	105	131	64	1707	561	1481

table

575	388	100	77	51
267	153	40	37	19
117	102	67	87	62
48	35	18	12	13
359	233	109	197	90
159	173	47	82	32

Marginals are unchanged

NewTable

1051	656	0	0	0
0	0	0	0	0
0	67	276	218	0
0	0	0	0	0
0	0	0	143	203
474	361	105	131	64

Fitting a Log-Linear Model to a Table

A second common usage of the IPF algorithm is to hypothesize that the table of observations can be fitted by a model with known effects and to ask whether the observed values indicate that the model hypothesis can be accepted or should be rejected. In this usage, you normally do not specify the *initab* argument to the IPF subroutine (but see the comment on structural zeros in the section "Additional Details" on page 741).

Example 3: Food Illness Korff, Taback, and Beard (1952) reported statistics related to the outbreak of food poisoning at a company picnic. A total of 304 people at the picnic were surveyed to determine who had eaten either of two suspect foods: potato salad and crabmeat. The predictor variables are whether the individual ate potato salad (Variable 1: "Yes" or "No") and whether the person ate crabmeat (Variable 2: "Yes" or "No"). The response variable is whether the person was ill (Variable 3: "Ill" or "Not Ill"). The order of the variables is determined by the *dim* and *table* arguments to the IPF subroutine. The variables are nested across columns, then across rows.

Crabmeat:	YES		NO	
Potato salad:	Yes	No	Yes	No
Ill	120	4	22	0
Not Ill	80	31	24	23

The following program defines the variables and observations, and then fits three separate models. How well each model fits the data is determined by computing a Pearson chi-square statistic $\chi^2 = \sum(O-E)^2/E$, where the sum is over all cells, O stands for the observed cell count, and E stands for the fitted estimate. Other statistics, such as the likelihood-ratio chi-square statistic $G^2 = -2\sum O\log(E/O)$, could also be used.

The program first fits a model that excludes the three-way interaction. The model fits well, so you can conclude that an association between illness and potato salad does not depend on whether an individual ate crabmeat. The next model excludes the interaction between potato salad and illness. This model is rejected with a large chi-square value, so the data support an association between potato salad and illness. The last model excludes the interaction between the crabmeat and the illness. This model fits moderately well.

```
/* Compute a chi-square score for a table of observed
   values, given a table of expected values. Compare
   this score to a chi-square value with given degrees
   of freedom at 95% confidence level. */
start ChiSqTest( obs, model, degFreedom );
   diff = (obs - model)##2 / model;
   chiSq = diff[+];
   chiSqCutoff = cinv(0.95, degFreedom);
   print chiSq chiSqCutoff;
   if chiSq > chiSqCutoff then
      print "Reject hypothesis";
   else
      print "No evidence to reject hypothesis";
finish;

dim={2 2 2};

/* Crab meat:   Y E S        N O
   Potato:   Yes   No      Yes  No   */
table={      120    4      22   0,   /* Ill */
              80   31      24  23 }; /* Not Ill */

crabmeat = "          C R A B      N O  C R A B";
potato  = {"YesPot" "NoPot" "YesPot" "NoPot"};
illness = {"Ill", "Not Ill"};
```

```
hypoth = "Hypothesis: no three-factor interaction";
config={1  1  2,
        2  3  3};
call ipf(fit,status,dim,table,config);

print hypoth, "Fitted Model:",
      fit[label=crabmeat colname=potato
          rowname=illness format=6.2];
run ChiSqTest(table, fit, 1); /* 1 deg of freedom */

/* Test for interaction between Var 3 (Illness) and
   Var 1 (Potato Salad) */
hypoth = "Hypothesis: no Illness-Potato Interaction";
config={1 2,
        2 3};
call ipf(fit,status,dim,table,config);

print hypoth, "Fitted Model:",
      fit[label=crabmeat colname=potato
          rowname=illness format=6.2];
run ChiSqTest(table, fit, 2); /* 2 deg of freedom */

/* Test for interaction between Var 3 (Illness) and
   Var 2 (Crab meat) */
hypoth = "Hypothesis: no Illness-Crab Interaction";
config={1 1,
        2 3};
call ipf(fit,status,dim,table,config);

print hypoth, "Fitted Model:",
      fit[label=crabmeat colname=potato
          rowname=illness format=6.2];
run ChiSqTest(table, fit, 2); /* 2 deg of freedom */
```

Figure 23.146 Fitting Log-Linear Models

```
                                hypoth

                  Hypothesis: no three-factor interaction

                              Fitted Model:

                            C R A B      N O  C R A B
                          YesPot  NoPot YesPot  NoPot

                   Ill     121.08   2.92  20.92   1.08
                 Not Ill  78.92  32.08  25.07  21.93

                             chiSq chiSqCutoff

                           1.7021335   3.8414588

                    No evidence to reject hypothesis
```

Figure 23.146 *continued*

```
                              hypoth

                 Hypothesis: no Illness-Potato Interaction

                              Fitted Model:

                            C R A B      N O  C R A B
                          YesPot  NoPot YesPot  NoPot

                Ill      105.53  18.47  14.67   7.33
                Not Ill   94.47  16.53  31.33  15.67

                            chiSq chiSqCutoff

                          44.344643   5.9914645

                           Reject hypothesis

                              hypoth

                Hypothesis: no Illness-Crab Interaction

                              Fitted Model:

                            C R A B      N O  C R A B
                          YesPot  NoPot YesPot  NoPot

                Ill      115.45   2.41  26.55   1.59
                Not Ill   84.55  32.59  19.45  21.41

                            chiSq chiSqCutoff

                          5.0945132   5.9914645

                        No evidence to reject hypothesis
```

Additional Details

Structural versus Random Zeros In the Marital-Status-By-Age example, the *initab* argument contained a zero for the "15–19 and Widowed/Divorced" category. Because the *initab* parameter determines the proportionality between cells, the fitted model retains a zero in that category. By contrast, in the Food-Illness example, the *table* parameter contained a zero for number of illnesses observed among those who did not eat either crabmeat or potato salad. This is a sampling (*random*) zero. Some models preserve that zero; others do not. If your table has a *structural zero* (for example, the number of ovarian cancers observed among male patients), then you can use the *initab* parameter to preserve that zero. see Bishop, Fienberg, and Holland (1975) or the documentation for the CATMOD procedure in the *SAS/STAT User's Guide* for more information about structural zeros and incomplete tables.

The *config* Parameter The columns of this matrix specify which interaction effects should be included in the model. The following table gives the model and the configuration parameter for common

interactions for an $I \times J \times K$ table in three dimensions. The so-called *noncomprehensive* models that do not include all variables (for example, *config* = {1}) are not listed in the table, but can be used. You can also specify combinations of main and interaction effects. For example, *config* = {1 3, 2 0}) specifies all main effects and the 1-2 interaction. Bishop, Fienberg, and Holland (1975) and Christensen (1997) explain how to compute the degrees of freedom associated with any model. For models with structural zeros, computing the degrees of freedom is complicated.

Model	config	Degrees of Freedom
No three-factor	{1 1 2, {2 3 3}	$(I-1)(J-1)(K-1)$
One two-factor absent	{1 2, {3 3}	$(I-1)(J-1)K$
	{1 2, 2 3}	$(I-1)J(K-1)$
	{1 1, {2 3}	$I(J-1)(K-1)$
Two two-factor absent	{2, 3}	$(I-1)(JK-1)$
	{1, 3}	$(J-1)(IK-1)$
	{1, 2}	$(K-1)(IJ-1)$
No two-factor	{1 2 3}	$IJK-(I+J+K)+2$
Saturated	{1,2,3}	IJK

The Shape of the *table* Parameter Since variables are nested across columns and then across rows, any shape that conforms to the *dim* parameter is equivalent.

For example, the section "Generating a Table with Given Marginals" on page 736 presents data on a person's self-esteem for people classified according to their religion and their father's educational level. To save space, the educational levels are subsequently denoted by labels that indicate the typical number of years spent in school: "<12," "12," "<16," "16," and ">16."

The table would be encoded as follows:

```
dim={5 2 3};

table={
/* Father's Education:
                  <12    12   <16   16   >16
          Self-
   Relig  Esteem                          */
/* Cath-   Hi */ 575   388  100    77   51,
/* olic    Lo */ 267   153   40    37   19,

/* Jew-    Hi */ 117   102   67    87   62,
/*  ish    Lo */  48    35   18    12   13,

/* Prot-   Hi */ 359   233  109   197   90,
/* estant  Lo */ 159   173   47    82   32
        };
```

The same information for the same variables in the same order could also be encoded into an $n \times m$ table in two other ways. Recall that the product of entries in *dim* is nm and that m must equal the product of the first k entries of *dim* for some k. For this example, the product of the entries in *dim* is

30, and so the table must be 6 × 5, 3 × 10, or 1 × 30. The 3 × 10 table is encoded as concatenating rows 1–2, 3–4, and 5–6 to produce the following:

```
table={
/* Esteem: H I G H           L O W                    */
/*     <12  ...  >16    <12   ...  >16                 */

       575  ...   51    267   ...   19, /* Catholic  */
       117  ...   62     48   ...   13, /* Jewish    */
       359  ...   90    159   ...   32  /* Protestant*/
      };
```

The 1 × 30 table is encoded by concatenating all rows, as follows:

```
table={
/*     CATHOLIC            ...        PROTESTANT
   High          Low       ...      High       Low
<12 ... >16 <12 ... >16    ...  <12 ... >16 <12 ... >16
*/
575 ...  51 267 ...  19    ...   359 ... 90 159 ... 32
      };
```

ITSOLVER Call

CALL ITSOLVER(*x, error, iter, method, A, b* < *, precon*> < *, tol*> < *, maxiter*> < *, start*> < *, history*> **);**

The ITSOLVER subroutine solves a sparse linear system by using iterative methods.

The ITSOLVER call returns the following values:

x is the solution to *Ax=b*.

error is the final relative error of the solution.

iter is the number of iterations executed.

The input arguments to the ITSOLVER call are as follows:

method is the type of iterative method to use. The following values are valid:

- `"CG"` specifies a conjugate gradient algorithm. The matrix *A* must be symmetric and positive definite.
- `"CGS"` specifies a conjugate gradient squared algorithm, for general *A*.
- `"MINRES"` specifies a minimum residual algorithm, when *A* is symmetric indefinite.
- `"BICG"` specifies a biconjugate gradient algorithm, for general *A*.

A is the sparse coefficient matrix in the equation *Ax=b*. You can use SPARSE function to convert a matrix from dense to sparse storage.

b is a column vector, the right side of the equation *Ax=b*.

precon is the name of a preconditioning technique to use. The following values are valid:

- `"NONE"` specifies no preconditioning. This is the default behavior if the argument is not specified.
- `"IC"` specifies an incomplete Cholesky factorization. Specify this value when you specify `"CG"` or `"MINRES"` for the *method* argument.
- `"DIAG"` specifies a diagonal Jacobi preconditioner. Specify this value when you specify `"CG"` or `"MINRES"` for the *method* argument.
- `"MILU"` specifies a modified incomplete LU factorization. Specify this value when you specify `"BICG"` for the *method* argument.

tol is the relative error tolerance.

maxiter is the iteration limit.

start is a starting point column vector.

history is a matrix to store the relative error at each iteration.

The ITSOLVER call solves a sparse linear system by iterative methods, which involve updating a trial solution over successive iterations to minimize the error. The ITSOLVER call uses the technique specified in the *method* parameter to update the solution.

The input matrix A represents the coefficient matrix in sparse format; it is an $n \times 3$ matrix, where n is the number of nonzero elements. The first column contains the nonzero values, and the second and third columns contain the row and column locations for the nonzero elements, respectively. For the algorithms that assume symmetric A, only the lower triangular elements should be specified. The algorithm continues iterating to improve the solution until either the relative error tolerance specified in *tol* is satisfied or the maximum number of iterations specified in *maxiter* is reached. The relative error is defined as

$$\text{error} = \|Ax - b\|_2 / (\|b\|_2 + \epsilon)$$

where the $\|\cdot\|_2$ operator is the Euclidean norm and ϵ is a machine-dependent epsilon value to prevent any division by zero. If *tol* or *maxiter* is not specified in the call, then a default value of 10^{-7} is used for *tol* and 100,000 for *maxiter*.

The convergence of an iterative algorithm can often be enhanced by preconditioning the input coefficient matrix. The preconditioning option is specified with the *precon* parameter.

A starting trial solution can be specified with the *start* parameter; otherwise the ITSOLVER call generates a zero starting point. You can supply a matrix to store the relative error at each iteration with the *history* parameter. The *history* matrix should be dimensioned with enough elements to store the maximum number of iterations you expect.

You should always check the returned *error* and *iter* parameters to verify that the desired relative error tolerance is reached. If the tolerance is not reached, the program might continue the solution process with another ITSOLVER call, with *start* set to the latest result. You might also try a different *precon* option to enhance convergence.

For example, the following linear system has a coefficient matrix that contains several zeros:

$$\begin{bmatrix} 3 & 2 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & -10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

You can represent the matrix in sparse form and use the biconjugate gradient algorithm to solve the linear system, as shown in the following statements:

```
/* value     row column */
A = {  3       1      1,
       2       1      2,
       1.1     2      1,
       4       2      2,
       1       3      2,
       3.2     4      2,
     -10       3      3,
       3       4      4};

/* right hand side */
b = {1, 1, 1, 1};
maxiter = 10;
hist = j(maxiter,1,.);
start = {1,1,1,1};
tol = 1e-10;
call itsolver(x, error, iter, "bicg", A, b, "milu", tol,
maxiter, start, hist);
print x;
print iter error;
print hist;
```

Figure 23.147 Solution of a Linear System

```
                                  x

                          0.2040816
                          0.1938776
                          -0.080612
                          0.1265306

                          iter     error

                             3 5.011E-16
```

Figure 23.147 *continued*

```
                                  hist

                              0.0254375
                              0.0080432
                              5.011E-16
                                      .
                                      .
                                      .
                                      .
                                      .
                                      .
                                      .
```

The following linear system also has a coefficient matrix with several zeros:

$$\begin{bmatrix} 3 & 1.1 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & 10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The following statements represent the matrix in sparse form and use the conjugate gradient algorithm solve the symmetric positive definite system:

```
/* value     row column */
A = { 3        1      1,
      1.1      2      1,
      4        2      2,
      1        3      2,
      3.2      4      2,
     10        3      3,
      3        4      4};

/* right hand side */
b = {1, 1, 1, 1};
call itsolver(x, error, iter, "CG", A, b);
print x, iter, error;
```

Figure 23.148 Solution to Sparse System

```
                                    x

                                 2.68
                                 -6.4
                                 0.74
                                 7.16

                               iter

                                    4
```

Figure 23.148 *continued*

```
                                    error

                                 2.847E-15
```

J Function

J(*nrow* < , *ncol* > < , *value* > **);**

The J function creates a matrix with *nrow* rows and *ncol* columns with all elements equal to *value*.

The arguments to the J function are as follows:

nrow	is a numeric matrix or literal that contains the number of rows.
ncol	is a numeric matrix or literal that contains the number of columns.
value	is a numeric or character matrix or literal for filling the rows and columns of the matrix.

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, 4);
c = j(5, 2, "xyz");
print b, c;
```

Figure 23.149 Constant Matrices

```
                                    b

                          1         1         1         1
                          1         1         1         1
                          1         1         1         1

                                    c

                                xyz xyz
                                xyz xyz
                                xyz xyz
                                xyz xyz
                                xyz xyz
```

JROOT Function

JROOT(*order, n***);**

The 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. The function returns an $n \times 2$ matrix with the computed roots in the first column and the derivatives in the second column. You can evaluate the Bessel function itself by calling the JBESSEL function.

The arguments to the JROOT function are as follows:

order	is a scalar that denotes the order of the Bessel function, with *order*> -1. The order of a Bessel function is often indicated with the Greek subscript ν, so that J_ν indicates the Bessel function of order ν.
n	is a positive integer that denotes 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

$$J_\nu(x_i) = 0, i = 1, \ldots, n$$

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. See Abramowitz and Stegun (1972) for more details concerning the Bessel and gamma functions.

The root-finding 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 statements compute the first few roots for the Bessel function of the first kind:

```
x = jroot(1, 4);
print x;
```

Figure 23.150 Roots of a Bessel Function

x	
3.831706	-0.402759
7.0155867	0.3001158
10.173468	-0.249705
13.323692	0.2183594

To obtain only the roots, you can use the following statement, which extracts the first column of the returned matrix:

```
r = x[,1];
```

KALCVF Call

CALL KALCVF(*pred, vpred, filt, vfilt, data, lead, a, f, b, h, var* < , *z0* > < , *vz0* > **);**

The KALCVF subroutine computes the one-step prediction $z_{t+1|t}$ and the filtered estimate $z_{t|t}$, in addition to their covariance matrices. The call uses forward recursions, and you can also use it to obtain k-step estimates.

The input arguments to the KALCVF subroutine are as follows:

data is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.

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 $(T + \text{lead})N_z \times 1$ vector that contains 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 $(T + \text{lead})N_z \times N_z$ matrix that contains 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 $(T + \text{lead})N_y \times 1$ vector that contains 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 $(T + \text{lead})N_y \times N_z$ matrix that contains measurement matrices in the measurement equation.

var is an $(N_z + N_y) \times (N_z + N_y)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + \text{lead})(N_z + N_y) \times (N_z + N_y)$ matrix that contains 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 $z'_{1|0}$.

vz0 is an optional $N_z \times N_z$ covariance matrix of an initial state vector $P_{1|0}$.

The KALCVF call returns the following values:

pred is a $(T + \text{lead}) \times N_z$ matrix that contains one-step predicted state vectors $(z_{1|0}, \ldots, z_{T+1|T}, z_{T+2|T}, \ldots, z_{T+\text{lead}|T})'$.

vpred is a $(T + \text{lead})N_z \times N_z$ matrix that contains mean square errors of predicted state vectors $(P_{1|0}, \ldots, P_{T+1|T}, P_{T+2|T}, \ldots, P_{T+\text{lead}|T})'$.

filt is a $T \times N_z$ matrix that contains filtered state vectors $(z_{1|1}, \ldots, z_{T|T})'$.

vfilt is a $TN_z \times N_z$ matrix that contains mean square errors of filtered state vectors $(P_{1|1}, \ldots, P_{T|T})'$.

The KALCVF call computes the conditional expectation of the state vector 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 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, V_t and R_t are variances of η_t and ϵ_t, respectively, and G_t is a covariance of η_t and ϵ_t, and A^- stands for the generalized inverse of A. The filtered value and its covariance matrix are denoted $z_{t|t}$ and $P_{t|t}$, respectively. For $k > 0$, $z_{t+k|t}$ and $P_{t+k|t}$ stand for the k-step forecast of z_{t+k} and its mean square error. The Kalman filtering algorithm for one-step prediction and filtering is given as follows:

$$\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + R_t \\
z_{t|t} &= z_{t|t-1} + P_{t|t-1} H_t' D_t^- \hat{\epsilon}_t \\
P_{t|t} &= P_{t|t-1} - P_{t|t-1} H_t' D_t^- H_t P_{t|t-1} \\
K_t &= (F_t P_{t|t-1} H_t' + G_t) D_t^- \\
z_{t+1|t} &= a_t + F_t z_{t|t-1} + K_t \hat{\epsilon}_t \\
P_{t+1|t} &= F_t P_{t|t-1} F_t' + V_t - K_t D_t K_t'
\end{aligned}$$

And for k-step forecasting for $k > 1$,

$$\begin{aligned}
z_{t+k|t} &= a_{t+k-1} + F_{t+k-1} z_{t+k-1|t} \\
P_{t+k|t} &= F_{t+k-1} P_{t+k-1|t} F_{t+k-1}' + V_{t+k-1}
\end{aligned}$$

When you use the alternative transition equation

$$z_t = a_t + F_t z_{t-1} + \eta_t$$

the forward recursion algorithm is written

$$\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + H_t \mathbf{G}_t + G_t' H_t' + \mathbf{R}_t \\
z_{t|t} &= z_{t|t-1} + (P_{t|t-1} H_t' + G_t) D_t^- \hat{\epsilon}_t \\
P_{t|t} &= P_{t|t-1} - (P_{t|t-1} H_t' + G_t) D_t^- (H_t P_{t|t-1} + G_t') \\
K_t &= (F_{t+1} P_{t|t-1} H_t' + G_t) D_t^- \\
z_{t+1|t} &= a_{t+1} + F_{t+1} z_{t|t-1} + K_t \hat{\epsilon}_t \\
P_{t+1|t} &= F_{t+1} P_{t|t-1} F_{t+1}' + V_{t+1} - K_t D_t K_t'
\end{aligned}$$

And for k-step forecasting ($k > 1$),

$$\begin{aligned}
z_{t+k|t} &= a_{t+k} + F_{t+k} z_{t+k-1|t} \\
P_{t+k|t} &= F_{t+k} P_{t+k-1|t} F_{t+k}' + V_{t+k}
\end{aligned}$$

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

$$\begin{aligned} z_{1|0} &= (I - F)^{-}a \\ P_{1|0} &= (I - F \otimes F)^{-}\mathrm{vec}(V) \end{aligned}$$

where F and V are the time-invariant transition matrix and the covariance matrix of transition equation noise, and $\mathrm{vec}(V)$ is an $N_z^2 \times 1$ column vector that is constructed by the stacking N_z columns of matrix V. Note that all eigenvalues of the matrix F are inside the unit circle when the SSM is stationary. When the preceding formula cannot be applied, the initial state vector estimate $z_{1|0}$ is set to a_1 and its covariance matrix $P_{1|0}$ is given by 10^6I. Optionally, you can specify initial values.

The KALCVF call accepts 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.

The following program gives an example of the KALCVF call:

```
q = 2;
p = 2;
n = 10;
lead = 3;
total = n + lead;

seed = 25735;
x = round(10*normal(j(n, p, seed)))/10;
f = round(10*normal(j(q*total, q, seed)))/10;
a = round(10*normal(j(total*q, 1, seed)))/10;
h = round(10*normal(j(p*total, q, seed)))/10;
b = round(10*normal(j(p*total, 1, seed)))/10;

do i = 1 to total;
   temp = round(10*normal(j(p+q, p+q, seed)))/10;
   var = var//(temp*temp`);
end;

call kalcvf(pred, vpred, filt, vfilt, x, lead, a, f, b, h, var);

/* default initial state and covariance */
call kalcvs(sm, vsm, x, a, f, b, h, var, pred, vpred);
print sm[format=9.4] vsm[format=9.4];
```

Figure 23.151 Smoothed Estimate and Covariance

sm		vsm	
-1.5236	-0.1000	1.5813	-0.4779
0.3058	-0.1131	-0.4779	0.3963
-0.2593	0.2496	2.4629	0.2426
-0.5533	0.0332	0.2426	0.0944
-0.5813	0.1251	0.2023	-0.0228
-0.3017	0.7480	-0.0228	0.5799
1.1333	-0.2144	0.8615	-0.7653
1.5193	-0.6237	-0.7653	1.2334
-0.6641	-0.7770	1.0836	0.8706
0.5994	2.3333	0.8706	1.5252
		0.3677	0.2510
		0.2510	0.2051
		0.3243	-0.4093
		-0.4093	1.2287
		0.1736	-0.0712
		-0.0712	0.9048
		1.3153	0.8748
		0.8748	1.6575
		8.6650	0.1841
		0.1841	4.4770

KALCVS Call

CALL KALCVS(*sm, vsm, data, a, f, b, h, var, pred, vpred* **<** *, un* **>** **<** *, vun* **>** **);**

The KALCVS subroutine uses backward recursions to compute the smoothed estimate $z_{t|T}$ and its covariance matrix, $P_{t|T}$, where T is the number of observations in the complete data set.

The input arguments to the KALCVS subroutine are as follows.

data is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.

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 that contains 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 that contains 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 that contains 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 that contains T time-variant H_t matrices in the measurement equation.

var is an $(N_z + N_y) \times (N_z + N_y)$ covariance matrix for the errors in the transition and the measurement equations, or a $T(N_z + N_y) \times (N_z + N_y)$ matrix that contains covariance matrices in the transition equation and measurement equation noises—that is, $(\eta_t', \epsilon_t')'$.

pred is a $T \times N_z$ matrix that contains one-step forecasts $(z_{1|0}, \ldots, z_{T|T-1})'$.

vpred is a $TN_z \times N_z$ matrix that contains mean square error matrices of predicted state vectors $(P_{1|0}, \ldots, P_{T|T-1})'$.

un is an optional $1 \times N_z$ vector that contains u_T. The returned value is u_0.

vun is an optional $N_z \times N_z$ matrix that contains U_T. The returned value is U_0.

The KALCVS call returns the following values:

sm is a $T \times N_z$ matrix that contains smoothed state vectors $(z_{1|T}, \ldots, z_{T|T})'$.

vsm is a $TN_z \times N_z$ matrix that contains covariance matrices of smoothed state vectors $(P_{1|T}, \ldots, P_{T|T})'$.

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, $z_{t|T}$ is the smoothed value of the state vector z_t, and the mean square error matrix is denoted $P_{t|T}$. For smoothing,

$$\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + R_t \\
K_t &= (F_t P_{t|t-1} H_t' + G_t) D_t^- \\
L_t &= F_t - K_t H_t \\
u_{t-1} &= H_t' D_t^- \hat{\epsilon}_t + L_t' u_t \\
U_{t-1} &= H_t' D_t^- H_t + L_t' U_t L_t \\
z_{t|T} &= z_{t|t-1} + P_{t|t-1} u_{t-1} \\
P_{t|T} &= P_{t|t-1} - P_{t|t-1} U_{t-1} P_{t|t-1}
\end{aligned}$$

where $t = T, T-1, \ldots, 1$. The initial values are $u_T = \mathbf{0}$ and $\mathbf{U}_T = \mathbf{0}$.

When the SSM is specified by using the alternative transition equation

$$z_t = a_t + F_t z_{t-1} + \eta_t$$

the fixed-interval smoothing is performed by using the following backward recursions:

$$\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + R_t \\
K_t &= F_{t+1} P_{t|t-1} H_t' D_t^- \\
L_t &= F_{t+1} - K_t H_t \\
u_{t-1} &= H_t' D_t^- \hat{\epsilon}_t + L_t' u_t \\
U_{t-1} &= H_t' D_t^- H_t + L_t' U_t L_t \\
z_{t|T} &= z_{t|t-1} + P_{t|t-1} u_{t-1} \\
P_{t|T} &= P_{t|t-1} - P_{t|t-1} U_{t-1} P_{t|t-1}
\end{aligned}$$

where it is assumed that $G_t = \mathbf{0}$.

You can use the KALCVS call regardless of the specification of the transition equation when $G_t = \mathbf{0}$. Harvey (1989) gives the following fixed-interval smoothing formula, which produces the same smoothed value:

$$\begin{aligned}
z_{t|T} &= z_{t|t} + P_t^*(z_{t+1|T} - z_{t+1|t}) \\
P_{t|T} &= P_{t|t} + P_t^*(P_{t+1|T} - P_{t+1|t}) P_t^{*\prime}
\end{aligned}$$

where

$$P_t^* = P_{t|t} F_t' P_{t+1|t}^-$$

under the shifted transition equation, but

$$P_t^* = P_{t|t} F_{t+1}' P_{t+1|t}^-$$

under the alternative transition equation.

The KALCVS call is accompanied by the KALCVF call, as shown in the following statements. 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, as shown in the following statements:

```
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;
```

The KALCVF call has an example program that includes the KALCVS call.

KALDFF Call

CALL KALDFF(*pred, vpred, initial, s2, data, lead, int, coef, var, intd, coefd* < , *n0*> < , *at*> < , *mt*> < , *qt*> **);**

The KALDFF subroutine computes the one-step forecast of state vectors in an SSM by using the diffuse Kalman filter. The call estimates the conditional expectation of z_t, and also estimates the initial random vector, δ, and its covariance matrix.

The input arguments to the KALDFF subroutine are as follows:

data is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.

lead is the number of steps to forecast after the end of the data set.

int is an $(N_z + N_y) \times N_\beta$ matrix for a time-invariant fixed matrix, or a $(T + \text{lead})(N_z + N_y) \times N_\beta$ matrix that contains fixed matrices for the time-variant model in the transition equation and the measurement equation—that is, $(W_t', X_t')'$.

coef is an $(N_z + N_y) \times N_z$ matrix for a time-invariant coefficient, or a $(T + \text{lead})(N_z + N_y) \times N_z$ matrix that contains coefficients at each time in the transition equation and the measurement equation—that is, $(F_t', H_t')'$.

var is an $(N_z + N_y) \times (N_z + N_y)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + \text{lead})(N_z + N_y) \times (N_z + N_y)$ matrix that contains 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 that contains the intercept term in the equation for the initial state vector z_0 and the mean effect β—that is, $(a', b')'$.

coefd is an $(N_z + N_\beta) \times N_\delta$ matrix that contains coefficients for the initial state δ in the equation for the initial state vector z_0 and the mean effect β—that is, $(A', B')'$.

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 $(y_1, \ldots, y_t)'$. If *n0*≤ 0 or *n0* is not specified, the denominator for $\hat{\sigma}_t^2$ is n_t. With *n0*≥ 0, the initial values, A_1, M_1, and Q_1, are assumed to be known and, hence, *at*, *mt*, and *qt* are used for input that contains 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*≥ 0, *at* contains $(A_1', \ldots, A_k')'$. However, only the first matrix $\mathbf{A}_1$ is used as input. When you specify the KALDFF call, *at* returns $(A'_{T-k+\text{lead}+1}, \ldots, A'_{T+\text{lead}})'$. 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*≥ 0, *mt* contains $(M_1, \ldots, M_k)'$. However, only the first matrix M_1 is used as input. If *n0* is negative or the matrix M_1 contains missing values, *mt* is used for output, and it contains $(M_{T-k+\text{lead}+1}, \ldots, M_{T+\text{lead}})'$. Note that the matrix M_1 can be used as an input matrix if either of the off-diagonal elements is not missing. The missing element $M_1(i, j)$ is replaced by the nonmissing element $M_1(j, i)$.

qt is an optional $k(N_\delta + 1) \times (N_\delta + 1)$ matrix. If *n0*≥ 0, *qt* contains $(Q_1, \ldots, Q_k)'$. However, only the first matrix Q_1 is used as input. If *n0* is negative or the matrix Q_1 contains missing values, *qt* is used for output and contains $(Q_{T-k+\text{lead}+1}, \ldots, Q_{T+\text{lead}})'$. The matrix Q_1 can also be used as an input matrix if either of the off-diagonal elements is not missing since the missing element $Q_1(i, j)$ is replaced by the nonmissing element $Q_1(j, i)$.

The KALDFF call returns the following values:

pred is a $(T + \text{lead}) \times N_z$ matrix that contains estimated predicted state vectors $(\hat{z}_{1|0}, \ldots, \hat{z}_{T+1|T}, \hat{z}_{T+2|T}, \ldots, \hat{z}_{T+\text{lead}|T})'$.

vpred is a $(T + \text{lead})N_z \times N_z$ matrix that contains estimated mean square errors of predicted state vectors $(P_{1|0}, \ldots, P_{T+1|T}, P_{T+2|T}, \ldots, P_{T+\text{lead}|T})'$.

initial is an $N_d \times (N_d + 1)$ matrix that contains an estimate and its variance for initial state δ, that is, $(\hat{\delta}_T, \hat{\Sigma}_{\delta,T})$.

s2 is a scalar that contains the estimated variance $\hat{\sigma}_T^2$.

The KALDFF call computes the one-step forecast of state vectors in an SSM by using the diffuse Kalman filter. The SSM for the diffuse Kalman filter is written

$$\begin{aligned} y_t &= X_t\beta + H_t z_t + \epsilon_t \\ z_{t+1} &= W_t\beta + F_t z_t + \eta_t \\ z_0 &= a + A\delta \\ \beta &= b + B\delta \end{aligned}$$

where z_t is an $N_z \times 1$ state vector, y_t is an $N_y \times 1$ observed vector, and

$$\begin{aligned} \begin{bmatrix} \eta_t \\ \epsilon_t \end{bmatrix} &\sim N\left(\mathbf{0}, \sigma^2 \begin{bmatrix} V_t & G_t \\ G_t' & R_t \end{bmatrix}\right) \\ \delta &\sim N(\mu, \sigma^2\Sigma) \end{aligned}$$

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, W_t, F_t, X_t, H_t, a, A, b, B, V_t, G_t, and R_t, are assumed to be known. The KALDFF call

estimates the conditional expectation of the state vector 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 $z_{t+k|t}$ and $P_{t+k|t}$ (for $k > 0$). $A_{t+k(\delta)}$ and $E_{t(\delta)}$ are last-column-deleted submatrices of A_{t+k} and E_t, respectively. The algorithm for one-step prediction is given as follows:

$$
\begin{aligned}
E_t &= (X_t B,\ y_t - X_t b) - H_t A_t \\
D_t &= H_t M_t H_t' + R_t \\
Q_{t+1} &= Q_t + E_t' D_t^- E_t \\
&= \begin{bmatrix} S_t & s_t \\ s_t' & q_t \end{bmatrix} \\
\hat{\sigma}_t^2 &= (q_t - s_t' S_t^- s_t)/n_t \\
\hat{\delta}_t &= S_t^- s_t \\
\hat{\Sigma}_{\delta,t} &= \hat{\sigma}_t^2 S_t^- \\
K_t &= (F_t M_t H_t' + G_t) D_t^- \\
A_{t+1} &= W_t(-B, b) + F_t A_t + K_t E_t \\
M_{t+1} &= (F_t - K_t H_t) M_t F_t' + V_t - K_t G_t' \\
z_{t+1|t} &= A_{t+1}(-\hat{\delta}_t', 1)' \\
P_{t+1|t} &= \hat{\sigma}_t^2 \mathbf{M}_{t+1} + A_{t+1(\delta)} \hat{\Sigma}_{\delta,t} A_{t+1(\delta)}'
\end{aligned}
$$

where n_t is the number of elements of $(y_1, \ldots, y_t)'$ plus $\max(n0, 0)$. Unless initial values are given and $n0 \geq 0$, initial values are set as follows:

$$
\begin{aligned}
A_1 &= W_1(-B, b) + F_1(-A, a) \\
M_1 &= V_1 \\
Q_1 &= \mathbf{0}
\end{aligned}
$$

For k-step forecasting where $k > 1$,

$$
\begin{aligned}
A_{t+k} &= W_{t+k-1}(-B, b) + F_{t+k-1} A_{t+k-1} \\
M_{t+k} &= F_{t+k-1} M_{t+k-1} F_{t+k-1}' + V_{t+k-1} \\
D_{t+k} &= H_{t+k} M_{t+k} H_{t+k}' + R_{t+k} \\
z_{t+k|t} &= A_{t+k}(-\hat{\delta}_t', 1)' \\
P_{t+k|t} &= \hat{\sigma}_t^2 M_{t+k} + A_{t+k(\delta)} \hat{\Sigma}_{\delta,t} A_{t+k(\delta)}'
\end{aligned}
$$

If there is a missing observation, the KALDFF call computes the one-step forecast for the observation that follows the missing observation as the two-step forecast from the previous observation.

An example that uses the KALDFF call is in the documentation for the KALDFS call.

KALDFS Call

CALL KALDFS(*sm, vsm, data, int, coef, var, bvec, bmat, initial, at, mt, s2* **<** *, un, vun* **>);**

The KALDFS subroutine 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 input arguments to the KALDFS subroutine are as follows:

data is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.

int is an $(N_z + N_y) \times N_\beta$ vector for a time-invariant intercept, or a $(T + \text{lead})(N_z + N_y) \times N_\beta$ vector that contains fixed matrices for the time-variant model in the transition equation and the measurement equation—that is, $(W_t', X_t')'$.

coef is an $(N_z + N_y) \times N_z$ matrix for a time-invariant coefficient, or a $(T + \text{lead})(N_z + N_y) \times N_z$ matrix that contains coefficients at each time in the transition equation and the measurement equation—that is, $(F_t', H_t')'$.

var is an $(N_z + N_y) \times (N_z + N_y)$ matrix for a time-invariant variance matrix for transition equation noise and the measurement equation noise, or a $(T + \text{lead})(N_z + N_y) \times (N_z + N_y)$ matrix that contains 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 that contains 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 that contains $(A_1', \ldots, A_T')'$.

mt is a $(TN_z) \times N_z$ matrix that contains $(M_1, \ldots, M_T)'$.

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 that contains u_T. The returned value is u_0.

vun is an optional $N_z \times N_z$ matrix that contains U_T. The returned value is U_0.

The KALDFS call returns the following values:

sm is a $T \times N_z$ matrix that contains smoothed state vectors $(z_{1|T}, \ldots, z_{T|T})'$.

vsm is a $TN_z \times N_z$ matrix that contains mean square error matrices of smoothed state vectors $(P_{1|T}, \ldots, P_{T|T})'$.

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 z_t given all

observations. Using the notations and results from the KALDFF section, the backward recursion algorithm for smoothing is denoted for $t = T, T-1, \ldots, 1$,

$$
\begin{aligned}
E_t &= (X_t B,\ y_t - X_t b) - H_t A_t \\
D_t &= H_t M_t H_t' + R_t \\
L_t &= F_t - (F_t M_t H_t' + G_t) D_t^- H_t \\
u_{t-1} &= H_t' D_t^- E_t + L_t' u_t \\
U_{t-1} &= H_t' D_t^- H_t + L_t' U_t L_t \\
z_{t|T} &= (A_t + M_t u_{t-1})(-\hat{\delta}_T', 1)' \\
C_t &= A_t + M_t u_{t-1} \\
P_{t|T} &= \hat{\sigma}_T^2 (M_t - M_t R_{t-1} M_t) + C_{t(\delta)} \hat{\Sigma}_{\delta,T} C_{t(\delta)}'
\end{aligned}
$$

where the initial values are $u_T = b0$ and $U_T = \mathbf{0}$, and $C_{t(\delta)}$ is the last-column-deleted submatrix of C_t. See De Jong, P. (1991) for details about smoothing in the diffuse Kalman filter.

The KALDFS call is accompanied by the KALDFF call as shown in the following statements:

```
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 statements perform smoothing. You should initialize UN and VUN as matrices in which all elements are zero.

```
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;
```

LAG Function

LAG(*x* < , *lags* > **);**

The LAG function computes one or more lagged (shifted) values for time series data. The arguments are as follows:

x	specifies an $n \times 1$ numerical matrix of time series data.
lags	specifies integer lags. The *lags* argument can be an integer matrix with d elements. If so, the LAG function returns an $n \times d$ matrix where the ith column represents the ith lag applied to the time series.

The values of the *lags* argument are usually positive integers. A positive lag shifts the time series data backwards in time. A lag of 0 represents the original time series. A negative value for the *lags* argument shifts the time series data forward in time; this is sometimes called a *lead effect*.

For example, the following statements compute several lags:

```
x = {1,3,4,7,9};
lag = lag(x, {0 1 3});
print lag;
```

Figure 23.152 Lagged Data

lag		
1	.	.
3	1	.
4	3	.
7	4	1
9	7	3

LAV Call

CALL LAV(*rc, xr, a, b* < , *x0* > < , *opt* > **);**

The LAV subroutine performs linear least absolute value regression by solving the L_1 norm minimization problem.

The LAV subroutine returns the following values:

rc is a scalar return code that indicates 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 A is rank-deficient (rank$(A) < n$)
-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 input arguments to the LAV subroutine are as follows:

a specifies an $m \times n$ matrix A with $m \geq n$ and full column rank, rank$(A) = n$. If you want to include an intercept in the model, you must include a column of ones in the matrix A.

b specifies the $m \times 1$ vector 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. If an element of the *opt* vector is missing, the default value is used.

- *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 always printed, and 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 A are printed.

- *opt[3]* specifies which estimate of the variance of the median of nonzero residuals 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 behavior is 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 behavior (*opt[4]*$= 0$) is 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 LAV subroutine is designed for solving the unconstrained linear L_1 norm minimization problem,

$$\min_x L_1(x) \text{ where } L_1(x) = \|Ax - b\|_1 = \sum_{i=1}^{m} \left| \sum_{j=1}^{n} a_{ij} x_j - b_i \right|$$

for m equations with n (unknown) parameters $x = (x_1, \ldots, x_n)$. This is equivalent to estimating the unknown parameter vector, x, by least absolute value regression in the model

$$b = Ax + \epsilon$$

where b is the vector of n observations, A is the design matrix, and ϵ 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 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*. By using PROC NLP and the 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 section "NLPNMS Call" on page 837 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 847 for details about 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 can provide fairly good initial estimates.

Gonin and Money (1989) show that the nonlinear L_1 norm estimation problem

$$\min_x \sum_{i=1}^{m} |f_i(x)|$$

can be reformulated as a linear optimization problem with nonlinear constraints in the following ways.

- as a linear optimization problem with $2m$ nonlinear inequality constraints in $m + n$ variables u_i and x_j,

$$\min_x \sum_{i=1}^{m} u_i \text{ subject to } \left.\begin{array}{lcl} f_i(x) - u_i & \leq & 0 \\ f_i(x) + u_i & \geq & 0 \\ u_i & \geq & 0 \end{array}\right\} \quad i = 1, \ldots, m$$

- as a linear optimization problem with $2m$ nonlinear equality constraints in $2m + n$ variables y_i, z_i, and x_j,

$$\min_x \sum_{i=1}^{m} (y_i + z_i) \text{ subject to } \left.\begin{array}{lcl} f_i(x) + y_i - z_i & = & 0 \\ y_i & \geq & 0 \\ z_i & \geq & 0 \end{array}\right\} \quad i = 1, \ldots, m$$

For linear functions $f_i(x) = \sum_{j=1}^{n} (a_{ij} x_j - b_i)$, $i = 1, \ldots, m$, 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 x^* that minimizes $L_1(x)$, you can also compute approximate standard errors and the approximate covariance matrix of x^*. The standard errors can be used to compute confidence limits.

The following example is the same one used for illustrating the LAV subroutine by Lee and Gentle (1986). A and b are as follows:

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & 2 \\ 1 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \\ 1 \\ -1 \\ 2 \\ 4 \end{bmatrix}$$

The following statements specify 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 be performed (*opt[4]*= 1).

```
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 output is shown in Figure 23.153. This output displays the least squares solution, which is used as the starting point. The estimates of the largest and smallest nonzero eigenvalues of $A'A$ give only an idea of the magnitude of these values, and they can be very crude approximations.

Figure 23.153 Least Squares Solution

```
                          LS Solution

                Est                1                1
```

The second part of the printed output shows the iteration history. It is shown in Figure 23.154.

Figure 23.154 Iteration History

```
                        LAV (L1) Estimation
                      Start with LS Solution
                  Start Iter: gamma=1 ActEqn=6
 Iter    N Huber    Act Eqn       Rank      Gamma        L1(x)     F(Gamma)

    1          1          2          2     0.9000     4.000000     2.200000
    1          1          2          2     0.0000     4.000000     2.200000
```

The third part of the output is shown in Figure 23.155. This output displays 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.

Figure 23.155 Parameter and Covariance Estimates

L1 Solution with ASE

Est	1	1
ASE	0.4482711811	0.3310702082

Cov Matrix: McKean-Schrader

0.2009470518	-0.054803741
-0.054803741	0.1096074828

The last part of the printed output shows the predicted values and residuals, as in Lee and Gentle (1986). It is shown in Figure 23.156.

Figure 23.156 Predicted and Residual Values

Predicted Values and Residuals

N	Observed	Predicted	Residual
1	1.0000	1.0000	0
2	2.0000	2.0000	0
3	1.0000	0.0000	1.000000
4	-1.0000	0.0000	-1.000000
5	2.0000	3.0000	-1.000000
6	4.0000	3.0000	1.000000

LCP Call

CALL LCP(*rc, w, z, m, q* < , *epsilon* > **);**

The LCP subroutine solves the linear complementarity problem:

$$\begin{aligned} \mathbf{w} &= \mathbf{Mz} + \mathbf{q} \\ \mathbf{w}'\mathbf{z} &= 0 \\ \mathbf{w}, \mathbf{z} &\geq 0 \end{aligned}$$

That is, given a matrix $\mathbf{M}$ and a vector $\mathbf{q}$, the LCP subroutine computes orthogonal, nonnegative vectors $\mathbf{w}$ and $\mathbf{z}$ which satisfy the previous equations.

The input arguments 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 that defines virtual zero. The default value of *epsilon* is 1E−8.

The LCP subroutine returns the following matrices:

rc returns one of the following scalar return codes:

rc	**Termination**
0	A solution is found.
1	No solution is possible.
5	The solution is numerically unstable.
6	The subroutine could not obtain enough memory.

w returns an m-element column vector

z returns an m-element column vector

The following statements give a simple example:

```
q = {1, 1};
m = {1 0,
     0 1};
call lcp(rc, w, z, m, q);
print rc, w, z;
```

Figure 23.157 Solution to a Linear Complementarity Problem

```
rc

    0

w

    1
    1

z

    0
    0
```

The next example shows the relationship between quadratic programming and the linear complementarity problem. Consider the linearly constrained quadratic program:

$$\begin{aligned} \min \mathbf{c}'\mathbf{x} &+ \frac{1}{2}\mathbf{x}'\mathbf{H}\mathbf{x} \\ \text{st. } \mathbf{G}\mathbf{x} &\geq \mathbf{b} \quad \text{(QP)} \\ \mathbf{x} &\geq 0 \end{aligned}$$

If $\mathbf{H}$ is positive semidefinite, then a solution to the Kuhn-Tucker conditions solves QP. The Kuhn-Tucker

conditions for QP are

$$\begin{aligned} \mathbf{c} + \mathbf{Hx} &= \mu + \mathbf{G}'\lambda \\ \lambda'(\mathbf{Gx} - \mathbf{b}) &= 0 \\ \mu'\mathbf{x} &= 0 \\ \mathbf{Gx} &\geq \mathbf{b} \\ x, \mu, \lambda &\geq 0 \end{aligned}$$

In the linear complementarity problem, let

$$\begin{aligned} \mathbf{M} &= \begin{bmatrix} H & -G' \\ G & 0 \end{bmatrix} \\ \mathbf{w}' &= (\mu'\mathbf{s}') \\ \mathbf{z}' &= (\mathbf{x}'\lambda') \\ \mathbf{q}' &= (\mathbf{c}' - \mathbf{b}) \end{aligned}$$

Then the Kuhn-Tucker conditions are expressed as finding **w** and **z** that satisfy

$$\begin{aligned} \mathbf{w} &= \mathbf{Mz} + \mathbf{q} \\ \mathbf{w}'\mathbf{z} &= 0 \\ \mathbf{w}, \mathbf{z} &\geq 0 \end{aligned}$$

From the solution **w** and **z** to this linear complementarity problem, the solution to QP is obtained; namely, **x** is the primal structural variable, $\mathbf{s} = \mathbf{Gx} - \mathbf{b}$ the surpluses, and $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$ 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 by using the LCP subroutine 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`;
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;
```

Figure 23.158 Solution to a Quadratic Programming Problem

```
                                    rc         x

                                     0 0.0307522
                                       0.0619692
                                       0.0929721
                                       0.1415983
```

LENGTH Function

LENGTH(*matrix***);**

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 follows:

```
c = {"Hello" "My name is Jenny"};
b = length(c);
print b;
```

Figure 23.159 Length of Elements of a Character Matrix

```
                                     b

                                     5        16
```

See also the description of the NLENG function.

LINK Statement

LINK(*label***);**

statements **;**

label:statements **;**

RETURN ;

The LINK statement provides a way of calling a group of statements as if they were defined as a subroutine. When the LINK statement is executed, the program jumps immediately to the statement with the given *label* and begins executing statements from that point as it does for the GOTO statement. However, when the program executes a RETURN statement, the program returns to the statement that immediately follows the LINK statement, which is different behavior than the GOTO statement.

The LINK statement can be used only inside modules and DO groups. 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.

Instead of using a LINK statement, you can define a module and call the module by using a RUN statement.

An example that uses the LINK statement follows:

```
start a;
   x=1;
   y=2;
   link sum1; /* go to label; execute until return stmt */
   print z;
   stop;
   sum1:
      z=x+y;
   return;
finish a;

run a;
```

Figure 23.160 Result of Linking to a Group of Statements

z
3

LIST Statement

LIST <*range*> <**VAR** *operand*> <**WHERE***(expression)*> **;**

The LIST statement displays observations of a data set.

The arguments to the LIST statement are as follows:

range	specifies a range of observations.
operand	specifies a set of variables.
expression	is an expression that selects certain observations.

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*.

Options for Specifying the Observations

You can use any of the following keywords for *range*:

ALL	specifies all observations.
CURRENT	specifies the current observation (this is the default for the LIST statement).
NEXT < *number* >	specifies the next observation or the next *number* of observations.
AFTER	specifies all observations after the current one.
POINT *value*	specifies observations specified by number, where *value* can be one of the following:

Value	Example
A single record number	`point 5`
A literal that contains several record numbers	`point {2 5 10}`
The name of a matrix that contains record numbers	`point p`
An expression in parentheses	`point (p+1)`

If the current data set has an index in use (see the INDEX statement), the POINT option is invalid.

Options for Specifying the Variables

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 that contains variable names
- the name of a matrix that contains variable names
- an expression in parentheses that yields variable names
- one of the keywords described in the following list:

  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 {x1 x5 x9};          /* a literal matrix of names         */
var x;                   /* a matrix that contains the names  */
var("x1":"x9");          /* an expression                     */
var _all_;               /* a keyword                         */
```

Options for Filtering the Observations

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)* **;**

The arguments to the WHERE clause are as follows:

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 like 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 by using the AND (&) and OR (|) operators. The general form is as follows:

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 by 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 following examples demonstrate the use of the LIST statement. The output is not shown.

```
data class;
   set sashelp.class;
run;

proc iml;
use class;

list all;                                /* lists whole data set */
list;                                 /* lists current observation */
list var{name age};          /* lists NAME and AGE in current obs */
list all where(age<=13); /* lists all obs where condition holds */
list next;                              /* lists next observation */
list point 18;                            /* lists observation 18 */
list point (10:15);           /* lists observations 10 through 15 */

close class;
```

LMS Call

CALL LMS(*sc, coef, wgt, opt, y* < , *x* > < , *sorb* > **);**

The LMS subroutine performs least median of squares (LMS) robust regression (sometimes called *resistant* regression) by minimizing the hth-ordered squared residual. The subroutine is able to detect outliers and perform a least squares regression on the remaining observations.

The algorithm used in the LMS subroutine is based on the PROGRESS program of Rousseeuw and Hubert (1996), which is an updated version of Rousseeuw and Leroy (1987). In the special case of regression through the origin with a single regressor, Barreto and Maharry (2006) show that the PROGRESS algorithm does not, in general, find the slope that yields the least median of squares. Starting with SAS/IML 9.2, the LMS subroutine uses the algorithm of Barreto and Maharry (2006) to obtain the correct LMS slope in the case of regression through the origin with a single regressor. In this case, input arguments that are specific to the PROGRESS algorithm are ignored and output specific to the PROGRESS algorithm is suppressed.

The value of h can be specified, but in most applications the default value works well and the results seem to be quite stable toward different choices of h.

In the following discussion, N is the number of observations and n is the number of regressors. The input arguments to the LMS subroutine 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 ones is added as the last column to the input matrix $\mathbf{X}$; that is, you do not need to add this column of ones 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.

- 0 prints no output except error messages.
- 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.
- 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.
- 3 additionally prints subsets that result in singular linear systems.

The default is *opt[2]*=0.

opt[3] specifies whether only LMS is computed or whether, additionally, least squares (LS) and weighted least squares (WLS) regression are computed.

- 0 computes only LMS.
- 1 computes, in addition to LMS, weighted least squares regression on the observations with *small* LMS residuals (where *small* is defined by opt[8]).
- 2 computes, in addition to LMS, unweighted least squares regression.
- 3 adds both unweighted and weighted least squares regression to LMS 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[4]*= $h = \left[\frac{N+n+1}{2}\right]$, 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 that contain 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}	10^6	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 chosen randomly. This means that an exhaustive search is performed for *opt[5]*=−1. If N is larger than N_{upper}, a note is printed in the log file that indicates how many subsets exist.

opt[6] is not used.

opt[7] specifies whether the last argument *sorb* contains a given parameter vector **b** or a given subset for which the objective function should be evaluated.

0 *sorb* contains a given subset index.

1 *sorb* contains a given parameter vector **b**.

The default is *opt[7]*=0.

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.

0 does not compute covariance matrix and ASEs.

1 computes covariance matrix and ASEs but prints neither of them.

2 computes the covariance matrix and ASEs but prints only the ASEs.

3 computes and prints both the covariance matrix and the ASEs.

The default is *opt[8]*=0.

y refers to an N response vector.

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 that contains 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[7]* 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 subroutine returns the following values:

sc is a column vector that contains the following scalar information, where rows 1–9 correspond to LMS 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} attained

sc[6] preliminary LMS scale estimate S_P

sc[7] final LMS scale estimate S_F

sc[8] robust R square (*coefficient of determination*)

sc[9] asymptotic consistency factor

If *opt[3]* > 0, then the following are also set:

sc[11] LS or WLS objective function (sum of squared residuals)

sc[12] LS or WLS scale estimate

sc[13] R square 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 that contains the following results in its rows:

coef[1,] LMS parameter estimates

coef[2,] indices of observations in the best subset

If *opt[3]* > 0, then the following are also 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, these rows correspond to LS estimates.

wgt is a matrix with N columns that contains the following results in its rows:

wgt[1,] weights (1 for small residuals; 0 for large residuals)

wgt[2,] residuals $r_i = y_i - \mathbf{x}_i\mathbf{b}$

wgt[3,] resistant diagnostic u_i (the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider results for Brownlee (1965) stackloss data. The three explanatory variables correspond to measurements for a plant that oxidizes 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). The data are also given by Rousseeuw and Leroy (1987) and Osborne (1985). Rousseeuw and Leroy (1987) cite a large number of papers where this data set was analyzed and state that most researchers "concluded that observations 1, 3, 4, and 21 were outliers," and that some people also reported observation 2 as an outlier.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5,985 different subsets of 4 observations out of 21. If you decide not to specify `opt[5]`, the LMS subroutine chooses N_{rep} =2,000 random sample subsets. Since there is a large number of subsets with singular linear systems, which you do not want to print, choose `opt[2]=2` for reduced printed output.

```
   /* X1  X2  X3   Y  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 };

a = aa[, 2:4]; b = aa[, 5];
opt = j(8, 1, .);
opt[2]= 2;    /* ipri */
opt[3]= 3;    /* ilsq */
opt[8]= 3;    /* icov */

call lms(sc, coef, wgt, opt, b, a);
```

The first portion of the output displays descriptive statistics, as shown in Figure 23.161:

Figure 23.161 Descriptive Statistics

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

There are 5985 subsets of 4 cases out of 21 cases.

The algorithm will draw 2000 random subsets of 4 cases.

Random Subsampling for LMS

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 is/are singular.

LMS Objective Function = 0.75

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 next portion of the output shows the least squares estimates and the covariance of the estimates. Information about the residuals are also displayed, but are not shown in Figure 23.162.

Figure 23.162 Least Squares Estimates

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

Unweighted Least-Squares Estimation

LS Parameter Estimates

Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.7156402	0.13485819	5.31	<.0001	0.45132301	0.97995739
VAR2	1.29528612	0.36802427	3.52	0.0026	0.57397182	2.01660043
VAR3	-0.1521225	0.15629404	-0.97	0.3440	-0.4584532	0.15420818
Intercep	-39.919674	11.8959969	-3.36	0.0038	-63.2354	-16.603949

Preliminary LMS Scale = 1.0478510755

Robust R Squared = 0.96484375

Final LMS Scale = 1.2076147288

Distribution of Residuals
Median(U)= 4.5864208797

Weighted Least-Squares Estimation

Weighted Sum of Squares = 20.400800254
Degrees of Freedom = 13
RLS Scale Estimate = 1.2527139846

Weighted R-squared = 0.9750062263

Figure 23.162 *continued*

```
                          Sum of Squares = 178.8299616
                            Degrees of Freedom = 17
                         LS Scale Estimate = 3.2433639182

                         Cov Matrix of Parameter Estimates

                         VAR1              VAR2              VAR3           Intercep

VAR1           0.0181867302      -0.036510675      -0.007143521      0.2875871057
VAR2           -0.036510675      0.1354418598      0.0000104768      -0.651794369
VAR3           -0.007143521      0.0000104768       0.024427828      -1.676320797
Intercep       0.2875871057      -0.651794369      -1.676320797      141.51474107
                        F(3,13) Statistic = 169.04317954
                           Probability = 1.158521E-10
                      There are 17 points with nonzero weight.
                           Average Weight = 0.8095238095

                             Distribution of Residuals

The run has been executed successfully.


                             R-squared = 0.9135769045
                          F(3,17) Statistic = 59.9022259
                            Probability = 3.0163272E-9
```

The LMS subroutine prints results for the 2,000 random subsets. Figure 23.163 shows the iteration history, the best subset of observations that are used to form estimates, and the estimated parameters. The subroutine also displays residual information (not shown).

Figure 23.163 Least Median Squares Estimates

```
                    MinRes            1st Qu.            Median

                -7.237712859      -1.814569436      -0.455092955

                      Mean            3rd Qu.            MaxRes

                3.383537E-16      1.8867828182      5.6977741706
```

Figure 23.163 *continued*

```
There are 5985 subsets of 4 cases out of 21 cases.

The algorithm will draw 2000 random subsets of 4 cases.

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 is/are singular.

        Observations of Best Subset

  15              11              19              10

          Estimated Coefficients

 VAR1            VAR2            VAR3         Intercep

 0.75             0.5               0           -39.25
```

Observations 1, 3, 4, and 21 have scaled residuals larger than 2.0 (table not shown) and are considered outliers. The corresponding WLS estimates are shown in Figure 23.164:

Figure 23.164 Weighted Least Squares Estimates

```
          Estimated Coefficients

 VAR1            VAR2            VAR3         Intercep

 0.75             0.5               0           -39.25
```

Figure 23.164 *continued*

LMS Objective Function = 0.75

Preliminary LMS Scale = 1.0478510755

Robust R Squared = 0.96484375

Final LMS Scale = 1.2076147288

LMS Residuals

N	Observed	Estimated	Residual	Res / S
1	42.000000	34.250000	7.750000	6.417610
2	37.000000	34.250000	2.750000	2.277216
3	37.000000	29.500000	7.500000	6.210590
4	28.000000	19.250000	8.750000	7.245688
5	18.000000	18.250000	−0.250000	−0.207020
6	18.000000	18.750000	−0.750000	−0.621059
7	19.000000	19.250000	−0.250000	−0.207020
8	20.000000	19.250000	0.750000	0.621059
9	15.000000	15.750000	−0.750000	−0.621059
10	14.000000	13.250000	0.750000	0.621059
11	14.000000	13.250000	0.750000	0.621059
12	13.000000	12.750000	0.250000	0.207020
13	11.000000	13.250000	−2.250000	−1.863177
14	12.000000	13.750000	−1.750000	−1.449138
15	8.000000	7.250000	0.750000	0.621059
16	7.000000	7.250000	−0.250000	−0.207020
17	8.000000	7.750000	0.250000	0.207020
18	8.000000	7.750000	0.250000	0.207020
19	9.000000	8.250000	0.750000	0.621059
20	15.000000	12.750000	2.250000	1.863177
21	15.000000	23.250000	−8.250000	−6.831649

Distribution of Residuals

MinRes	1st Qu.	Median
−8.25	−0.5	0.25
Mean	**3rd Qu.**	**MaxRes**
0.9047619048	0.75	8.75

Figure 23.164 *continued*

Resistant Diagnostic

N	U	Resistant Diagnostic
1	10.448052	2.278040
2	7.931751	1.729399
3	10.000000	2.180349
4	11.666667	2.543741
5	2.729730	0.595176
6	3.486486	0.760176
7	4.729730	1.031246
8	4.243243	0.925175
9	3.648649	0.795533
10	3.759835	0.819775
11	4.605767	1.004218
12	4.925169	1.073859
13	3.888889	0.847914
14	4.586421	1.000000
15	5.297030	1.154938
16	4.009901	0.874299
17	6.679576	1.456381
18	4.305340	0.938715
19	4.019976	0.876495
20	3.000000	0.654105
21	11.000000	2.398384

Median(U)= 4.5864208797

Weighted Least-Squares Estimation

RLS Parameter Estimates Based on LMS

Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.79768556	0.06743906	11.83	<.0001	0.66550742	0.9298637
VAR2	0.57734046	0.16596894	3.48	0.0041	0.25204731	0.9026336
VAR3	-0.0670602	0.06160314	-1.09	0.2961	-0.1878001	0.05367975
Intercep	-37.652459	4.73205086	-7.96	<.0001	-46.927108	-28.37781

Figure 23.164 *continued*

```
                    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          0.0045480273      -0.007921409      -0.001198689      0.0015681747
VAR2          -0.007921409      0.0275456893       -0.00046339      -0.065017508
VAR3          -0.001198689       -0.00046339      0.0037949466      -0.246102248
Intercep      0.0015681747      -0.065017508      -0.246102248      22.392305355

                       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
```

LOAD Statement

LOAD < **MODULE**=*(module-list)* > < *matrix-list* > **;**

The LOAD statement loads modules and matrix values from the current library storage into the current workspace.

The arguments to the LOAD statement are as follows:

module-list is a list of modules.

matrix-list is a list of matrices.

For example, to load three modules A, B, and C and one matrix X, use the following 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, use the following statement:

```
load _all_;
```

If you want to load all modules, use the following statement:

```
load module=_all_;
```

To load all matrices and modules stored in the library storage, you can enter the LOAD command without any arguments, as follows:

```
load;
```

The storage library can be specified by using a RESET STORAGE command. The default library is Work.Imlstor. For more information, see Chapter 17 and the descriptions of the STORE, REMOVE, RESET, and SHOW statements.

LOC Function

LOC(*matrix***);**

The LOC function finds nonzero elements of a matrix. It creates a $1 \times n$ row vector, where n is the number of nonzero elements in the argument matrix. 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).

For example, consider the following statements:

```
a = {1 0 2 3 0};
b = loc(a);
print b;
```

Because the first, third, and fourth elements of **A** are nonzero, these statements result in the row vector shown in Figure 23.165:

Figure 23.165 Location of Nonzero Elements

```
                                b

                    1           3           4
```

If every element of the argument vector is 0, the result is empty; that is, **B** has zero rows and zero columns.

The LOC function is useful for subscripting parts of a matrix that satisfy some condition. For example, the following statements create a matrix **Y** that contains the rows of **X** that have a positive element in the diagonal of **X**:

```
x = {1  1  0,
     0 -2  2,
     0  0  3};
y = x[loc(vecdiag(x)>0), ];
print y;
```

Figure 23.166 Rows with Positive Diagonal Elements

```
                                y

                    1           1           0
                    0           0           3
```

LOG Function

LOG(*matrix***);**

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 follows:

```
c = {1 2 3};
b = log(c);
print b;
```

Figure 23.167 Natural Logarithms

```
                                    b

                0 0.6931472 1.0986123
```

LP Call

CALL LP(*rc, x, dual, a, b* < , *cntl* > < , *u* > < , *l* > < , *basis* > **);**

The LP subroutine solves the linear programming problem.

The input arguments to the LP subroutine are as follows:

a is an $m \times n$ vector that specifies the technological coefficients, where m is less than or equal to n.

b is an $m \times 1$ vector that specifies the right-side vector.

cntl is an optional row vector with one to five 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 999,999, *ndual* equals 999,999, *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 that specifies 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 that specifies 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 that specifies 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 the corresponding 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 **A** and that no nonbasic variables are at their upper bound.

rc returns one of the following scalar return codes:

rc	**Termination**
0	The solution is optimal.
1	The solution is primal infeasible and dual feasible.
2	The solution is dual infeasible and primal feasible.
3	The solution is neither primal nor dual feasible.
4	A singular basis was encountered.
5	The solution is numerically unstable.
6	The subroutine could not obtain enough memory.
7	The number of iterations was 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:

$$\max(0, \ldots, 0, 1, 0, \ldots, 0)\mathbf{x}$$
$$\text{st. } \mathbf{Ax} = \mathbf{b}$$
$$l \leq \mathbf{x} \leq \mathbf{u}$$

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 **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.

Care must be taken when solving a sequence of linear programs that use the NPRIMAL or NDUAL control parameters. 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 $X_1 + X_2 \leq 10$, and $X_1 \geq 0$, and $X_2 \geq 0$. The problem is solved by using the following statements:

```
/* 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);
print rc, x, dual;
```

Figure 23.168 Solution to a Constrained Linear Problem

```
                                    rc

                                     0

                                    x

                                    10
                                     0
                                    10

                                  dual

                                 -1          1
```

LTS Call

CALL LTS(*sc, coef, wgt, opt, y* < , *x* > < , *sorb* > **);**

The LTS subroutine performs least trimmed squares (LTS) robust regression by minimizing the sum of the h smallest squared residuals. The subroutine also detects outliers and perform a least squares regression on the remaining observations. The LTS subroutine implements the FAST-LTS algorithm described by Rousseeuw and Van Driessen (1998).

The value of h can be specified, but for many applications the default value works well and the results seem to be quite stable toward different choices of h.

In the following discussion, N is the number of observations and n is the number of regressors. The input arguments to the LTS subroutine 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]*≠ 0). If *opt[1]*=0, then a column of ones is added as the last column to the input matrix **X**; that is,

you do not need to add this column of ones 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.

0 prints no output except error messages.

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.

2 additionally prints arrays of $O(N)$, such as weights, residuals, and diagnostics; it also prints the case numbers of the observations in the best subset and some basic history of the optimization process.

3 additionally prints subsets that result in singular linear systems.

The default is *opt[2]*=0.

opt[3] specifies whether only LTS is computed or whether, additionally, least squares (LS) and weighted least squares (WLS) regression are computed:

0 computes only LTS.

1 computes, in addition to LTS, weighted least squares regression on the observations with *small* LTS residuals (where *small* is defined by opt[8]).

2 computes, in addition to LTS, unweighted least squares regression.

3 adds both unweighted and weighted least squares regression to 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[4]*$= h = \left[\frac{N+n+1}{2}\right]$, 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 that contain 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.

When *opt[5]* is zero or missing:

- If $N > 600$, the default FAST-LTS algorithm constructs up to five disjoint random subsets with sizes as equal as possible, but not to exceed 300. Inside each subset, the algorithm chooses $500/5 = 100$ subset combinations of n observations.
  The 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}	10^6	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, fixed 500 subsets for FAST-LTS or N_{Rep} subsets for algorithm before SAS/IML 8.1 are chosen randomly. This means that an exhaustive search is performed for *opt[5]*=−1. If N is larger than N_{upper}, a note is printed in the log file that indicates how many subsets exist.

opt[6] is not used.

opt[7] specifies whether the last argument *sorb* contains a given parameter vector **b** or a given subset for which the objective function should be evaluated.

0 *sorb* contains a given subset index.

1 *sorb* contains a given parameter vector **b**.

The default is *opt[7]*=0.

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.

0 does not compute covariance matrix and ASEs.

1 computes covariance matrix and ASEs but prints neither of them.

2 computes the covariance matrix and ASEs but prints only the ASEs.

3 computes and prints both the covariance matrix and the ASEs.

The default is *opt[8]*=0.

opt[9] is relevant only for LTS. If *opt[9]*=0, the algorithm FAST-LTS of Rousseeuw and Van Driessen (1998) is used. If *opt[9]* = 1, the algorithm of Rousseeuw and Leroy (1987) is used. The default is *opt[9]*=0.

y a response vector with N observations.

x 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 that contains 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[7]* 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 LTS subroutine returns the following values:

sc is a column vector that contains the following scalar information, where rows 1–9 correspond to 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_{LTS} attained
- *sc[6]* preliminary LTS scale estimate S_P
- *sc[7]* final LTS scale estimate S_F
- *sc[8]* robust R square (*coefficient of determination*)
- *sc[9]* asymptotic consistency factor

If *opt[3]* > 0, then the following are also set:

- *sc[11]* LS or WLS objective function (sum of squared residuals)
- *sc[12]* LS or WLS scale estimate
- *sc[13]* R square 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 that contains the following results in its rows:

- *coef[1,]* LTS parameter estimates
- *coef[2,]* indices of observations in the best subset

If *opt[3]* > 0, then the following are also 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, these rows correspond to LS estimates.

wgt is a matrix with N columns that contains the following results in its rows:

wgt[1,] weights (1 for small residuals; 0 for large residuals)

wgt[2,] residuals $r_i = y_i - \mathbf{x}_i\mathbf{b}$

wgt[3,] resistant diagnostic u_i (the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider Brownlee (1965) stackloss data used in the example for the LMS subroutine.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5,985 different subsets of 4 observations out of 21. If you decide not to specify `opt[5]`, the FAST-LTS algorithm chooses 500 random sample subsets, as in the following statements:

```
   /* X1  X2  X3   Y  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 };

a = aa[, 2:4]; b = aa[, 5];
opt = j(8, 1, .);
opt[2]= 1;    /* ipri */
opt[3]= 3;    /* ilsq */
opt[8]= 3;    /* icov */

call lts(sc, coef, wgt, opt, b, a);
```

Figure 23.169 Least Trimmed Squares

LTS: The sum of the 13 smallest squared residuals 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

Least Trimmed Squares (LTS) Method
Minimizing Sum of 13 Smallest Squared Residuals.
Highest Possible Breakdown Value = 42.86 %
Random Selection of 517 Subsets
Among 517 subsets 17 is/are singular.

The best half of the entire data set obtained after full iteration consists of the cases:

LTS Objective Function = 0.474940583

Preliminary LTS Scale = 0.9888435617

Robust R Squared = 0.9745520119

Final LTS Scale = 1.0360272594

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

Figure 23.169 *continued*

Unweighted Least-Squares Estimation

LS Parameter Estimates

Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.7156402	0.13485819	5.31	<.0001	0.45132301	0.97995739
VAR2	1.29528612	0.36802427	3.52	0.0026	0.57397182	2.01660043
VAR3	−0.1521225	0.15629404	−0.97	0.3440	−0.4584532	0.15420818
Intercep	−39.919674	11.8959969	−3.36	0.0038	−63.2354	−16.603949

Weighted Least-Squares Estimation

Weighted Sum of Squares = 10.273044977
Degrees of Freedom = 11
RLS Scale Estimate = 0.9663918355

Weighted R-squared = 0.9622869127
F(3,11) Statistic = 93.558645037
Probability = 4.1136826E−8
There are 15 points with nonzero weight.
Average Weight = 0.7142857143

The run has been executed successfully.

Sum of Squares = 178.8299616
Degrees of Freedom = 17
LS Scale Estimate = 3.2433639182

Cov Matrix of Parameter Estimates

	VAR1	VAR2	VAR3	Intercep
VAR1	0.0181867302	−0.036510675	−0.007143521	0.2875871057
VAR2	−0.036510675	0.1354418598	0.0000104768	−0.651794369
VAR3	−0.007143521	0.0000104768	0.024427828	−1.676320797
Intercep	0.2875871057	−0.651794369	−1.676320797	141.51474107

Figure 23.169 *continued*

```
                          R-squared = 0.9135769045
                       F(3,17) Statistic = 59.9022259
                          Probability = 3.0163272E-9


                     Least Trimmed Squares (LTS) Method

                     Least Trimmed Squares (LTS) Method
                Minimizing Sum of 13 Smallest Squared Residuals.
                  Highest Possible Breakdown Value =  42.86 %
                        Random Selection of 517 Subsets
                     Among 517 subsets 17 is/are singular.


The best half of the entire data set obtained after full iteration consists of
                                    the cases:

    5     6     7     8     9    10    11    12    15    16    17    18    19

                            Estimated Coefficients

             VAR1               VAR2               VAR3           Intercep

       0.7409210642       0.3915267228       0.0111345398      -37.32332647

                    LTS Objective Function = 0.474940583


                    Preliminary LTS Scale = 0.9888435617


                      Robust R Squared = 0.9745520119


                      Final LTS Scale = 1.0360272594


                     Weighted Least-Squares Estimation


                    RLS Parameter Estimates Based on LTS

                                 Approx                 Pr >
  Variable       Estimate       Std Err    t Value      |t|     Lower WCI    Upper WCI

  VAR1         0.75694055    0.07860766       9.63    <.0001    0.60287236    0.91100874
  VAR2         0.45353029    0.13605033       3.33    0.0067    0.18687654    0.72018405
  VAR3           -0.05211    0.05463722      -0.95    0.3607     -0.159197     0.054977
  Intercep      -34.05751    3.82881873      -8.90    <.0001    -41.561857    -26.553163
```

Figure 23.169 *continued*

```
                    Weighted Sum of Squares = 10.273044977
                          Degrees of Freedom = 11
                      RLS Scale Estimate = 0.9663918355

                      Cov Matrix of Parameter Estimates

                     VAR1              VAR2              VAR3           Intercep

VAR1          0.0061791648      -0.005776855      -0.002300587      -0.034290068
VAR2          -0.005776855      0.0185096933      0.0002582502      -0.069740883
VAR3          -0.002300587      0.0002582502      0.0029852254      -0.131487406
Intercep      -0.034290068      -0.069740883      -0.131487406      14.659852903

                       Weighted R-squared = 0.9622869127
                        F(3,11) Statistic = 93.558645037
                           Probability = 4.1136826E-8
                    There are 15 points with nonzero weight.
                         Average Weight = 0.7142857143

The run has been executed successfully.
```

The preceding program produces the following output associated with the LTS analysis. In this analysis, observations, 1, 2, 3, 4, 13, and 21 have scaled residuals larger than 2.5 (table not shown) and are considered outliers.

See the documentation for the LMS subroutine for additional details.

LUPDT Call

CALL LUPDT(*lup, bup, sup, L, z* < , *b* > < , *y* > < , *ssq* > **);**

The LUPDT subroutine provides updating and downdating for rank deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses.

The LUPDT subroutine returns the following values:

lup is an $n \times n$ lower triangular matrix L that is updated or downdated by using the q rows in Z.

bup is an $n \times p$ matrix B of right-hand sides that is updated or downdated by using the q rows in 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 Y. If *ssq* is not specified, *sup* is not accessible.

The input arguments 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 Z. Only the lower triangle of *L* is used; the upper triangle can contain any information.

z is a $q \times n$ matrix Z used rowwise to update or downdate the matrix L.

b specifies an optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with *L*. If *b* is specified, the argument *y* must be specified.

y specifies an optional $q \times p$ matrix Y used rowwise to update or downdate the right-hand-side matrix *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 *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 example in the documentation for the RZLIND call.

MAD Function

MAD(*x* < , *method*> **);**

The MAD function computes the univariate (scaled) median absolute deviation of each column of the input matrix.

The arguments to the MAD function are as follows:

x is an $n \times p$ input data matrix.

method is an optional string argument with the following values:

"MAD" for computing the median absolute deviation (MAD); this is the default.

"NMAD" for computing the normalized version of MAD

"SN" for computing S_n

"QN" for computing Q_n

For simplicity, the following descriptions assume that the input argument *x* is a column vector. The notation x_i means the *i*th element of the column vector *x*.

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,

$$\text{MAD}_n = b * med_i^n \; |x_i - med_j^n \; x_j|$$

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 that depends on sample size n.

- Q_n is another efficient alternative to MAD. It is based on the kth-order statistic of the $\binom{n}{2}$ inter-point distances,

$$Q_n = d_n * \{|x_i - x_j|; \quad i < j\}_{(k)} \quad \text{with} \quad k \approx \binom{n}{2}/4$$

  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:

$$c_n = 1.1926* \begin{cases} 0.743 & \text{for n=2} \\ 1.851 & \text{for n=3} \\ 0.954 & \text{for n=4} \\ 1.351 & \text{for n=5} \\ 0.993 & \text{for n=6} \\ 1.198 & \text{for n=7} \\ 1.005 & \text{for n=8} \\ 1.131 & \text{for n=9} \\ n/(n-0.9) & \text{for other odd n} \\ 1.0 & \text{otherwise} \end{cases} \qquad d_n = 2.2219* \begin{cases} 0.399 & \text{for n=2} \\ 0.994 & \text{for n=3} \\ 0.512 & \text{for n=4} \\ 0.844 & \text{for n=5} \\ 0.611 & \text{for n=6} \\ 0.857 & \text{for n=7} \\ 0.669 & \text{for n=8} \\ 0.872 & \text{for n=9} \\ n/(n+1.4) & \text{for other odd n} \\ n/(n+3.8) & \text{otherwise} \end{cases}$$

Example

The following example uses the univariate data set of Barnett and Lewis (1978). The data set is used in Chapter 12 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;
```

Figure 23.170 Median Absolute Deviations

```
                                rmad1

                Default MAD=         4

                                rmad2

                Common MAD =         4

                                rmad3

                MAD*1.4826 = 5.9304089

                                rmad4

                Robust S_n =  7.143674

                                rmad5

                Robust Q_n = 5.7125049
```

MARG Call

CALL MARG(*locmar, marginal, dim, table, config***);**

The MARG subroutine evaluates marginal totals in a multiway contingency table.

The input arguments to the MARG subroutine are as follows:

locmar	is a returned matrix that contains a vector of indices to each new set of marginal totals under the model specified by *config*. A marginal total is exhibited for each level of the specified marginal. These indices help locate particular totals.
marginal	is a return vector of marginal totals.
dim	is an input matrix. If the problem contains v variables, then *dim* is $1 \times v$ row vector. The value *dim*[i] is the number of possible levels for variable i in a contingency table.
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	is an input matrix. The *config* argument specifies which marginal totals to evaluate. Each column of *config* specifies a distinct marginal in the model under consideration.

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 about specifying *table*.

For example, consider the three-dimensional table discussed in the IPF call, based on data that appear in Christensen (1997). The table presents data on a person's self-esteem for people classified according to their religion and their father's educational level.

		Father's Educational Level				
Religion	Self-Esteem	Not HS Grad	HS Grad	Some Coll	Coll Grad	Post Coll
Catholic	High	575	388	100	77	51
	Low	267	153	40	37	19
Jewish	High	117	102	67	87	62
	Low	48	35	18	12	13
Protestant	High	359	233	109	197	90
	Low	159	173	47	82	32

As explained in the IPF documentation, the father's education level is Variable 1, self-esteem is Variable 2, and religion is Variable 3.

The following program encodes this table, uses the MARG call to compute a two-way marginal table by summing over the third variable and a one-way marginal by summing over the first two variables.

```
dim={5 2 3};

table={
/* Father's Education:
           NotHSGrad HSGrad Col ColGrad PostCol
          Self-
   Relig  Esteem                          */
/* Cath-   Hi */ 575   388  100    77   51,
/* olic    Lo */ 267   153   40    37   19,

/* Jew-    Hi */ 117   102   67    87   62,
/*  ish    Lo */  48    35   18    12   13,

/* Prot-   Hi */ 359   233  109   197   90,
/* estant  Lo */ 159   173   47    82   32
        };

config = { 1 3,
           2 0 };
call marg(locmar, marginal, dim, table, config);
print locmar, marginal;
```

Figure 23.171 Marginal Totals in a Three-Way Table

```
                                locmar

                                 1        11
```

Figure 23.171 *continued*

```
                                  marginal
             COL1       COL2       COL3       COL4       COL5       COL6       COL7

ROW1         1051        723        276        361        203        474        361

                                  marginal
                    COL8       COL9      COL10      COL11      COL12      COL13

         ROW1        105        131         64       1707        561       1481
```

The first marginal total is contained in locations 1 through 10 of the **marginal** vector, which is shown in Figure 23.171. It represents the results of summing **table** over the religion variable. The first entry of **marginal** is the number of subjects with high self-esteem whose fathers did not graduate from high school ($1051 = 575 + 117 + 359$). The second entry is the number of subjects with high self-esteem whose fathers were high school graduates ($723 = 388 + 102 + 233$). The tenth entry is the number of subjects with low self-esteem whose fathers had some post-collegiate education ($64 = 19 + 13 + 32$).

The second marginal is contained in locations 11 through 13 of the **marginal** vector. It represents the results of summing **table** over the education and self-esteem variables. The eleventh entry of the **marginal** vector is the number of Catholics in the study. The thirteenth entry is the number of Protestants.

You can also extract the marginal totals into separate vectors, as shown in the following statements:

```
/* Examine marginals: The name indicates the
   variable(s) that are NOT summed over.
   The locmar variable tells where to index
   into the marginal variable. */
Var12_Marg = marginal[1:(locmar[2]-1)];
Var12_Marg = shape(Var12_Marg, dim[2], dim[1]);
Var3_Marg  = marginal[locMar[2]:ncol(marginal)];
print Var12_Marg, Var3_Marg;
```

Figure 23.172 Marginal Totals

```
                          Var12_Marg

                1051        723        276        361        203
                 474        361        105        131         64

                          Var3_Marg

                           1707
                            561
                           1481
```

MATTRIB Statement

MATTRIB *name* <**ROWNAME**=*row-name*> <**COLNAME**=*column-name*> <**LABEL**=*label*> <**FORMAT**=*format*> **;**

The MATTRIB subroutine associates printing attributes with matrices.

The input arguments to the MATTRIB subroutine are as follows:

name	is a character matrix or quoted literal that contains the name of a matrix.
row-name	is a character matrix or quoted literal that specifies row names.
column-name	is a character matrix or quoted literal that specifies column names.
label	is a character matrix or quoted literal that associates a label with the matrix. The *label* argument has a maximum length of 256 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 are 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 associates a row name RA and a column name CA to **a**, and a column name CB to **b**:

```
a = {1 2 3, 4 5 6};
ra = {"Row 1", "Row 2"};
ca = 'C1':'C3';
b = 1:4;
cb = {"A" "B" "C" "D"};
mattrib a rowname=ra colname=ca  b colname=cb;
print a, b;
```

Figure 23.173 Matrix Attributes

```
                              a
                        C1          C2          C3

              Row 1      1           2           3
              Row 2      4           5           6

                              b
                   A          B           C          D

                   1          2           3          4
```

You cannot group names. The following statement does not associate anything with **a**. In fact, it clears any attributes that were previously associated with **a**.

```
mattrib a b colname=cb;
```

```
print a, b;
```

Figure 23.174 Modified Matrix Attributes

```
                              a
                        C1          C2          C3

              Row 1      1           2           3
              Row 2      4           5           6

                              b
                   A          B           C           D

                   1          2           3           4
```

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). Use the SHOW NAMES statement to view current matrix attributes.

The following example uses all options in the MATTRIB statement:

```
rows = "xr1":"xr3";
cols = "cl1":"cl4";
x = {1 1 1 1,
     2 2 2 2,
     3 3 3 3};
mattrib x rowname=rows
          colname=cols
          label={"My Matrix, x"}
          format=5.2;
print x;
```

Figure 23.175 Matrix Attributes

```
                          My 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

MAX(*matrix1* < , *matrix2, . . . , matrix15*> **);**

The MAX function returns the maximum value of a matrix or set of matrices. The matrices can be numeric or character.

For numeric arguments, the MAX function returns a single numeric value that is the largest element among all arguments. For character arguments, the MAX function returns the character string that is largest in the ASCII order. For character arguments, the size of the result is the maximum number of characters among the 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 (<>).

An example that uses the MAX function follows:

```
c = {1 -123 13 56 128 -81 12};
b = max(c);
print b;
```

Figure 23.176 Maximum Value of a Matrix

```
b

  128
```

MAXQFORM Call

CALL MAXQFORM(*rc, maxq, V, b* < *, best*> **);**

The MAXQFORM subroutine computes the subsets of a matrix system that maximize the quadratic form.

If V and 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 $b'[s]V^{-1}[s,s]b[s]$ 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 b is too large to process
2	error: 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 of *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 *V* and *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 *V* and *b* are included in the subset.

The input arguments to the MAXQFORM subroutine are as follows:

V specifies an $n \times n$ positive semidefinite matrix. Often this is generated as a crossproduct matrix, $X'X$, where X is a $k \times n$ matrix.

b specifies an $n \times 1$ vector. Often this arises as $X'y$, where X is a $k \times n$ matrix, and 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.

```
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 statement to center the data:

```
fitc = fit - fit[:,];
```

Now compute the crossproduct matrices, as follows:

```
x = fitc[, 1:6];
y = fitc[, 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;
```

Figure 23.177 Best Three Regression Sums of Squares

	maxq					
1	632.9001	0	0	1	0	0	0
1	135.78285	0	0	0	1	0	0
1	134.84474	0	0	0	0	1	0
2	650.66573	1	0	1	0	0	0
2	648.26218	0	0	1	0	1	0
2	634.46746	0	0	1	0	0	1
3	690.55086	1	0	1	0	1	0
3	689.60921	0	0	1	0	1	1
3	665.55064	1	0	1	0	0	1
4	712.45153	1	0	1	0	1	1
4	695.14669	1	1	1	0	1	0
4	694.5988	0	1	1	0	1	1
5	721.97309	1	1	1	0	1	1
5	712.63302	1	0	1	1	1	1
5	696.05218	1	1	1	1	1	0
6	722.54361	1	1	1	1	1	1

MCD Call

CALL MCD(*sc, coef, dist, opt, x***);**

The MCD subroutine computes the minimum covariance determinant estimator. The MCD call is the robust estimation of multivariate location and scatter, defined by minimizing the determinant of the covariance matrix computed from h points. The algorithm for the MCD subroutine is based on the FAST-MCD algorithm given by Rousseeuw and Van Driessen (1999).

These robust locations and covariance matrices can be used to detect multivariate outliers and leverage points. For this purpose, the MCD subroutine provides a table of robust distances.

In the following discussion, N is the number of observations and n is the number of regressors. The input arguments to the MCD 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.

- 0 prints no output except error messages.
- 1 prints most of the output.
- 2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.
- 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. 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. The default is *opt[3]*=0.

- 0 does not return or print.
- 1 prints the robust correlation matrix.
- 2 returns the final robust correlation matrix in *coef*.
- 3 prints and returns the final robust correlation matrix.

opt[4] specifies the quantile h used in the objective function. The default is *opt[4]*= $h = [\frac{N+n+1}{2}]$. 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 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.

When *opt[5]* is zero or missing:

- If $N > 600$, up to five disjoint random subsets are constructed with sizes as equal as possible, but not to exceed 300. Inside each subset, $N_{\text{Rep}} = 500/5 = 100$ subset combinations of n observations are chosen.
- If $N \leq 600$, the number of subsets is taken from the following table.

n	1	2	3	4	5	6	7 or more
N_{lower}	500	50	22	17	15	14	0

- If the number of observations N is smaller than N_{lower}, as given in the table, then all possible subsets are used; otherwise, $N_{\text{Rep}} = 500$ subsets are chosen randomly. This means that an exhaustive search is performed for *opt[5]*=−1.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors.

Missing values are not permitted in x. Missing values in *opt* cause default values to be used for each option.

The MCD subroutine returns the following values:

sc is a column vector that contains the following scalar information:

- *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_{MCD} attained (smallest determinant)
- *sc[6]* Mahalanobis-like distance used in the computation of the lowest value of the objective function F_{MCD}
- *sc[7]* the cutoff value used for the outlier decision

coef is a matrix with n columns that contains 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
- *coef[2+n+1:2+2n,]* the final robust correlation matrix for *opt[3]*=1 or *opt[3]*=3

dist is a matrix with N columns that contains 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 robust distances; 0 for large robust distances)

Example

Consider the Brownlee (1965) stackloss data used in the example for the MVE subroutine.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5,985 different subsets of 4 observations out of 21. If you decide not to specify `opt[5]`, the MCD algorithm chooses 500 random sample subsets, as in the following statements:

```
     /* X1  X2  X3   Y  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 };

a = aa[,2:4];
opt = j(8, 1, .);
opt[1] = 2;              /* ipri */
opt[2] = 1;              /* pcov: print COV */
opt[3] = 1;              /* pcor: print CORR */

call mcd(sc, xmcd, dist, opt, a);
```

A portion of the output is shown in the following figures. Figure 23.178 shows a summary of the MCD algorithm and the final *h* points selected.

Figure 23.178 Summary of MCD

```
                Fast MCD by Rousseeuw and Van Driessen

            Number of Variables                          3
            Number of Observations                      21
            Default Value for h                         12
            Specified Value for h                       12
            Breakdown Value                          42.86
            - Highest Possible Breakdown Value -
```

Figure 23.179 shows the observations that were chosen that are used to form the robust estimates.

Figure 23.179 Selected Observations

```
            MCD Estimates (Obtained by Subsampling and Iteration)


The best half of the entire data set obtained after full iteration consists of
                                the cases:


  4      5      6      7      8      9     10     11     12     13     14     20
```

Figure 23.180 shows the MCD estimators of the location, scatter matrix, and correlation matrix. The MCD scatter matrix is multiplied by a factor to make it consistent with the data that come from a single Gaussian distribution.

Figure 23.180 MCD Estimators

MCD Location Estimate

VAR1	VAR2	VAR3
59.5	20.833333333	87.333333333

MCD Scatter Matrix Estimate

	VAR1	VAR2	VAR3
VAR1	5.1818181818	4.8181818182	4.7272727273
VAR2	4.8181818182	7.6060606061	5.0606060606
VAR3	4.7272727273	5.0606060606	19.151515152

Consistent Scatter Matrix

	VAR1	VAR2	VAR3
VAR1	8.6578437815	8.0502757968	7.8983838007
VAR2	8.0502757968	12.708297013	8.4553211199
VAR3	7.8983838007	8.4553211199	31.998580526

Figure 23.181 shows the classical Mahalanobis distances, the robust distances, and the weights that identify the outlying observations (that is, leverage points when explaining y with these three regressor variables).

Figure 23.181 Robust Distances

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	12.173282	0
2	2.324745	12.255677	0
3	1.593712	9.263990	0
4	1.271898	1.401368	1.000000
5	0.303357	1.420020	1.000000
6	0.772895	1.291188	1.000000
7	1.852661	1.460370	1.000000
8	1.852661	1.460370	1.000000
9	1.360622	2.120590	1.000000
10	1.745997	1.809708	1.000000
11	1.465702	1.362278	1.000000
12	1.841504	1.667437	1.000000
13	1.482649	1.416724	1.000000
14	1.778785	1.988240	1.000000
15	1.690241	5.874858	0
16	1.291934	5.606157	0
17	2.700016	6.133319	0
18	1.503155	5.760432	0
19	1.593221	6.156248	0
20	0.807054	2.172300	1.000000
21	2.176761	7.622769	0

Robust distances are based on reweighted estimates.

The cutoff value is the square root of the 0.975 quantile of the chi square distribution with 3 degrees of freedom.

Points whose robust distance exceeds 3.0575159206 have received a zero weight in the last column above.

There were 9 such points in the data.
These may include boundary cases.
Only points whose robust distance is substantially larger than the cutoff should be considered outliers.

MEAN Function

MEAN(*x* < , *method*> < , *param*> **);**

The MEAN function computes a sample mean of data. The arguments are as follows:

x specifies an $n \times p$ numerical matrix. The MEAN function computes means of the p columns of this matrix.

method specifies the method used to compute the mean. This argument is optional. The following are valid values:

- `"arithmetic"` specifies that arithmetic means be computed. This is the default value.
- `"trimmed"` specifies that trimmed means be computed. The number of observations that are trimmed is determined by the *param* option.
- `"winsorized"` specifies that Winsorized means be computed. The number of observations that are Winsorized is determined by the *param* option.

param specifies the number of observations trimmed or Winsorized. (This argument is ignored when `"arithmetic"` is specified for the *method* argument.) The default value for *param* is 0.1, which corresponds to trimming or Winsorizing 10% of the observations with the lowest values and 10% of the observations with the largest values.

The *method* argument is not case-sensitive. The first four characters are used to determine the value. For example, `"WINS"`, `"Winsor"`, and `"winsorized"` specify the same option.

The MEAN function uses the same algorithms as the UNIVARIATE procedure for computing the means, trimmed means, and Winsorized means. For additional details and formulas, see the UNIVARIATE procedure documentation (especially the TRIMMED= and WINSORIZED= options) in the *Base SAS Procedures Guide: Statistical Procedures*.

The *param* argument determines how many observations are trimmed (or Winsorized). The value for this argument can be an integer or a proportion. If the value is an integer k, then k observations are trimmed, provided that k is between 0 and half the number of nonmissing observations. If value is a proportion p in the interval $[0, 0.5)$, then the number of observations trimmed is equal to the smallest integer that is greater than or equal to np, where n is the number of nonmissing observations.

The following example demonstrates basic usage:

```
x = {5,6,6,6,7,7,7,8,8,15};
mean = mean(x);
trim = mean(x, "trimmed", 0.2); /* 20% of obs */
winsor = mean(x, "winsorized", 1); /* one obs */
print mean trim winsor;
```

Figure 23.182 Arithmetic, Trimmed, and Winsorized Means

mean	trim	winsor
7.5	6.8333333	6.9

The MEAN function operates on columns of matrices. If *x* is an $n \times p$ matrix, the function returns a $1 \times p$ row vector. The value of the jth element is the mean for the jth column of the matrix, as the following example demonstrates:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
```

```
     7 2 13};
mean = mean(x);
print mean;
```

Figure 23.183 Arithmetic Mean of Columns

mean		
6	4	8

Missing values in a column are excluded from the computation. The default behavior of the MEAN function is identical to the subscript reduction operator that computes the mean. That is, **mean(x)** and **x[:,]** both compute the means of the columns of **x**. See the section "Subscript Reduction Operators" on page 59 for more information about subscript reduction operators.

MIN Function

MIN(*matrix1* <, *matrix2, . . . , matrix15*> **);**

The MIN function returns the minimum value of a matrix or set of matrices. The matrices can be numeric or character.

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.

The following statements use the MIN function to compute the minimum value of a vector:

```
c = {1 -123 13 56 128 -81 12};
b = min(c);
print b;
```

Figure 23.184 Minimum Value

b
-123

MOD Function

MOD(*value, divisor***);**

The MOD function returns the remainder of the division of elements of the first argument by elements of the second argument.

The arguments to the MOD function are as follows:

value is a numeric matrix or literal that contains the dividend.

divisor is a numeric matrix or literal that contains the divisor.

If either operand is a scalar, the MOD function performs the operation for each element of the matrix with the scalar value. If either operand is a row or column vector, then the operation is performed by using that vector on each of the rows or columns of the matrix.

Unlike the MOD function in Base SAS software, the MOD function in SAS/IML software does not perform any numerical "fuzzing" to return an exact zero when the result would otherwise be very small. Thus the results of the SAS/IML MOD function is more similar to the MODZ function in Base SAS software.

An example of a valid statement follows:

```
c = {-7 14 20 -81 23};
b = mod(c, 4);
print b;
```

Figure 23.185 Remainders after Division

b				
-3	2	0	-1	3

MODULEI Call

CALL MODULEI(*control, modname, < matrix1, . . . , matrix13>* **);**

The MODULEI subroutine calls an external routine that does not return a value.

The input arguments to the MODULEI subroutine are as follows:

control is a character matrix that contains a control string.

modname is a character matrix that contains the name of the external routine to be called.

matrix specifies matrix parameters to be passed to the external routine.

The CALL MODULEI routine executes a routine *modname* that resides in an external shared library with the specified arguments.

The MODULEI call routine is similar to the MODULE call routine that is available in the SAS DATA step. It is also closely related to the MODULEIN function, which returns a scalar numeric value, and the MODULEIC function, which returns a character value. CALL MODULEI builds a parameter list by using the information in the arguments and a routine description and argument attribute table that you define in a separate file. The attribute table is a sequential text file that contains descriptions of the routines that you can invoke with the CALL MODULEI routine and MODULEIN and MODULEIC functions. The purpose of the table is to define how CALL MODULEI should interpret its supplied arguments when it builds a parameter list to pass to the external routine. The attribute table should contain a description for each external routine that you intend to call, and descriptions of each argument associated with that routine. This enables you to call external routines that have been compiled in different programming languages that use different calling and matrix representation conventions.

Before you invoke CALL MODULEI, you must define the fileref of SASCBTBL to point to the external file that contains the attribute table. You can name the file whatever you want when you create it. You can then use matrices as arguments to CALL MODULEI and ensure that these arguments are properly converted before being passed to the external routine. The exact syntax for the attribute table is system-dependent, and can be found in the *SAS Companion* for your operating system. Attempting to use CALL MODULEI for a module without a correct attribute table entry can cause the SAS System to fail.

MODULEIC Function

MODULEIC(*control, modname, < matrix1, . . . , matrix13>* **);**

The MODULEIC subroutine calls an external routine that returns a character value.

The arguments to the MODULEIC function are as follows:

control is a character matrix that contains a control string.

modname is a character matrix that contains the name of the external routine to be called.

matrix specifies matrix parameters to be passed to the external routine.

The MODULEIC routine executes a routine *modname* that resides in an external shared library with the specified arguments and that returns a character value.

The description of this function is identical to the description of the MODULEI call, except that the MODULEIC function returns a character value from the external routine. See the MODULEI call for a full description of the function and its arguments.

MODULEIN Function

MODULEIN(*control, modname, < matrix1, . . . , matrix13>* **);**

The MODULEIN subroutine calls an external routine that returns a numerical value.

The arguments to the MODULEIN function are as follows:

control is a character matrix that contains a control string.

modname is a character matrix that contains the name of the external routine to be called.

matrix specifies matrix parameters to be passed to the external routine.

The MODULEIN routine executes a routine *modname* that resides in an external shared library with the specified arguments and that returns a numeric value.

The description of this function is identical to the description of the MODULEI call, except that the MODULEIN function returns a scalar numeric value from the external routine. See the MODULEI call for a full description of the function and its arguments.

This example invokes the CHANGI routine from the TRYMOD.DLL module on a Windows platform. Use the following attribute table.

```
routine changi module=trymod returns=long;
arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;
```

The following statements call the CHANGI function:

```
proc iml;
   ones = J(4,5,1);
   i = do(10, 40, 10);
   j = 4:8;
   x1 = i` # ones + j;

   y1=x1;
   x2=x1;
   y2=y1;
   rc=modulein("*i","changi",6,x2);
```

MVE Call

CALL MVE(*sc, coef, dist, opt, x* < , *s* > **);**

The MVE subroutine computes the robust estimation of multivariate location and scatter, defined by minimizing the volume of an ellipsoid that contains *h* points.

The MVE subroutine computes the minimum volume ellipsoid estimator. These robust locations and covariance matrices can be used to detect multivariate outliers and leverage points. For this purpose, the MVE subroutine provides a table of robust distances.

In the following discussion, N is the number of observations and n is the number of regressors. The input arguments to the 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.

- 0 prints no output except error messages.
- 1 prints most of the output.
- 2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.
- 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. 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. The default is *opt[3]*=0.

- 0 does not return or print.
- 1 prints the robust correlation matrix.
- 2 returns the final robust correlation matrix in *coef*.
- 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]*= $h = \left[\frac{N+n+1}{2}\right]$. If the value of h is specified outside the range $\frac{N}{2}+1 \le h \le \frac{3N}{4}+\frac{n+1}{4}$, 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[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}	10^6	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}, as given in the table, then all possible subsets are used; otherwise, N_{Rep} subsets are chosen randomly. This means that an exhaustive search is performed for *opt[5]*= -1. If N is larger than N_{upper}, a note is printed in the log file that indicates how many subsets exist.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors. Missing values are not permitted in x.

s refers to an $n + 1$ vector that contains $n + 1$ observation numbers of a subset for which the objective function should be evaluated, where n is the number of parameters. In other words, the MVE algorithm computes the minimum volume of the ellipsoid that contains the observation numbers contained in s.

The MVE subroutine returns the following values:

sc is a column vector that contains the following scalar information:

- *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_{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

coef is a matrix with n columns that contains 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
- *coef[2+n+1:2+2n,]* the final robust correlation matrix for *opt[3]*=1 or *opt[3]*=3

dist is a matrix with N columns that contains 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 robust distances; 0 for large robust distances)

Example

Consider results for Brownlee (1965) stackloss data. The three explanatory variables correspond to measurements for a plant that oxidizes 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 and Leroy (1987).

```
        /* X1  X2  X3   Y    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 and Leroy (1987) cite a large number of papers where this data set was analyzed 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 chooses only 2,000 randomly selected subsets in its search. There are in total 5,985 subsets of 4 cases out of 21 cases, as shown in Figure 23.186, which is produced by the following statements:

```
a = aa[, 2:4];
opt = j(8, 1, .);
opt[1] = 2;              /* ipri */
opt[2] = 1;              /* pcov: print COV */
opt[3] = 1;              /* pcor: print CORR */
opt[5] = -1;             /* nrep: use all subsets */

call mve(sc, xmve, dist, opt, a);
```

The first part of the output (Figure 23.186) shows the classical scatter and correlation matrix, along with the means of each variable.

Figure 23.186 Classical Estimates of Scatter and Location

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

Figure 23.186 *continued*

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

The second part of the output (Figure 23.187) shows the results of the optimization (complete subset sampling):

Figure 23.187 Subset Sampling and Optimal Subset

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

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 (Figure 23.188) shows the optimization results after local improvement:

Figure 23.188 Robust Estimates of Scatter and Location

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 (Figure 23.189) presents a table that contains the classical Mahalanobis distances, the robust distances, and the weights that identify the outlying observations (that is leverage points when explaining y with these three regressor variables):

Figure 23.189 Distances and Weights

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

MinRes	1st Qu.	Median
0.6751244996	1.5084120761	1.7159242054

Mean	3rd Qu.	MaxRes
2.2282960174	2.0831826658	5.6373573538

NAME Function

NAME(*arguments***);**

The NAME function returns the names of the arguments in a column vector. The *arguments* parameter specifies the names of existing matrices.

In the following example, **N** is a 3×1 character matrix that contains the character values 'A', 'B', and 'C':

```
Seq = 1:3;
Const = -1;
N = name(Seq, Const);
do i = 1 to nrow(N);
   msg = "Values of Matrix " + N[i];
   x = value(N[i]);
```

```
   print x[label=msg];
end;
```

Figure 23.190 Matrix Names

```
                          Values of Matrix Seq

                              1         2         3

                          Values of Matrix Const

                                              -1
```

A primary use of the NAME function is in writing macros in which you want to use an argument for both its name and its value.

NCOL Function

NCOL(*matrix***);**

The NCOl function returns the number of columns in its matrix argument. If the matrix has not been given a value, the NCOL function returns a value of 0.

For example, following statements display the number of columns of the matrix **m**:

```
m = {1 2 3, 4 5 6, 3 2 1, 4 3 2, 5 4 3};
p = ncol(m);
print p;
```

Figure 23.191 Number of Columns in a Matrix

```
                                        p

                                        3
```

NLENG Function

NLENG(*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. For English text, this size is also the number of characters that can be stored in a matrix element.

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 statements demonstrate the NLENG function:

```
m = {"ab " "ijklm  ",
     "x"   " "       };
len = nleng(m);
print len;
```

Figure 23.192 Number of Bytes in Each Matrix Element

len
7

Nonlinear Optimization and Related Subroutines

The following list shows the syntax for nonlinear optimization subroutines. Subsequent sections describe each subroutine in detail.

- 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" >* **);**

The following list shows the syntax for optimization subroutines that use least squares methods. Subsequent sections describe each subroutine in detail.

- 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"* > **);**

The following list shows the syntax for supplementary subroutines that are often used in conjunction with optimization subroutines. Subsequent sections describe each subroutine in detail.

- 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"* argument, which specifies a 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 the previous lists, not all input arguments apply to each subroutine.

Note that you can specify optional arguments with the *keyword=argument* syntax.

The following list describes each argument:

opt 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 the section "Options Vector" on page 356 for details.

blc specifies a constraint matrix that defines lower and upper bounds for the n parameters in addition to general linear equality and inequality constraints. For details, see the section "Parameter Constraints" on page 354.

tc specifies a vector of thresholds that correspond to the termination criteria tested in each iteration. See the section "Termination Criteria" on page 360 for details.

par 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 section "Control Parameters Vector" on page 367.

"ptit" specifies a 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 the section "Termination Criteria" on page 360 for details.

"grd" specifies a module that computes the gradient vector, $g = \nabla f$, at a given input point x. See the section "Objective Function and Derivatives" on page 347 for details.

"hes" specifies a module that computes the $n \times n$ Hessian matrix, $\mathbf{G} = \nabla^2 f$, at a given input point x. See the section "Objective Function and Derivatives" on page 347 for details.

"jac" specifies a 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 the section "Objective Function and Derivatives" on page 347 for details.

"nlc" specifies a module that computes general equality and inequality constraints. This is the method by which nonlinear constraints must be specified. For details, see the section "Parameter Constraints" on page 354.

"jacnlc" specifies a module that computes the Jacobian matrix of first-order derivatives of the equality and inequality constraints specified by the NLC module. For details, see the section "Parameter Constraints" on page 354.

lin specifies the linear part of the quadratic optimization problem. See the section "NLPQUA Call" on page 851 for details.

The modules that can be used as input arguments for the subroutines (*"fun," "grd," "hes," "jac," "ptit," "nlc,"* and *"jacnlc"*) accept 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 the section "Using the GLOBAL Clause" on page 77 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 a successful termination that corresponds to one of the specified termination criteria. A return code $rc < 0$ indicates unsuccessful termination—that is, that the result *xr* is unreliable. See the section "Definition of Return Codes" on page 346 for more details.
- The row vector *xr*, which has length n, contains the optimal point when $rc > 0$.

NLPCG Call

CALL NLPCG(*rc, xr, "fun", x0 <, opt> <, blc> <, tc> <, par> <, "ptit"> <, "grd"> ***);**

The NLPCG subroutine uses the conjugate gradient method to solve a nonlinear optimization problem.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 a module with the *"grd"* argument, the first-order derivatives are approximated by finite difference formulas by 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 requires less memory than other optimization methods. (The NLPCG method allocates memory proportional to n, whereas other methods allocate memory proportional to n^2.) 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 the section "Constrained Betts Function" on page 341.

NLPDD Call

CALL NLPDD(*rc, xr, "fun", x0 <, opt> <, blc> <, tc> <, par> <, "ptit"> <, "grd"> ***);**

The NLPDD subroutine uses the double-dogleg method to solve a nonlinear optimization problem.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 (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 the section "Constrained Betts Function" on page 341):

```
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};
opt = {0 1};
call nlpdd(rc, xres, "F_BETTS", x, opt, con);
```

Figure 23.193 shows the iteration history. The optimization converged after six iterations.

Figure 23.193 Constrained Optimization

```
    NOTE: Initial point was changed to be feasible for boundary and linear
                                constraints.

                          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

                                                          Max Abs             Slope
         Rest     Func      Act     Objective  Obj Fun Gradient             Search
 Iter    arts    Calls      Con      Function   Change  Element  Lambda     Direc

    1       0        2        0    -99.54678   1.0092   0.1346   6.012   -1.805
    2       0        3        0    -99.59120   0.0444   0.1279       0  -0.0228
    3       0        5        0    -99.90252   0.3113   0.0624       0   -0.209
    4       0        6        1    -99.96000   0.0575  0.00432       0  -0.0975
    5       0        7        1    -99.96000  4.66E-6 0.000079       0  -458E-8
    6       0        8        1    -99.96000 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.
```

The optimal value for the function is returned in the **xres** vector, which is displayed in Figure 23.194.

Figure 23.194 The Optimal Value

```
                                    xres

                                  2 -9.612E-8
```

NLPFDD Call

CALL NLPFDD(*f, g, h, "fun", x0, < , par> < , "grd">* **);**

The NLPFDD subroutine uses the finite-differences method to approximate derivatives.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 $\mathbf{J}$, and the crossproduct of the Jacobian matrix $\mathbf{J}'\mathbf{J}$ at the point *x0*. 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 a 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 the objective function comprises.
- 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. 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 a module that returns a vector that defines 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 that contains 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 by using calls of the function module in a neighborhood of *x0*.
- *h* is a matrix that contains 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 that contains the values of the m functions that comprise 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 the section "Finite-Difference Approximations of Derivatives" on page 352 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 the section "Unconstrained Rosenbrock Function" on page 337), the objective function is

$$f(x) = 50(x_2 - x_1^2)^2 + \frac{1}{2}(1 - x_1)^2$$

The gradient and the Hessian are

$$g'(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 200x_1^3 - 200x_1x_2 + x_1 - 1 \\ -100x_1^2 + 100x_2 \end{bmatrix}$$

$$\mathbf{H}(x,y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 600x_1^2 - 200x_2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix}$$

At the point $x = (2, 7)$, these matrices evaluate to

$$g'(2,7) = \begin{bmatrix} -1199 \\ 300 \end{bmatrix}$$

$$\mathbf{H}(2,7) = \begin{bmatrix} 1001 & -400 \\ -400 & 100 \end{bmatrix}$$

The following statements define the Rosenbrock function and use the NLPFDD call to compute the gradient and the Hessian:

```
start F_ROSEN(x);
   y1 = 10 * (x[2] - x[1] * x[1]);
   y2 =  1 - x[1];
   f  = 0.5 * (y1 * y1 + y2 * y2);
   return(f);
finish F_ROSEN;
x = {2 7};
call nlpfdd(crit, grad, hess, "F_ROSEN", x);
print grad;
print hess;
```

Figure 23.195 Gradient and Hessian at a Point

```
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

$$f_1(x) = 10(x_2 - x_1^2)$$
$$f_2(x) = 1 - x_1$$

The Jacobian and the crossproduct of the Jacobian are

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \end{bmatrix}$$

$$\mathbf{J}^T\mathbf{J} = \begin{bmatrix} 400x_1^2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix}$$

At the point $x = (2, 7)$, these matrices evaluate to

$$\mathbf{J}(2,7) = \begin{bmatrix} -40 & 10 \\ -1 & 0 \end{bmatrix}$$

$$\mathbf{J}^T\mathbf{J}|_{(2,7)} = \begin{bmatrix} 1601 & -400 \\ -400 & 100 \end{bmatrix}$$

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 $f_2(x)$.

```
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;
```

Figure 23.196 Jacobian and Crossproduct Matrix at a Point

```
                jac

          -40        10
           -1         0

                crpj

         1601      -400
         -400       100
```

NLPFEA Call

CALL NLPFEA(*xr, x0, blc* < , *par*> **);**

The NLPFEA subroutine computes feasible points subject to constraints.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 returns missing values.

You can specify the following input arguments with the NLPFEA subroutine:

- *x0* is a row vector that defines 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 that defines a set of m boundary and linear constraints. See the section "Parameter Constraints" on page 354 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 section "Control Parameters Vector" on page 367.

The NLPFEA subroutine returns the *xr* argument. The result is a vector that contains 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 the section "Constrained Betts Function" on page 341) and an initial infeasible point $x_0 = (-17, -61)$. The subroutine returns the feasible point $(2, -50)$ as the vector XFEAS.

```
con = {  2 -50  .   .,
        50  50  .   .,
        10  -1  1  10};
x = {-17. -61};
call nlpfea(xfeas, x, con);
print xfeas;
```

Figure 23.197 Feasible Point

xfeas	
6.8	-40

NLPHQN Call

CALL NLPHQN(*rc, xr, "fun", x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"jac"*> **);**

The NLPHQN subroutine uses a hybrid quasi-Newton least squares method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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) and Al-Baali and Fletcher (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 ((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 the section "Options Vector" on page 356 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 the section "Unconstrained Rosenbrock Function" on page 337).

```
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};
opt = {2 2};
call nlphqn(rc, xr, "F_ROSEN", x, opt);
```

Figure 23.198 Optimization Results

Test of NLPHQN subroutine: No Derivatives

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

Test of NLPHQN subroutine: No Derivatives

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					

Iter	Rest arts	Func Calls	Act Con	Objective Function	Obj Fun Change	Max Abs Gradient Element	Step Size	Slope Search Direc
1	0	3	0	7.22423	4.8758	56.9322	0.0616	-628.8
2*	0	5	0	0.97090	6.2533	2.3017	0.266	-14.448
3*	0	7	0	0.81911	0.1518	3.7839	0.119	-1.942
4	0	9	0	0.69103	0.1281	5.5103	2.000	-0.144
5	0	19	0	0.47345	0.2176	8.8638	11.854	-0.194
6*	0	21	0	0.35906	0.1144	9.8734	0.253	-0.947
7*	0	22	0	0.23342	0.1256	10.1490	0.398	-0.718
8*	0	24	0	0.14799	0.0854	11.6248	1.346	-0.467
9*	0	26	0	0.00948	0.1385	2.6275	1.443	-0.296
10*	0	28	0	1.98834E-6	0.00947	0.00609	0.938	-0.0190
11*	0	30	0	7.0768E-10	1.988E-6	0.000748	1.003	-398E-8
12*	0	32	0	2.0246E-21	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	2.024647E-21
Max Abs Gradient Element	1.816858E-10	Slope of Search Direction	-1.415366E-9

Figure 23.198 *continued*

```
ABSGCONV convergence criterion satisfied.

                    Test of NLPHQN subroutine: No Derivatives

                              Optimization Results
                               Parameter Estimates
                                                          Gradient
                                                         Objective
N Parameter                 Estimate                      Function

1 X1                        1.000000                1.816858E-10
2 X2                        1.000000                -1.22069E-10

Value of Objective Function = 2.024647E-21
```

NLPLM Call

CALL NLPLM(*rc, xr, "fun", x0, opt* < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"jac"*> **);**

The NLPLM subroutine uses the Levenberg-Marquardt least squares method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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, $\mathbf{J}$, 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 $\mathbf{J}$ with the *"jac"* module argument; otherwise, the subroutine computes 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 1 indicate optimization difficulties. Values close to or larger than 1 indicate that the trust region radius can be increased.

See the section "Unconstrained Rosenbrock Function" on page 337 for an example that uses the NLPLM subroutine to solve the unconstrained Rosenbrock problem.

NLPNMS Call

CALL NLPNMS(*rc, xr, "fun", x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"nlc"*> **);**

The NLPNMS subroutine use the Nelder-Mead simplex method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 can 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, can require many calls of the function and constraint modules and can 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 the section "Parameter Constraints" on page 354 for details.

For more information about the special sets of termination criteria used by the NLPNMS algorithms, see the section "Termination Criteria" on page 360.

In addition to the standard iteration history, the NLPNMS subroutine prints the following information. For unconstrained or boundary-constrained problems, the subroutine also prints the following:

- *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 statements uses the NLPNMS call to solve the Rosen-Suzuki problem (see the section "Rosen-Suzuki Problem" on page 342), which has three nonlinear constraints. Figure 23.199 is a partial listing of the output:

```
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);
opt = j(1, 11, .);
opt[2] = 3; opt[10] = 3; opt[11] = 0;
call nlpnms(rc, xres, "F_HS43", x, opt, , , , , "C_HS43");
```

Figure 23.199 Nelder-Mead Simplex Optimization

```
                          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
```

Figure 23.199 *continued*

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	Merit Function	Merit Function Change	Ratio Between Actual and Predicted Change
1	0	12	-52.80342	4.3411	-42.3031	12.803	1.000
2	0	17	-39.51475	0.0227	-39.3797	-2.923	0.250
3	0	53	-44.02098	0.00949	-43.9727	4.593	0.0625
4	0	62	-44.00214	0.000833	-43.9977	0.0249	0.0156
5	0	72	-44.00009	0.000033	-43.9999	0.00226	0.0039
6	0	79	-44.00000	1.783E-6	-44.0000	0.00007	0.0010
7	0	90	-44.00000	1.363E-7	-44.0000	1.74E-6	0.0002
8	0	94	-44.00000	1.543E-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		

Figure 23.199 *continued*

```
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    *?*

The 2 nonlinear constraints which are marked with *?* are not satisfied at the
accuracy specified by the LCEPSILON= option. However, the default value of this
     option seems to be too strong to be applied to nonlinear constraints.
```

NLPNRA Call

CALL NLPNRA(*rc, xr, "fun", x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"grd"*> < , *"hes"*> **);**

The NLPNRA subroutine uses the Newton-Raphson method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 ((Eskow and Schnabel 1991)).

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$. 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 can perform well for medium to large problems.

Using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and can 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 the section "Options Vector" on page 356 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 the section "Objective Function and Derivatives" on page 347 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 the section "Constrained Betts Function" on page 341). The iteration history follows.

```
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};
opt = {0 2};
call nlpnra(rc, xres, "F_BETTS", x, opt, con);
```

Figure 23.200 Newton-Raphson Optimization

```
NOTE: Initial point was changed to be feasible for boundary and linear
                                constraints.
```

Figure 23.200 *continued*

Optimization Start
Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound 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		

Iter	Rest arts	Func Calls	Act Con	Objective Function	Obj Fun Change	Max Abs Gradient Element	Step Size	Slope Search Direc
1	0	2	0	-98.81551	0.2779	1.8000	0.100	-2.925
2*	0	3	0	-99.40840	0.5929	1.2713	0.294	-2.365
3*	0	4	1	-99.87460	0.4662	0.5845	0.540	-1.182
4	0	5	1	-99.96000	0.0854	0.000025	1.000	-0.171
5	0	6	1	-99.96000	1.54E-10	0	1.000	-31E-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	-3.07388E-10	Ridge	0

GCONV convergence criterion satisfied.

Figure 23.200 *continued*

```
                          Optimization Results
                           Parameter Estimates
                                                Gradient     Active
                                               Objective     Bound
          N Parameter          Estimate         Function     Constraint

          1 X1                 2.000000         0.040000      Lower BC
          2 X2              -4.860653E-9                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

CALL NLPNRR(*rc, xr, "fun", x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"grd"*> < , *"hes"*> **);**

The NLPNRR subroutine uses a Newton-Raphson ridge method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of 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 $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$. 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 can 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 might 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 1 indicates possible optimization difficulties.

The following statements invoke the NLPNRR subroutine to solve the constrained Betts optimization problem (see the section "Constrained Betts Function" on page 341). The iteration history follows.

```
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};
opt = {0 2};
call nlpnrr(rc, xres, "F_BETTS", x, opt, con);
```

Figure 23.201 Newton-Raphson Optimization

NOTE: Initial point was changed to be feasible for boundary and linear constraints.

Optimization Start
Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound 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

Figure 23.201 *continued*

```
                    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

                                                                        Actual
                                                          Max Abs         Over
          Rest     Func       Act     Objective  Obj Fun Gradient         Pred
 Iter     arts    Calls       Con      Function   Change  Element  Ridge  Change

    1        0        2         1    -99.87337   1.3358   0.5887      0   0.706
    2        0        3         1    -99.96000   0.0866 0.000040      0   1.000
    3        0        4         1    -99.96000 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
                                                 Gradient     Active
                                                Objective     Bound
          N Parameter         Estimate          Function     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

CALL NLPQN(*rc, xr, "fun", x0 <, opt> <, blc> <, tc> <, par> <, "ptit"> <, "grd"> <, "nlc"> <, "jacnlc"*> **);**

The NLPQN subroutine uses a quasi-Newton method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of NLP subroutines.

The NLPQN subroutine uses (dual) quasi-Newton optimization techniques, and it is one of the two available subroutines 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 the section "Termination Criteria" on page 360.

Unconstrained or Linearly Constrained Quasi-Newton 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 that results 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.

The following statements invoke the NLPQN subroutine to solve the Rosenbrock problem (see the section "Unconstrained Rosenbrock Function" on page 337):

```
start F_ROSEN(x);
   y1 = 10 * (x[2] - x[1] * x[1]);
   y2 =  1 - x[1];
   f  = 0.5 * (y1 * y1 + y2 * y2);
   return(f);
finish F_ROSEN;
x = {-1.2 1};
opt = {0 2 . 2};
call nlpqn(rc, xr, "F_ROSEN", x, opt);
```

Since *opt[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.

Figure 23.202 Quasi-Newton Optimization

```
                          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
```

Figure 23.202 *continued*

Iter	Rest arts	Func Calls	Act Con	Objective Function	Obj Fun Change	Max Abs Gradient Element	Step Size	Slope Search Direc
1	0	4	0	2.06405	10.0359	0.7917	0.0340	-628.8
2	0	7	0	1.92035	0.1437	8.6301	6.557	-0.0363
3	0	10	0	1.78089	0.1395	11.0943	8.193	-0.0288
4	0	13	0	1.33331	0.4476	7.6069	33.376	-0.0269
5	0	17	0	1.13400	0.1993	0.9386	15.438	-0.0260
6	0	22	0	0.93915	0.1948	3.5290	11.537	-0.0233
7	0	24	0	0.84821	0.0909	4.8308	8.124	-0.0193
8	0	30	0	0.54334	0.3049	4.1770	35.143	-0.0186
9	0	32	0	0.46593	0.0774	0.9479	8.708	-0.0178
10	0	37	0	0.35322	0.1127	2.5981	10.964	-0.0147
11	0	40	0	0.26381	0.0894	3.3028	13.590	-0.0121
12	0	41	0	0.20282	0.0610	0.6451	10.000	-0.0116
13	0	46	0	0.11714	0.0857	1.6603	11.395	-0.0102
14	0	51	0	0.07149	0.0456	2.4050	11.559	-0.0074
15	0	53	0	0.04746	0.0240	0.5628	6.868	-0.0071
16	0	58	0	0.02759	0.0199	1.3282	5.365	-0.0055
17	0	60	0	0.01625	0.0113	1.9246	5.882	-0.0035
18	0	62	0	0.00475	0.0115	0.6357	8.068	-0.0032
19	0	66	0	0.00167	0.00307	0.4810	2.336	-0.0022
20	0	70	0	0.0005952	0.00108	0.6043	3.287	-0.0006
21	0	72	0	0.0000771	0.000518	0.0289	2.329	-0.0004
22	0	76	0	1.92121E-6	0.000075	0.0365	1.772	-0.0001
23	0	78	0	2.39914E-8	1.897E-6	0.00158	1.159	-331E-8
24	0	80	0	5.0936E-11	2.394E-8	0.000016	0.967	-46E-9
25	0	119	0	3.9538E-11	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

N	Parameter	Estimate	Gradient Objective Function
1	X1	0.999991	-0.000000796
2	X2	0.999982	0.000000430

Value of Objective Function = 3.953804E-11

Nonlinearly Constrained Quasi-Newton Optimization

The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's (1978a, 1982b) 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 (Gill et al. 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 (1982). However, the VMCWD program seems to have a bug in the implementation of formula (4.4) in Powell (1982). If you specify *opt[6]*= 1, the original update of μ used in the VF02AD algorithm in Powell (1978) 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, the algorithm restarts 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. See Gill et al. (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. Therefore, 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 the section "Control Parameters Vector" on page 367 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 n_c, where n_c is the total number of constraints. Letting n_e be the number of equality constraints, the constraints must be of the following form:

$$\begin{aligned} c_i(x) &= 0, \quad i = 1, \ldots, n_c \\ c_i(x) &\geq 0, \quad i = n_e + 1, \ldots, n_c \end{aligned}$$

The first n_c 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 their number, n_e, with the eleventh element of the *opt* argument.

The nonlinear NLPQN algorithm requires the Jacobian matrix of the first-order derivatives of the n_c 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 $n_c \times n$ matrix such that the entry in the ith row and jth 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 14.8 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 the section "Rosen-Suzuki Problem" on page 342.

NLPQUA Call

CALL NLPQUA(*rc, xr, quad, x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *lin*> **);**

The NLPQUA subroutine computes an optimum value of a quadratic objective function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of NLP subroutines.

The NLPQUA subroutine uses a fast algorithm for maximizing or minimizing the quadratic objective func-

tion

$$\frac{1}{2}x^T\mathbf{G}x + g^T x + con$$

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 (Gill et al. 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 that contains 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 $Z_k^T\mathbf{G}Z_k$ 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 is solved with the LCP call, which solves the linear complementarity problem.

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 stores and processes 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 the section "Constrained Betts Function" on page 341) 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

$$\frac{1}{2}x^T G x - g^T x + con = 0.5[0.02x_1^2 + 2x_2^2] - 100 = 0.01x_1^2 + x_2^2 - 100$$

The following statements use the NLPQUA subroutine to solve the Betts problem:

```
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.};
opt = {0 2};
call nlpqua(rc, xres, quad, x, opt, 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.

Figure 23.203 Quadratic Optimization

NOTE: Initial point was changed to be feasible for boundary and linear constraints.

Optimization Start Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound Constraint
1	X1	6.800000	0.136000	2.000000	50.000000
2	X2	-1.000000	-2.000000	-50.000000	50.000000

Figure 23.203 *continued*

```
                  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

                                                      Max Abs                Slope
        Rest    Func      Act    Objective  Obj Fun Gradient    Step    Search
 Iter    arts   Calls      Con     Function   Change  Element    Size     Direc

    1       0       2        1    -99.87349   1.3359   0.5882   0.706   -2.925
    2       0       3        1    -99.96000   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

CALL NLPTR(*rc, xr, "fun", x0 < , opt> < , blc> < , tc> < , par> < , "ptit"> < , "grd"> < , "hes">);*

The NLPTR subroutine uses a trust-region method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 823 for a listing of all NLP subroutines. See Chapter 14 for a description of the arguments of NLP subroutines.

The NLPTR subroutine is a trust-region method. The algorithm uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$ and 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 $\mathbf{G}$ 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 that corresponds to the quality of the quadratic approximation. The method is implemented by using Dennis, Gay, and Welsch (1981), Gay (1983), and Moré and Sorensen (1983).

Finite difference approximations for second-order derivatives that use 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 $\mathbf{G}$ 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 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 might 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 the section "Unconstrained Rosenbrock Function" on page 337.

NORMAL Function

NORMAL(*seed*);

The NORMAL function generates pseudorandom numbers from the standard normal distribution. The *seed* argument is a numeric matrix or literal. The elements of the *seed* argument can be any integer value up to $2^{31} - 1$.

The NORMAL function returns pseudorandom numbers from 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; if that value is 0, the system time is used for the seed. This function is equivalent to the DATA step function RANNOR.

The Box-Muller transformation of the UNIFORM function deviates is used to generate the numbers. The following statements produce the output shown in Figure 23.204:

```
seed = 123456;
c = j(5, 1, seed);        /* generate 5 number from the same seed */
b = normal(c);
print b;
```

Figure 23.204 Random Values Generated from a Normal Distribution

b
-0.109483
-0.348785
1.1202546
-2.513766
1.3630022

For generating millions of pseudorandom numbers, use the RANDGEN subroutine.

NROW Function

NROW(*matrix*);

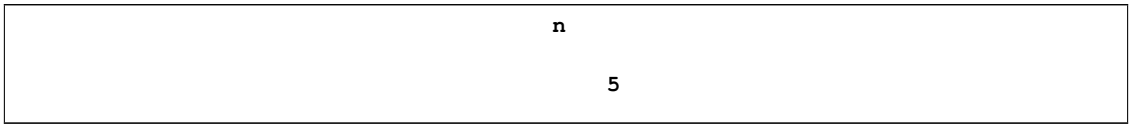
The NROW function returns the number of rows in its matrix argument. If the matrix has not been given a value, the NROW function returns a value of 0.

For example, following statements display the number of rows of the matrix **m**:

```
m = {1 2 3, 4 5 6, 3 2 1, 4 3 2, 5 4 3};
```

```
n = nrow(m);
print n;
```

Figure 23.205 Number of Rows in a Matrix



n
5

NUM Function

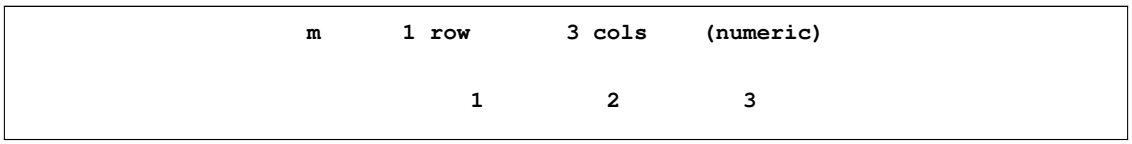
NUM(*matrix***);**

The NUM function produces a numeric representation of elements in a character matrix. If you have a character matrix for which each element is a string representation of a number, the NUM function 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.

For example, following statements display the result of converting a character matrix to a numeric matrix:

```
c = {"1" "2" "3"};
reset print;       /* display values and type of matrices */
m = num(c);
```

Figure 23.206 Numeric Matrix



m	1 row	3 cols	(numeric)
1	2	3	

You can also use the PUTN function in Base SAS software to apply a SAS format to each element of a numeric matrix. The resulting matrix is character-valued.

See also the description of the CHAR function, which converts numeric matrices into character matrices.

ODE Call

CALL ODE(*r, "dername", c, t, h* **<** **,** **J=***"jacobian"***>** **<** **,** **EPS=***eps***>** **<** **,** *"SAS-data-set"***>** **);**

The ODE subroutine performs numerical integration of first-order vector differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}(t)) \quad \text{with } \mathbf{y}(0) = \mathbf{c}$$

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 are not reported in *r*.

The input arguments to the ODE subroutine are as follows:

"dername" specifies the name of a module used to evaluate the integrand.

c specifies an initial value vector for the variable 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 subroutine 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*. 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 a 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 subroutine uses a finite-difference method to approximate the Jacobian. The keyword for this option is J.

eps specifies a scalar that indicates the required accuracy. It has a default value of 1E−4. The keyword for this option is EPS.

SAS-data-set is an optional argument that specifies the name of a valid predefined SAS data set name. The data set is used to save the successful independent and dependent variables of the integration at each step. The keyword for this option is DATA.

The ODE subroutine is an adaptive, variable-order, variable-step-size, stiff integrator based on implicit backward-difference methods. See 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.

A Linear Differential Equation

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 $t = 1$:

```
/* Define the integrand */
start fun(t,y);
   v = -t*y;
   return(v);
finish;

/* Call ODE */
c   = 0.5;
t   = {0 1};
h   = {1E-12 1 1E-5};
call ode(r1, "FUN", c, t, h);
print r1[format=E21.14];
```

Figure 23.207 Solution to a Differential Equation at $t = 1$

r1
3.03432290135600E-01

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:

```
call ode(r2, "FUN", c, t, h) eps=1E-7;
print r2[format =E21.14];
```

Figure 23.208 A Solution with Increased Accuracy

r2
3.03265687354960E-01

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 1E−7, you would use the following statements:

```
t   = {0 1 2};
h   = {1E-12 1 1E-5};
call ode(r3, "FUN", c, t, h) eps=1E-7;
print r3[format=E21.14];
```

Figure 23.209 A Solution at Two Times

```
                                    r3

          3.03265687354960E-01  6.76677185425360E-02
```

Note that **r3** contains the solution at $t = 1$ in the first column and at $t = 2$ in the second column.

A Discontinuous Forcing Function

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 does 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:

```
/* Define the integrand */
start fun(t,y);
    if t < 1 then v = t;
    else v = .5*t*t;
    return(v);
finish;

c   = 0;
t   = {0 2};
h   = {1E-12 1. 1E-5};
call ode(r1, "FUN", c, t, h) eps=1E-12;
print r1[format=E21.14];
```

Figure 23.210 Numerical Solution Across a Discontinuity

```
                                    r1

                    1.66666626639430E+00
```

In the preceding case, the integration is carried out over a single interval, $(0, 2)$. The optional parameter *eps* is specified to be 1E−12. The optional parameter *"jacobian"* is not specified, so finite-difference methods are used to estimate the Jacobian.

Note that the value of **r1** 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 statements implement this method:

```
c   = 0;
t   = {0 1 2};
h   = {1E-12 1 1E-5};
call ode(r2, "FUN", c, t, h) eps=1E-12;
print r2[format=E21.14];
```

Figure 23.211 Numerical Solution on Subintervals

r2
5.00000000003280E-01 1.66666666667280E+00

The variable **r2** 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.

A Piecewise Continuous Forcing Function

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 statements finds the solution at $t = 2$. Since the function $f(\cdot)$ is continuous, the requirements for error control are satisfied.

```
/* Define the integrand */
start fun(t,y);
    if t < 1 then v = t;
    else v = t*t;
    return(v);
finish;

c   = 0.5;
t   = {0 2};
h   = {1E-12 1 1E-5};
call ode(r, "FUN", c, t, h) eps=1E-12;
print r[format=E21.14];
```

Figure 23.212 Numerical Solution Across a Discontinuity

r
3.33333333334290E+00

Figure 23.212 *continued*

```
                                count

                                  437

                                 z1

          1.85787839378720E-06  6.58580950202810E-06
         -1.76251639451200E-03 -6.35540294085790E-03
         -1.56685625917120E-03 -5.72429205508220E-03
          1.01207878768800E-03  3.73655890904620E-03
```

Comparing Numerical Integration with an Eigenvalue Decomposition

This example compares the ODE subroutine to an eigenvalue decomposition for stiff-linear systems. In the problem

$$\frac{dy}{dt} = Ay \quad \text{with} \quad y(0) = c$$

where A is a symmetric constant matrix, the solution can be written in terms of the eigenvalue decomposition as

$$y(t) = Ue^{Dt}U'c$$

where U is the matrix of eigenvectors and D is a diagonal matrix with the eigenvalues on its diagonal.

The following statements produce two solutions, one by using the ODE subroutine and the other by using the eigenvalue decomposition:

```
/* 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);
   return(a);
finish;

a = {-1000 -1 -2 -3,
        -1 -2  3 -1,
        -2  3 -4 -3,
        -3 -1 -3 -5 };

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;
```

Figure 23.213 Numerical Integration of a Linear System

```
                                  z

                 1.85787365492010E-06  6.58581431443360E-06
                -1.76251618648210E-03 -6.35540480231360E-03
                -1.56685608329260E-03 -5.72429355490000E-03
                 1.01207978491490E-03  3.73655984699120E-03

                               count

                                   437

                                  z1

                 1.85787839378720E-06  6.58580950202810E-06
                -1.76251639451200E-03 -6.35540294085790E-03
                -1.56685625917120E-03 -5.72429205508220E-03
                 1.01207878768800E-03  3.73655890904620E-03
```

```
/* 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];
```

Figure 23.214 Analytic Solution of a Linear System

```
                                  z

                 1.85787365492010E-06  6.58581431443360E-06
                -1.76251618648210E-03 -6.35540480231360E-03
                -1.56685608329260E-03 -5.72429355490000E-03
                 1.01207978491490E-03  3.73655984699120E-03

                               count

                                   437
```

Figure 23.214 *continued*

```
                                z1

          1.85787839378720E-06  6.58580950202810E-06
         -1.76251639451200E-03 -6.35540294085790E-03
         -1.56685625917120E-03 -5.72429205508220E-03
          1.01207878768800E-03  3.73655890904620E-03
```

Is this an $O(1E{-}9)$ result? Note that for the problem

$$d = \max_{0 \le t \le T} (\|y(t)\|_\infty, 1) = 1$$

with the 1E−6 result, the ODE subroutine 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 **z** is 6.58597048842310E−06, while its more accurate value is 6.58580950203220E−06, showing an $O(1E{-}10)$ error.

Troubleshooting

The ODE subroutine can 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.

ODSGRAPH Call

CALL ODSGRAPH(*name, template, matrix1* **<** *, matrix2, . . . , matrix13* **>);**

The ODSGRAPH subroutine renders an ODS statistical graph that is defined by a template.

The input arguments to the ODSGRAPH subroutine are as follows:

name is a character matrix or quoted literal that assigns a name to the graph. The name is used to identify the output graph in the SAS Results window.

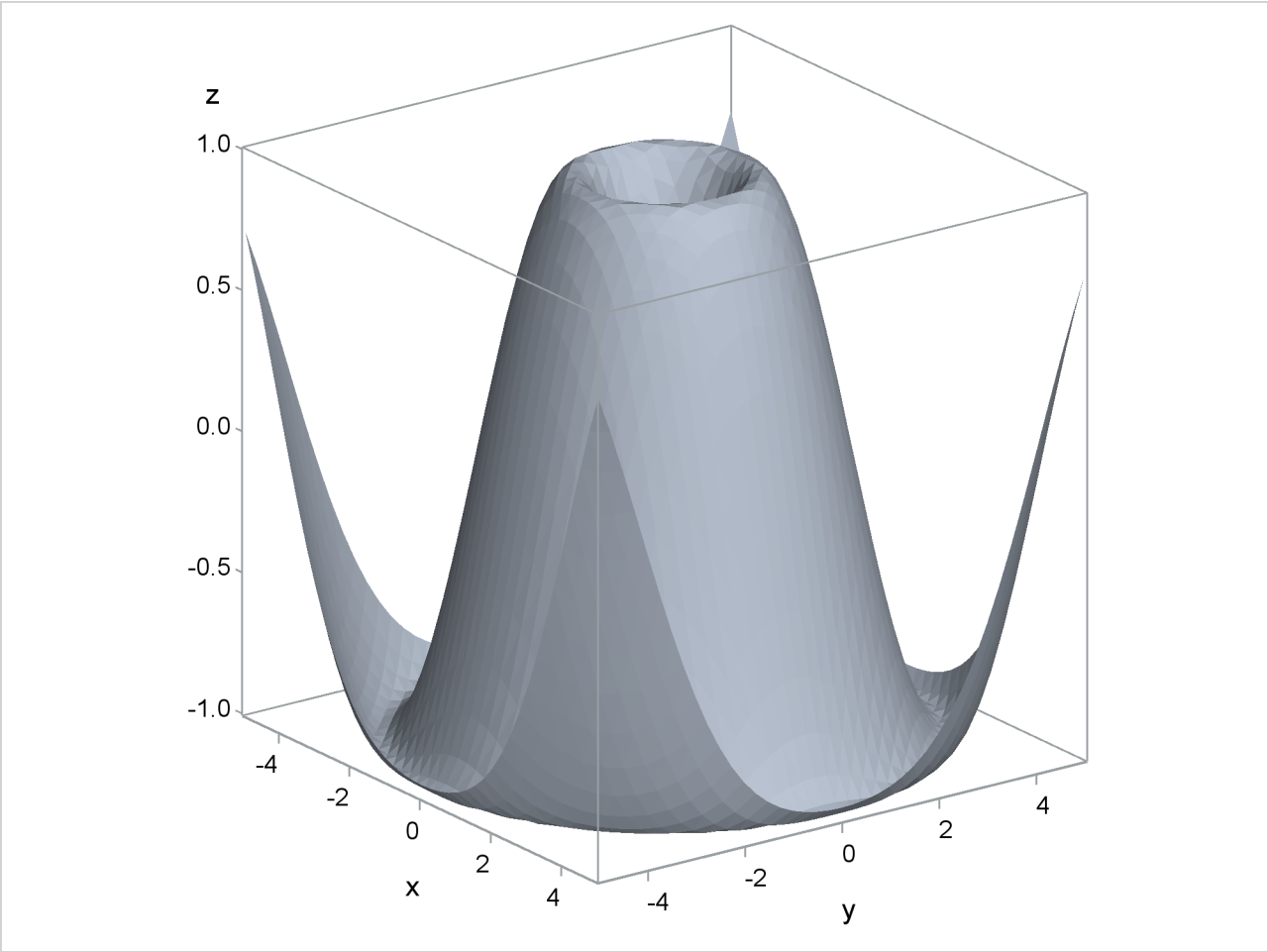
template is a character matrix or quoted literal that names the template used to render the graph.

matrix is a matrix whose columns are supplied to the template. You can specify up to 13 arguments. The name of each column must be specified by using the MATTRIB statement or the COLNAME= option in a READ statement.

The ODSGRAPH subroutine (which requires a SAS/GRAPH license) renders a graph defined by the input template. Data for the graph are in the columns of the matrix arguments. Column names are assigned to the matrices by using the MATTRIB statement or by using the COLNAME= option in a READ statement. This is illustrated in the following example, which produces a three-dimensional surface plot:

```
proc template;
  define statgraph SurfacePlot;
    BeginGraph;
    layout overlay3d;
      surfaceplotparm x=x y=y z=z / surfacetype=fill;
    endlayout;
    EndGraph;
  end;
run;

ods graphics on;
title "Surface Plot";

proc iml;
XDiv = do( -5, 5, 0.25 );
YDiv = do( -5, 5, 0.25 );
nX = ncol(XDiv);
nY = ncol(YDiv);
x = shape(repeat(XDiv, nY, 1), 0, 1);
y = shape(repeat(YDiv`, 1, nX), 0, 1);
z = sin( sqrt( x##2 + y##2 ) );
matrix = x || y || z;
mattrib matrix colname={"x" "y" "z"};
call odsgraph("surface", "SurfacePlot", matrix);
quit;

ods graphics off;
```

Figure 23.215 ODS Graphic



In the example, the TEMPLATE procedure defines a template for a surface plot. The ODSGRAPH subroutine calls ODS to render the graph by using the layout in the template. (You can render the graph in any ODS destination.) The data for the graph are contained in a matrix. The MATTRIB statement associates the columns of the matrix with the variable names required by the template.

You can also create graphs from data that are read from a data set. If x, y, and z are variables in a data set, then the following statements plot these variables:

```
use myData;
read all into matrix[colname = c];
call odsgraph("surface", "SurfacePlot", matrix);
```

Since column names created via a READ statement are permanently associated with the INTO matrix, you do not need to use a MATTRIB statement for this example.

The sample programs distributed with SAS/IML software include other examples of plots that are available by using ODS Statistical Graphics.

OPSCAL Function

OPSCAL(*mlevel, quanti* < *, qualit*> **);**

The OPSCAL function rescales qualitative data to be a least squares fit to quantitative data.

The arguments 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 and 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. 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 that results has the following properties:

- It is a least squares fit to the quantitative data in *quanti*.
- It preserves the qualitative measurement level of *qualit*.

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, the following statements create the vector shown in Figure 23.216:

```
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);
print os;
```

Figure 23.216 Optimal Scaling Transformation of Nominal Data

	os				
	COL1	COL2	COL3	COL4	COL5
ROW1	5	5	6	7	3
	os				
	COL6	COL7	COL8	COL9	COL10
ROW1	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 that correspond 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, which produce the transformation shown in Figure 23.217:

```
os2 = opscal(2, quanti, qualit);
print os2;
```

Figure 23.217 Optimal Scaling Transformation of Ordinal Data

```
                                os2
                 COL1       COL2       COL3       COL4       COL5

          ROW1      5          5          4          7          4

                                os2
                 COL6       COL7       COL8       COL9      COL10

          ROW1      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.

You can omit the *qualit* argument, as shown in the following statements:

```
quanti = {5  3  6  7  5  7  8  6  7  8};
os3 = opscal(2, quanti);
print os3;
```

These statements are equivalent to specifying

```
qualit = 1:10;
```

The result is shown in Figure 23.218.

Figure 23.218 Optimal Scaling Transformation

			os3		
	COL1	COL2	COL3	COL4	COL5
ROW1	4	4	6	6	6

			os3		
	COL6	COL7	COL8	COL9	COL10
ROW1	7	7	7	7	8

ORPOL Function

ORPOL(*x* < , *maxdegree*> < , *weights*> **);**

The ORPOL function generates orthogonal polynomials on a discrete set of points.

The arguments to the ORPOL function are as follows:

x	is an $n \times 1$ vector of values on which the polynomials are to be defined.
maxdegree	specifies the maximum degree polynomial to be computed. If *maxdegree* is omitted, the default value is $\min(n, 19)$. If *weights* is specified, you must also specify *maxdegree*.
weights	specifies an $n \times 1$ vector of nonnegative weights associated with the points in *x*. 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 evaluated at the n points contained in *x* by using the algorithm of Emerson (1968). The result is a column-orthonormal matrix $\mathbf{P}$ with n rows and *maxdegree*+1 columns such that $\mathbf{P}'\text{diag}(\text{weights})\mathbf{P} = \mathbf{I}$. The result of evaluating the polynomial of degree $j-1$ at the ith element of *x* is stored in $\mathbf{P}[i, j]$.

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 $r+1, r+2, \ldots,$ *maxdegree*+1 of the result are set to 0. In this case,

$$\mathbf{P}'\text{diag}(\text{weights})\mathbf{P} = \begin{bmatrix} I(r) & 0 \\ 0 & 0 \end{bmatrix}$$

The following statements create a matrix with three orthogonal columns, as shown in Figure 23.219:

```
x = T(1:5);
P = orpol(x,2);
print P;
```

Figure 23.219 Orthogonal Polynomials

```
                                P

                0.4472136 -0.632456 0.5345225
                0.4472136 -0.316228 -0.267261
                0.4472136 1.755E-17 -0.534522
                0.4472136 0.3162278 -0.267261
                0.4472136 0.6324555 0.5345225
```

The first column is a polynomial of degree 0 (a constant polynomial) evaluated at each point of x. The second column is a polynomial of degree 1 evaluated at each point of x. The third column is a polynomial of degree 2 evaluated at each point of x.

Normalization of the Polynomials

The columns of **P** are orthonormal with respect to the inner product

$$\langle f, g \rangle = \sum_{i=1}^{n} f(x_i) g(x_i) w_i$$

as shown by the following statements:

```
reset fuzz;          /* print tiny numbers as zero */
w = j(ncol(x),1,1); /* default weight is all ones */
/* Verify orthonormal */
L = P`*diag(w)*P;
print L;
```

Some reference books on orthogonal polynomials do not normalize the columns of the matrix that represents the orthogonal polynomials. For example, a textbook might give the following as a fourth-degree polynomial evaluated on evenly spaced data:

```
textbookPoly = { 1 -2  2 -1  1,
                 1 -1 -1  2 -4,
                 1  0 -2  0  6,
                 1  1 -1 -2 -4,
                 1  2  2  1  1 };
```

To compare this representation to the normalized representation that the ORPOL function produces, use the following program:

```
/* Normalize the columns of textbook representation */
normalPoly = textbookPoly;
do i = 1 to ncol( normalPoly );
   v = normalPoly[,i];
   norm = sqrt(v[##]);
   normalPoly[,i] = v / norm;
end;

/* Compare the normalized matrix with ORPOL */
x = T(1:5); /* Any evenly spaced data gives the same answer */
```

```
imlPoly = orpol( x, 4 );

diff = imlPoly - normalPoly;
maxDiff = abs(diff)[<>];
reset fuzz; /* print tiny numbers as zero */
print maxDiff;
```

Figure 23.220 Normalizing a Matrix

maxDiff
0

Polynomial Regression

A typical use for orthogonal polynomials is to fit a polynomial to a set of data. Given a set of points (x_i, y_i), $i = 1, \ldots, m$, the classical theory of orthogonal polynomials says that the best approximating polynomial of degree d is given by

$$f_d = \sum_{i=1}^{d+1} c_i P_i$$

where $c_i = \langle y, P_i \rangle / \langle P_i, P_i \rangle$ and where P_i is the ith column of the matrix **P** returned by ORPOL. But the matrix is orthonormal with respect to the inner product, so $\langle P_i, P_i \rangle = 1$ for all i. Thus you can easily compute a regression onto the span of polynomials.

In the following program, the weight vector is used to overweight or underweight particular data points. The researcher has reasons to doubt the accuracy of the first measurement. The last data point is also underweighted because it is a leverage point and is believed to be an outlier. The second data point was measured twice and is overweighted. (Rerunning the program with a weight vector of all ones and examining the new values of the `fit` variable is a good way to understand the effect of the weight vector.)

```
x = {0.1, 2,   3,  5,    8,   10, 20};
y = {0.5, 1, 0.1, -1, -0.5, -0.8, 0.1};

/* The second measurement was taken twice.
   The first and last data points are underweighted
   because of uncertainty in the measurements.  */
w = {0.5, 2,   1,  1,    1,    1, 0.2};
maxDegree = 4;
P = orpol(x,maxDegree,w);

/* The best fit by a polynomial of degree k is
   Sum c_i P_i  where c_i = <f,P_i> */
start InnerProduct(f,g,w);
   h = f#g#w;
   return (h[+]);
finish;

c = j(1,maxDegree+1);
do i = 1 to maxDegree+1;
```

```
   c[i] = InnerProduct(y, P[,i], w);
end;

FitResults = j(maxDegree+1,2);
do k = 1 to maxDegree+1;
   fit = P[,1:k] * c[1:k];
   resid = y - fit;
   FitResults[k,1] = k-1;       /* degree of polynomial */
   FitResults[k,2] = resid[##]; /* sum of square errors */
end;
print FitResults[colname={"Degree" "SSE"}];
```

Figure 23.221 Statistics for an Orthogonal Polynomial Regression

FitResults	
Degree	SSE
0	3.1733014
1	4.6716722
2	1.3345326
3	1.3758639
4	0.8644558

Testing Linear Hypotheses

The ORPOL function can also be used to test linear hypotheses. Suppose you have an experimental design with k factor levels. (The factor levels can be equally or unequally spaced.) At the ith level, you record n_k observations, $i = 1 \ldots k$. If $n_1 = n_2 = \ldots = n_k$, then the design is said to be *balanced*; otherwise it is *unbalanced*. You want to fit a polynomial model to the data and then ask how much variation in the data is explained by the linear component, how much variation is explained by the quadratic component after the linear component is taken into account, and so on for the cubic, quartic, and higher-level components.

To be completely concrete, suppose you have four factor levels (1, 4, 6, and 10) and that you record seven measurements at first level, two measurements at the second level, three measurements at the third level, and four measurements at the fourth level. This is an example of an unbalanced and unequally spaced factor-level design. The following program uses orthogonal polynomials to compute the Type I sum of squares for the linear hypothesis. (The program works equally well for balanced designs and for equally spaced factor levels.)

The following program calls the ORPOL function to generate the orthogonal polynomial matrix **P**, and uses it to form the Type I hypothesis matrix **L**. The program then uses the DESIGN function to generate **X**, the design matrix associated with the experiment. The program then computes **b**, the estimated parameters of the linear model. Since **L** was expressed in terms of the orthogonal polynomial matrix **P**, the computations involved in forming the Type I sum of squares are considerably simplified.

```
/* unequally spaced and unbalanced factor levels */
levels = { 1,1,1,1,1,1,1,
           4,4,
           6,6,6,
           10,10,10,10};
```

```
/* data for y. Make sure the data are sorted
   according to the factor levels */
y = {2.804823, 0.920085, 1.396577, -0.083318,
     3.238294, 0.375768, 1.513658,            /* level 1 */
     3.913391, 3.405821,                      /* level 4 */
     6.031891, 5.262201, 5.749861,            /* level 6 */
    10.685005, 9.195842, 9.255719,  9.204497 /* level 10 */
};

a      = {1,4,6,10}; /* spacing */
trials = {7,2,3,4};  /* sample sizes */
maxDegree = 3; /* model with Intercept,a,a##2,a##3 */

P = orpol(a,maxDegree,trials);

/* Test linear hypotheses:
   How much variation is explained by the
   i_th polynomial component after components
   0..(i-1) have been taken into account? */

/* the columns of L are the coefficients of the
   orthogonal polynomial contrasts */
L = diag(trials)*P;

/* form design matrix */
x = design(levels);

/* compute b, the estimated parameters of the
   linear model. b is the mean of the y values
   at each level.
   b = ginv(x'*x) * x` * y
   but since x is the output from DESIGN, then
   x`*x = diag(trials) and so
   ginv(x`*x) = diag(1/trials) */
b = diag(1/trials)*x`*y;

/* (L`*b)[i] is the best linear unbiased estimated
   (BLUE) of the corresponding orthogonal polynomial
   contrast */
blue = L`*b;

/* The variance of (L`*b) is
   var(L`*b) = L`*ginv(x`*x)*L
     =[P`*diag(trials)]*diag(1/trials)*[diag(trials)*P]
     = P`*diag(trials)*P
     = Identity         (by definition of P)

   Therefore the standardized square of
   (L`*b) is computed as
   SS1[i] = (blue[i]*blue[i])/var(L`*b)[i,i])
          = (blue[i])##2                          */

SS1 = blue # blue;
rowNames = {"Intercept" "Linear" "Quadratic" "Cubic"};
```

```
print SS1[rowname=rowNames format=11.7 label="Type I SS"];
```

Figure 23.222 indicates that most of the variation in the data can be explained by a first-degree polynomial.

Figure 23.222 Statistics for an Orthogonal Polynomial Regression

	Type I SS
Intercept	331.8783538
Linear	173.4756050
Quadratic	0.4612604
Cubic	0.0752106

Generating Families of Orthogonal Polynomials

There are classical families of orthogonal polynomials (for example, Legendre, Laguerre, Hermite, and Chebyshev) that arise in the study of differential equations and mathematical physics. These "named" families are orthogonal on particular intervals (a, b) with respect to the inner product $\int_b^a f(x)g(x)w(x)\,dx$. The functions returned by the ORPOL function are *different* from these named families because the ORPOL function uses a different inner product. There are no built-in functions that can automatically generate these families; however, you can write a program to generate them.

Each named polynomial family $\{p_j\}$, $j \geq 0$ satisfies a three-term recurrence relation of the form

$$p_j = (A_j + xB_j)p_{j-1} - C_j p_{j-2}$$

where the constants A_j, B_j, and C_j are relatively simple functions of j. To generate these "named" families, use the three-term recurrence relation for the family. The recurrence relations are found in references such as Abramowitz and Stegun (1972) or Thisted (1988).

For example, the so-called Legendre polynomials (represented P_j for the polynomial of degree j) are defined on $(-1, 1)$ with the weight function $w(x) = 1$. They are standardized by requiring that $P_j(1) = 1$ for all $j \geq 0$. Thus $P_0(x) = 1$. The linear polynomial $P_1(x) = a + bx$ is orthogonal to P_0 so that

$$\int_{-1}^{1} P_1(x)P_0(x)\,dx = \int_{-1}^{1} a + bx\,dx = 0$$

which implies $a = 0$. The standardization $P_1(1) = 1$ implies that $P_1(x) = x$. The remaining Legendre polynomials can be computed by looking up the three-term recurrence relation: $A_j = 0$, $B_j = (2j-1)/j$, and $C_j = (j-1)j$. The following program computes Legendre polynomials evaluated at a set of points:

```
maxDegree = 6;
/* evaluate polynomials at these points */
x = T( do(-1,1,0.05) );

/* define the standard Legendre Polynomials
   Using the 3-term recurrence with
   A[j]=0, B[j]=(2j-1)/j, and C[j]=(j-1)/j
   and the standardization P_j(1)=1
   which implies P_0(x)=1, P_1(x)=x. */
legendre = j(nrow(x), maxDegree+1);
legendre[,1] = 1; /* P_0 */
```

```
legendre[,2] = x; /* P_1 */

do j = 2 to maxDegree;
   legendre[,j+1] = (2*j-1)/j # x # legendre[,j] -
                    (j-1)/j # legendre[,j-1];
end;
```

ORTVEC Call

CALL ORTVEC(*w, r, rho, lindep, v* <, *q*> **);**

The ORVEC subroutine provides columnwise orthogonalization and stepwise QR decomposition by using the Gram-Schmidt process.

The ORTVEC subroutine returns the following values:

w is an $m \times 1$ vector. If the Gram-Schmidt process converges (*lindep*=0), *w* is orthonormal to the columns of 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 $(n+1)$th orthogonal column of the matrix Q. If the *q* argument is not specified, *w* is the normalized value of the vector *v*,

$$w = \frac{v}{\sqrt{v'v}}$$

r is a $n \times 1$ vector. If the Gram-Schmidt process converges (*lindep*=0), *r* contains 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 $(n+1)$th column of R.

rho is a scalar value. If the Gram-Schmidt process converges (*lindep*=0), *rho* specifies the distance from *w* to the range of Q. Even if the Gram-Schmidt process converges, if *rho* is sufficiently small, the vector *v* can be linearly dependent on the columns of Q. If the Gram-Schmidt process does not converge (*lindep*=1), *rho* is set to 0. For stepwise QR decomposition, *rho* contains the diagonal element of the $(n+1)$th column of R. In formulas, the value *rho* is denoted by ρ.

lindep returns a value of 1 if the Gram-Schmidt process does not converge in 10 iterations. A value of 1 often indicates that the input vector *v* is linearly dependent on the n columns of the input matrix Q. In that case, *rho* 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 *rho* are computed.

The input arguments to the ORTVEC subroutine are as follows:

v specifies an $m \times 1$ vector *v* that is to be orthogonalized to the n columns of Q. For stepwise QR decomposition of a matrix, *v* is the $(n+1)$th matrix column before its orthogonalization.

q specifies an optional $m \times n$ matrix Q that is assumed to have $n \leq m$ (nearly) orthonormal columns. Thus, the $n \times n$ matrix $Q'Q$ 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

$$v = \mathbf{Qr} + \rho w$$

In the formula, if the $m \times n$ matrix Q has n (nearly) orthonormal columns, the vector *v* is orthogonal to the columns of Q and ρ is the distance from *w* to the range of Q.

There are two special cases:

- If $m > n$, ORTVEC normalizes the result *w*, so that *w′ w*= 1.
- If $m = n$, the output vector *w* is the null vector.

The case $m < n$ is not possible since Q is assumed to have n (nearly) orthonormal columns.

To initialize a stepwise QR decomposition, the ORTVEC subroutine can be called to normalize *v* only (that is, to compute $w = v/\sqrt{v'v}$ and $\rho = \sqrt{v'v}$). There are two ways of using the ORTVEC subroutine for this reason:

- Omit the last argument *q*, as in `call ortvec(w,r,rho,lindep,v);`.
- Provide a matrix *q* with zero rows and columns (for example, by using the `free q;` command).

In both cases, *r* is a column vector with zero rows.

The ORTVEC subroutine is useful for the following applications:

- performing stepwise QR decomposition. Compute Q and R, so that $A = QR$, where Q is column orthonormal, $Q'Q = I$, and R is upper triangular. The jth step is applied to the jth column, *v*, of A, and it computes the jth column *w* of Q and the jth column, $(r\ \rho\ 0)'$, of R.
- computing the $m \times (m - n)$ null space matrix, Q_2, that corresponds to an $m \times n$ range space matrix, Q_1 $(m > n)$, by the following stepwise process:
  1. Set $v = e_i$ (where e_i is the ith unit vector) and try to make it orthogonal to all column vectors of Q_1 and the already generated Q_2.
  2. If the subroutine is successful, append *w* to Q_2; otherwise, try $v = e_{i+1}$.

The 4×3 matrix Q contains the unit vectors e_1, e_3, and e_4. The column vector *v* is pairwise linearly independent with the three columns of Q. As expected, the ORTVEC subroutine computes the vector *w* as the unit vector e_2 with $u = (1, 1, 1)$ and $\rho = 1$.

```
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;
```

Figure 23.223 Matrix Orthogonalization

rho	u	w
1	1	0
	1	1
	1	0
		0

Stepwise QR Decomposition Example

You can perform the QR decomposition of the linearly independent columns of an $m \times n$ matrix $\mathbf{A}$ with the following statements:

```
a = {1  2  1,
     2  4  2,
     1  4 -1,
     1  0  3}; /* use any matrix A */
nind = 0;  ndep = 0;  dmax = 0.;
n = ncol(a);  m = nrow(a); ind = j(1,n,0);
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;
print q r;
```

Figure 23.224 QR Decomposition of Independent Columns

```
                          q                   r

                  0.3779645          0 2.6457513 5.2915026
                  0.7559289          0         0 2.8284271
                  0.3779645  0.7071068         0         0
                  0.3779645 -0.707107
```

Next, process the remaining (dependent) 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;
print q r;
```

Figure 23.225 QR Decomposition of Dependent Columns

```
                  q                              r

          0.3779645          0 -0.239046 2.6457513 5.2915026 2.6457513
          0.7559289          0 -0.478091         0 2.8284271 -2.828427
          0.3779645  0.7071068 0.5976143         0         0 1.327E-16
          0.3779645 -0.707107 0.5976143
```

You can also use the ORTVEC subroutine to 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;
```

```
print q;
```

Figure 23.226 Final Orthogonal Matrix

q			
0.3779645	0	−0.239046	0.8944272
0.7559289	0	−0.478091	−0.447214
0.3779645	0.7071068	0.5976143	0
0.3779645	−0.707107	0.5976143	−3.1E−17

In the example, if you define $\mathbf{Q}_2$ to be the last two columns of $\mathbf{Q}$, then $\mathbf{Q}_2' A = \mathbf{0}$.

PAUSE Statement

PAUSE < *expression* > < * > **;**

The PAUSE statement interrupts the execution of a module.

The arguments to the PAUSE statement are as follows:

expression	is a character matrix or quoted literal that contains 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 following default message is printed:

```
Paused in module XXX.
```

In this case, *XXX* is the name of the module that contains the pause. If you want to suppress all messages in a PAUSE statement, use an asterisk as the operand, as follows:

```
pause *;
```

The PAUSE statement should be specified only in modules. It generates a warning if executed in immediate mode.

When an error occurs while executing inside a module, PROC IML automatically behaves as though a PAUSE statement was issued. The following note is displayed in the SAS Log:

```
Paused in module XXX.
```

PROC IML also enters "immediate mode" within the module environment. You can correct the error and then resume execution by issuing a RESUME command.

PROC IML supports pause processing of both subroutine and function modules. See also the description of the SHOW statement which uses the PAUSE option.

PGRAF Call

CALL PGRAF(*xy* < , *id*> < , *xlabel*> < , *ylabel*> < , *title*> **);**

The PGRAF subroutine displays a low-resolution scatter plot, sometimes called a "line-printer plot."

The arguments 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 beneath 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 following statements 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");
```

```
                Plot of X vs Y
    |
  4 +                            *
    |
    |
    |
Y   |
  3 +               *
    |
    |
    |
    |
  2 + *                           *
    --+------+------+------+------+------+-
     1.0    2.0    3.0    4.0    5.0    6.0

                        X
```

POLYROOT Function

POLYROOT(*vector***);**

The POLYROOT function computes the zeros of a real polynomial. The *vector* argument is an $n \times 1$ (or $1 \times n$) vector that contains 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 that contains 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 is 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. If r is given as a root of the polynomial $P(x)$, then $1 + P(R) = 1$, based on the rounding error of the computer that is employed.

For example, you can use the following statements to find the roots of the polynomial

$$P(x) = 0.2567x^4 + 0.1570x^3 + 0.0821x^2 - 0.3357x + 1$$

```
p = {0.2567 0.1570 0.0821 -0.3357 1};
r = polyroot(p);
print r;
```

Figure 23.227 Roots of a Quartic Polynomial

```
                    r

          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

PRINT < *matrices* > < *(expression)* > < *"message"* > < *pointer-controls* > < *[options]* > **;**

The PRINT statement displays the values of matrices or literals.

The arguments 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, a comma (,) skips a single line and a slash (/) skips to a new page.
options	are described in the following list.

The following *options* 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];
```

LABEL=*label*
specifies the name of a scalar character matrix or literal to use as a label when printing the matrix. For example:

```
print x[label="Net Pay"];
```

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 following RESET AUTONAME statement to automatically label rows as ROW1, ROW2, and so on:

```
reset autoname;
```

For example, the following statements print a matrix in the 12.2 format with column and row labels:

```
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];
```

Figure 23.228 Matrix with Row and Column Labels

```
                                  x
                              Amount      Net Pay

                     Div A     45.13        50.50
                     Div B     75.38        90.83
```

To permanently associate the preceding options with a matrix name, see 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 continued on a subsequent line.

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, including the PRINTADV option, see the description of the RESET statement.

To print part of a matrix or a temporary expression, enclose the expression in parentheses:

```
y=1:10;
print (y[1:3])[format=5.1]; /* prints first few elements */
print (sum(y))[label="sum"];
```

Figure 23.229 Printing Temporary Matrices

```
                                  x
                              Amount      Net Pay

                     Div A     45.13        50.50
                     Div B     75.38        90.83

                               1.0
                               2.0
                               3.0

                              sum

                                  55
```

PROD Function

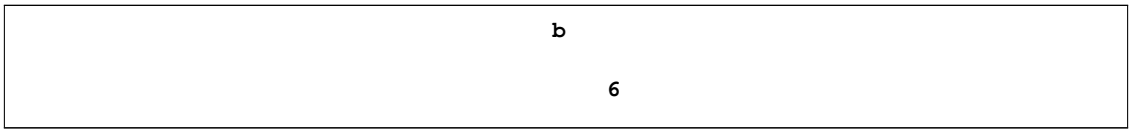
PROD(*matrix1* **<** *, matrix2, . . . , matrix15***>);**

The PROD function returns as a single numeric value the product of all nonmissing elements in all arguments. You can pass in as many as 15 numeric matrices as arguments. The PROD function checks for missing values and does not include them in the product. It returns missing if all values are missing.

For example, consider the following statements:

```
a = {2 1, . 3};
b = prod(a);
print b;
```

Figure 23.230 Output from the PROD Function



b
6

For a single argument with at least one nonmissing value, the PROD function is identical to the subscript reduction operator that computes the product. That is, **prod(x)** and **x[#]** both compute the product of the elements of **x**. See the section "Subscript Reduction Operators" on page 59 for more information about subscript reduction operators.

PRODUCT Function

PRODUCT(*a, b* **<** *, dim* **>);**

The PRODUCT function multiplies matrices of polynomials.

The arguments 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 the dimension p of the matrix b. 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.

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 following statements demonstrate the PRODUCT function:

```
m1 = {1 2 3 4,
      5 6 7 8};
m2 = {1 2 3,
      4 5 6};
r = product(m1, m2, 1);
print r;
```

Figure 23.231 A Product of Matrices of Polynomials

r			
9	31	41	33
29	79	105	69

PURGE Statement

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 can be time-consuming because it involves rewriting the entire data set.

CAUTION: Any indexes associated with the data set are lost after a purge.

When you quit PROC IML, observations marked for deletion are *not* automatically purged.

The following example creates a data set named A. The EDIT statement opens the data set for editing. The DELETE statement marks several observations for deletion. As shown in Figure 23.232, the observations are not removed and renumbered until the PURGE statement executes.

```
data a;
do i=1 to 10;
   output;
end;
run;

proc iml;
edit a;
pts = 3:8;
delete point pts;
list all;

purge;
list all;
```

Figure 23.232 Deleting and Purging Observations

```
   OBS         i
------ ----------
     1    1.0000
     2    2.0000
     9    9.0000
    10   10.0000


   OBS         i
------ ----------
     1    1.0000
     2    2.0000
     3    9.0000
     4   10.0000
```

PUSH Call

CALL PUSH(*argument1* < *, argument2, . . . , argument15*> **);**

The PUSH subroutine pushes character arguments that contain valid SAS statements (usually SAS/IML statements or global statements) to the input command stream. You can specify up to 15 arguments. Any statements in the input command queue are executed when the module is paused (see the PAUSE statement), which happens when one of the following occurs:

- An execution error occurs within a module.
- An interrupt is issued.
- A PAUSE statement executes.

The pushed string 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 many statements at one time. Pushing too many statements causes problems that can result in exiting the SAS System.

For more information about the input command stream, see Chapter 18.

An example that uses the PUSH subroutine follows:

```
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 main;
```

Figure 23.233 Result of a PUSH Statement

```
Options: noautoname center noclip
         deflib=WORK (system-specific-pathname)
         nodetails noflow nofuzz fw=9
         imlmlib=SASHELP.IMLMLIB  linesize=80 nolog
         name pagesize=25 noprint noprintall spaces=1
         userlib=WORK.IMLSTOR(not open)
```

PUT Statement

PUT < *operand* > < *record-directives* > < *positionals* > < *format* > **;**

The PUT statement writes data to an external file.

The arguments 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 @* is used at the end of a PUT statement to hold the current record so that you can continue to write more data to the record with later PUT statements. Otherwise, the next record is used for the next PUT statement.
- / writes out the current record and begins forming a new record.
- > *operand* specifies that the next record written 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 $NAMEw.d$, 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 matrices. The statement is described in detail in Chapter 8.

The PUT statement is a sequence of positionals and record directives, variables, and formats. An example that uses the PUT statement follows:

```
/* output variable A in column 1 using a 6.4 format */
/* Skip 3 columns and output X using an 8.4 format  */
put @1 a 6.4 +3 x 8.4;
```

PV Function

PV(*times,flows,freq,rates***);**

The PV function returns a scalar that contains the present value of the cash flows based on the specified frequency and rates.

The arguments to the function are as follows:

times	is an $n \times 1$ column vector of times. Elements should be nonnegative.
flows	is an $n \times 1$ column vector of cash flows.
freq	is a scalar that 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 per period. If zero, it represents continuous compounding. If –1, the rates represent per-period discount factors. No other negative values are accepted.
rates	is an $n \times 1$ 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, $\{c(k)\}, k = 1, \ldots, K$, is the sequence of cash flows from the asset, t_k is the time to the kth cash flow in periods from the present, and $D(t)$ is the discount function for time t. The discount factors are as follows:

- with per-unit-time-period discount factors d_t:

$$D(t) = d_t^t$$

- with continuous compounding:

$$D(t) = e^{-r_t t}$$

- with discrete compounding:

$$D(t) = (1 + fr)^{-t/f}$$

  where $f > 0$ is the frequency, the reciprocal of the number of compoundings per unit time period.

The following statements present an example of using the PV function in the DATA step:

```
data a;
  pv = mort(., 438.79, 0.10/12, 30*12);
run;
proc print data=a; run;
```

Figure 23.234 Present Value Computation (DATA Step)

OBS	pv
1	50000.48

You can do the same computation by using the PV function in SAS/IML software. The first example uses a monthly rate; the second example uses an annual rate.

```
proc iml;
/*  If rate is specified as annual rate divided by 12 and FREQ=1,
 *  then results are equal to those computed by the MORT function. */
timesn = t(1:360);
flows = repeat(438.79, 360);
rate = repeat(0.10/12, 360);
freq = 1;
pv = pv(timesn, flows, freq, rate);
print pv;

/*  If rate is specified as annual rate, then the cash flow TIMES
 *  need to be specified in 1/12 increments and the FREQ=1/12.
 *  This produces the same result as the previous PV call. */
timesn = t(do(1/12, 30, 1/12));
flows = repeat(438.79, 360);
rate = repeat(0.10, 360); /* specify annual rate  */
freq = 1/12;              /* 12 compoundings annually */
pv = pv(timesn, flows, freq, rate);
print pv;
```

Figure 23.235 Present Value Computation (PROC IML)

pv
50000.48

pv
50000.48

QNTL Call

CALL QNTL(*q, x,* < *, probs*> < *, method*> **);**

The QNTL subroutine computes sample quantiles for data. The arguments are as follows:

q specifies a matrix to contain the quantiles of the *x* matrix.

x specifies an $n \times p$ numerical matrix of data. The QNTL subroutine computes quantiles for each column of the matrix.

probs specifies a numeric vector of probabilities used to compute the quantiles. If this option is not specified, the vector $\{0.25, 0.5, 0.75\}$ is used, resulting in the quartiles of the data.

method specifies the method used to compute the quantiles. These methods correspond to those defined by using the PCTLDEF= option in the UNIVARIATE procedure. For details, see the section "Calculating Percentiles" of the documentation for the CORR procedure in the *Base SAS Procedures Guide: Statistical Procedures*.

The following values are valid:

1 specifies that quantiles are computed according to a weighted average.

2 specifies that quantiles are computed by choosing an observation closest to some quantity.

3 specifies that quantiles are computed by using the empirical distribution function.

4 specifies that quantiles are computed according to a different weighted average.

5 specifies that quantiles are computed by using average values of the empirical distribution function. This is the default value.

If *x* is an $n \times p$ matrix, the QNTL subroutine computes a $k \times p$ matrix where k is the dimension of the PROBS= option. The quantiles are returned in the *q* matrix, as shown in the following example:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
     7 2 13};
```

```
call qntl(q, x);
print q[rowname={"P25", "P50", "P75"}];
```

Figure 23.236 Quantiles

	q		
P25	6	2	5
P50	6	2	9
P75	6	7	10

You can use the MATTRIB statement to permanently assign row names to the matrix that contains the quantiles, as shown in the following statements:

```
p = {0.25 0.50 0.75};
labels = "P" + strip(putn(100*p, "best5."));
mattrib q rowname=labels;
print q;
```

Figure 23.237 Rownames for Quantiles

	q		
P25	6	2	5
P50	6	2	9
P75	6	7	10

You can specify the optional arguments in either of two ways: by specifying an argument positionally or by specifying a keyword/value pair, as shown in the following statements.

```
x = T(1:100);
p = do(0.1, 0.9, 0.1);
call qntl(q1, x, p);
call qntl(q2, x) probs=p; /* equivalent */
```

QR Call

CALL QR(*q, r, piv, lindep, a* < , *ord*> < , *b*> **);**

The QR subroutine produces the QR decomposition of a matrix by using Householder transformations.

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 (by using the 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 $[\mathbf{A}_{piv[1]} \ldots \mathbf{A}_{piv[n]}]$. 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 input arguments 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 $\mathbf{A}$. When you specify the *ord* argument, the columns of $\mathbf{A}$ can be divided into the following groups:

ord[jj]>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 n_l columns of $\mathbf{A}$ that are to be processed.

ord[jj]=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 n_l initial columns and before the n_u final columns.

ord[jj]<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 n_u columns of $\mathbf{A}$ that are to be processed. If $n > m$, some of these columns are not processed.

The default is *ord[jj]*=j, in which case the Householder transformations are processed in the same order in which 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

$$\mathbf{A\Pi} = \mathbf{Q}\widetilde{\mathbf{R}}, \mathbf{Q}'\mathbf{Q} = \mathbf{QQ}' = \mathbf{I}_m$$

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 in the following example:

```
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 $\mathbf{Q}$ might require too much memory or time.

In the usual case where $m > n$,

$$\mathbf{A} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

$$\widetilde{\mathbf{R}} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}$$

$$\mathbf{Q} = [\mathbf{Q}_1 \; \mathbf{Q}_2], \qquad \widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

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. 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 $\mathbf{A}$.

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:

```
a = shape(1:20, 5);
m = nrow(a); n = ncol(a);
ord = j(1, n, 0);
call qr(q, r, piv, lindep, a);
ss0 = ssq(a[ ,piv] - q[,1:n] * r);
ss1 = ssq(q * q` - i(m));
ss2 = ssq(q` * q - i(m));
print ss0 ss1 ss2;
```

Figure 23.238 Result of a QR Decomposition

ss0	ss1	ss2
6.231E-28	2.948E-31	2.862E-31

If the QR subroutine detects linearly dependent columns while processing matrix **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 **A** is found to be linearly dependent on columns already processed, this column is swapped to the end of matrix **A**. The order of columns in the result matrix **R** corresponds to the order of columns processed in **A**. The swapping of a linearly dependent column of **A** to the end of the matrix corresponds to the swapping of the same column in **R** and leads to a zero row at the end of the upper triangular matrix **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 singularity criterion, which is 1E−8 by default.

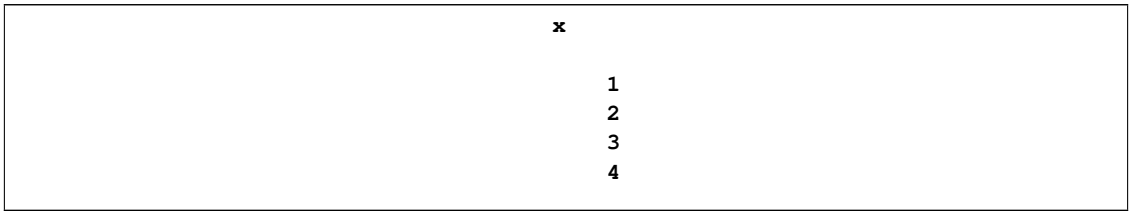
Solving the linear system $Rx = Q'b$ with an upper triangular matrix **R** whose columns are permuted corresponding to the result vector *piv* leads to a solution x with permuted components. You can reorder the components of x by using the index vector *piv* at the left-hand side of an expression, as follows:

```
a = {3  0  0 -1,
     0  1  2  0,
     4 -4 -1  1,
    -1  2  3  4};
b = {-1, 8, -3, 28};

n = ncol(a); p = ncol(b);
ord = j(1, n, 0);
call qr(qtb, r, piv, lindep, a, ord, b);
print piv;

x = j(n,1);
x[piv] = inv(r) * qtb[1:n, 1:p];
print x;
```

Figure 23.239 Solution to a Linear System

piv			
1	4	2	3

Figure 23.239 *continued*



The Full-Rank Linear Least Squares Problem

This example solves the full-rank linear least squares problem. Specify the argument *b* as an $m \times p$ matrix **B**, as follows:

```
call qr(q, r, piv, lindep, a, ord, b);
```

When you specify the *b* argument, the QR subroutine computes the matrix $Q'B$ (instead of Q) as the result *q*. Now you can compute the p least squares solutions x_k of an overdetermined linear system with an $m \times n, m > n$ coefficient matrix A, rank$(A) = n$, and p right-hand sides b_k stored as the columns of the $m \times p$ matrix B:

$$\min_{x_k} \|Ax_k - b_k\|^2, k = 1, \ldots, p$$

where $\|\cdot\|$ is the Euclidean vector norm. This is accomplished by solving the p upper triangular systems with back substitution:

$$x_k = Pi' R^{-1} Q_1' b_k, k = 1, \ldots, p$$

For most applications, the number of rows of **A**, m, is much larger than n, the number of columns of **A**, or p, the number of right-hand sides. In these cases, you are advised not to compute the large $m \times m$ matrix **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 example, use the first five columns of the 6×6 Hilbert matrix **A**, as follows:

```
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 };
aa = a[, 1:5];
b= { 463, -13860, 97020, -258720, 291060, -116424};

m = nrow(aa); n = ncol(aa); p = ncol(b);
call qr(qtb, r, piv, lindep, aa, , b);

if lindep=0 then do;
```

```
   x=inv(r)*qtb[1:n];
   print x;  /* x solves aa*x=b */
end;
else /* handle linear dependence */;
```

Figure 23.240 Solution to Least Squares Problem

x
1
0.5
0.3333333
0.25
0.2

Note that you are using only the first n rows, $\mathbf{Q}_1'\mathbf{B}$, of the **qtb** matrix. The IF-THEN statement of the preceding example can be replaced by the more efficient TRISOLV function:

```
if lindep=0 then
   x = trisolv(1, r, qtb[1:n], piv);
```

For information about solving rank-deficient linear least squares problems, see the RZLIND call.

QUAD Call

CALL QUAD(*r, "fun", points* **<** *, eps* **>** **<** *, peak* **>** **<** *, scale* **>** **<** *, msg* **>** **<** *, cycles* **>** **);**

The QUAD subroutine performs numerical integration of scalar functions in one dimension over infinite, connected semi-infinite, and connected finite intervals.

The QUAD subroutine returns the following value:

r is a numeric vector that contains the results of the integration. The size of *r* is equal to the number of subintervals defined by the argument *points*. If the numerical integration fails on a particular subinterval, the corresponding element of *r* is set to missing.

The input arguments to the QUAD subroutine are as follows:

"fun" specifies the name of a 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 hinders the algorithm.

eps is an optional scalar that specifies the desired relative accuracy. It has a default value of 1E−7. You can specify *eps* by using the EPS= keyword.

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* by using the PEAK= keyword.

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* by using the SCALE= keyword.

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* by using the MSG= keyword.

cycles is an optional integer that indicates the maximum number of refinements the QUAD subroutine can make in order to achieve the required accuracy. It has a default value of 8. You can specify *cycles* by using the CYCLES= keyword.

If the dimensions of any optional argument are 0×0, the QUAD subroutine uses its default value.

The QUAD subroutine is a numerical integrator based on adaptive Romberg-type integration techniques. See Rice (1973), Sikorsky (1982), Sikorsky and Stenger (1984), Stenger (1973a), Stenger (1973b), and Stenger (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 subroutine is an adaptive global-type integrator that produces a quick, rough estimate of the integration result and then refines the estimate until it achieves 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.

A Simple Example

Consider the integral

$$\int_0^{\infty} e^{-t}\, dt$$

The following statements evaluate this integral:

```
/* Define the integrand */
start fun(t);
   v = exp(-t);
   return(v);
finish;

a = {0  .P};
call quad(z, "fun", a);
print z[format=E21.14];
```

Figure 23.241 Result of Numerical Integration on a Semi-Infinite Domain

z
9.99999999595190E-01

The integration is carried out over the interval $(0, \infty)$, as specified by the **a** variable. 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 statements integrate the same exponential function over two subintervals:

```
a   = {0  3 .P };
call quad(z2, "fun", a);
print z2[format=E21.14];
```

Figure 23.242 Result of Numerical Integration on Two Intervals

z2
9.50212930994570E-01
4.97870683477090E-02

Notice 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 **z2** vector, as follows:

```
b = sum(z2);
print b[format=E21.14];
```

Figure 23.243 Result of Numerical Integration on Two Intervals

b
9.99999999342280E-01

Using the PEAK= Option

The *peak* and *scale* options 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:

```
start fun2(t);
   v = exp(-10000*t);
   return(v);
finish;

a = {0  .P};
/* Either syntax can be used                    */
/*   call quad(z, "fun2", a, 1E-10, 0.0001); or */
call quad(z3, "fun2", a) eps=1E-10 peak=0.0001;
print z3[format=E21.14];
```

Figure 23.244 Result of Specifying PEAK= Option

z3
9.99999999998990E-05

The integration is performed over the semi-infinite interval $(0, \infty)$. The default values of SCALE=1 and CYCLES=8 are used. However, the default value of *peak* is 1 for this semi-infinite interval, which is not a good estimate of the location of the function's maximum. 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 nonzero 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 can lead to an increase in the number of function evaluations per step at a linear rate.

Using the SCALE= Option

Consider the integration

$$\int_{-\infty}^{\infty} e^{-100000(t-3)^2}\, dt$$

The integrand is essentially zero except on a small interval close to $t = 3$. The following statements evaluate this integral:

```
/* Define the integrand */
start fun3(t);
    v = exp(-100000*(t-3)*(t-3));
    return(v);
finish;

a = { .M  .P };
call quad(z4, "fun", a) eps=1E-10 peak=3 scale=0.001;
print z4[format=E21.14];
```

The integration is carried out over the infinite interval $(-\infty, \infty)$. The default value of CYCLES=8 has been used. The integrand has its maximum value at $t = 3$, so the PEAK=3 option is specified.

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 increase the number of possible function evaluations; the number of possible function evaluations increases linearly with the reciprocal of *scale*, but it potentially increases in an exponential manner when *cycles* is increased. Increasing the number of function evaluations increases execution time.

Two-Dimensional Integration

When you perform double integration, you must separate the variables between the iterated integrals. There should be a clear distinction between the variable of the one-dimensional integration and the parameters that are passed to the integrand. Another important consideration is specifying the correct limits of integration.

For example, suppose you want to compute probabilities for the standard bivariate normal distribution with correlation ρ. In particular, if an observation (x, y) is drawn from the distribution, what is probability that $x \leq a$ and $y \leq b$ for given values of a and b?

The bivariate normal probability is given by the following double integral:

$$\text{probbnrm}(a, b, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{a} \int_{-\infty}^{b} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}\right) dy\, dx$$

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 a function module with parameters a, b, ρ that computes the bivariate normal probability. In the following statements, the function module is called NORCDF2 because it compute the CDF of the bivariate normal distribution. The NORCDF2 module calls the QUAD subroutine on the MARGINAL module, which computes the outer integral. The MARGINAL module, in turn, uses the QUAD function to evaluate inner integral. The integrand of the inner integral is defined in the NORPDF2 module.

```
/*-----------------------------------------------------*/
/* This function is the density function and requires  */
/* the variable T (passed in the argument)             */
/* and a list of global parameters, YV, RHO, COUNT     */
/*-----------------------------------------------------*/
start norpdf2(t) global(yv,rho,count);
   count = count+1;
   q=(t#t-2#rho#t#yv+yv#yv)/(1-rho#rho);
   p=exp(-q/2);
   return(p);
finish;
```

```
/*-------------------------------------------------*/
/* The outer integral                              */
/* The limits of integration are .M to YY          */
/* YV is passed as a parameter to the inner integral*/
/*-------------------------------------------------*/
start marginal(v) global(yy,yv,eps);
   interval = .M || yy;
   if ( v < -12 ) then return(0);
   yv = v;
   call quad(pm, "NORPDF2", interval) eps=eps;
   return(pm);
finish;

/*-------------------------------------------------*/
/* Global parameters: YY, RHO, EPS                 */
/* EPS is set from IML                             */
/*-------------------------------------------------*/
start norcdf2(a, b, rrho) global(yy,rho,eps);
   rho = rrho; /* copy arguments (local variables) to global list */
   yy = b;

   interval= .M || a; /* upper/lower limits for outer integral */
   call quad(p,"MARGINAL",interval) eps=eps;

   pi = constant("Pi");
   per = p /(2#pi#sqrt(1-rho#rho)); /* scale the value from QUAD */
   return(per);
finish;

/*-------------------------------------------------------*/
/* Main Program: set up global constants and call QUAD */
/*-------------------------------------------------------*/
count = 0;
eps = 1E-11;

p =  norcdf2(2, 1, 0.1);
print p[format=E21.14], count;
```

Figure 23.245 Result of Numerical Integration of a Double Iterated Integral

p
8.23640898880300E-01

count
250453

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, which computes probabilites for the bivariate normal model, as shown by the following statements:

```
/* Compute the value with the PROBBNRM function */
pp = probbnrm(2,1,0.1);
print pp[format=E21.14];
```

Figure 23.246 Result of Numerical Integration of a Double Iterated Integral

pp
8.23640898880500E-01

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 computed to the required relative accuracy. Therefore, either cut off the function (when $v \leq -12$), as in the MARGINAL module in the preceding example, or have the intervals start from a reasonable cutoff value. In addition, the QUAD subroutine 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 subroutine for the inner integration lacks information about the final value that the QUAD subroutine 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 close to 0.8, as in this example. Any computation of the inner execution of the QUAD call that yields 0.8E−11 or less does not contribute to the final answer of the QUAD subroutine 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 cutoffs. Although this method can be inefficient, the final calculations are accurate.

QUEUE Call

CALL QUEUE(*argument1 < , argument2, . . . , argument15 >* **);**

The QUEUE subroutine places character arguments that contain 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. Each argument to the QUEUE subroutine is a character matrix or quoted literal that contains valid SAS statements.

The queued string is read after other lines of input already in 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 paused (see the PAUSE statement), which happens when one of the following occurs:

- An execution error occurs within a module.
- An interrupt is issued.
- A PAUSE statement executes.

The strings you queue do not appear on the log.

CAUTION: Do not queue too many statements at one time. Queuing too many statements can cause problems that can result in exiting the SAS System.

For more examples, see Chapter 18.

An example that uses the QUEUE subroutine follows:

```
start mod(x);
   code="x=0;";
   call queue (code,"resume;");
   pause;
finish;

x=1;
run mod(x);
print x;
```

Figure 23.247 Result of Evaluating Queued Statements

x
0

QUIT Statement

QUIT ;

Use the QUIT statement to exit PROC 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). However, you can use the ABORT statement as an executable 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.

RANCOMB Function

RANCOMB(*n, k* < *, numcomb* > **);**

RANCOMB(*set, k* < *, numcomb* > **);**

The RANCOMB function generates random combinations of *k* elements taken from a set of *n* elements. The random number seed is set by the RANDSEED subroutine.

The first argument, *set*, can be a scalar or a vector. If *set* is a scalar, the function returns indices in the range 1–*n*. If *set* is a vector, the number of elements of the vector determines *n* and the RANCOMB function returns elements of *set*.

By default, the RANCOMB function returns a single random combination with one row and *k* columns. If the *numcomb* argument is specified, the function returns a matrix with *numcomb* rows and *k* columns. Each row of the returned matrix represents a single combination.

The following statements generate five random combinations of two elements from the set $\{1, 2, 3, 4\}$:

```
n = 4;
k = 2;
call randseed(1234);
c = rancomb(n, k, 5);
print c;
```

Figure 23.248 Random Pairwise Combinations of Four Items

```
                c

            1       4
            1       2
            2       4
            2       3
            1       3
```

The function can return combinations for arbitrary numerical or character matrices. For example, the following statements generate five random pairwise combinations of four elements:

```
d = rancomb({A B C D}, 2, 5);
print d;
```

Figure 23.249 Random Pairwise Combinations of Four Characters

```
                d

            A D
            A B
            A D
            B D
            A B
```

RANDGEN Call

CALL RANDGEN(*result, distname* < *, parm1* > < *, parm2* > < *, parm3* > **);**

The RANDGEN subroutine generates random numbers from a specified distribution.

The input arguments to the RANDGEN subroutine are as follows:

result	is a matrix that is to be filled with random samples from the specified distribution.
distname	is the name of the distribution that is to be sampled.
parm1	is a distribution shape parameter.
parm2	is a distribution shape parameter.
parm3	is a distribution shape parameter.

The RANDGEN subroutine generates random numbers by using the same numerical method as the RAND function in Base SAS software, with the efficiency optimized for matrices. You can initialize the random number stream that is used by RANDGEN by calling the RANDSEED subroutine. The *result* parameter should be preallocated to a size equal to the number of samples you want to generate. If *result* is not initialized, then it receives a single random sample.

The following distributions can be sampled.

Bernoulli Distribution

The random sample x is from the probability density function:

$$f(x) = \begin{cases} 1 & \text{for } p = 0, x = 0 \\ p^x(1-p)^{1-x} & \text{for } 0 < p < 1, x = 0, 1 \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

x is in the range: $x = 0, 1$

p is the success probability, with range: $0 \leq p \leq 1$

Beta Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

x is in the range: $0 < x < 1$

a and b are shape parameters, with range: $a > 0$ and $b > 0$

Binomial Distribution

The random sample x is from the probability density function:

$$f(x) = \begin{cases} 1 & \text{for } p = 0, x = 0 \\ nxp^x(1-p)^{1-x} & \text{for } 0 < p < 1, x = 0, \ldots, n \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

x is in the range: $x = 0, 1, \ldots, n$

p is the success probability, with range: $0 \leq p \leq 1$

n specifies the number of independent trials, with range: $n = 1, 2, \ldots$

Cauchy Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{1}{\pi(1+x^2)}$$

x is in the range: $-\infty < x < \infty$

Chi-Square Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{2^{-df/2}}{\Gamma(\frac{df}{2})} x^{df/2-1} e^{-x/2}$$

x is in the range: $x > 0$

df is degrees of freedom, with range: $df > 0$

Erlang Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

x is in the range: $x > 0$

a is an integer shape parameter, with range: $a = 1, 2, \ldots$

Exponential Distribution

The random sample x is from the probability density function:

$$f(x) = e^{-x}$$

x is in the range: $x > 0$

F Distribution ($F_{n,d}$)

The random sample x is from the probability density function:

$$f(x) = \frac{\Gamma(\frac{n+d}{2}) n^{\frac{n}{2}} d^{\frac{d}{2}} x^{\frac{n}{2}-1}}{\Gamma(\frac{n}{2})\Gamma(\frac{d}{2})(d+nx)^{\frac{n+d}{2}}}$$

x is in the range: $x > 0$

n and d are degrees of freedom, with range: $n > 0$ and $d > 0$

Gamma Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{x^{a-1}}{\Gamma(a)} e^{-x}$$

x is in the range: $x > 0$

a is a shape parameter: $a > 0$

Geometric Distribution

The random sample x is from the probability density function:

$$f(x) = \begin{cases} (1-p)^{x-1} p & \text{for } 0 < p < 1, x = 1, 2, \ldots \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

x is in the range: $x = 1, 2, \ldots$

p is the success probability, with range: $0 < p \leq 1$

Hypergeometric Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{RxN - Rn - x}{Nn}$$

x is in the range: $x = \max(0, (n - (N - R))), \ldots, \min(n, R)$

N is the population size, with range: $N = 1, 2, \ldots$

R is the size of the category of interest, with range: $R = 0, 1, \ldots, N$

n is the sample size, with range: $n = 0, 1, \ldots, N$

Lognormal Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{e^{-\ln^2(x)/2}}{x\sqrt{2\pi}}$$

x is in the range: $x \geq 0$

Negative Binomial Distribution

The random sample x is from the probability density function:

$$f(x) = \begin{cases} x + k - 1k - 1(1-p)^x p^k & \text{for } 0 < p < 1, x = 0, 1, \ldots \\ 1 & \text{for } p = 1, x = 0 \end{cases}$$

x is in the range: $x = 0, 1, \ldots$

p is the success probability with range: $0 < p \leq 1$

k is an integer number that counts the number of successes, with range: $k = 1, 2, \ldots$

Normal Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{1}{\lambda\sqrt{2\pi}} \exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

x is in the range: $-\infty < x < \infty$

θ is the mean, with range: $-\infty < \theta < \infty$. This parameter is optional and defaults to 0.

λ is the standard deviation, with range: $\lambda > 0$. This parameter is optional and defaults to 1.

Poisson Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{m^x e^{-m}}{x!}$$

x is in the range: $x = 0, 1, \ldots$

m is the mean, with range $m > 0$

t Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{df\pi}\,\Gamma\left(\frac{df}{2}\right)}\left(1 + \frac{x^2}{df}\right)^{-\frac{df+1}{2}}$$

x is in the range: $-\infty < x < \infty$

df is the degrees of freedom, with the range: $df > 0$

Table Distribution

The random sample i is from the probability density function:

$$f(i) = \begin{cases} p_i & \text{for } i = 1, 2, \ldots, n \\ 1 - \sum_{j=1}^{n} p_j & \text{for } i = n + 1 \end{cases}$$

where p is a vector of probabilities, such that $0 \leq p \leq 1$, and n is the largest integer such that $n \leq$ size of p and

$$\sum_{j=1}^{n} p_j \leq 1$$

Triangle Distribution

The random sample x is from the probability density function:

$$f(x) = \begin{cases} \frac{2x}{h} & \text{for } 0 \leq x \leq h \\ \frac{2(1-x)}{1-h} & \text{for } h < x \leq 1 \end{cases}$$

x is in the range: $0 \leq x \leq 1$

h is the horizontal location of the peak of the triangle, with range: $0 \leq h \leq 1$

Uniform Distribution

The random sample x is from the probability density function:

$$f(x) = 1$$

x is in the range: $0 < x < 1$

Weibull Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{a}{b^a} x^{a-1} e^{-(\frac{x}{b})^a}$$

x is in the range: $x \geq 0$

a and b are shape parameters, with range $a > 0$ and $b > 0$

The following table describes how parameters of the RANDGEN call correspond to the distribution parameters.

Table 23.1 Parameter Assignments for Distributions

Distribution	*distname*	*parm1*	*parm2*	*parm3*
Bernoulli	'BERNOULLI'	p		
Beta	'BETA'	a	b	
Binomial	'BINOMIAL'	p	n	
Cauchy	'CAUCHY'			
Chi-Square	'CHISQUARE'	df		
Erlang	'ERLANG'	a		
Exponential	'EXPONENTIAL'			
$F_{n,d}$	'F'	n	d	

Gamma	'GAMMA'	a		
Geometric	'GEOMETRIC'	p		
Hypergeometric	'HYPERGEOMETRIC'	N	R	n
Lognormal	'LOGNORMAL'			
Negative Binomial	'NEGBINOMIAL'	p	k	
Normal	'NORMAL' or 'GAUSSIAN'	θ	λ	
Poisson	'POISSON'	m		
t	'T'	df		
Table	'TABLE'	p		
Triangle	'TRIANGLE' or 'TRIANGULAR'	h		
Uniform	'UNIFORM'			
Weibull	'WEIBULL'	a	b	

The *distname* argument can be in lowercase or uppercase, and you need to specify only enough letters to distinguish one distribution from the others, as shown by the following statements:

```
/* generate 10 samples from a Bernoulli distribution */
r = j(10, 1, .);         /* allocate room for samples */
call randgen(r, "ber", 0.5);
```

Except for the normal distribution, you must specify the parameters listed for each of the preceding distributions. For the normal distribution, default values of $\theta = 0$ and $\lambda = 1$ are used if no values are supplied.

The following example illustrates the use of the RANDGEN call:

```
call randseed(12345);

/* get four random observations from each distribution */
x = j(1, 4, .);
/* each row of m comes from a different distribution */
m = j(20, 4, .);
call randgen(x, 'BERN', 0.75);        m[ 1, ] = x;
call randgen(x, 'BETA', 3, 0.1);      m[ 2, ] = x;
call randgen(x, 'BINOM', 0.75, 10);   m[ 3, ] = x;
call randgen(x, 'CAUCHY');            m[ 4, ] = x;
call randgen(x, 'CHISQ', 22);         m[ 5, ] = x;
call randgen(x, 'ERLANG',  7);        m[ 6, ] = x;
call randgen(x, 'EXPO');              m[ 7, ] = x;
call randgen(x, 'F', 12, 322);        m[ 8, ] = x;
call randgen(x, 'GAMMA', 7.25);       m[ 9, ] = x;
call randgen(x, 'GEOM', 0.02);        m[10, ] = x;
call randgen(x, 'HYPER', 10, 3, 5);   m[11, ] = x;
call randgen(x, 'LOGN');              m[12, ] = x;
call randgen(x, 'NEGB', 0.8, 5);      m[13, ] = x;
call randgen(x, 'NORMAL');            m[14, ] = x;
call randgen(x, 'POISSON', 6.1);      m[15, ] = x;
call randgen(x, 'T', 4);              m[16, ] = x;
p = {0.1 0.2 0.25 0.1 0.15 0.1 0.1};
call randgen(x, 'TABLE', p);          m[17, ] = x;
```

```
call randgen(x, 'TRIANGLE', 0.7);     m[18, ] = x;
call randgen(x, 'UNIFORM');           m[19, ] = x;
call randgen(x, 'WEIB', 0.25, 2.1);   m[20, ] = x;

dist = {'BERN','BETA','BINOM','CAUCHY','CHISQ','ERLANG','EXPO',
        'F','GAMMA','GEOM','HYPER','LOGN','NEGB','NORMAL',
        'POISSON','T','TABLE','TRIANGLE','UNIFORM','WEIB'};
print m[rowname=dist];
```

Figure 23.250 Random Numbers from Various Distributions

	m			
BERN	1	0	1	0
BETA	1	0.9999234	0.9842784	0.9997739
BINOM	7	8	5	10
CAUCHY	-1.209834	3.9732282	-0.048339	-1.337284
CHISQ	30.300691	20.653151	27.301922	26.878221
ERLANG	10.636299	4.6455449	7.5284821	2.5558646
EXPO	0.2449632	2.7656037	4.2254588	0.2866158
F	0.7035829	1.2676112	0.9806787	1.4811389
GAMMA	8.475216	8.8723256	8.2993617	8.0409742
GEOM	109	4	33	30
HYPER	1	1	2	1
LOGN	0.7784513	0.9792472	0.6018993	0.3643607
NEGB	3	2	0	2
NORMAL	0.0053637	1.4026784	-0.271338	-0.416685
POISSON	5	11	8	4
T	1.3237918	0.0505162	-0.660845	-0.634447
TABLE	2	3	2	3
TRIANGLE	0.5270875	0.6909336	0.8607548	0.5450831
UNIFORM	0.4064393	0.7464901	0.3463207	0.2615394
WEIB	0.4183405	0.9981923	16.812803	0.0001131

RANPERM Function

RANPERM(*n*);

RANPERM(*set*, < , *numperm* >);

The RANPERM function generates random permutations of a set with n elements. The random number seed is set by the RANDSEED subroutine.

The first argument, *set*, can be a scalar or a vector. If *set* is a scalar, the function returns indices in the range 1–n. If *set* is a vector, the number of elements of the vector determines n and the RANPERM function returns elements of *set*, which can be numeric or character.

By default, the RANPERM function returns a single random combination with one row and n columns. If the *numperm* argument is specified, the function returns a matrix with *numperm* rows and n columns. Each row of the returned matrix represents a single permutation.

The following statements generate five random permutations of the set $\{1, 2, 3\}$:

```
call randseed(1234);
n = 3;
p = ranperm(n, 5);
print p;
```

Figure 23.251 Random Permutations of Three Items

p		
1	2	3
3	2	1
3	2	1
3	1	2
3	1	2

Alternatively, the following statements compute five random permutations of an unsorted character vector:

```
a = ranperm({C B A}, 5);
print a;
```

Figure 23.252 Random Permutations of a Character Vector

a		
B	C	A
B	A	C
C	B	A
B	C	A
B	C	A

RANDSEED Call

CALL RANDSEED(*seed* < , *reinit* > **);**

The RANDSEED subroutine sets the initial random seed for the RANDGEN subroutine.

The input arguments to the RANDSEED call are as follows:

seed is a number to be used to initialize the RANDGEN random number generator.

reinit specifies whether the random number stream can be reinitialized after the first initialization, within the same PROC IML session.

The RANDSEED subroutine creates an initial random seed for subsequent RANDGEN calls. If RANDSEED is not called, an initial seed is generated from the system clock. This subroutine is normally used when it is desirable to reproduce the same random number stream in different PROC IML sessions. The optional *reinit* parameter controls whether the seed is reinitialized within the same PROC IML session. If it is set to one, identical seeds produce the same random number sequence; otherwise a second call to RAND-

SEED within the same PROC IML session is ignored. Normally you should not specify *reinit*, or you should set it to zero to ensure that you are working with an independent random number stream within your PROC IML session.

RANGE Function

RANGE(*matrix1* **<** *, matrix2, . . . , matrix15* **>);**

The RANGE function returns the range of values of a numerical matrix or set of matrices.

Missing values are excluded in the computation. When the arguments contain at least one nonmissing value, the range is defined as the maximum value minus the minimum value. If all arguments are missing, the RANGE function returns a missing value.

The following example uses the RANGE function:

```
c = {1 -123 13 56 128 -81 12};
r = range(c);
print r;
```

Figure 23.253 Range of Values

r
251

RANK Function

RANK(*matrix***);**

The RANK function creates a new matrix that contains elements that are the ranks of the corresponding elements of the numerical argument, *matrix*. The rank of a missing value is a missing value. The ranks of tied values are assigned arbitrarily. (See the description of the RANKTIE function for alternate approaches.)

For example, the following statements produce the ranks of a vector:

```
x = {2 2 1 0 5};
r = rank(x);
print r;
```

Figure 23.254 Ranks of a Vector

r				
3	4	2	1	5

Provided that a vector, **x**, does not contain missing values, the RANK function can be used to sort the vector, as shown in the following statements:

```
b = x;
x[,rank(x)] = b;
print x;
```

Figure 23.255 Sorted Vector

x				
0	1	2	2	5

You can also sort a matrix by using the SORT subroutine. The SORT subroutine handles missing values in the data.

The RANK function can also be used to find anti-ranks of **x**, as follows:

```
x = {2 2 1 0 5};
r = rank(x);
a = r;
a[,r] = 1:ncol(x);
print a;
```

Figure 23.256 Anti-Ranks of a Vector

a				
4	3	1	2	5

Although the RANK function ranks only the elements of numerical matrices, you can rank the elements of a character matrix by using the UNIQUE function, as demonstrated by the following statements:

```
/* Create RANK-like functionality for character matrices */
start rankc(x);
   s = unique(x);              /* the unique function returns a sorted list */
   idx = j(nrow(x), ncol(x));
   ctr = 1;                    /* there can be duplicate values in x */
   do i = 1 to ncol(s);        /* for each unique value */
      t = loc(x = s[i]);
      nDups = ncol(t);
      idx[t] = ctr : ctr+nDups-1;
      ctr = ctr + nDups;
   end;
   return (idx);
finish;

/* call the RANKC module */
x = {every good boy does fine and good and well every day};
rc = rankc(x);
print rc[colnam=x];
```

```
/* Notice that ranking is in ASCII order, in which capital
   letters precede lower case letters.  To get case-insensitive
   behavior, transform the matrix before comparison */
x = {"a" "b" "X" "Y" };
asciiOrder = rankc(x);
alphaOrder = rankc(upcase(x));
print x, asciiOrder, alphaOrder;
```

Figure 23.257 Ranks of Character Matrices

```
                                rc
            EVERY      GOOD      BOY      DOES      FINE       AND

ROW1           6         9         3         5         8         1

                                rc
                GOOD       AND      WELL     EVERY       DAY

ROW1              10         2        11         7         4

                                x

                              a b X Y

                           asciiOrder

                        3         4         1         2

                           alphaOrder

                        1         2         3         4
```

There is no SAS/IML function that directly computes the linear algebraic rank of a matrix. In linear algebra, the rank of a matrix is the maximal number of linearly independent columns (or rows). You can use the following technique to compute the numerical rank of matrix **a**:

```
/* Only four linearly independent columns */
A = {1 0 1 0 0,
     1 0 0 1 0,
     1 0 0 0 1,
     0 1 1 0 0,
     0 1 0 1 0,
     0 1 0 0 1 };
rank = round(trace(ginv(a)*a));
print rank;
```

Figure 23.258 Numerical Rank of a Matrix

```
                              rank

                                 4
```

Another common technique used to examine the rank of a matrix is to look at the number of nonzero singular values in the singular value decomposition of a matrix (see the SVD call). However, keep in mind that numerical computations might result in singular values for a rank-deficient matrix that are small but nonzero.

RANKTIE Function

RANKTIE(*matrix* < , *method*> **);**

The RANKTIE function creates a new matrix that contains elements that are the ranks of the corresponding elements of *matrix*. The rank of a missing value is a missing value. The ranks of tied values are computed by using one of several methods.

The arguments to the function are as follows:

matrix specifies the data.

method specifies the method used to compute the ranking of tied values. These methods correspond to those defined by using the TIES= option in the RANK procedure. For details, see the "Concepts" section of the documentation for the RANK procedure in the *Base SAS Procedures Guide*.

The following values are valid:

"Mean" specifies that tied elements are assigned rankings equal to the mean of the tied elements. This is the default method. This method is known as a fractional competition ranking.

"Low" specifies that tied elements are assigned rankings equal to the minimum order rank of the tied elements. This method is known as a standard competition ranking.

"High" specifies that tied elements are assigned rankings equal to the maximum rank of the tied elements. This method is known as a modified competition ranking.

"Dense" specifies that ranks are consecutive integers that begin with 1 and end with the number of unique, nonmissing values. Tied values are assigned the same rank. This method is known as a dense ranking.

The RANKTIE function differs from the RANK function in that the RANK function breaks ties arbitrarily.

For example, the following statements produce ranks of a vector by using several different methods of breaking ties:

```
x = {4 4 0 6};
rMean = ranktie(x);   /* default is "Mean" */
rLow = ranktie(x, "Low");
rHigh = ranktie(x, "High");
rDense = ranktie(x, "Dense");
print rMean, rLow, rHigh, rDense;
```

Figure 23.259 Numerical Ranks of a Vector

	rMean		
2.5	2.5	1	4
	rLow		
2	2	1	4
	rHigh		
3	3	1	4
	rDense		
2	2	1	3

Although the RANKTIE function ranks only the elements of numerical matrices, you can rank the elements of a character matrix by using the UNIQUE function, as demonstrated by the following statements:

```
/* Create RANKTIE-like functionality for character matrices */
start ranktiec(x);
   s = unique(x);
   idx = j(nrow(x), ncol(x));
   ctr = 1;                      /* there can be duplicate values in x */
   do i = 1 to ncol(s);          /* for each unique value */
      t = loc(x = s[i]);
      nDups = ncol(t);
      idx[t] = ctr+(nDups-1)/2; /* =(ctr:ctr+nDups-1)[:] */
      ctr = ctr + nDups;
   end;
   return (idx);
finish;

/* call the RANKTIEC module */
x = {every good boy does fine and good and well every day};
rtc = ranktiec(x);
print rtc[colname=x];
```

Figure 23.260 Numerical Ranks of a Character Vector

	rtc					
	EVERY	GOOD	BOY	DOES	FINE	AND
ROW1	6.5	9.5	3	5	8	1.5

	rtc				
	GOOD	AND	WELL	EVERY	DAY
ROW1	9.5	1.5	11	6.5	4

RATES Function

RATES(*rates, oldfreq, newfreq***);**

The RATES function computes a column vector of (per-period, such as per-year) interest rates converted from one base to another. The arguments to the RATES function are as follows:

rates is an $n \times 1$ column vector of rates that correspond to the old base. Elements should be positive.

oldfreq is a scalar that represents the old base. If positive, it represents discrete compounding as the reciprocal of the number of compoundings per period. If zero, it represents continuous compounding. If –1, the rates represent discount factors. No other negative values are accepted.

newfreq is a scalar that represents the new base. If positive, it represents discrete compounding as the reciprocal of the number of compoundings per period. If zero, it represents continuous compounding. If –1, the rates represent discount factors. No other negative values are accepted.

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 the following ways:

- with per-unit-time-period discount factors d_t:

$$D(t) = d_t^t$$

- with continuous compounding:

$$D(t) = e^{-r_t t}$$

- with discrete compounding:

$$D(t) = (1 + fr)^{-t/f}$$

  where $0 < f < 1$ is the frequency, the reciprocal of the number of compoundings per unit time period. The RATES function converts between these three representations.

For example, the following example produces the output shown in Figure 23.261:

```
rates = T(do(0.1, 0.3, 0.1));
oldfreq = 0;
newfreq = 0;
rates = rates(rates, oldfreq, newfreq);
print rates;
```

Figure 23.261 Interest Rates

```
rates

    0.1
    0.2
    0.3
```

RATIO Function

RATIO(*ar, ma, terms* < *, dim* > **);**

The RATIO function divides matrix polynomials.

The arguments to the RATIO function are as follows:

ar is an $n \times (ns)$ matrix that represents 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 that represents 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 that contains the number of terms to be computed, denoted by r in the following discussion. This value must be positive.

dim is a scalar that contains the value of m, a dimension of the matrix *ma*. 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.

The value returned is an $n \times (mr)$ matrix that contains the terms of $\Phi(B)^{-1}\Theta(B)$ considered as a matrix of rational functions in B that have been expanded as power series.

The RATIO function can be used to consolidate the matrix operators that are used 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 used 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 **x** can be obtained by using the following statement:

```
ratio({ 1 -1}, x, ncol(x));
```

The following example defines polynomial coefficients that are used in a multivariate ARMA(1,1) model and computes the ratio of the polynomials:

```
ar = {1 0 -0.5  2,
      0 1  3   -0.8};
ma = {1 0  0.9  0.7,
      0 1  2   -0.4};
psi = ratio(ar, ma, 4, 2);
print psi;
```

Figure 23.262 The Ratio of Polynomials

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

CALL RDODT(*def, rup, bup, sup, r, z* < , *b* > < , *y* > < , *ssq* > **);**

CALL RUPDT(*rup, bup, sup, r, z* < , *b* > < , *y* > < , *ssq* > **);**

If $\mathbf{A} = \mathbf{QR}$ is the QR decomposition of the matrix $\mathbf{A}$, the RUPDT subroutine enables you to efficiently recompute the $\mathbf{R}$ matrix when a new row is added to A. This is called an update. Similarly, the RDODT subroutine enables you to efficiently recompute the $\mathbf{R}$ matrix when an existing row is deleted from A. This is called a downdate. You can also use the RDODT and RUPDT subroutines to downdate and update Cholesky decompositions.

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 input arguments to the RDODT and RUPDT subroutines are as follows:

r	specifies an $n \times n$ upper triangular matrix **R** to be updated or downdated by the q rows in **Z**. Only the upper triangle of **R** is used; the lower triangle can contain any information.
z	specifies a $q \times n$ matrix **Z** used rowwise to update or downdate the matrix **R**.
b	specifies an optional $n \times p$ matrix **B** of right-hand sides that have to be updated or downdated simultaneously with **R**. If *b* is specified, the argument *y* must also be specified.
y	specifies an optional $q \times p$ matrix **Y** used rowwise to update or downdate the right-hand side matrix **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 **R** and **B**.

The upper triangular matrix **R** of the QR decomposition of an $m \times n$ matrix **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 *z* is added to matrix **A**.
- *downdate*: An n vector *z* is deleted from matrix **A**.

Computing the whole QR decomposition of matrix **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 *z* requires only $2n^2$ floating-point operations.

If the QR decomposition is used to solve the full-rank linear least squares problem

$$\min_x \|\mathbf{A}x - b\|^2 = \text{ssq}$$

by solving the nonsingular upper triangular system

$$x = \mathbf{R}^{-1}\mathbf{Q}'b$$

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 *z* added to or deleted from **A** there is also a p vector *y* added to or deleted from the $m \times p$ right-hand-side matrix **B**.

If the arguments *z* and *y* of the subroutines RUPDT and RDODT contain $q > 1$ row vectors for which **R** (and $\mathbf{Q}'\mathbf{B}$, and eventually *ssq*) is to be updated or downdated, the process is performed stepwise by processing the rows z_k (and y_k), $k = 1, \ldots, q$, in the order in which they are stored.

The QR decomposition of an $m \times n$ matrix **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 z to matrix $\mathbf{A}$ corresponds to the rank-1 modification of the crossproduct matrix $\mathbf{C}$

$$\widetilde{\mathbf{C}} = \mathbf{C} + z'z, \text{ where } \widetilde{\mathbf{C}} = \widetilde{\mathbf{A}}'\widetilde{\mathbf{A}}$$

and the $(m+1) \times n$ matrix $\widetilde{A}$ contains all rows of $\mathbf{A}$ with the row z added.

Deleting a row vector z from matrix $\mathbf{A}$ corresponds to the rank-1 modification

$$\mathbf{C}^* = \mathbf{C} - z'z, \text{ where } \mathbf{C}^* = \mathbf{A}^{*\prime}\mathbf{A}^*$$

and the $(m-1) \times n$ matrix $\mathbf{A}^*$ contains all rows of $\mathbf{A}$ with the row 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 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 b_k (stored columnwise in the $m \times p$ matrix $\mathbf{B}$).

The methods in this section use the following simple example:

```
a = { 1 3 ,
      2 2 ,
      3 1 };
b = { 1, 1, 1};
m = nrow(a);
n = ncol(a);
p = ncol(b);
```

- Cholesky decomposition of crossproduct matrix:

```
/* form and solve the normal equations */
aa = a` * a; ab = a` * b;
r  = root(aa);
x  = trisolv(2,r,ab);
x  = trisolv(1,r,x);
print 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 **a**, each method obtains the unique LS estimator:

```
ss = ssq(a * x - b);
print ss x;
```

Figure 23.263 Least Squares Solution and Sum of Squared Residuals

```
                                ss          x

                        2.465E-31       0.25
                                        0.25
```

To compute the (transposed) matrix **Q**, you can use the following technique:

```
r = repeat(0,n,n);
y = i(m);
qt = repeat(0,n,m);
call rupdt(rup, qtup, sup, r, a, qt, y);
print qtup;
```

Figure 23.264 Transposed Matrix

qtup
0.2672612 0.5345225 0.8017837
−0.872872 −0.218218 0.4364358

READ Statement

READ < *range* > < **VAR** *operand* > < **WHERE**(*expression*) > < **INTO** *name* < [**ROWNAME**=*row-name* **COLNAME**=*column-name*] > > **;**

The READ statement reads observations from the current SAS data set.

The arguments to the READ statement 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 that contains descriptive row labels.
column-name	is a character matrix or quoted literal that contains descriptive column labels.

The clauses and options are explained in the following lists.

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 use any of the following keywords to specify a *range* of observations:

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 *value* observations specified by number, where *value* can be one of the following.

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

If the current data set has an index in use (see the INDEX statement, 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 that contains variable names
- the name of a matrix that contains variable names
- an expression in parentheses that yields variable names
- one of keywords described in the following list:

ALL for all variables

CHAR for all character variables

NUM for all numeric variables.

The following examples demonstrate each possible way you can use the VAR clause.

```
var {x1 x5 x9};          /* a literal matrix of names         */
var x;                   /* a matrix that contains the names  */
var("x1":"x9");          /* 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 as follows:

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

The arguments to the WHERE clause are as follows:

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 like 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 by 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 by 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 and column 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 vectors with the same names as the data set variables. Here is the statement:

```
read all var{name age};
```

To read all variables for the 23rd observation only, use the following statement:

```
read point 23;
```

To read the data set variables NAME and ADDR for all observations with a STATE value of 'NJ', use the following statement:

```
read all var{name addr} where(state="NJ");
```

See Chapter 7 for further information.

REMOVE Function

REMOVE(*matrix, indices***);**

The REMOVE function discards elements from a matrix. The arguments to the REMOVE function are as follows:

matrix	is a numeric or character matrix or literal.
indices	specifies the indices of elements of *matrix* to remove.

The REMOVE function returns (as a row vector) a subset of the elements of the first argument. Elements that correspond to indices in the second argument are removed. The elements of the first argument are enumerated in row-major order, and the indices must be in the range 1 to np, where *matrix* is an $n \times p$ matrix. Nonintegral indices are truncated to their integer part. You can repeat the indices and give them in any order. If all elements are removed, the result is a null matrix (zero rows and zero columns).

The following statement removes the third element, which creates a row vector with three elements:

```
x = {5 6, 7 8};
a = remove(x, 3);       /* remove element 3 */
print a;
```

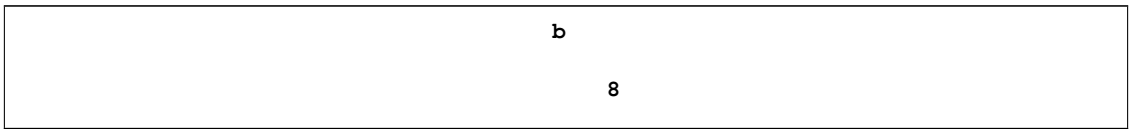
Figure 23.265 Result of Removing an Element

a		
5	6	8

The following statement causes all but the fourth element to be removed:

```
r = {3 2 3 1};
b = remove(x, r);       /* equivalent to removing elements 1:3 */
print b;
```

Figure 23.266 Result of Removing Several Elements



```
b

8
```

The output shown in Figure 23.266 shows that repeated indices are ignored.

REMOVE Statement

REMOVE < **MODULE**=*(module-list)* > < *matrix-list* > **;**

The REMOVE statement removes modules and matrices from storage.

The arguments 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 following statement 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 17, "Storage Features," and also the descriptions of the LOAD, STORE, RESET, and SHOW statements for related information.

RENAME Call

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

The RENAME subroutine renames a SAS data set.

The arguments to the RENAME subroutine are as follows:

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

member-name is a character matrix or quoted literal that contains the current name of the data set.

new-name is a character matrix or quoted literal that contains 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-level 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

REPEAT(*matrix, nrow, ncol***);**

The REPEAT function creates a matrix of repeated values.

The arguments to the REPEAT function are as follows:

matrix is a numeric matrix or literal.

nrow gives the number of times *matrix* is repeated down 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 **Y**, repeating the **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

REPLACE < *range* > < **VAR** *operand* > < **WHERE** *(expression)* > **;**

The REPLACE statement replaces values of observations in a SAS data set.

The arguments 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 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 following statements 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 use any of the following keywords to specify a *range* of observations:

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 *value*	observations by number, where *value* can be one of the following:

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

If the current data set has an index in use (see the INDEX statement, 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 that contains variable names
- the name of a matrix that contains variable names
- an expression in parentheses that yields variable names

- one of the keywords described in the following list:

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

The following examples demonstrate each possible way you can use the VAR clause.

```
var {x1 x5 x9};          /* a literal matrix of names         */
var x;                   /* a matrix that contains the names  */
var("x1":"x9");          /* 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)* **;**

The arguments to the WHERE clause are as follows:

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 like 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 by 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 by 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 following statement replaces all variables in the current observation:

```
replace;
```

RESET Statement

RESET < *options* > **;**

The RESET statement sets processing options. The options are described in the following list. 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 that contains the value, or an expression in parentheses. The SHOW OPTIONS statement displays the current settings of 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. This resets the global CENTER/NOCENTER option for the SAS session.

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 is 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. This resets the global LINESIZE option for the SAS session.

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 the SAS windowing environment and to the listing file in batch mode. 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 inherited from the SAS environment. Changing the

PAGESIZE=
option also changes the global PAGESIZE option.

PRINT

NOPRINT
specifies whether the final results from assignment statements are printed automatically. NOPRINT is the default.

PRINTADV=*n*
inserts blank lines into the log before printing out the value of a matrix. The default, PRINTADV=2, causes two blank lines to be inserted.

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 WORK.IMLSTOR. The *libname* argument is optional and defaults to Sasuser. It can be specified with or without quotes.

RESUME Statement

RESUME ;

The RESUME statement enables you to continue execution from the line in a 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

RETURN < *(operand)* > **;**

The RETURN statement causes a program to return to a previous calling point.

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. The RETURN statement without an argument is used to return from a user-defined subroutine.

You can also use the RETURN statement in conjunction with a LINK statement. If a LINK statement has been issued, the RETURN statement returns control to the statement that follows the LINK statement. See the description of the LINK statement. Also, see Chapter 6 for details.

If a RETURN statement is encountered outside of a module, execution is stopped as with a STOP statement.

The following examples use the RETURN statement to exit from modules:

```
start sum1(a, b);
   sum = a+b;
   return(sum);
finish;
start sum2(s, a, b);
   s = a+b;
   return;
finish;

x = sum1(2, 3);
run sum2(y, 4, 5);
print x y;
```

Figure 23.267 Return from Module Calls

```
                                    x          y

                                    5          9
```

ROOT Function

ROOT(*matrix***);**

The ROOT function performs the Cholesky decomposition of a symmetric and positive definite matrix, **A**. The Cholesky decomposition factors **A** into the product

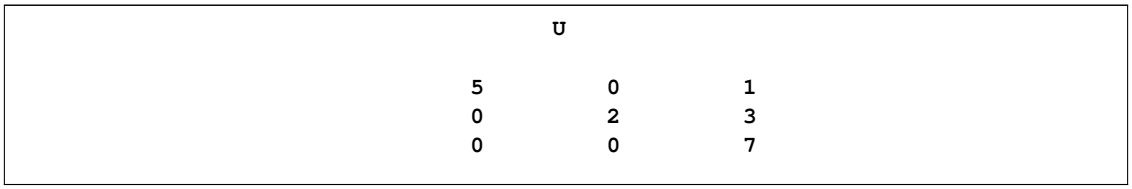
$$\mathbf{A} = \mathbf{U}'\mathbf{U}$$

where **U** is upper triangular.

For example, the following statements compute the upper-triangular matrix, `U`, in the Cholesky decomposition of a matrix:

```
A = {25  0  5,
       0  4  6,
       5  6 59};
U = root(A);
print U;
```

Figure 23.268 Cholesky Decomposition

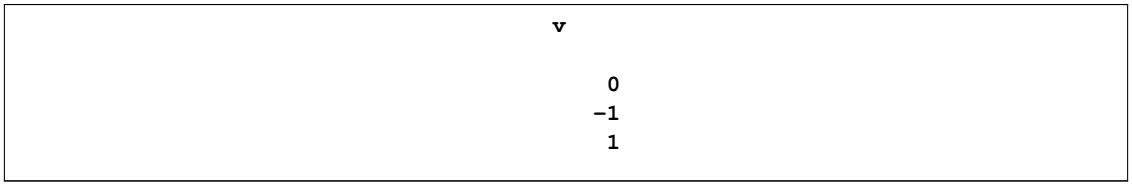


U		
5	0	1
0	2	3
0	0	7

If you need to solve a linear system and you already have a Cholesky decomposition of your matrix, then use the TRISOLV function as illustrated by the following statements:

```
b = {5, 2, 53};
/* Want to solve A * v = b.
   First solve U` z = b,
   then solve U v = z */
z = trisolv(2, U, b);
v = trisolv(1, U, z);
print v;
```

Figure 23.269 Solution to a Linear System



v
0
−1
1

The ROOT function performs most of its computations in the memory allocated for returning the Cholesky decomposition.

ROWCAT Function

ROWCAT(*matrix* < , *rows* > < , *columns* > **);**

The ROWCAT function concatenates rows of a character matrix without using blank compression. In particular, the 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.

The arguments 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*.

If the input matrix has *n* rows and *m* columns, the result will have *n* rows and 1 column. The element length of the result is *m* times the element length of the argument. The optional rows and columns arguments can be used to select which rows and columns are concatenated.

For example, the following statements produce the 2×1 matrix shown:

```
b = {"ABC"  "D  "  "EF ",
     " GH"  " I "  " JK"};
a = rowcat(b);
```

```
   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 syntax

ROWCAT(*matrix, rows, columns***);**

returns the same result as

ROWCAT(*matrix[rows, columns]***);**

The syntax

ROWCAT(*matrix, rows***);**

returns the same result as

ROWCAT(*matrix[rows,]***);**

ROWCATC Function

ROWCATC(*matrix* < , *rows* > < , *columns* > **);**

The ROWCATC function concatenates rows of a character matrix by using blank compression.

The arguments 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 following statements produce the matrix **A** as shown:

```
b={"ABC"  "D  "  "EF ",
   " GH"  " I "  " JK"};
a=rowcatc(b);
```

```
   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

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

The RUN statement executes a user-defined module or invokes PROC IML's built-in subroutines.

The arguments to the RUN statement are as follows:

name	is the name of a user-defined module or a built-in subroutine.
arguments	are arguments to the subroutine. Arguments can be both local and global.

The resolution order for the RUN statement is

1. A user-defined module
2. A built-in function or subroutine

This resolution order need only be considered if you have defined a module that has the same name as a 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, an attempt is made 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 6 for further details.

RUPDT Call

CALL RUPDT(*rup, bup, sup, r, z* **<** *, b* **>** **<** *, y* **>** **<** *, ssq* **>** **);**

See the entry for the RDODT subroutine for details.

RZLIND Call

CALL RZLIND(*lindep, rup, bup, r* **<** *, sing* **>** **<** *, b* **>** **);**

The RZLIND subroutine computes rank-deficient linear least squares solutions, complete orthogonal factorizations, and Moore-Penrose inverses.

The RZLIND subroutine returns the following values:

lindep is a scalar that contains the number of linear dependencies that are recognized in **R** (number of zeroed rows in `rup[n,n]`).

rup is the updated $n \times n$ upper triangular matrix **R** that contains zero rows corresponding to zero recognized diagonal elements in the original **R**.

bup is the $n \times p$ matrix **B** of right-hand sides that is updated simultaneously with **R**. If *b* is not specified, *bup* is not accessible.

The input arguments to the RZLIND subroutine are as follows:

r specifies the $n \times n$ upper triangular matrix **R**. Only the upper triangle of *r* is used; the lower triangle can contain any information.

sing is an optional scalar that specifies a relative singularity criterion for the diagonal elements of **R**. The diagonal element r_{ii} is considered zero if $r_{ii} \leq sing\|r_i\|$, where $\|r_i\|$ is the Euclidean norm of column r_i of **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 **B** of right-hand sides that have to be updated or downdated simultaneously with **R**.

The singularity test used in the RZLIND subroutine is a relative test that uses the Euclidean norms of the columns r_i of **R**. The diagonal element r_{ii} is considered as nearly zero (and the ith row is zeroed out) if the following test is true:

$$r_{ii} \leq \text{sing}\|r_i\|, \quad \text{where } \|r_i\| = \sqrt{r_i' r_i}$$

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 **R** is computed by the QR decomposition $\mathbf{A} = \mathbf{QR}$, then the Euclidean norm of column i of **R** is the same (except for rounding errors) as the Euclidean norm of column i of **A**.

Consider the following possible application of the RZLIND subroutine. Assume that you want to compute the upper triangular Cholesky factor **R** of the $n \times n$ positive semidefinite matrix $\mathbf{A}'\mathbf{A}$,

$$A'A = R'R \text{ where } A \in \mathcal{R}^{m \times n},\ \text{rank}(A) = r,\ r \leq n \leq m$$

The Cholesky factor **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, **A** is a 12×8 matrix with linearly dependent columns $a_1 = a_2 + a_3 + a_4$ and $a_1 = a_5 + a_6 + a_7$ with $r = 6$, $n = 8$, and $m = 12$.

```
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 generates an upper triangular matrix $\mathbf{R}_1$ where linearly dependent rows are zeroed out. You can verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_1'\mathbf{R}_1$. Here is the code:

```
r1 = root(aa);
ss1 = ssq(aa - r1` * r1);
print ss1 r1 [format=best6.];
```

Applying the QR subroutine with column pivoting on the original matrix **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 $\mathbf{A}'\mathbf{A}$. Here is the code:

```
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 **R** by the m rows of **A** finally results 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 generates 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. Here is the code:

```
call rzlind(lind,r4,bup,r3);
ss4 = ssq(aa - r4` * r4);
print ss4 r4 [format=best6.];
```

Consider the rank-deficient linear least squares problem:

$$\min_x \|Ax - b\|^2 \text{ where } \mathbf{A} \in \mathcal{R}^{m \times n},\ \text{rank}(\mathbf{A}) = r,\ r \leq n \leq m$$

For $r = n$, the optimal solution, $\hat{x}$, 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:

$$\text{ss} = (\mathbf{A}\hat{x} - b)'(\mathbf{A}\hat{x} - b)$$

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. The following example uses the 12×8 matrix from the preceding section and generates a new column vector b. The vector 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 $\mathbf{A} = \mathbf{UDV}'$. 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 $\hat{x}_1$ obtained by singular value decomposition, $\hat{x}_1 = \mathbf{V}\mathbf{D}^{-}\mathbf{U}'b/4$, 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 $\hat{x}_2$:

$$\mathbf{R}^{-} = \begin{bmatrix} \mathbf{R}_1^{-1} \\ \mathbf{0} \end{bmatrix} \hat{x}_2 = \mathbf{\Pi}\mathbf{R}^{-}\mathbf{Y}'b$$

```
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 $\hat{x}_2$ is not of minimum Euclidean length.

- Use the result $\mathbf{R}_1$ of the ROOT function on page 936 to obtain the vector *piv* which indicates 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 $\hat{x}_3$ by solving the equation $\hat{x}_3 = \mathbf{R}^{-1}\mathbf{R}^{-\prime}\mathbf{A}'b$.

```
r  = r1[piv[1:nr],piv[: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 $\hat{x}_3$ is not of minimum Euclidean length.

- Use the result $\mathbf{R}_3$ of the RUPDT call on page 940 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 $\hat{x}_4 = \mathbf{R}^{-1}\mathbf{Y}'b$.

```
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 $\hat{x}_4$ is the same as $\hat{x}_3$.

- Use the result $\mathbf{R}_4$ of the RZLIND subroutine 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 $\hat{x}_5$ 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 **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 **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 $\hat{x}_6$ 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 $\hat{x}_7$ 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 $\hat{x}_8$ 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 orthogonal methods to compute the Moore-Penrose inverse $\mathbf{A}^-$ of a rectangular matrix $\mathbf{A}$. The entries in the following list find the Moore-Penrose inverse of the matrix $\mathbf{A}$ shown on page 942.

- Use the GINV operator. The GINV operator 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

$$\mathbf{A} = \mathbf{Y}\begin{bmatrix} \bar{\mathbf{R}}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\bar{\mathbf{Y}}'\mathbf{\Pi}'$$

where $\bar{\mathbf{R}}'$ is lower triangular, yields the Moore-Penrose inverse

$$\mathbf{A}^{-} = \mathbf{\Pi}\bar{\mathbf{Y}}\begin{bmatrix} \bar{\mathbf{R}}^{-\prime} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\mathbf{Y}'$$

```
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

SAVE ;

The SAVE statement saves data to a SAS data set.

The SAVE statement flushes 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

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 SEQ, SEQSCALE, and SEQSHIFT subroutines perform discrete sequential tests.

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 input arguments to the SEQSHIFT subroutine are as follows:

domain specifies an $m \times n$ matrix that contains 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. The values `.M` and `.P` represent $-\infty$ 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 are 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 are 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 are internally set to 1. The 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 is internally set to 1E−7. The 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 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 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 that contains 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 jth 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 subroutine 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 computed 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 $g^{\mu}_{ij} = g_{ij} - \mu j$.

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 statements computes the probability from the sequential test at each boundary point specified in the geometry.

```
/* 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 statements verify this by using the *tscale* parameter to compute this problem.

```
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);
   /* You 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 output shows the values returned for the variables **m**, **prob1**, **mm** and **prob2**.

Some internal limitations are imposed on the geometry. Consider the three-level case with geometry m in the preceding statements. Since the *tscale* variable is not specified, it is set to its default value, $(1, 1)$. The variance at the jth 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 statements verify for the given domain that the significance level is 0.05 and that the power is $1 - \beta$ under the alternative hypothesis:

```
/*****************************************/
/* 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:

```
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

$$-\Gamma\sqrt{j} \le C_j \le \Gamma\sqrt{j}$$

for five groups. The overall significance level is α (the probability $palpha = 1-\alpha$), and the power is $1-\beta$.

```
%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).

```
          SUMMARY

0.99359    1.31083    1.59229    1.75953    2.07153
```

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

$$-\Gamma j^{\delta} \le C_j \le \Gamma j^{\delta}$$

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):

```
%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 about the NLPDD subroutine, see the section "NLPDD Call" on page 826. For details about the *opt*, *tc*, and *par* arguments in the NLPDD call, see the section "Options Vector" on page 356, the section "Termination Criteria" on page 360, and the section "Control Parameters Vector" on page 367, 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 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 program does the computations for a power of 0.9.

```
%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     */
/* you 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 can choose to run the 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. Larger numbers produce more output.

SEQSCALE Call

CALL SEQSCALE(*prob, gscale, domain, level* **<,** **IGUESS=***iguess***>** **<,** **TSCALE=***tscale***>** **<,** **EPS=***eps***>** **<,** **DEN=***den***>** **);**

The SEQSCALE subroutine computes estimates of scales associated with discrete sequential tests.

See the entry for the SEQ subroutine for details.

SEQSHIFT Call

CALL SEQSHIFT(*prob, shift, domain, plevel* **<,** **IGUESS=***iguess***>** **<,** **TSCALE=***tscale***>** **<,** **EPS=***eps***>** **<,** **DEN=***den***>** **);**

The SEQSHIFT subroutine computes estimates of means associated with discrete sequential tests.

See the entry for the SEQ subroutine for details.

SETDIF Function

SETDIF(*matrix1, matrix2***);**

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 arguments 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 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 following statements produce the matrix **C**, as shown:

```
a = {1 2 4 5};
b = {3 4};
c = setdif(a,b);
```

```
              C             1 row       3 cols     (numeric)

                              1          2          5
```

SETIN Statement

SETIN *SAS-data-set* <**NOBS** *name*> <**POINT** *value*> **;**

The SETIN data set makes a data set the current input data set.

The arguments to the SETIN statement are as follows:

SAS-data-set can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). For more information about 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.

value 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

SETOUT *SAS-data-set* <**NOBS** *name*> <**POINT** *value*> **;**

The SETOUT data set makes a data set the current output data set.

The arguments to the SETOUT statement are as follows:

SAS-data-set can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). For more information about 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.

value 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

SHAPE(*matrix, nrow* <*, ncol*> <*, pad-value*> **);**

The SHAPE function reshapes and repeats values in a matrix. The arguments are as follows:

matrix is a numeric or character matrix or literal.

nrow specifies the number of rows fo the new matrix.

ncol specifies the number of columns for the new matrix.

pad-value specifies a value to use for elements of the new matrix if the quantity *nrow*×*ncol* is greater than the number of elements in *matrix*.

The SHAPE function creates a new matrix from data in *matrix*. The values *nrow* and *ncol* specify the number of rows and columns, respectively, in the new matrix. The function can reshape both numeric and character matrices.

There are three ways of using the function:

- 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 following statements create constant matrices of a given size:

```
r = shape(12, 3, 4);        /* 3 x 4 matrix with constant value 12 */
s = shape({99 31}, 3, 3);   /* 3 x 3 matrix with alternating values */
print r, s;
```

Figure 23.270 Constant and Repeated Matrices

```
                              r

          12          12          12          12
          12          12          12          12
          12          12          12          12

                              s

                99          31          99
                31          99          31
                99          31          99
```

Notice that the SHAPE function produces the result matrix by traversing the argument matrix in row-major order until the specified number of elements is reached. If necessary, the SHAPE function reuses elements.

You can also use the SHAPE function to reshape an existing matrix, as shown in the following statements:

```
t = shape(1:6, 2);
print t;
```

Figure 23.271 Reshaped Matrix

t		
1	2	3
4	5	6

SHAPECOL Function

SHAPECOL(*matrix, nrow* < , *ncol* > < , *pad-value* > **);**

The SHAPECOL function reshapes and repeats values in a matrix. It is similar to the SHAPE function except that the SHAPECOL function produces the result matrix by traversing the argument matrix in column-major order.

```
A = {1 2 3, 4 5 6};
c = shapecol(A, 3);
v = shapecol(A, 0, 1);
print c v;
```

Figure 23.272 Reshaped Matrices

c		v
1	5	1
4	3	4
2	6	2
		5
		3
		6

The vector **v** in the example is called the "vec of **A**" and is written vec(**A**). Uses of the vec operator in matrix algebra are described in Harville (1997). One important property is the relationship between the vec operator and the Kronecker product. If **A**, **B**, and **X** have the appropriate dimensions, then

$$\text{vec}(\mathbf{AXB}) = (\mathbf{B}' \otimes \mathbf{A})\text{vec}(\mathbf{X})$$

There is also a relationship between the SHAPECOL function and the SHAPE function. If **A** is a matrix, then the following two computations are equivalent:

```
b = shapecol(A, m, n, padVal);
c = T(shape(A`, n, m, padVal));
```

See the VECH function for a similar function that is useful for computing with symmetric matrices.

SHOW Statement

SHOW *operands* **;**

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 PROC 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 PROC 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 PROC IML library storage.
WINDOWS	shows all active windows opened by WINDOW statements.

An example of a valid statement follows:

```
show all;
```

SOLVE Function

SOLVE(*A, B***);**

The SOLVE function solves a system of linear equations.

The arguments 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 $\mathbf{X}$. The matrix $\mathbf{A}$ must be square and nonsingular.

The expression **`X = SOLVE(A,B)`** is mathematically equivalent to using the INV function in the expression **`X = INV(A)*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, Malcom, and Moler (1967).

The SOLVE function uses a criterion to determine whether the input matrix is singular. See the INV function for details.

If $\mathbf{A}$ is an $n \times n$ matrix, the SOLVE function temporarily allocates an n^2 array in addition to the memory allocated for the return matrix.

SOLVELIN Call

CALL SOLVELIN(*x, status, A, b, method***);**

The SOLVELIN subroutine uses direct decomposition to solve sparse symmetric linear systems.

The SOLVELIN subroutine returns the following values:

x	is the solution to $Ax = b$.
status	is the final status of the solution.

The input arguments to the SOLVELIN subroutine are as follows:

A	is the sparse coefficient matrix in the equation $Ax = b$. You can use SPARSE function to convert a matrix from dense to sparse storage.
b	is the right side of the equation $Ax = b$.
method	is the name of the decomposition to be used.

The input matrix A represents the coefficient matrix in sparse format; it is an n by 3 matrix, where n is the number of nonzero elements. The first column contains the nonzero values, while the second and third columns contain the row and column locations for the nonzero elements, respectively. Since A is assumed to be symmetric, only the elements on and below the diagonal should be specified, and it is an error to specify elements above the diagonal.

The solution to the system is returned in x. Your program should also check the returned $status$ to make sure that a solution was found.

status = 0	indicates success.
status = 1	indicates the matrix A is not positive-definite.

status = 2 indicates the system ran out of memory.

If the SOLVELIN subroutine is unable to solve your system, you can try the iterative method subroutine ITSOLVER.

Two different factorization methods are available from the call, Cholesky and Symbolic LDL, specified as 'CHOL' or 'LDL' with the *method* parameter. Both these factorizations are applicable only to positive-definite symmetric systems; if your system is not positive-definite or not symmetric, you can use an ITSOLVER call.

The following example uses SOLVELIN to solve the system:

$$\begin{bmatrix} 3 & 1.1 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & 10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

```
/* value      row column */
A = {  3        1      1,
      1.1       2      1,
       4        2      2,
       1        3      2,
       3.2      4      2,
      10        3      3,
       3        4      4};

/* right hand side */
b = {1, 1, 1, 1};

call solvelin(x, status, A, b, 'LDL');
print status x;
```

The results are as follows:

```
                STATUS          X

                     0       2.68
                            -6.4
                             0.74
                             7.16
```

SORT Call

CALL SORT(*matrix* < , *by*> < , *descend*> **);**

The SORT subroutine sorts a matrix by the values of one or more columns. The arguments to the SORT subroutine are as follows:

matrix is the input matrix. It is sorted in place by the call. If you want to preserve the original order of the data, make a copy of *matrix*.

by specifies the columns used to sort the matrix. The argument *by* is either a numeric matrix that contains column numbers, or a character matrix that contains the names of columns assigned to *matrix* by a MATTRIB statement or READ statement. If *by* is not specified, then the first column is used.

descend specifies which columns, if any, should be sorted in descending order. Any *by* columns not specified as descending will be ascending. If *descend* = *by*, then all *by* columns will be descending; if *descend* is skipped or is a null matrix, then all *by* columns will be ascending.

The SORT subroutine is used to sort a matrix according to the values in the columns specified by the *by* and *descend* arguments. Because the sort is done in place, very little additional memory space is required. The SORT subroutine is not as fast as the SORTNDX call for matrices with a large number of rows. After a matrix has been sorted, the unique combinations of values in the *by* columns can be obtained from the UNIQUEBY function.

For example, the following statements sort a matrix:

```
m = { 1 1 0,
      2 2 0,
      1 1 1,
      2 2 2};
call sort(m, {1 3}, 3); /* ascending by col 1; descending by col 3 */
print m;
```

Figure 23.273 Sorted Matrix

m		
1	1	1
1	1	0
2	2	2
2	2	0

SORT Statement

SORT <**DATA**=*SAS-data-set*> <**OUT**=*SAS-data-set*> **BY** <**DESCENDING** *variables*> **;**

The SORT statement sorts a SAS data set. 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-level name (for example, A) or a two-level name (for example, Sasuser.A). For more information about 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. In contrast with other 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 always retains the same relative positions of the observations with identical BY variable values.

For example, the following statement 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:

```
sort class out=sclass by descending age height;
```

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 in the preceding list.

SORTNDX Call

CALL SORTNDX(*index, matrix* < *, by*> < *, descend*> **);**

The SORTNDX subroutine creates an index to reorder a matrix by specified columns.

The SORTNDX subroutine returns the following value:

index is a vector such that *index*[*i*] is the row index of the *i*th element of *matrix* when sorted according to *by* and *descend*. Consequently, *matrix*[*index*,] is the sorted matrix.

The arguments to the SORTNDX subroutine are as follows:

matrix is the input matrix, which is not modified by the call.

by specifies the columns used to sort the matrix. The argument *by* is either a numeric matrix that contains column numbers, or a character matrix that contains the names of columns assigned to *matrix* by a MATTRIB statement or READ statement. If *by* is not specified, then the first column is used.

descend specifies which columns, if any, should be sorted in descending order. Any *by* columns not specified as descending will be ascending. If *descend* = *by*, then all *by* columns will be descending; if *descend* is skipped or is a null matrix, then all *by* columns will be ascending.

The SORTNDX subroutine can be used to process the rows of a matrix in a sorted order, without having to actually modify the matrix.

For example, the following statements return a vector that gives the order of the rows in a matrix:

```
m = { 1 1 0,
      2 0 0,
      1 3 1,
      2 2 2 };
call sortndx(ndx, m, {1 3}, 3);
print ndx;
```

Figure 23.274 Sort Index

ndx
3
1
4
2

The output is shown in Figure 23.274. The SORTNDX subroutine returns the vector **ndx** that indicates how rows of **m** will appear if you sort **m** in ascending order by column 1 and in descending order by column 3. The values of the vector **ndx** indicate that row 3 of **m** will be the first row in the sorted matrix. Row 1 of **m** will become the second row. Row 4 will become the third row, and row 2 will become the last row.

The matrix can be physically sorted with the SORT call), as follows:

```
sorted = m[ndx,];
```

The SORTNDX subroutine can be used with the UNIQUEBY function to extract the unique combinations of values in the *by* columns.

SOUND Call

CALL SOUND(*freq* < , *dur*> **);**

The SOUND subroutine generates a tone with a frequency (in hertz) given by the *freq* parameter and a duration (in seconds) given by the *dur* parameter.

The arguments to the SOUND subroutine are as follows:

freq is a numeric matrix or literal that contains the frequency in hertz.

dur is a numeric matrix or literal that contains the duration in seconds. Note that the *dur* argument differs from that in the DATA step.

Matrices can 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);
```

SPARSE Function

SPARSE(*x* <, *type*> **);**

The SPARSE function converts an $n \times p$ matrix that contains many zeros into a matrix stored in a sparse format which suitable for use with the ITSOLVER call or the SOLVELIN call.

The arguments are as follows:

x specifies an $n \times p$ numerical matrix. Typically, *x* contains many zeros and only k nonzeros, where k is much smaller than np.

type specifies whether the *x* matrix is symmetric. The following values are valid:

- `"symmetric"` specifies that only the lower triangular nonzero values of the *x* matrix are used.
- `"unsymmetric"` specifies that all nonzero values of the *x* matrix are used. This is the default value.

The *type* argument is not case-sensitive. The first three characters are used to determine the value. For example, `"SYM"` and `"symmetric"` specify the same option.

The matrix returned by the SPARSE function is a $k \times 3$ matrix that contains the folling values:

- The first column contains the nonzero values of the *x* matrix.
- The second column contains the row numbers for each value.
- The third column contains the column numbers for each value.

For example, the following statements compute a sparse representation of a dense matrix with many zeros:

```
x = {3   1.1  0   0  ,
     1.1 4    0   3.2,
     0   1   10   0  ,
     0   3.2  0   3  };
a = sparse(x, "sym");
print a[colname={"Value" "Row" "Col"}];
```

Figure 23.275 Sparse Data Representation

a		
Value	Row	Col
3	1	1
1.1	2	1
4	2	2
1	3	2
10	3	3
3.2	4	2
3	4	4

SPLINE and SPLINEC Calls

CALL SPLINE(*fitted, data* < , *smooth*> < , *delta*> < , *nout*> < , *type*> < , *slope*> **);**

CALL SPLINEC(*fitted, coeff, endSlopes, data* < , *smooth*> < , *delta*> < , *nout*> < , *type*> < , *slope*> **);**

The SPLINE and SPLINEC subroutines fit cubic splines to data. The SPLINE subroutine is the same as SPLINEC but does not return the matrix of spline coefficients needed to call SPLINEV, nor does it return the slopes at the endpoints of the curve.

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 subroutine returns *coeff*.

endSlopes is a 1×2 matrix that contains the slopes of the two ends of the curve expressed as angles in degrees. The SPLINEC subroutine returns the *endSlopes* argument.

The input arguments 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.

smooth is an optional scalar that specifies 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.

delta is an optional scalar that specifies 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 that specifies 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.

type is an optional 1×1 (or 1×2) character matrix or quoted literal that contains 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 strictly 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.

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).

see Stoer and Bulirsch (1980), Reinsch (1967), and Pizer (1975) for descriptions of the methods used to fit the spline. For simplicity, the following explanation assumes that the *data* matrix does not contain a weighting column.

Nonparametric splines can be used to fit data for which you believe there is a functional relationship between the X and Y variables. The unique values of X (stored in the first column of *data*) form a partition $\{a = x_1 < x_2 < \ldots < x_n = b\}$ of the interval $[a, b]$. You can use a spline to interpolate the data (produce a

curve that passes through each data point) provided that there is a single Y value for each X value. The spline is created by constructing cubic polynomials on each subinterval $[x_i, x_{i+1}]$ so that the value of the cubic polynomials and their first two derivatives coincide at each x_i.

An interpolating spline is not uniquely determined by the set of Y values. To achieve a unique interpolant, S, you must specify two constraints on the endpoints of the interval $[a, b]$. You can achieve uniqueness by specifying one of the following conditions:

- $S''(a) = 0, S''(b) = 0$. The second derivative at both endpoints is zero. This is the default condition, but can be explicitly set by using `type='zero'`.
- Periodic conditions. If your data are periodic so that x_1 can be identified with x_n, and if $y_1 = y_n$, then the interpolating spline already satisfies $S(a) = S(b)$. Setting `type='periodic'` further requires that $S'(a) = S'(b)$ and $S''(a) = S''(b)$.
- Fixed slopes at endpoints. Setting slope=$\{y'_1, y'_n\}$ requires that $S'(a) = y'_1$ and $S'(b) = y'_n$.

The following codeu give three examples of computing an interpolating spline for data. Note that the first and last Y values are the same, so you can ask for a periodic spline.

```
data = { 0 5, 1 3, 2 5, 3 4, 4 6, 5 7, 6 6, 7 5 };

/* Compute three spline interpolants of the data */
/* (1) a cubic spline with type=zero (the default) */
call spline(fitted,data);

/* (2) A periodic spline */
call spline(periodicFitted,data) type='periodic';

/* (3) A spline with specific slopes at endpoints */
call spline(slopeFitted,data) slope={45 30};
```

You can also use a spline to smooth data. In general, a smoothing spline will not pass through any data pair exactly. A very small value of the *smooth* smoothing parameter will approximate an interpolating polynomial for data in which each unique X value is assigned the mean of the Y values that correspond to that X value. As the *smooth* parameter gets very large, the spline will approximate a linear regression.

The following statements compute two smoothing splines for the same data as in the previous example. The spline coefficients are passed to the SPLINEV function which evaluates the smoothing spline at the original X values. Note that the smoothing spline does not pass through the original Y values. Note also that the smoothing parameter for the periodic spline is smaller, so the periodic spline has more "wiggles" than the corresponding spline with the larger smoothing parameter.

```
data = { 0 5, 1 3, 2 5, 3 4, 4 6, 5 7, 6 6, 7 5 };

/* Compute spline smoothers of the data. */
call splinec(fitted,coeff,endSlopes,data) smooth=1;

/* Evaluate the smoother at the original X positions */
smoothFit = splinev(coeff, data[,1]);
```

```
/* Compute periodic spline smoother of the data. */
call splinec(fitted,coeff,endSlopesP,data)
             smooth=0.1 type="periodic";

/* Evaluate the smoother at the original X positions */
smoothPeriodicFit = splinev(coeff, data[,1]);

/* Compare the two fits. Verify that the periodic
   spline has identical slopes at the end points. */
print smoothFit smoothPeriodicFit, endSlopesP;
```

```
             SMOOTHFIT           SMOOTHPERIODICFIT

             0 4.4761214             0 4.7536432
             1  4.002489             1 3.5603915
             2 4.2424509             2 4.3820561
             3 4.8254655             3   4.47148
             4 5.7817386             4 5.8811872
             5 6.3661254             5 6.8331581
             6 6.0606327             6 6.1180839
             7 5.2449764             7 4.7536432

                        ENDSLOPESP

                      -58.37255 -58.37255
```

A parametric spline can be used to interpolate or smooth data for which there does not seem to be a functional relationship between the X and Y variables. A partition $\{t_i\}$ is formed as explained in the documentation for the *type* parameter. Splines are then used to fit the X and Y values independently.

The following program fits a parametric curve to data that is shaped like an "S." The variable *fitted* is returned as a *numParam* $\times 2$ matrix that contains the ordered pairs that correspond to the parametric spline. These ordered pairs correspond to *numParam* evenly spaced points in the domain of the parameter t.

The purpose of the SPLINEV function is to evaluate (*score*) an interpolating or smoothing spline at an arbitrary set of points. The following program shows how to construct the parameters that correspond to the original data by using the formula specified in the documentation for the *type* argument. These parameters are used to construct the evenly spaced parameters that correspond to the data in the *fitted* matrix.

```
data = {3 7, 2 7, 1 6, 1 5, 2 4, 3 3, 3 2, 2 1, 1 1};

/* Compute parametric spline interpolant */
numParam = 20;
call splinec(fitted,coeff,endSlopes,data)
             nout=numParam type={"zero" "parametric"};

/* Form the parameters mapped onto the data */
/* Evaluating the splines at t would return data */
t = j(nrow(data),1,0); /* first parameter is zero */
```

```
do i = 2 to nrow(t);
   t[i] = t[i-1] + sqrt( (data[i,1]-data[i-1,1])##2 +
                         (data[i,2]-data[i-1,2])##2 );
end;

/* construct numParam evenly-spaced parameters
   between 0 and t[nrow(t)] */
params = do(0, t[nrow(t)], t[nrow(t)]/(numParam-1))`;

/* evaluate the parametric spline at these points */
xy = splinev(coeff, params);
print params fitted xy;
```

The output from this program is as follows:

```
   PARAMS       FITTED                    XY

        0            3         7           3         7
0.6897753    2.3002449 7.0492667   2.3002449 7.0492667
1.3795506    1.6566257 6.8416091   1.6566257 6.8416091
2.0693259    1.1581077 6.3289203   1.1581077 6.3289203
2.7591012    0.9203935 5.6475064   0.9203935 5.6475064
3.4488765    1.0128845 4.9690782   1.0128845 4.9690782
4.1386518    1.4207621 4.4372889   1.4207621 4.4372889
4.8284271            2         4           2         4
5.5182024    2.5792379 3.5627111   2.5792379 3.5627111
6.2079777    2.9871155 3.0309218   2.9871155 3.0309218
 6.897753    3.0796065 2.3524936   3.0796065 2.3524936
7.5875283    2.8418923 1.6710797   2.8418923 1.6710797
8.2773036    2.3433743 1.1583909   2.3433743 1.1583909
8.9670789    1.6997551 0.9507333   1.6997551 0.9507333
9.6568542            1         1           1         1
```

Attempting to evaluate a spline outside of its domain of definition will result in a missing value. For example, the following statements define a spline on the interval $[0, 7]$. Attempting to evaluate the spline at points outside of this interval (-1 or 20) results in missing values.

```
data = { 0 5, 1 3, 2 5, 3 4, 4 6, 5 7, 6 6, 7 5 };
call splinec(fitted,coeff,endSlopes,data) slope={45 45};
v =  splinev(coeff,{-1 1 2 3 3.5 4 20});
print v;
```

```
                          v

                       -1         .
                        1         3
                        2         5
                        3         4
                      3.5 4.7073171
                        4         6
                       20         .
```

One use of splines is to estimate the integral of a function that is known only by its value at a discrete set of points. Many people are familiar with elementary methods of numerical integration such as the Left-Hand Rule, the Trapezoid Rule, and Simpson's Rule. In the Left-Hand Rule, the integral of discrete data is estimated by the exact integral of a piecewise constant function between the data. In the Trapezoid Rule, the integral is estimated by the exact integral of a piecewise linear function connecting the data. In Simpson's Rule, the integral is estimated as the exact integral of a piecewise quadratic function between the data points.

Since a cubic spline is a sequence of cubic polynomials, it is possible to compute the exact integral of the cubic spline and use this as an estimate for the integral of the discrete data. The next example takes a function defined by discrete data and finds the integral and the first moment of the function.

The implementation of the integrand function (`fcheck`) uses a useful trick to evaluate a spline at a single point. Note that if you pass in a scalar argument to the SPLINEV function, you get back a vector that represents the evaluation of the spline along evenly spaced points. To get around this, the following statements evaluate the spline at the vector `x // x` and then extract the entry in the first row, second column. This number is the value of the spline evaluated at x.

```
x = { 0, 2, 5, 7, 8, 10 };
y = x + 0.1*sin(x);
a = x || y;
call splinec(fit,coeff,endSlopes,a);

start fcheck(x) global(coeff,pow);
   /* The first column of v contains the points of evaluation
      while the second column contains the evaluation. */
   v = x##pow # splinev(coeff,x //x)[1,2];
   return(v);
finish;

/* use QUAD to integrate */
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;

/* Check the computation by using Gauss-Legendre integration: this
   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; /* should agree with earlier result */
```

The program prints the following results:

```
                          MASS

                     50.204224
                      6.658133
                     49.953307

                          MASS

                     50.204224
                      6.658133
                     49.953307
```

SPLINEV Function

SPLINEV(*coeff* < , *delta* > < , *nout* > **);**

The SPLINEV function evaluates a cubic spline at a set of points. The function returns a two-column matrix that contains the points of evaluation in the first column and the corresponding fitted values of the spline in the second column.

The arguments 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 that specifies evaluation points. If *delta* is a scalar, the spline is evaluated at equally spaced points *delta* apart. If *delta* is a vector arranged in ascending order, the spline is 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 that specifies 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 972 for details and examples.

SPOT Function

SPOT(*times, forward_rates***);**

The SPOT function returns an $n \times 1$ vector of (per-period) spot rates, given vectors of forward rates and times. The function takes the following arguments:

times is an $n \times 1$ column vector of times in consistent units. Elements should be nonnegative.

forward_rates is an $n \times 1$ column vector of corresponding (per-period) forward rates. Elements should be positive.

The SPOT function transforms the given spot rates as

$$s_1 = f_1$$

$$s_i = \left(\Pi_{j=1}^{j=i} (1 + f_j)^{t_j - t_{j-1}} \right)^{\frac{1}{t_i}} - 1; \quad i = 2, \ldots, n$$

where, by convention, $t_0 = 0$.

For example, the following statements produce the output shown in Figure 23.276:

```
time = T(do(1, 5, 1));
forward = T(do(0.05, 0.09, 0.01));
```

```
spot = spot(time, forward);
print spot;
```

Figure 23.276 Spot Rates

spot
0.05
0.0549882
0.0599686
0.0649413
0.0699065

SQRSYM Function

SQRSYM(*matrix***);**

where *matrix* is a symmetric numeric matrix.

The SQRSYM function takes a packed-symmetric matrix (such as generated by the SYMSQR function) and transforms it back into a dense square matrix. The elements of the argument are unpacked (rowwise) into the lower triangle of the result and reflected across the diagonal into the upper triangle.

For example, the following statements return a symmetric matrix:

```
v = T(1:6);
sqr = sqrsym(v);
print sqr;
```

Figure 23.277 Symmetric Matrix

sqr		
1	2	4
2	3	5
4	5	6

The SQRSYM function and the SYMSQR function are inverse operations on the set of symmetric matrices.

SQRT Function

SQRT(*matrix***);**

The SQRT function returns the positive square roots of each element of the argument matrix. An example of a valid statement follows.

```
a = {1 2 3 4};
c = sqrt(a);
print c;
```

Figure 23.278 Square Roots

```
                                    c

                1 1.4142136 1.7320508         2
```

SQRVECH Function

SQRVECH(*matrix***);**

The SQRVECH function transforms a packed-symmetric matrix into a dense square matrix. The elements of the argument are unpacked (columnwise) into the lower triangle of the result and reflected across the diagonal into the upper triangle. The argument *matrix* should be a column-stacked, packed-symmetric matrix, such as generated by the VECH function.

For example, the following statements return a symmetric matrix:

```
v = T(1:6);
sqr = sqrvech(v);
print sqr;
```

Figure 23.279 Symmetric Matrix

```
                                   sqr

                          1         2         3
                          2         4         5
                          3         5         6
```

The SQRVECH function and the VECH function are inverse operations on the set of symmetric matrices.

SSQ Function

SSQ(*matrix1* **<** *, matrix2, . . . , matrix15* **>);**

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);
print x;
```

Figure 23.280 Sums of Squares

x
91

START Statement

START < *name* > < *(arguments)* > < **GLOBAL***(arguments)* > **;**

language statements

FINISH < *name* > **;**

The START statement defines the start of a module definition. Subsequent statements are not executed immediately, but are instead collected for later execution. The FINISH statement signals the end of a module definition.

The arguments to the START statement 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.
language statements	are statements making up the body of the module.

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 6 for details.

The following example 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;

x = 1:4;
call mymod(2, 1);
print y;
```

Figure 23.281 Results of Calling a Module

```
                                y

                3         5         7         9
```

STD Function

STD(*x*);

The STD function computes a sample standard deviation of data. The sample standard deviation of a column vector is computed as the square root of the sample variance. See the VAR function for details.

When *x* is a matrix, the sample variance is computed for each column, as the following example shows:

```
x = {5 1 10,
     6 2 3,
     6 8 .,
     6 7 9,
     7 2 13};

std = std(x);
print std;
```

Figure 23.282 Standard Deviation of Columns

```
                                std

                0.7071068 3.2403703 4.1932485
```

The STD function returns a missing value for columns with fewer than two nonmissing observations.

STOP Statement

STOP ;

The STOP statement stops the program, and no further matrix statements are executed. However, PROC IML does not exit, and continues to execute if more statements are submitted. See also the descriptions of the RETURN and ABORT statements.

If execution was interrupted by a PAUSE statement or by a break, the STOP statement clears all the paused states and returns to immediate mode.

STORAGE Function

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";
```

Issue the following command to get the resulting matrix:

```
a = storage();
print a;
```

STORE Statement

STORE <**MODULE**=*(module-list)*> <*matrix-list*> **;**

The STORE statement stores matrices and modules in a storage library.

The arguments to the STORE statement are as follows.

module-list is a list of module names.

matrix-list is a list of matrix names.

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 by using the RESET STORAGE statement and defaults to WORK.IMLSTOR. The SHOW STORAGE statement lists the current contents of the storage library. The following statement stores all matrices:

```
store;
```

See Chapter 17, "Storage Features," and also the descriptions of the LOAD, REMOVE, RESET, and SHOW statements for related information.

SUBMIT Statement

SUBMIT <*parameters*> </ *options*> **;**

language statements

ENDSUBMIT ;

The SUBMIT statement enables you to submit SAS statements for processing from within a SAS/IML program. You can use the SUBMIT statement to call SAS procedures, DATA steps, and macros. All statements between the SUBMIT and the ENDSUBMIT statements are referred to as a *SUBMIT block*. The SUBMIT block is processed by the SAS language processor.

If you use the R option, the SUBMIT statement enables you to submit statements to the R language for processing.

The SUBMIT statement must appear on a line by itself. All SAS/IML matrices that are defined prior to the SUBMIT statement remain defined after the ENDSUBMIT statement.

parameters specifies one or more optional SAS/IML matrices whose values are substituted into the language statements in the SUBMIT block. To reference a parameter in the SUBMIT block, prefix the name of the parameter with an ampersand (&). If you do not specify the *parameters* argument, the SUBMIT block is sent without modification to the SAS (or R) language processor.

The following options are available in the SUBMIT statement after a slash (/).

OK=*ok-matrix* specifies the name of a matrix. The matrix is set to 1 if the SUBMIT block executes without error, and to 0 otherwise.

R specifies that statements in the SUBMIT block are processed by the R statistical software. You can use the R option to call functions in the R language, provided that the following statements are true:

1. the R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See "The RLANG System Option" on page 190.)

The following example calls a SAS procedure from a PROC IML program. The example passes in a parameter which is used by the FREQ procedure:

```
proc iml;
VarName = "Sex";
submit VarName;
proc freq data=Sashelp.Class;
   table &VarName / out=OutFreq;
```

```
run;
endsubmit;
```

Prior to the SUBMIT statement, the program defines the **VarName** matrix. The matrix contains the name of a variable in the Sashelp.Class data set. The **VarName** matrix is listed in the SUBMIT statement, which means that the contents of the matrix is available for substitution into the SUBMIT block. The SUBMIT block references the contents of the matrix by preceding the matrix name by an ampersand (&). Consequently, the FREQ procedure carries out a one-way frequency analysis for the Sex variable. The output from PROC FREQ is shown in Figure 23.283.

Figure 23.283 Result of Calling a SAS Procedure

The FREQ Procedure

Sex	Frequency	Percent	Cumulative Frequency	Cumulative Percent
F	9	47.37	9	47.37
M	10	52.63	19	100.00

The preceding statements also create output data set, OutFreq. The following statements read the data into SAS/IML matrices:

```
use OutFreq;
read all var VarName into Levels;
read all var {Count};
close OutFreq;

print Count[rowname=Levels];
```

Notice that the **VarName** matrix is still defined, even after the FREQ procedure has finished execution. The statements read portions of the PROC FREQ output data set into two SAS/IML vectors. The output from the program is shown in Figure 23.284.

Figure 23.284 Result of Calling a SAS Procedure

	COUNT
F	9
M	10

Chapter 10, "Submitting SAS Statements," provides details and further examples of submitting SAS statements. Chapter 11, "Calling Functions in the R Language," describes how to submit R statements and provides examples.

You cannot use the SUBMIT statement in code that is pushed to the input command queue with the EXECUTE, PUSH, or QUEUE subroutines.

SUBSTR Function

SUBSTR(*matrix, position* < *, length* > **);**

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.

The arguments to the SUBSTR function are as follows:

matrix	is a character matrix or quoted literal.
position	is a numeric matrix or scalar that contains the starting position.
length	is a numeric matrix or scalar that contains the length of the substring.

Each substring is constructed by 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

$$\text{NLENG(matrix)} - \text{MIN(position)} + 1$$

The following statements return the output shown:

```
B = {abc def ghi, jkl mno pqr};
a = substr(b, 3, 2);
```

```
        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

SUM(*matrix1* < *, matrix2, . . . , matrix15* > **);**

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, consider the following statements:

```
a={2 1, 0 -1};
b=sum(a);
```

These statements return the following scalar:

```
B             1 row       1 col     (numeric)

                          2
```

SUMMARY Statement

SUMMARY <**CLASS** *operand*> <**VAR** *operand*> <**WEIGHT** *operand*> <**STAT** *operand*> <**OPT** *operand*> <**WHERE***(expression)*> **;**

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 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 that contains the names of the variables, for example:

```
summary class { age sex} ;
```

Both numeric and character variables can be used as CLASS variables.

VAR *operand*

computes statistics for a set of numeric variables from the current input data set. The *operand* is a character matrix that contains 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 that contains 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 that contains the names of statistics. For example, to get the mean and standard deviation, specify the following:

```
summary stat{mean std};
```

The following list describes 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, 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 that contains 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, that contains 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)* **;**

The arguments to the WHERE clause are as follows:

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 like 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, by 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 by using operator precedence. Notice that 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 7 for an example that uses the SUMMARY statement.

SVD Call

CALL SVD(*u, q, v, a***);**

The SVD subroutine computes the singular value decomposition for a numerical matrix. The input to the SVD subroutine is as follows:

a	is the $m \times n$ input matrix that is factored as described in the following discussion.
	The SVD subroutine returns the following output arguments:
u	is an $m \times n$ orthonormal matrix
q	is an $n \times 1$ vector that contains the singular values
v	is an $n \times n$ orthonormal matrix

If $m \geq n$, the SVD subroutine factors a real $m \times n$ matrix $\mathbf{A}$ 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 $\mathbf{A}$. The columns of $\mathbf{U}$ contains of the orthonormal eigenvectors of $\mathbf{A}\mathbf{A}'$, and $\mathbf{V}$ contains 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 $m < n$, a corresponding decomposition is done where $\mathbf{U}$ and $\mathbf{V}$ switch roles:

$$\mathbf{A} = \mathbf{U}\text{diag}(\mathbf{Q})\mathbf{V}'$$

where

$$\mathbf{U}'\mathbf{U} = \mathbf{U}\mathbf{U}' = \mathbf{V}'\mathbf{V} = \mathbf{I}_w$$

The singular values are sorted in descending order.

For information about the method used in the SVD subroutine, see Wilkinson and Reinsch (1971).

The following example is taken from Wilkinson and Reinsch (1971):

```
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);
print u, q, v;
```

```
/* check correctness of factors */
zero = ssq(a - u*diag(q)*v`);
reset fuzz;           /* print small numbers as zero */
print zero;
```

The matrix is rank-3 with exact singular values $\sqrt{1248}$, 20, $\sqrt{384}$, 0, and 0. Because of the repeated singular values, the last two columns of the **U** matrix are not uniquely determined. A valid result is shown in Figure 23.285:

Figure 23.285 Singular Value Decomposition

```
                              u

0.7071068 0.1581139 -0.176777 -0.328209 -0.328056
0.5303301 0.1581139 0.3535534 0.5309976 0.0489362
0.1767767 -0.790569 0.1767767 -0.413567 0.1307398
        0 0.1581139 0.7071068 -0.266418 0.0321656
0.3535534 -0.158114         0 0.0253566 -0.041441
0.1767767 0.1581139  -0.53033  -0.19666 0.3666144
        0 0.4743416 0.1767767 -0.500944 0.4145131
0.1767767 -0.158114         0 0.2793571 0.7509412

                              q

                         35.327043
                                20
                         19.595918
                           1.1E-15
                         5.501E-16

                              v

0.8006408 0.3162278 -0.288675 -0.419095         0
0.4803845 -0.632456         0 0.4405091 0.4185481
0.1601282 0.3162278 0.8660254 -0.052005 0.3487901
        0 0.6324555 -0.288675 0.6760591  0.244153
0.3202563         0 0.2886751 0.4129773 -0.802217

                             zero

                                  0
```

The SVD routine performs most of its computations in the memory allocated for returning the singular value decomposition.

SWEEP Function

SWEEP(*matrix, index-vector***);**

The SWEEP function sweeps *matrix* on the pivots indicated in *index-vector* to produce a new matrix.

The arguments the SWEEP function are as follows:

matrix is a numeric matrix or literal.

index-vector is a numeric vector that indicates the pivots.

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 **A** is partitioned into

$$\begin{bmatrix} \mathbf{R} & \mathbf{S} \\ \mathbf{T} & \mathbf{U} \end{bmatrix}$$

such that **R** is $q \times q$ and **U** is $(m-q) \times (n-q)$. Let

```
I = [1 2 3 . . . q]
```

Then, the statement

```
B=sweep(A,I);
```

becomes

$$\begin{bmatrix} \mathbf{R}^{-1} & \mathbf{R}^{-1}\mathbf{S} \\ -\mathbf{T}\mathbf{R}^{-1} & \mathbf{U}-\mathbf{T}\mathbf{R}^{-1}\mathbf{S} \end{bmatrix}$$

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(**A**,1),2)=SWEEP(**A**,{ 1 2 })
- SWEEP(SWEEP(**A**,**I**),**I**)=**A**

See Beaton (1964) for more information about these properties.

To use the SWEEP function for regression, suppose the matrix **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

$$\begin{bmatrix} (\mathbf{X}'\mathbf{X})^{-1} & (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y} \\ -\mathbf{Y}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} & \mathbf{Y}'(\mathbf{I}-\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}')\mathbf{Y} \end{bmatrix}$$

The partitions of **B** form the beta values, SSE, and a matrix proportional to the covariance of the beta values for the least squares estimates of **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.

An example that uses the SWEEP function for regression follows:

```
x = { 1  1  1,
      1  2  4,
      1  3  9,
      1  4 16,
      1  5 25,
      1  6 36,
      1  7 49,
      1  8 64 };

y = {  3.929,
       5.308,
       7.239,
       9.638,
      12.866,
      17.069,
      23.191,
      31.443 };

n = nrow(x);          /* number of observations */
k = ncol(x);          /* number of variables */
xy = x||y;            /* augment design matrix */
A = xy` * xy;         /* form cross products */
S = sweep( A, 1:k );

beta = S[1:k,4];      /* parameter estimates */
sse = S[4,4];         /* sum of squared errors */
mse = sse / (n-k);    /* mean squared error */
cov = S[1:k, 1:k] # mse; /* covariance of estimates */
print cov, beta, sse;
```

```
                          COV

             0.9323716 -0.436247 0.0427693
             -0.436247 0.2423596 -0.025662
             0.0427693 -0.025662 0.0028513

                          BETA

                        5.0693393
                        -1.109935
                        0.5396369

                          SSE

                        2.395083
```

The SWEEP function performs most of its computations in the memory allocated for the result matrix.

SYMSQR Function

SYMSQR(*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 symmetry, but usually *matrix* is a symmetric matrix.

The following statement produces the output shown:

```
sym=symsqr({1 2, 3 4});
```

```
              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

T(*matrix***);**

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 following statements produce the matrix **Y**, as shown:

```
x={1 2, 3 4};
y=t(x);
```

```
              Y             2 rows      2 cols    (numeric)

                                    1         3
                                    2         4
```

TABULATE Call

CALL TABULATE(*levels, freq, x* < *, method*> **);**

The TABULATE subroutine counts the number of elements in each of the unique categories of the *x* argument.

The output arguments are as follows:

levels	contains the unique sorted elements of the *x* argument. See also the UNIQUE function.
freq	contains the number of elements of *x* that match each element of *levels*.

The input arguments are as follows:

x	specifies a vector of values.
method	specifies whether missing values are included in the analysis. The following values are valid:

"nomissing"	specifies that missing values are excluded from the analysis. This is the default value for the option.
"missing"	specifies that missing values are counted as a valid separate level.

The *method* argument is not case-sensitive. The first two characters are used to determine the value. For example, "MISS" and "missing" specify the same option.

The following statements demonstrate the TABULATE subroutine:

```
x = {C, A, B, A, C, A};
call tabulate(labels, freq, x);
print freq[colname=labels];

x = {C, A, B, " ", A, C, A, " "};
call tabulate(labels, freq, x, "Missing");
labels = "Missing" || remove(labels, 1);
print freq[colname=labels];
```

Figure 23.286 Frequencies of Levels

```
                              freq
                         A         B         C

                         3         1         2


                              freq
                Missing       A         B         C

                      2       3         1         2
```

TOEPLITZ Function

TOEPLITZ(*a*);

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, $\mathbf{R}$, are the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, the first p rows of $\mathbf{R}$ are 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

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2' & \mathbf{A}_1 & \mathbf{A}_2 & \cdots & \mathbf{A}_{n-1} \\ \mathbf{A}_3' & \mathbf{A}_2' & \mathbf{A}_1 & \cdots & \mathbf{A}_{n-2} \\ \vdots & & & & \\ \mathbf{A}_n' & \mathbf{A}_{n-1}' & \mathbf{A}_{n-2}' & \cdots & \mathbf{A}_1 \end{bmatrix}$$

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

CALL TPSPLINE(*fitted, coeff, adiag, gcv, x, y* < , *lambda*> **);**

The TSPLINE subroutine fits a thin-plate smoothing spline (TPSS) to data. 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 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. (see 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 x respectively. The length of *coeff* equals to $k + \text{nuobs} + 1$. 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 you with both the ability to study the GCV curves by plotting *gcv* against *lambda* and the chance to identify a possible local minimum.

The input arguments 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 that contains λ values in $log_{10}(n\lambda)$ scale. This option gives you the power to control how you 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 that uses this particular *lambda*. This gives you the freedom to choose the λ that you deem 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.

No missing values are accepted 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, the TPSS method is illustrated with a two-dimensional independent variable $\mathbf{X} = (x^1, x^2)$. More details can be found in Wahba (1990), or in Bates et al. (1987).

Assume that the data are from the model

$$y_i = f(x_i) + \epsilon_i,$$

where $(x_i, y_i), i = 1, \ldots, n$ 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 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

$$S_\lambda(f) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda J_2(f).$$

Duchon (1976) derived that the minimizer f_λ can be represented as

$$f_\lambda(x) = \sum_{i=1}^{3} \beta_i \phi_i(x) + \sum_{i=1}^{n} \delta_i E_2(x - x_i),$$

where $(\phi_1(x), \phi_2(x), \phi_3(x)) = (1, x^1, x^2)$ and $E_2(s) = \frac{1}{2^3\pi}\|s\|^2 ln(\|s\|)$.

Let matrix $\mathbf{K}$ have entries $(\mathbf{K})_{ij} = E_2(x_i - x_j)$ and matrix $\mathbf{T}$ have entries $(\mathbf{T})_{ij} = \phi_j(x_i)$. Then the minimization problem can be rewritten as finding coefficients β and δ to minimize

$$S_\lambda(\beta, \delta) = \frac{1}{n}\|y - \mathbf{T}\beta - \mathbf{K}\delta\|^2 + \lambda\delta^T\mathbf{K}\delta$$

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

$$\hat{y} = \mathbf{A}(\lambda)y$$

and call the $\mathbf{A}(\lambda)$ as the "hat" matrix, the GCV function $V(\lambda)$ is defined as

$$V(\lambda) = \frac{(1/n)\|(\mathbf{I} - \mathbf{A}(\lambda)y\|^2}{[(1/n)\mathrm{tr}(\mathbf{I} - \mathbf{A}(\lambda))]^2}$$

The returned values from this function call provide the $\hat{y}$ as *fitted*, the (β, δ) as *coeff*, and $diag(A(\lambda))$ as *adiag*.

To evaluate the TPSS fit $f_\lambda(x)$ at new data points, you can use the TPSPLNEV call.

Suppose $\mathbf{X}^{\mathrm{new}}$, a $m \times k$ matrix, contains the m new data points at which you want to evaluate f_λ. Let $(\mathbf{T}^{\mathrm{new}}_{ij}) = \phi_j(x^{\mathrm{new}}_i)$ and $(\mathbf{K}^{\mathrm{new}}_{ij}) = E_2(x^{\mathrm{new}}_i - x_j)$ be the ijth elements of $\mathbf{T}^{\mathrm{new}}$ and $\mathbf{K}^{\mathrm{new}}$ respectively. The prediction at new data points $\mathbf{X}^{\mathrm{new}}$ is

$$y_{\mathrm{pred}} = T^{\mathrm{new}}\beta + K^{\mathrm{new}}\delta$$

Therefore, the y_{pred} can be easily evaluated by using the coefficient (β, δ) obtained from the TPSPLINE call.

An example is given in the documentation for the TPSPLNEV call.

TPSPLNEV Call

CALL TPSPLNEV(*pred, xpred, x, coeff***);**

The TPSPLNEV subroutine evaluates the thin-plate smoothing spline (TPSS) at new data points. It is used after the TPSPLINE subroutine fits a thin-plate spline model to data.

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 input arguments 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 that 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 23.287.

```
x={ -1.0 -1.0,    -1.0 -1.0,    -0.5 -1.0,    -0.5 -1.0,
     0.0 -1.0,     0.0 -1.0,     0.5 -1.0,     0.5 -1.0,
     1.0 -1.0,     1.0 -1.0,    -1.0 -0.5,    -1.0 -0.5,
    -0.5 -0.5,    -0.5 -0.5,     0.0 -0.5,     0.0 -0.5,
     0.5 -0.5,     0.5 -0.5,     1.0 -0.5,     1.0 -0.5,
    -1.0  0.0,    -1.0  0.0,    -0.5  0.0,    -0.5  0.0,
     0.0  0.0,     0.0  0.0,     0.5  0.0,     0.5  0.0,
     1.0  0.0,     1.0  0.0,    -1.0  0.5,    -1.0  0.5,
    -0.5  0.5,    -0.5  0.5,     0.0  0.5,     0.0  0.5,
     0.5  0.5,     0.5  0.5,     1.0  0.5,     1.0  0.5,
    -1.0  1.0,    -1.0  1.0,    -0.5  1.0,    -0.5  1.0,
     0.0  1.0,     0.0  1.0,     0.5  1.0,     0.5  1.0,
     1.0  1.0,     1.0  1.0 };

y = {15.54, 15.76, 18.67, 18.50, 19.66, 19.80, 18.60, 18.52,
     15.87, 16.04, 10.92, 11.14, 14.81, 14.83, 16.56, 16.44,
     14.91, 15.06, 10.92, 10.94,  9.61,  9.65, 14.03, 14.03,
     15.77, 16.00, 14.00, 14.03,  9.56,  9.58, 11.21, 11.09,
     14.84, 14.99, 16.55, 16.51, 14.98, 14.72, 11.15, 11.17,
     15.83, 15.96, 18.64, 18.56, 19.54, 19.81, 18.57, 18.61,
     15.87, 15.90 };
```

Now generate a sequence of λ from -3.8 to -3.3 so that you can study the GCV function within this range. Use the following statement:

```
lambda = T( do(-3.8, -3.3, 0.1) );
```

Use the following 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 subroutine 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 23.287. Also in Figure 23.287, 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 $f_\lambda(x)$ more closely, you can use the following statements to generate a more dense grid on $[-1, 1] \times [-1, 1]$.

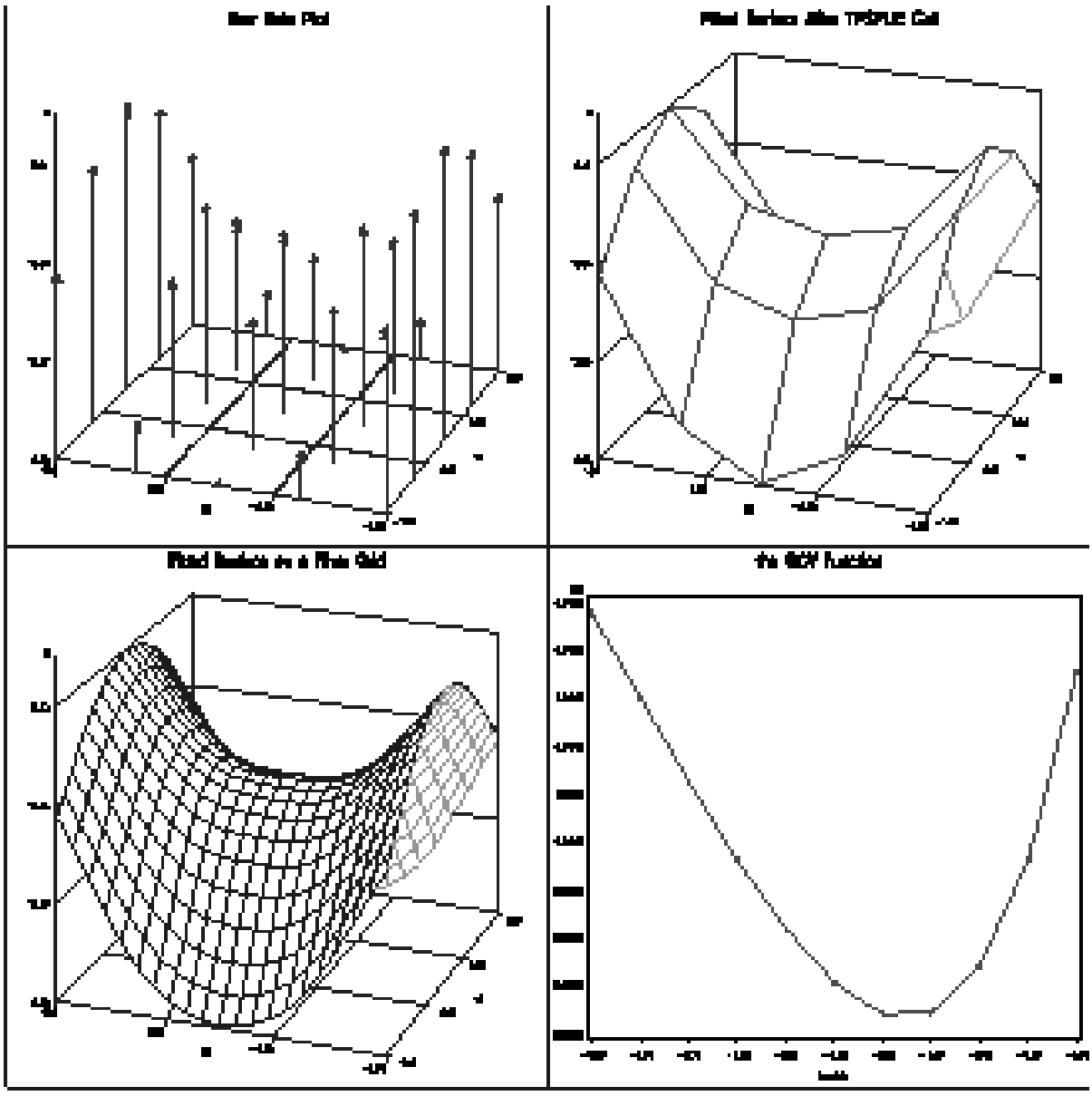
```
xGrid = T( do(-1, 1, 0.1) );
yGrid = T( do(-1, 1, 0.1) );
do i = 1 to nrow(xGrid);
   x1 = x1 // repeat(xGrid[i], nrow(yGrid));
   x2 = x2 // yGrid;
end;
xpred = x1 || x2;
```

Now you can use the function TPSPLNEV to evaluate $f_\lambda(x)$ on this dense grid. Here is the statement:

```
call tpsplnev(pred, xpred, x, coef);
```

The final fitted surface is plotted in Figure 23.287, panel 3.

Figure 23.287 Plots of Fitted Surface



TRACE Function

TRACE(*matrix***);**

The TRACE function produces a single numeric value that is the sum of the diagonal elements of *matrix*. For example, the following statement produces the output shown:

```
a = trace({5 2, 1 3});

              A             1 row       1 col     (numeric)

                                        8
```

TRISOLV Function

TRISOLV(*form, R, b* **<** *, piv* **>);**

The TRISOLV function efficiently solves linear systems that involve a triangular matrix.

The TRISOLV function returns the $n \times p$ matrix **X** that contains p solutions of the p linear systems specified by *form*, *R*, and *b*.

The arguments to the TRISOLV function are as follows:

form specifies which of the following form of a triangular linear system is to be solved:

form=1	solve $\mathbf{R}x = b$, **R** upper triangular
form=2	solve $\mathbf{R}'x = b$, **R** upper triangular
form=3	solve $\mathbf{R}'x = b$, **R** lower triangular
form=4	solve $\mathbf{R}x = b$, **R** lower triangular

R specifies the $n \times n$ nonsingular upper (*form*=1,2) or lower (*form*=3,4) triangular coefficient matrix **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, **B**, of p right-hand sides $b_k, k = 1 \ldots p$.

piv specifies an optional n vector that relates the order of the columns of matrix **R** to the order of the columns of an original coefficient matrix **A** for which matrix **R** has been computed as a factor. For example, the vector *piv* can be the result of the QR decomposition of a matrix **A** whose columns were permuted in the order $\mathbf{A}_{piv[1]}, \ldots, \mathbf{A}_{piv[n]}$.

For *form*=1 and *form*=3, the solution is obtained by backward elimination. For *form*=2 and *form*=4, the solution is obtained by forward substitution.

If TRISOLV recognizes the upper or lower triangular matrix **R** as a singular matrix (that is, one that contains at least one zero diagonal element), it exits with an error message.

Consider the following example:

```
R = { 1  0  0  0,
      3  2  0  0,
      1 -3  5  0,
      2  7  9 -1 };

b = {1, 1, 4, -6 };
x = trisolv(4, R, b);
print x;
```

Figure 23.288 Solution of a Triangular System

```
                x

                1
               -1
                0
                1
```

Also see the example in the QR call section.

TSBAYSEA Call

CALL TSBAYSEA(*trend, season, series, adjust, abic, data* < *,order*> < *, sorder*> < *, rigid*> < *, npred*> < *, opt*> < *, cntl*> < *, print*> **);**

The TSBAYSEA subroutine performs Bayesian seasonal adjustment modeling.

The input arguments 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 section "Missing Values" on page 300.
- *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 are 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.
- *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.

The TSBAYSEA subroutine performs Bayesian seasonal adjustments. 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 subroutine 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

$$\sum_{t=1}^{N} \left\{ (y_t - T_t - S_t)^2 + d^2 \left[s^2 (\nabla^k T_t)^2 + (\nabla_L^l S_t)^2 + z^2 (S_t + \ldots + S_{t-L+1})^2 \right] \right\}$$

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" on page 296 for more information.

To analyze the monthly data with rigidity 0.5, you can specify either of the following two statements:

```
call tsbaysea(trend,season,series,adj,abic) data=z order=2
         sorder=1 rigid=0.5 npred=10 print=2;

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 in addition to the smoothed estimates. The detailed result is also printed since the PRINT=2 option is specified.

TSDECOMP Call

CALL TSDECOMP(*comp, est, aic, data, < , xdata> < , order> < , sorder> < , nar> < , npred> < , init> < , opt> < , icmp> < , print>* **);**

The TSDECOMP subroutine analyzes nonstationary time series by using smoothness priors modeling.

The input arguments 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

$$\psi_i = 0.88 \times (-0.6)^{i-1} i = 1, 2, \ldots, nar$$

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 accepted.
- *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.

- 1 specifies a line search method that requires the same number of objective function and gradient calls for cubic interpolation and extrapolation.
- 2 specifies a line search method that requires more objective function calls than gradient calls for cubic interpolation and extrapolation.
- 3 specifies a line search method that requires the same number of objective function and gradient calls for cubic interpolation and extrapolation.
- 4 specifies a line search method that requires the same number of objective function and gradient calls for cubic interpolation and stepwise extrapolation.
- 5 specifies a line search method that is a modified version of *opt[8]*=4.
- 6 specifies the golden section line search method that uses only function values for linear approximation.
- 7 specifies the bisection line search method that uses only function values for linear approximation.
- 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 computed.

- 1 requests the estimate and forecast of trend component.
- 2 requests the estimate and forecast of seasonal component.
- 3 requests the estimate and forecast of AR component.
- 4 requests the trading day adjustment component.
- 5 requests the regression component.

6 requests the time-varying regression coefficients.

You can compute 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 section "Smoothness Priors Modeling" on page 284 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 subroutine 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:

$$\nabla^k T_t = w_{1t}, w_{1t} \sim N(0, \tau_1^2)$$

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:

$$\left(1 + \sum_{i=1}^{L-1} \mathbf{B}^i\right)^l S_t = w_{2t}, w_{2t} \sim N(0, \tau_2^2)$$

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:

$$u_t = \sum_{i=1}^{p} \alpha_i u_{t-i} + w_{3t}, w_{3t} \sim N(0, \tau_3^2)$$

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:

$$\beta_{jt} = \beta_{jt-1} + v_{jt}, v_{jt} \sim N(0, \sigma_j^2)$$

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" on page 298, 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

CALL TSMLOCAR(*arcoef, ev, nar, aic, start, finish, data* **<** *, maxlag* **>** **<** *, opt* **>** **<** *, missing* **>** **<** *, print* **>**
);

The TSMLOCAR subroutine analyzes nonstationary or locally stationary time series by using the minimum AIC procedure.

The input arguments 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 are processed assuming that the mean value of the input series is 0. The default is *opt[1]*=0.
- *opt[2]* specifies the span length to be used when breaking up the time series into separate blocks. By default, $opt[2] = 0$, which forces all of the time series values into a single span.
- *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 in addition to the AR estimates.

The TSMLOCAR subroutine returns the following values:

arcoef refers to an nar×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 section "Nonstationary Time Series" on page 287 for details.

TSMLOMAR Call

CALL TSMLOMAR(*arcoef, ev, nar, aic, start, finish, data* < *, maxlag*> < *, opt*> < *, missing*> < *, print*> **);**

The TSMLOMAR subroutine analyzes nonstationary or locally stationary multivariate time series by using the minimum AIC procedure.

The input arguments 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 are processed assuming that the mean values of input series are zeros. The default is *opt[1]*=0.
- *opt[2]* specifies the span length to be used when breaking up the time series into separate blocks. By default, $opt[2] = 0$, which forces all of the time series values into a single span.
- *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 $M \times (M * \text{nar})$ 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 section "Nonstationary Time Series" on page 287 for details.

TSMULMAR Call

CALL TSMULMAR(*arcoef, ev, nar, aic, data* < *, maxlag*> < *, opt*> < *, missing*> < *, print*> **);**

The TSMULMAR subroutine estimates VAR processes by using the minimum AIC procedure.

The input arguments 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 are 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 291 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 $M \times (M * \text{nar})$ 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 section "Multivariate Time Series Analysis" on page 291 for details.

TSPEARS Call

CALL TSPEARS(*arcoef, ev, nar, aic, data* < , *maxlag*> < , *opt*> < , *missing*> < , *print*> **);**

The TSPEARS subroutine analyzes periodic AR models with the minimum AIC procedure.

The input arguments 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 are processed assuming that the mean values of input series are zeros. 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 section "Multivariate Time Series Analysis" on page 291 for details.

TSPRED Call

CALL TSPRED(*forecast, impulse, mse, data, coef, nar, nma* **<** *, ev* **>** **<** *, npred* **>** **<** *, start* **>** **<** *, constant* **>**
);

The TSPRED subroutine provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are input.

The input arguments 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

CALL TSROOT(*matout, matin, nar, nma, < , qcoef> < , print>* **);**

The TSROOT subroutine computes AR and MA coefficients from the characteristic roots of the model or computes the characteristic roots of the model from the AR and MA coefficients.

The input arguments to the TSROOT subroutine are as follows:

matin refers to the (nar + nma) × 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 computed (*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

CALL TSTVCAR(*arcoef, variance, est, aic, data < , nar> < , init> < , opt> < , outlier> < , print>* **);**

The TSTVCAR subroutine analyzes time series that are nonstationary in the covariance function.

The input arguments 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 section "Smoothness Priors Modeling" on page 284 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 section "Nonstationary Time Series" on page 287 for details.

TSUNIMAR Call

CALL TSUNIMAR(*arcoef, ev, nar, aic, data* < *, maxlag*> < *, opt*> < *, missing*> < *, print*> **);**

The TSUNIMAR subroutine determines the order of an AR process with the minimum AIC procedure and estimates the AR coefficients.

The input arguments 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 are 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 × 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, then *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" on page 295 for more information.

TYPE Function

TYPE(*matrix***);**

The TYPE function returns a single character value that represents the type of a matrix. The value 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.

The following statements determine the type for three different matrices:

```
cMat = {"Rick" "Nancy"};
t1 = type(cMat);
nMat = {3.14159 2.71828};
t2 = type(nMat);
```

```
free uMat;
t3 = type(uMat);
print t1 t2 t3;
```

Figure 23.289 The Types of Matrices

```
                                t1 t2 t3

                                C  N  U
```

UNIFORM Function

UNIFORM(*seed*);

The UNIFORM function generates a pseudorandom numbers from the uniform distribution on [0, 1]. The *seed* argument is a numeric matrix or literal. The elements of the *seed* argument can be any integer value up to $2^{31} - 1$.

The UNIFORM function returns one or more pseudorandom 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.

The following statements produce the output shown in Figure 23.204:

```
seed = 123456;
c = j(10, 1, seed);         /* generate 10 number from the same seed */
b = uniform(c);
print b;
```

Figure 23.290 Random Values Generated from a Uniform Distribution

```
                                   b

                             0.73902
                           0.2724794
                           0.7095326
                           0.3191636
                            0.367853
                            0.104491
                           0.0368003
                           0.5333324
                           0.3712995
                           0.0401944
```

For generating millions of pseudorandom numbers, use the RANDGEN subroutine.

UNION Function

UNION(*matrix1 < , matrix2, . . . , matrix15* **>);**

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.

The following statements produce the output shown:

```
a={1 2 4 5};
b={3 4};
c=union(a,b);
```

```
   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

UNIQUE(*matrix1 < , matrix2, . . . , matrix15* **>);**

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.

UNIQUEBY Function

UNIQUEBY(*matrix < , by > < , index* **>);**

The UNIQUEBY function returns the locations of the unique BY-group combinations for a sorted or indexed matrix. The arguments to the UNIQUEBY function are as follows:

matrix is the input matrix, which must be sorted or indexed according to the *by* columns.

by is either a numeric matrix of column numbers, or a character matrix that contains the names of columns that correspond to column labels assigned to *matrix* by a MATTRIB statement or READ statement. If *by* is not specified, then the first column is used.

index is a vector such that *index*[*i*] is the row index of the *i*th element of *matrix* when sorted according to *by*. Consequently, *matrix*[*index*,] is the sorted matrix. *index* can be computed for a matrix and a given set of *by* columns with the SORTNDX call. If the matrix is known to be sorted according to the *by* columns already, then *index* should be 1:nrow(*matrix*). In this case, you can also omit the *index* argument.

The UNIQUEBY function returns a column vector whose *i*th row gives the row in *index* whose value is the row in *matrix* of the *i*th unique combination of values in the *by* columns.

For example, the following statements use the SORTNDX subroutine to create a sort index for a matrix. The UNIQUEBY function is then used to determine the unique combinations of the columns of the matrix:

```
m = { 1 0,
      2 0,
      2 2,
      2 0,
      1 0,
      2 0,
      1 1 };
cols = 1:2;
call sortndx(ndx, m, cols);

sorted = m[ndx,];
unique_rows = uniqueby(m, cols, ndx);
unique_vals = m[ndx[unique_rows], cols ];
print sorted, unique_rows unique_vals;
```

Figure 23.291 Unique Values of the Sort Variables

```
                                sorted

                                 1         0
                                 1         0
                                 1         1
                                 2         0
                                 2         0
                                 2         0
                                 2         2

                      unique_rows unique_vals

                                1           1         0
                                3           1         1
                                4           2         0
                                7           2         2
```

In addition, the following statements gives the number of unique values and the number of elements in each BY-group:

```
n = nrow(unique_rows);
size = j(n,1);
do i = 1 to n-1;
   size[i] = unique_rows[i+1] - unique_rows[i];
end;
size[n] = nrow(m) - unique_rows[n] + 1;
print n, size;
```

Figure 23.292 Number of BY Groups and Number of Elements in Each Group

n
4

size
2
1
3
1

If *matrix* is already sorted according to the *by* columns (see the SORT call), then UNIQUEBY can be called with 1:nrow(*matrix*) for the *index* argument, or the last argument can be omitted as shown in the following statement:

```
unique_loc = uniqueby(sorted, cols);
print unique_loc;
```

Figure 23.293 Position of Unique Rows for a Sorted Matrix

unique_loc
1
3
4
7

USE Statement

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

The USE statement opens a SAS data set for reading.

The arguments to the USE statement are as follows:

SAS-data-set can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). For more information about 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 that contains variable names
- the name of a matrix that contains variable names
- an expression in parentheses that yields variable names
- one of the following keywords:

  ALL for all variables

  CHAR for all character variables

  NUM for all numeric variables

The following examples demonstrate each possible way you can use the VAR clause:

```
var {x1 x5 x9};          /* a literal matrix of names         */
var x;                   /* a matrix that contains the names  */
var("x1":"x9");          /* 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 as follows:

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

The arguments to the WHERE clause are as follows:

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 like 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 by 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 by 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

CALL VALSET(*matrix-name, new-value***);**

The VALSET subroutine performs indirect assignment. The subroutine takes the name of a matrix and assigns a new value to that matrix.

The arguments to the VALSET subroutine are as follows:

matrix-name is a character matrix or literal that specifies the name of a matrix.

new-value is a value to which the matrix is set.

For example, the following statements assign the string "A" to the value of the **B** matrix. The VALSET subroutine assigns the value 99 to the matrix **A**.

```
B = "A";
call valset(B, 99);
print A;
```

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

VALUE(*matrix-name***);**

The VALUE function assigns values by indirect reference. The function takes the name of a matrix and returns the value of that matrix.

For example, the following statements find that the value of the argument **B** is **A**, then look up **A** and copy the value 1 2 3 to **C**:

```
a={1 2 3};
b="A";
c=value(b);
```

Here is the resulting output:

```
C             1 row       3 cols     (numeric)

                1           2          3
```

VAR Function

VAR(*x***);**

The VAR function computes a sample variance of data. The arguments are as follows:

x specifies an $n \times p$ numerical matrix. The VAR function computes the variance of the p columns of this matrix.

The VAR function computes the sample variance of a column vector *x* as $\Sigma_{i=1}^{n}(x_i - \bar{x})^2/(n-1)$ where n is the number of nonmissing values of *x* and any missing values have been excluded. When *x* is a matrix, the sample variance is computed for each column, as the following example shows:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
     7 2 13};
var = var(x);
print var;
```

Figure 23.294 Variance of Columns

var		
0.5	10.5	16

The following statemen computes the standard deviation of each column:

```
sd = sqrt(var(x));
```

The VAR function returns a missing value for columns with fewer than two nonmissing observations.

VARMACOV Call

CALL VARMACOV(*cov, phi, theta, sigma* < , *p*> < , *q*> < , *lag*> **);**

The VARMACOV subroutine computes the theoretical cross-covariance matrices for a stationary VARMA(p, q) model.

The input arguments to the VARMACOV subroutine are as follows:

phi specifies a $km_p \times k$ matrix, Φ, that contains the autoregressive coefficient matrices, where m_p is the number of elements in the subset of the AR order and $k \geq 2$ is the number of variables. All the roots of $|\Phi(B)| = 0$ should be greater than one in absolute value, where $\Phi(B)$ is the finite order matrix polynomial in the backshift operator B, such that $B^j \mathbf{y}_t = \mathbf{y}_{t-j}$. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

sigma specifies a $k \times k$ symmetric positive-definite covariance matrix of the innovation series. If *sigma* is not specified, then an identity matrix is used.

p specifies the subset of the AR order. The quantity m_p is defined as

$$m_p = nrow(phi)/ncol(phi)$$

where *nrow(phi)* is the number of rows of the matrix *phi* and *ncol(phi)* is the number of columns of the matrix *phi*.

If you do not specify *p*, the default subset is *p*= $\{1, 2, \ldots, m_p\}$.

For example, consider a 4-dimensional vector time series, and *phi* is a 4×4 matrix. If you specify *p*=1 (the default, since $m_p = 4/4 = 1$), the VARMACOV subroutine computes the theoretical cross-covariance matrices of VAR(1) as $\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t$.

If you specify *p*=2, the VARMACOV subroutine computes the cross-covariance matrices of VAR(2) as $\mathbf{y}_t = \Phi \mathbf{y}_{t-2} + \boldsymbol{\epsilon}_t$.

Let *phi* $= [\Phi_1', \Phi_2']'$ be an 8×4 matrix. If you specify *p*= $\{1, 3\}$, the VARMACOV subroutine computes the cross-covariance matrices of VAR(3) as $\mathbf{y}_t = \Phi_1 \mathbf{y}_{t-1} + \Phi_2 \mathbf{y}_{t-3} + \boldsymbol{\epsilon}_t$. If you do not specify *p*, the VARMACOV subroutine computes the cross-covariance matrices of VAR(2) as $\mathbf{y}_t = \Phi_1 \mathbf{y}_{t-1} + \Phi_2 \mathbf{y}_{t-2} + \boldsymbol{\epsilon}_t$.

q specifies the subset of the MA order. The quantity m_q is defined as

$$m_q = nrow(theta)/ncol(theta)$$

where *nrow(theta)* is the number of rows of matrix *theta* and *ncol(theta)* is the number of columns of matrix *theta*.

If you do not specify *q*, the default subset is *q*= $\{1, 2, \ldots, m_q\}$.

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 cross-covariance matrices from lag zero to lag h. By default, *lag* = 12.

The VARMACOV subroutine returns the following value:

cov is a $k(lag + 1) \times k$ matrix that contains the theoretical cross-covariance matrices of the VARMA(p, q) model.

Consider the following bivariate ($k = 2$) VARMA(1,1) model:

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t - \Theta \boldsymbol{\epsilon}_{t-1}$$

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To compute the cross-covariance matrices of this model, you can use the following statements:

```
phi   = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
sigma= { 1.0  0.5, 0.5 1.25};
call varmacov(cov, phi, theta, sigma) lag=5;
```

VARMALIK Call

CALL VARMALIK(*lnl, series, phi, theta, sigma* **<** *, p* **>** **<** *, q* **>** **<** *, opt* **>** **);**

The VARMALIK subroutine computes the log-likelihood function for a VARMA(p, q) model.

The input arguments to the VARMALIK subroutine are as follows:

series specifies an $n \times k$ matrix that contains the vector time series (assuming mean zero), where n is the number of observations and $k \geq 2$ is the number of variables.

phi specifies a $km_p \times k$ matrix that contains the autoregressive coefficient matrices, where m_p is the number of the elements in the subset of the AR order. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

sigma specifies a $k \times k$ covariance matrix of the innovation series. If you do not specify *sigma*, an identity matrix is used.

p specifies the subset of the AR order. See the VARMACOV subroutine.

q specifies the subset of the MA order. See the VARMACOV subroutine.

opt specifies the method of computing the log-likelihood function:

- *opt=0* requests the multivariate innovations algorithm. This algorithm requires that the time series is stationary and does not contain missing observations.
- *opt=1* requests the conditional log-likelihood function. This algorithm requires that the number of the observations in the time series must be greater than *p*+*q* and that the series does not contain missing observations.
- *opt=2* requests the Kalman filtering algorithm. This is the default and is used if the required conditions in *opt=0* and *opt=1* are not satisfied.

The VARMALIK subroutine returns the following value:

lnl is a 3×1 matrix that contains the log-likelihood function, the sum of log determinant of the innovation variance, and the weighted sum of squares of residuals. The log-likelihood function is computed as $-0.5\times$ *(the sum of last two terms)*.

The options *opt=0* and *opt=2* are equivalent for stationary time series without missing values. Setting *opt=0* is useful for a small number of the observations and a high order of *p* and *q*; *opt=1* is useful for a high order of *P* and *q*; *opt=2* is useful for a low order of *p* and *q*, or for missing values in the observations.

Consider the following bivariate ($k = 2$) VARMA(1,1) model:

$$\mathbf{y}_t = \Phi\mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t - \Theta\boldsymbol{\epsilon}_{t-1}$$

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To compute the log-likelihood function of this model, you can use the following statements:

```
phi  = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
sigma= { 1.0  0.5, 0.5 1.25};
call varmasim(yt, phi, theta) sigma=sigma;
call varmalik(lnl, yt, phi, theta, sigma);
```

VARMASIM Call

CALL VARMASIM(*series, phi, theta, mu, sigma, n* < , *p*> < , *q*> < , *initial*> < , *seed*> **);**

The VARMASIM subroutine generates a VARMA(p,q) time series.

The input arguments to the VARMASIM subroutine are as follows:

phi specifies a $km_p \times k$ matrix that contains the autoregressive coefficient matrices, where m_p is the number of the elements in the subset of the AR order and $k \geq 2$ is the number of variables. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

mu specifies a $k \times 1$ (or $1 \times k$) mean vector of the series. If *mu* is not specified, a zero vector is used.

sigma specifies a $k \times k$ covariance matrix of the innovation series. If *sigma* is not specified, an identity matrix is used.

n specifies the length of the series. If *n* is not specified, $n = 100$ is used.

p specifies the subset of the AR order. See the VARMACOV subroutine.

q specifies the subset of the MA order. See the VARMACOV subroutine.

initial specifies the initial values of random variables. If $initial = a_0$, then $\mathbf{y}_{-p+1}, \ldots, \mathbf{y}_0$ and $\boldsymbol{\epsilon}_{-q+1}, \ldots, \boldsymbol{\epsilon}_0$ all take the same value a_0. If the *initial* option is not specified, the initial values are estimated for the stationary vector time series; the initial values are assumed as zero for the nonstationary vector time series.

seed is a scalar that contains the random number seed. At the first execution of the subroutine, the seed variable is used as follows:

If *seed* > 0, the input seed is used for generating the series.

If *seed* = 0, the system clock is used to generate the seed.

If *seed* < 0, the value $(-1)\times$(*seed*) is used for generating the series.

If the seed is not supplied, the system clock is used to generate the seed.

On subsequent calls of the subroutine in the DO loop like environment the seed variable is used as follows: If seed > 0, the seed remains unchanged. In other cases, after each execution of the subroutine, the current seed is updated internally.

The VARMASIM subroutine returns the following value:

series is an $n \times k$ matrix that contains the generated VARMA(p, q) time series. When either the *initial* option is specified or zero initial values are used, these initial values are not included in *series*.

Consider the following bivariate ($k = 2$) stationary VARMA(1,1) time series:

$$\mathbf{y}_t - \boldsymbol{\mu} = \Phi(\mathbf{y}_{t-1} - \boldsymbol{\mu}) + \boldsymbol{\epsilon}_t - \Theta\boldsymbol{\epsilon}_{t-1}$$

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \boldsymbol{\mu} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To generate this series, you can use the following statements:

```
phi   = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
mu    = { 10, 20 };
sigma= { 1.0  0.5, 0.5 1.25};
call varmasim(yt, phi, theta, mu, sigma, 100);
```

Consider a bivariate ($k = 2$) nonstationary VARMA(1,1) time series with the same $\boldsymbol{\mu}$, Σ, and Θ in the previous example and the following AR coefficient:

$$\Phi = \begin{bmatrix} 1.0 & 0 \\ 0 & 0.3 \end{bmatrix}$$

To generate this series, you can use the following statements:

```
phi   = { 1.0 0.0, 0.0 0.3 };
call varmasim(yt, phi, theta, mu, sigma, 100) initial=3;
```

VECDIAG Function

VECDIAG(*matrix***);**

The VECDIAG function creates a column vector whose elements are the elements on the main diagonal of *matrix*. For example, the following statements produce the column vector shown in Figure 23.295:

```
a = {2 1, 0 -1};
d = vecdiag(a);
print d;
```

Figure 23.295 Diagonal of a Matrix

d
2
-1

VECH Function

VECH(*matrix***);**

The VECH function creates a column vector whose elements are the stacked columns of the lower triangular elements of *matrix*. Often, the argument is a symmetric matrix, in which case the VECH function has the effect of discarding the "duplicate" elements that are above the matrix diagonal.

Uses of the VECH function in matrix algebra are described in Harville (1997). "Vech" is an abbreviation for "vector-half."

The following statements produce the column vector shown in Figure 23.296:

```
a = {1 2 3, 4 5 6, 7 8 9};
v = vech(a);
print v;
```

Figure 23.296 Stacked Columns of Lower Triangular Matrix

v
1
4
7
5
8
9

The SQRVECH function and the VECH function are inverse operations on the set of symmetric matrices.

VNORMAL Call

CALL VNORMAL(*series, mu, sigma, n* < , *seed*> **);**

The VNORMAL subroutine generates a multivariate normal random series.

The input arguments to the VNORMAL subroutine are as follows:

mu specifies a $k \times 1$ (or $1 \times k$) mean vector, where $k \geq 2$ is the number of variables. You must specify either *mu* or *sigma*. If *mu* is not specified, a zero vector is used.

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*. If *sigma* is not specified, an identity matrix is used.

n specifies the length of the series. If *n* is not specified, $n = 100$ is used.

seed is a scalar that contains the random number seed. At the first execution of the subroutine, the seed variable is used as follows:

If *seed* > 0, the input seed is used for generating the series.

If *seed* = 0, the system clock is used to generate the seed.

If *seed* < 0, the value $(-1)\times$(*seed*) is used for generating the series.

If the seed is not supplied, the system clock is used to generate the seed.

On subsequent calls of the subroutine in the DO loop like environment the seed variable is used as follows: If seed > 0, the seed remains unchanged. In other cases, after each execution of the subroutine, the current seed is updated internally.

The VNORMAL subroutine returns the following value:

series is an $n \times k$ matrix that contains the generated normal random series.

Consider a bivariate ($k = 2$) normal random series with mean $\boldsymbol{\mu}$ and covariance matrix Σ, where

$$\boldsymbol{\mu} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \text{ and } \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To generate this series, you can use the following statements:

```
mu    = { 10, 20 };
sigma= { 1.0  0.5, 0.5 1.25};
call vnormal(et, mu, sigma, 100);
```

VTSROOT Call

CALL VTSROOT(*root, phi, theta* **<** *, p* **>** **<** *, q* **>** **);**

The VTSROOT subroutine computes the characteristic roots of the model from AR and MA characteristic functions.

The input arguments to the VTSROOT subroutine are as follows:

phi specifies a $km_p \times k$ matrix that contains the autoregressive coefficient matrices, where m_p is the number of the elements in the subset of the AR order and $k \geq 2$ is the number of variables. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

p specifies the subset of the AR order. See the VARMACOV subroutine.

q specifies the subset of the MA order. See the VARMACOV subroutine.

The VTSROOT subroutine returns the following value:

root is a $k(p_{max} + q_{max}) \times 5$ matrix, where p_{max} is the maximum order of the AR characteristic function and q_{max} is the maximum order of the MA characteristic function. The first kp_{max} rows refer to the results of the AR characteristic function; the last kq_{max} rows refer to the results of the MA characteristic function.

The first column contains the real parts, x, of eigenvalues of companion matrix associated with the AR(p_{max}) or MA(q_{max}) characteristic function; the second column contains the imaginary parts, y, of the eigenvalues; the third column contains the moduli of the eigenvalues, $\sqrt{x^2 + y^2}$; the fourth column contains the arguments ($\arctan(y/x)$) of the eigenvalues, measured in radians from the positive real axis. The fifth column contains the arguments expressed in degrees rather than radians.

Consider the roots of the characteristic functions, $\Phi(B) = I - \Phi B$ and $\Theta(B) = I - \Theta B$, where I is an identity matrix with dimension 2 and

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix}$$

To compute these roots, you can use the following statements:

```
phi   = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
call vtsroot(root, phi, theta);
```

WAVFT Call

CALL WAVFT(*decomp, data, opt* < *, levels* > **);**

The fast wavelet transform (WAVFT) subroutine computes a specified discrete wavelet transform of the input data by using the algorithm of Mallat (1989). This transform decomposes the input data into sets of detail and scaling coefficients defined at a number of scales or "levels."

The input data are used as scaling coefficients at the top level in the decomposition. The fast wavelet transform then recursively computes a set of detail and a set of scaling coefficients at the next lower level by respectively applying "low pass" and "high pass" conjugate mirror filters to the scaling coefficients at the current level. The number of coefficients in each of these new sets is approximately half the number of scaling coefficients at the level above them. Depending on the filters being used, a number of additional scaling coefficients, known as boundary coefficients, can be involved. These boundary coefficients are obtained by using a specified method to extend the sequence of interior scaling coefficients

Details of the discrete wavelet transform and the fast wavelet transformation algorithm are available in many references, including Mallat (1989), Daubechies (1992), and Ogden (1997).

The input arguments to the WAVFT subroutine are as follows:

data specifies the data to transform. These data must be in either a row or column vector.

opt refers to an options vector with the following components:

opt[1] specifies the boundary handling used in computing the wavelet transform. At each level of the wavelet decomposition, necessary boundary scaling coefficients are obtained by extending the interior scaling coefficients at that level as follows:

- 0 specifies extension by zero.
- 1 specifies periodic extension.
- 2 specifies polynomial extension.
- 3 specifies extension by reflection.
- 4 specifies extension by anti-symmetric reflection.

opt[2] specifies the polynomial degree that is used for polynomial extension. (The value of *opt[2]* is ignored if *opt[1]*$\neq$ 2.)

0 specifies constant extension.

1 specifies linear extension.

2 specifies quadratic extension.

opt[3] specifies the wavelet family.

1 specifies the Daubechies Extremal phase family (Daubechies 1992).

2 specifies the Daubechies Least Asymmetric family (also known as the Symmlet family) (Daubechies 1992).

opt[4] specifies the wavelet family member. Valid values are

1 through 10, if *opt[3]*=1

4 through 10, if *opt[3]*=2

Some examples of wavelet specifications are

opt={1 . 1 1}; specifies the first member (more commonly known as the Haar system) of the Daubechies extremal phase family with periodic boundary handling.

opt={2 1 2 5}; specifies the fifth member of the Symmlet family with linear extension boundary handling.

levels is an optional scalar argument that specifies the number of levels from the top level to be computed in the decomposition. If you do not specify this argument, then the decomposition terminates at level 0. Usually, you do not need to specify this optional argument. You use this option to avoid unneeded computations in situations where you are interested in only the higher level detail and scaling coefficients.

The WAVFT subroutine returns

decomp a row vector that encapsulates the specified wavelet transform. The information that is encoded in this vector includes:

- the options specified for computing the transform
- the number of detail coefficients at each level of the decomposition
- all detail coefficients
- the scaling coefficients at the bottom level of the decomposition
- boundary scaling coefficients at all levels of the decomposition

NOTE: *decomp* is a private representation of the specified wavelet transform and is not intended to be interpreted in its raw form. Rather, you should use this vector as an input argument to the WAVIFT, WAVPRINT, WAVGET, and WAVTHRSH subroutines

The following program shows an example that uses wavelet calls to estimate and reconstruct a piecewise constant function:

```
/* define a piecewise constant step function */
start blocky(t);
   /* positions (p) and magnitudes (h) of jumps */
   p = {0.1 0.13 0.15 0.23 0.25 0.4 0.44 0.65 0.76 0.78 0.81};
   h  = {4 -5 3 -4 5 -4.2 2.1 4.3 -3.1 2.1 -4.2};

   y=j(1, ncol(t), 0);
   do i=1 to ncol(p);
      diff=( (t-p[i])>=0 );
      y=y+h[i]*diff;
   end;
   return (y);
finish blocky;

n=2##8;
x=1:n;
x=(x-1)/n;
y=blocky(x);

opt = { 2, /* polynomial extension at boundary */
         1, /* using linear polynominal */
         1, /* Daubechies Extremal phase */
         3  /* family member 3 */
         };

call wavft(decomp, y, opt);
call wavprint(decomp,1); /* print summary information */


/* perform permanent thresholding */
threshOpt = { 2, /* soft thresholding */
              2, /* global threshold */
              ., /* ignored */
             -1  /* apply to all levels */
              };
call wavthrsh(decomp, threshOpt );

/* request detail coefficients at level 4 */
call wavget(detail4,decomp,2,4);

/* reconstruct function by using wavelets */
call wavift(estimate,decomp);

errorSS=ssq(y-estimate);
print errorSS;
```

Decomposition Summary

Decomposition Name	DECOMP
Wavelet Family	Daubechies Extremal Phase
Family Member	3
Boundary Treatment	Recursive Linear Extension
Number of Data Points	256
Start Level	0

```
ERRORSS

1.746E-25
```

WAVGET Call

CALL WAVGET(*result, decomp, request* < **,** *options* > **);**

The WAVGET subroutine is used to return information that is encoded in a wavelet decomposition.

The required input arguments are

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

request specifies a scalar that indicates what information is to be returned.

You can specify different optional arguments depending on the value of *request*:

request=1 requests the number of points in the input data vector.

result returns as a scalar that contains this number.

request=2 requests the detail coefficients at a specified level. Valid syntax is

CALL WAVGET(*result, decomp, 2, level* < **,** *opt* > **);**

The arguments are as follows:

level is the level at which the detail coefficients are requested.

opt is an optional vector which specifies the thresholding to be applied to the returned detail coefficients. See the WAVIFT subroutine for details. If you omit this argument, no thresholding is applied.

result returns as a column vector that contains the specified detail coefficients.

request=3 requests the scaling coefficients at a specified level. Valid syntax is

CALL WAVGET(*result, decomp, 3, level* < **,** *opt* > **);**

The arguments are as follows:

level is the level at which the scaling coefficients are requested.

opt is an optional vector that specifies the thresholding to be applied. See the WAVIFT subroutine for a description of this vector. The scaling coefficients at the requested level are obtained by using the inverse wavelet transform, after applying the specified thresholding. If you omit this argument, no thresholding is applied.

result returns as a column vector that contains the specified scaling coefficients.

request=4 requests the thresholding status of the detail coefficients in *decomp*.

result returns as a scalar whose value is

0, if the detail coefficients have not been thresholded

1, otherwise

request=5 requests the wavelet options vector that you specified in the WAVFT subroutine to compute *decomp*.

result returns as a column vector with 4 elements that contains the specified options vector. See the WAVFT subroutine for the interpretation of the vector entries.

request=6 requests the index of the top level in *decomp*.

result returns as a scalar that contains this number.

request=7 requests the index of the lowest level in *decomp*.

result returns as a scalar that contains this number.

request=8 requests a vector evaluating the father wavelet used in *decomp*, at an equally spaced grid spanning the support of the father wavelet. The number of points in the grid is specified as a power of 2 times the support width of the father wavelet. For wavelets in the Daubechies extremal phase and least asymmetric families, the support width of the father wavelet is $2m - 1$, where m is the family member. Valid syntax is

CALL WAVGET(*result, decomp, 8 < , power* **>);**

The optional argument has the following meaning:

power is the exponent of 2 that determines the number of grid points used. *power* defaults to 8 if you do not specify this argument.

result returns as a column vector that contains the specified evaluation of the father wavelet.

An example is available in the documentation for the WAVFT subroutine.

WAVIFT Call

CALL WAVIFT(*result, decomp* **<** *, opt* **>** **<** *, level* **>** **);**

The Inverse Fast Wavelet Transform (WAVIFT) subroutine computes the inverse wavelet transform of a wavelet decomposition computed by using the WAVFT subroutine. Details of this algorithm are available in many references, including Mallat (1989), Daubechies (1992), and Ogden (1997).

The inverse transform yields an exact reconstruction of the original input data, provided that no smoothing is specified. Alternatively, a smooth reconstruction of the input data can be obtained by thresholding the detail coefficients in the decomposition prior to applying the inverse transformation. Thresholding, also known as shrinkage, replaces the detail coefficient $d_j^{(i)}$ at level i by $\delta_{T_i}(d_j^{(i)})$, where the $\delta_T(x)$ is a shrinkage function

and T_i is the threshold value used at level i. The wavelet subroutines support hard and soft shrinkage functions (Donoho and Johnstone 1994) and the nonnegative garrote shrinkage function (Breiman 1995). These functions are defined as follows:

$$\delta_T^{\text{hard}}(x) = \begin{cases} 0 & |x| \le T \\ x & |x| > T \end{cases}$$

$$\delta_T^{\text{soft}}(x) = \begin{cases} 0 & |x| \le T \\ x - T & x > T \\ x + T & x < -T \end{cases}$$

$$\delta_T^{\text{garrote}}(x) = \begin{cases} 0 & |x| \le T \\ x - T^2/x & |x| > T \end{cases}$$

You can specify several methods for choosing the threshold values. Methods in which the threshold T_i varies with the level i are called adaptive. Methods where the same threshold is used at all levels are called global.

The input arguments to the WAVIFT subroutine are as follows:

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

opt refers to an options vector that specifies the thresholding algorithm. If this optional argument is not specified, then no thresholding is applied.

The options vector has the following components:

opt[1] specifies the thresholding policy.

- 0 specifies that no thresholding be done. If *opt[1]*=0 then all other entries in the options vector are ignored.
- 1 specifies hard thresholding.
- 2 specifies soft thresholding.
- 3 specifies garrote thresholding.

opt[2] specifies the method for selecting the threshold.

- 0 specifies a global user-supplied threshold.
- 1 specifies a global threshold chosen by using the minimax criterion of Donoho and Johnstone (1994).
- 2 specifies a global threshold defined by using the universal criterion of Donoho and Johnstone (1994).
- 3 specifies an adaptive method where the thresholds at each level i are chosen to minimize an approximation of the L^2 risk in estimating the true data values by using the reconstruction with thresholded coefficients (Donoho and Johnstone 1995).

4 specifies a hybrid method of Donoho and Johnstone (1995). The universal threshold as specified by *opt[2]*=2 is used at levels where most of the detail coefficients are essentially zero. The risk minimization method as specified by *opt[2]*=4 is used at all other levels.

opt[3] specifies the value of the global user-supplied threshold if *opt[2]*=1. It is ignored if *opt[2]* $\neq$ 1.

opt[4] specifies the number of levels starting at the highest detail coefficient level at which thresholding is to be applied. If this value is negative or missing, thresholding is applied at all levels in *decomp*.

Some common examples of threshold options specifications are:

opt={1 3 . -1}; specifies hard thresholding with a minimax threshold applied at all levels in the decomposition. This threshold is named "*RiskShrink*" in Donoho and Johnstone (1994).

opt={2 2 . -1}; specifies soft thresholding with a universal threshold applied at all levels in the decomposition. This threshold is named "*VisuShrink*" in Donoho and Johnstone (1994).

opt={2 4 . -1}; specifies soft thresholding with level dependent thresholds which minimize the Stein Unbiased Estimate of Risk (SURE). This threshold is named "*SureShrink*" in Donoho and Johnstone (1995).

level is an optional scalar argument that specifies the level at which the reconstructed data are to be returned. If this argument is not specified then the reconstructed data are returned at the top level defined in *decomp*.

The WAVIFT subroutine returns

result a vector obtained by inverting, after thresholding the detail coefficients, the discrete wavelet transform encoded in *decomp*. The row or column orientation of *result* is the same as that of the input data specified in the corresponding WAVFT subroutine. If you specify the optional *level* argument, *result* contains the reconstruction at the specified level, otherwise the reconstruction corresponds to the top level in the decomposition.

An example is available in the documentation for the WAVFT subroutine.

WAVPRINT Call

CALL WAVPRINT(*decomp, request* < *, options* > **);**

The WAVPRINT subroutine is used to display the information that is encoded in a wavelet decomposition.

The required input arguments are

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

request specifies a scalar that indicates what information is to be displayed.

You can specify different optional arguments depending on the value of *request*:

request=1 displays information about the wavelet family used to perform the wavelet transform. No additional arguments need to be specified.

request=2 displays the detail coefficients by level. Valid syntax is

CALL WAVPRINT(*decomp, 2 < , lower> < , upper>* **);**

The optional arguments are as follows:

lower specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.

upper specifies the upper level to be displayed. The default value of *upper* is the highest detail level in *decomp*.

request=3 displays the scaling coefficients by level. Valid syntax is

CALL WAVPRINT(*decomp,3 < , lower> < , upper>* **);**

The optional arguments are as follows:

lower and specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.

upper specifies the upper level to be displayed. The default value of *upper* is the top level in *decomp*.

request=4 displays thresholded detail coefficients by level. Valid syntax is

CALL WAVPRINT(*decomp, 4 < , opt> < , lower> < , upper>* **);**

The optional arguments are as follows:

opt specifies the thresholding to be applied to the displayed detail coefficients. See the WAVIFT subroutine for details. If you omit this argument, no thresholding is applied.

lower specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.

upper specifies the upper level to be displayed. The default value of *upper* is the highest detail level in *decomp*.

An example is available in the documentation for the WAVFT subroutine.

WAVTHRSH Call

CALL WAVTHRSH(*decomp, opt* **);**

The wavelet threshold (WAVTHRSH) subroutine thresholds the detail coefficients in a wavelet decomposition.

The required input arguments are

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

opt refers to an options vector that specifies the thresholding algorithm used. See the WAVIFT subroutine for a description of this options vector.

On return, the detail coefficients encoded in *decomp* are replaced by their thresholded values. Note that this action is not reversible. If you want to retain the original detail coefficients, you should not use the WAVTHRSH subroutine to do thresholding. Rather, you should supply the thresholding argument where appropriate in the WAVIFT, WAVGET, and WAVPRINT subroutines.

An example is available in the documentation for the WAVFT subroutine.

WINDOW Statement

WINDOW <**CLOSE**=*window-name*> <*window-options*> <**GROUP**=*group-name field-specs*> <. . . **GROUP**=*group-name field-specs*> **;**

The WINDOW statement defines and opens 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 that contains the number, or an expression in parentheses that yields the number. The default value is 23 rows.

Both the WINDOW and DISPLAY statements accept field specifications, which have the following general form:

< positionals > field-operand < format > < field-options > **;**

The arguments to these statements are as follows:

positionals	are directives determining the position on the screen to begin the field. There are four kinds of positionals; any number of positionals are accepted 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

XMULT(*matrix1, matrix2***);**

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.

The following program uses the XMULT function:

```
a=1e13;
b=1e13;
c=100*a;
a=a+1;
x=c || a || b || c;
y=c || a || (-b) || (-c);

z=xmult(x,y`);  /* correct answer */
print z [format=16.0];

wrong = x * y`;  /* loss of precision */
print wrong [format=16.0];
```

```
                    Z

20000000000001

    WRONG

19997367730176
```

XSECT Function

XSECT(*matrix1 < , matrix2, . . . , matrix15* **>);**

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 following statements return the result shown:

```
a={1 2 4 5};
b={3 4};
c=xsect(a,b);
```

```
              C             1 row       1 col     (numeric)

                                        4
```

YIELD Function

YIELD(*times, flows, freq, value***);**

The YIELD function returns a scalar that contains 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 nonnegative.
flows	is an n-dimensional column vector of cash flows.
freq	is a scalar that 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 accepted.
value	is a scalar that 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 kth cash flow in periods from the present, and $D(t)$ is the discount function for time t.

With continuous compounding:

$$D(t) = e^{-yt}$$

With discrete compounding:

$$D(t) = (1 + fy)^{-t/f}$$

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.

For example, the following statements produce the output shown in Figure 23.297:

```
timesn = T(do(1, 100, 1));
flows = repeat(10, 100);
freq = 50;
value = 682.31027;
yield = yield(timesn, flows, freq, value);
print yield;
```

Figure 23.297 Yield to Maturity

yield
0.01

Base SAS Functions Accessible from SAS/IML Software

You can call most functions available in Base SAS software from SAS/IML programs. If you call a Base SAS function with a matrix argument, the function will usually act elementwise on each element of teh matrix.

The following Base SAS functions are either not available from SAS/IML software, or behave differently from the Base SAS function of the same name.

Function	Comment
CALL CATS	return variable must be preinitialized
DIF*n*	not supported; use the SAS/IML DIF function instead.
DIM	not supported
HBOUND	not supported
LAG*n*	not supported; use the SAS/IML LAG function instead.
LBOUND	not supported
MOD	base function performs "fuzzing;" the SAS/IML function does not

PUT	Use the PRINT statement instead
CALL PRXNEXT	return variables must be preinitialized
CALL PRXPOSN	return variables must be preinitialized
CALL PRXSUBSTR	return variables must be preinitialized
CALL RXCHANGE	return variables must be preinitialized
CALL RXMATCH	return variables must be preinitialized
CALL RXSUBSTR	return variables must be preinitialized
CALL SCAN	return variables must be preinitialized
CALL SCANQ	return variable must be preinitialized
VVALUE	not applicable: interrogates DATA step variables
VVALUEX	not applicable: interrogates DATA step variables
VNEXT	not applicable: interrogates DATA step variables

There are also some Base SAS features that are not supported by the SAS/IML language. For example, the DATA step permits N-literals (strings that end with 'N') to be interpreted as the name of a variable, but the SAS/IML language does not.

The following Base SAS functions can be called from SAS/IML. The functions are documented in the *SAS Language Reference: Dictionary*. In some cases, SAS/IML does not accept all variations in the syntax. For example, SAS/IML does not accept the OF keyword as a way to generate an argument list in a function.

The functions displayed in italics are documented elsewhere in this user's guide. These functions operate on matrices in addition to scalar values, as do many of the mathematical and statistical functions.

Bitwise Logical Operation Functions

BAND	returns the bitwise logical AND of two arguments
BLSHIFT	performs a bitwise logical left shift of an argument by a specified amount
BNOT	returns the bitwise logical NOT of an argument
BOR	returns the bitwise logical OR of two arguments
BRSHIFT	performs a bitwise logical right shift of an argument by a specified amount
BXOR	returns the bitwise logical EXCLUSIVE OR of two arguments

Character and Formatting Functions

ANYALNUM	searches a character string for an alphanumeric character and returns the first position at which it is found
ANYALPHA	searches a character string for an alphabetic character and returns the first position at which it is found

ANYCNTRL	searches a character string for a control character and returns the first position at which it is found
ANYDIGIT	searches a character string for a digit and returns the first position at which it is found
ANYFIRST	searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
ANYGRAPH	searches a character string for a graphical character and returns the first position at which it is found
ANYLOWER	searches a character string for a lowercase letter and returns the first position at which it is found
ANYNAME	searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
ANYPRINT	searches a character string for a printable character and returns the first position at which it is found
ANYPUNCT	searches a character string for a punctuation character and returns the first position at which it is found
ANYSPACE	searches a character string for a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found
ANYUPPER	searches a character string for an uppercase letter and returns the first position at which it is found
ANYXDIGIT	searches a character string for a hexadecimal character that represents a digit and returns the first position at which that character is found
BYTE	returns one character in the ASCII or EBCDIC collating sequence
CAT	concatenates character strings without removing leading or trailing blanks
CATS	concatenates character strings and removes leading and trailing blanks
CALL CATS	concatenates character strings and removes leading and trailing blanks
CATT	concatenates character strings and removes trailing blanks
CALL CATT	concatenates character strings and removes trailing blanks
CATX	concatenates character strings, removes leading and trailing blanks, and inserts separators
CALL CATX	concatenates character strings, removes leading and trailing blanks, and inserts separators
CHOOSEC	returns a character value that represents the results of choosing from a list of arguments
CHOOSEN	returns a numeric value that represents the results of choosing from a list of arguments
COLLATE	returns an ASCII or EBCDIC collating sequence character string
COMPARE	returns the position of the left-most character by which two strings differ, or returns 0 if there is no difference
COMPBL	removes multiple blanks from a character string

CALL COMPCOST	sets the costs of operations for later use by the COMPGED function
COMPGED	compares two strings by computing the generalized edit distance
COMPLEV	compares two strings by computing the Levenshtein edit distance
COMPRESS	removes specific characters from a character string
COUNT	counts the number of times that a specific substring of characters appears within a character string that you specify
COUNTC	counts the number of specific characters that either appear or do not appear within a character string that you specify
COUNTQ	counts the number of "words" and quoted strings in a character expression
COUNTW	counts the number of words in a character expression
FIND	searches for a specific substring of characters within a character string that you specify
FINDC	searches for specific characters that either appear or do not appear within a character string that you specify
IFC	returns a character value that matches an expression
IFN	returns a numeric value that matches an expression
INDEX	searches a character expression for a string of characters
INDEXC	searches a character expression for specific characters
INDEXW	searches a character expression for a specified string as a word
INPUTC	applies a character informat at run time
INPUTN	applies a numeric informat at run time
LEFT	left aligns a character expression
LENGTH	returns the length of a character string
LENGTHC	returns the length of a character string, including trailing blanks
LENGTHM	returns the amount of memory (in bytes) that is allocated for a character string
LENGTHN	returns the length of a nonblank character string, excluding trailing blanks, and returns 0 for a blank character string
LOWCASE	converts all letters in an argument to lowercase
CALL MISSING	assigns a missing value to the specified character or numeric variable
NLITERAL	converts a character string that you specify to a SAS name literal (N-literal)
NOTALNUM	searches a character string for a nonalphanumeric character and returns the first position at which it is found
NOTALPHA	searches a character string for a nonalphabetic character and returns the first position at which it is found
NOTCNTRL	searches a character string for a character that is not a control character and returns the first position at which it is found
NOTDIGIT	searches a character string for any character that is not a digit and returns the first position at which that character is found
NOTFIRST	searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
NOTGRAPH	searches a character string for a nongraphical character and returns the first position at which it is found

NOTLOWER	searches a character string for a character that is not a lowercase letter and returns the first position at which that character is found
NOTNAME	searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
NOTPRINT	searches a character string for a nonprintable character and returns the first position at which it is found
NOTPUNCT	searches a character string for a character that is not a punctuation character and returns the first position at which it is found
NOTSPACE	searches a character string for a character that is not a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found
NOTUPPER	searches a character string for a character that is not an uppercase letter and returns the first position at which that character is found
NOTXDIGIT	searches a character string for a character that is not a hexadecimal digit and returns the first position at which that character is found
NVALID	checks a character string for validity for use as a SAS variable name in a SAS statement
PROPCASE	converts all words in an argument to proper case
PUTC	applies a character format at run time
PUTN	applies a numeric format at run time
REPEAT	repeats a character expression
REVERSE	reverses a character expression
RIGHT	right aligns a character expression
SCAN	selects a given word from a character expression
CALL SCAN	returns the position and length of a given word from a character expression
CALL SCANQ	returns the position and length of a given word from a character expression, and ignores delimiters that are inside quotation marks
ROUNDEX	encodes a string to facilitate searching
SPEDIS	determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words
STRIP	returns a character string with all leading and trailing blanks removed
SUBPAD	returns a substring that has specified length and is padded with blanks, if necessary
SUBSTRN	returns a substring, allowing a result with a length of zero
SUBSTR	extracts substrings of character expressions
TRANSLATE	replaces specific characters in a character expression
TRANWRD	replaces or removes all occurrences of a word in a character string
TRIM	removes trailing blanks from character expressions and returns one blank if the expression is missing
TRIMN	removes trailing blanks from character expressions and returns a null string (zero blanks) if the expression is missing
UPCASE	converts all letters in an argument to uppercase
UUIDGEN	returns the short or binary form of a Universal Unique Identifier (UUID)

VERIFY	returns the position of the first character that is unique to an expression

Character String Matching Functions and Subroutines

CALL RXCHANGE	changes one or more substrings that match a pattern
CALL RXFREE	frees memory allocated by other regular expression (RX) functions and CALL routines
RXMATCH	finds the beginning of a substring that matches a pattern
RXPARSE	parses a pattern
CALL RXSUBSTR	finds the position, length, and score of a substring that matches a pattern
CALL PRXCHANGE	performs a pattern matching substitution
CALL PRXDEBUG	enables Perl regular expressions in a DATA step to send debug output to the SAS log
CALL PRXFREE	frees unneeded memory that was allocated for a Perl regular expression
PRXMATCH	searches for a pattern match and returns the position at which the pattern is found
CALL PRXNEXT	returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string
PRXPAREN	returns the last bracket match for which there is a match in a pattern
PRXPARSE	compiles a Perl regular expression that can be used for pattern matching of a character value
CALL PRXPOSN	returns the start position and length for a capture buffer
CALL PRXSUBSTR	returns the position and length of a substring that matches a pattern

Date and Time Functions

DATDIF	returns the number of days between two dates
DATE	returns the current date as a SAS date value
DATEJUL	converts a Julian date to a SAS date value
DATEPART	extracts the date from a SAS datetime value
DATETIME	returns the current date and time of day as a SAS datetime value
DAY	returns the day of the month from a SAS date value
DHMS	returns a SAS datetime value from date, hour, minute, and seconds
HMS	returns a SAS time value from hour, minute, and seconds
HOUR	returns the hour from a SAS time or datetime value
INTCK	returns the integer number of time intervals in a given time span
INTNX	advances a date, time, or datetime value by a given interval, and returns a date, time, or datetime value
JULDATE	returns the Julian date from a SAS date value
JULDATE7	returns a seven-digit Julian date from a SAS date value

MDY	returns a SAS date value from month, day, and year values
MINUTE	returns the minute from a SAS time or datetime value
MONTH	returns the month from a SAS date value
QTR	returns the quarter of the year from a SAS date value
SECOND	returns the second from a SAS time or datetime value
TIME	returns the current time of day
TIMEPART	extracts a time value from a SAS datetime value
TODAY	returns the current date as a SAS date value
WEEKDAY	returns the day of the week from a SAS date value
YEAR	returns the year from a SAS date value
YRDIF	returns the difference in years between two dates
YYQ	returns a SAS date value from the year and quarter

Descriptive Statistics Functions and Subroutines

CMISS	returns the number of nonmissing values
CSS	returns the corrected sum of squares
CV	returns the coefficient of variation
GEOMEAN	returns the geometric mean
EUCLID	returns the Euclidean norm of the nonmissing arguments
GEOMEANZ	returns the geometric mean without fuzzing the values of the arguments that are approximately 0
HARMEAN	returns the harmonic mean
HARMEANZ	returns the harmonic mean without fuzzing the values of the arguments that are approximately 0
IQR	returns the interquartile range
KURTOSIS	returns the kurtosis
LARGEST	returns the kth largest nonmissing value
LPNORM	returns the L_p norm of the nonmissing arguments
MAX	returns the largest value
MAD	returns the median absolute deviation from the median
MEDIAN	computes median values
MEAN	returns the arithmetic mean (average)
MIN	returns the smallest value
N	returns the number of nonmissing values
ORDINAL	returns any specified order statistic
PCTL	computes percentiles
RANGE	returns the range of values
RMS	returns the root mean square
SKEWNESS	returns the skewness
SMALLEST	returns the kth smallest nonmissing value
SUM	returns the sum of the nonmissing arguments
STD	returns the standard deviation
CALL STDIZE	standardizes the values of one or more variables
STDERR	returns the standard error of the mean

SUMABS	returns the sum of the absolute values of the nonmissing arguments
USS	returns the uncorrected sum of squares
VAR	returns the variance

Double-Byte Character String Functions

Many of the Base SAS character functions have analogous companion functions that take double-byte character strings (DBCS) as arguments. These functions (for example, KCOMPARE, KCVT, KINDEX, and KSUBSTR) are accessible from SAS/IML. See the *SAS Language Reference: Dictionary* for a complete list of DBCS functions.

External Files Functions

DROPNOTE	deletes a note marker from a SAS data set or an external file and returns a value
EXIST	verifies the existence of a SAS data library member
FAPPEND	appends the current record to the end of an external file and returns a value
FCLOSE	closes an external file, directory, or directory member, and returns a value
FCOL	returns the current column position in the File Data Buffer (FDB)
FDELETE	deletes an external file or an empty directory
FEXIST	verifies the existence of an external file associated with a fileref and returns a value
FGET	copies data from the File Data Buffer (FDB) into a variable and returns a value
FILEEXIST	verifies the existence of an external file by its physical name and returns a value
FILENAME	assigns or deassigns a fileref for an external file, directory, or output device and returns a value
FILEREF	verifies that a fileref has been assigned for the current SAS session and returns a value
FINFO	returns the value of a file information item
FNOTE	identifies the last record that was read and returns a value that FPOINT can use
FOPEN	opens an external file and returns a file identifier value
FOPTNAME	returns the name of an item of information about a file
FOPTNUM	returns the number of information items that are available for an external file
FPOINT	positions the read pointer on the next record to be read and returns a value
FPOS	sets the position of the column pointer in the File Data Buffer (FDB) and returns a value

FPUT	moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position, and returns a value
FREAD	reads a record from an external file into the File Data Buffer (FDB) and returns a value
FREWIND	positions the file pointer to the start of the file and returns a value
FRLEN	returns the size of the last record read, or, if the file is opened for output, returns the current record size
FSEP	sets the token delimiters for the FGET function and returns a value
FWRITE	writes a record to an external file and returns a value
MOPEN	opens a file by directory identifier and member name, and returns the file identifier or a 0
PATHNAME	returns the physical name of a SAS data library or of an external file, or returns a blank
SYSMSG	returns the text of error messages or warning messages from the last data set or external file function execution
SYSRC	returns a system error number

File I/O Functions

ATTRC	returns the value of a character attribute for a SAS data set
ATTRN	returns the value of a numeric attribute for the specified SAS data set
CEXIST	verifies the existence of a SAS catalog or SAS catalog entry and returns a value
CLOSE	closes a SAS data set and returns a value
CUROBS	returns the observation number of the current observation
DROPNOTE	deletes a note marker from a SAS data set or an external file and returns a value
DSNAME	returns the SAS data set name that is associated with a data set identifier
EXIST	verifies the existence of a SAS data library member
FETCH	reads the next nondeleted observation from a SAS data set into the Data Set Data Vector (DDV) and returns a value
FETCHOBS	reads a specified observation from a SAS data set into the Data Set Data Vector (DDV) and returns a value
GETVARC	returns the value of a SAS data set character variable
GETVARN	returns the value of a SAS data set numeric variable
LIBNAME	assigns or deassigns a libref for a SAS data library and returns a value
LIBREF	verifies that a libref has been assigned and returns a value
NOTE	returns an observation ID for the current observation of a SAS data set
OPEN	opens a SAS data set and returns a value
PATHNAME	returns the physical name of a SAS data library or of an external file, or returns a blank

POINT	locates an observation identified by the NOTE function and returns a value
REWIND	positions the data set pointer at the beginning of a SAS data set and returns a value
SYSMSG	returns the text of error messages or warning messages from the last data set or external file function execution
SYSRC	returns a system error number
VARFMT	returns the format assigned to a SAS data set variable
VARINFMT	returns the informat assigned to a SAS data set variable
VARLABEL	returns the label assigned to a SAS data set variable
VARLEN	returns the length of a SAS data set variable
VARNAME	returns the name of a SAS data set variable
VARNUM	returns the number of a variable's position in a SAS data set
VARTYPE	returns the data type of a SAS data set variable

Financial Functions

COMPOUND	returns compound interest parameters
CONVX	returns the convexity for an enumerated cash flow
CONVXP	returns the convexity for a periodic cash flow stream
DACCDB	returns the accumulated declining balance depreciation
DACCDBSL	returns the accumulated declining balance with conversion to a straight-line depreciation
DACCSL	returns the accumulated straight-line depreciation
DACCSYD	returns the accumulated sum-of-years-digits depreciation
DACCTAB	returns the accumulated depreciation from specified tables
DEPDB	returns the declining balance depreciation
DEPDBSL	returns the declining balance with conversion to a straight-line depreciation
DEPSL	returns the straight-line depreciation
DEPSYD	returns the sum-of-years-digits depreciation
DEPTAB	returns the depreciation from specified tables
DUR	returns the modified duration for an enumerated cash flow
INTRR	returns the internal rate of return as a decimal
IRR	returns the internal rate of return as a percentage
MORT	returns amortization parameters
NETPV	returns the net present value as a decimal
NPV	returns the net present value as a percentage
PVP	returns the present value for a periodic cash flow stream
SAVING	returns the future value of a periodic saving
YIELDP	returns the yield-to-maturity for a periodic cash flow stream

Macro Functions and Subroutines

CALL RESOLVE	resolves the value of a text expression at execution time
SYMGET	returns the character value of a macro variable
SYMGETN	returns the numeric value of a macro variable
SYMEXIST	indicates the existence of a macro variable
CALL SYMPUT	sets the character value of a macro variable
CALL SYMPUTX	assigns a value to a macro variable and removes both leading and trailing blanks

Mathematical Functions and Subroutines

ALLCOMB	generates all combinations of n elements taken k at a time
ALLPERM	generates all permutations of n elements
ABS	returns the absolute value
AIRY	returns the Airy function
BETA	returns the value of the beta function.
COALESCE	returns the first non-missing value from a list of numeric arguments
COALESCEC	returns the first non-missing value from a list of character arguments
COMB	returns the number of combinations of n items taken r at a time
COMPFUZZ	returns the result of a fuzzy comparison of numeric values
CONSTANT	returns some machine and mathematical constants
CNONCT	returns the noncentrality parameter from a chi-squared distribution
DAIRY	returns the derivative of the Airy function
DEVIANCE	returns the deviance from a specified distribution
DIGAMMA	returns the DIGAMMA function
ERF	returns the normal error function
ERFC	returns the complementary normal error function
EXP	returns the exponential function
FACT	returns the factorial of an integer
FNONCT	returns the noncentrality parameter of an F distribution
GAMMA	returns the gamma function
IBESSEL	returns a modified Bessel function
JBESSEL	returns a Bessel function
LOGBETA	returns the logarithm of the beta function
LGAMMA	returns the natural logarithm of the gamma function
LOG	returns the natural (base e) logarithm
LOG2	returns the logarithm base 2
LOG10	returns the logarithm base 10
CALL LOGISTIC	returns the logistic value of each argument
MOD	returns the remainder value
RANCOMB	returns random combinations of n elements taken k at a time

CALL RANPERK	randomly permutes the values of the arguments, and returns a permutation of k out of n values
RANPERM	returns random permutations of n elements
PERM	returns the number of permutations of n items taken r at a time
SIGN	returns the sign of a value
CALL SOFTMAX	returns the softmax value for each argument
SQRT	returns the square root of a value
TNONCT	returns the value of the noncentrality parameter from the student's t distribution
TRIGAMMA	returns the value of the TRIGAMMA function

Probability Functions

CDF	computes cumulative distribution functions
LOGCDF	returns the logarithm of a left cumulative distribution function
LOGPDF	computes the logarithm of a probability function
LOGSDF	computes the logarithm of a survival function
PDF	computes probability density functions
POISSON	returns the probability from a Poisson distribution
PROBBETA	returns the probability from a beta distribution
PROBBNML	returns the probability from a binomial distribution
PROBBNRM	returns the probability from the bivariate normal distribution
PROBCHI	returns the probability from a chi-squared distribution
PROBF	returns the probability from an F distribution
PROBGAM	returns the probability from a gamma distribution
PROBHYPR	returns the probability from a hypergeometric distribution
PROBMC	returns a probability or a quantile from various distributions for multiple comparisons of means
PROBNEGB	returns the probability from a negative binomial distribution
PROBNORM	returns the probability from the standard normal distribution
PROBT	returns the probability from a t distribution
SDF	computes a survival function

Quantile Functions

BETAINV	returns a quantile from the beta distribution
CINV	returns a quantile from the chi-squared distribution
FINV	returns a quantile from the F distribution
GAMINV	returns a quantile from the gamma distribution
PROBIT	returns a quantile from the standard normal distribution
QUANTILE	returns the quantile from the specified distribution
TINV	returns a quantile from the t distribution

Random Number Functions and Subroutines

NORMAL	returns a random variate from a normal distribution
RANBIN	returns a random variate from a binomial distribution
RANCAU	returns a random variate from a Cauchy distribution
RAND	returns a random variate from a specified distribution. (See the RANDGEN subroutine.)
RANEXP	returns a random variate from an exponential distribution
RANGAM	returns a random variate from a gamma distribution
RANNOR	returns a random variate from a normal distribution
RANPOI	returns a random variate from a Poisson distribution
RANTBL	returns a random variate from a tabled probability
RANTRI	returns a random variate from a triangular distribution
RANUNI	returns a random variate from a uniform distribution
CALL STREAMINIT	specifies a seed value to use for subsequent random number generation by the RAND function. (See the RANDSEED subroutine.)
UNIFORM	returns a random variate from a uniform distribution

State and Zip Code Functions

FIPNAME	converts FIPS codes to uppercase state names
FIPNAMEL	converts FIPS codes to mixed-case state names
FIPSTATE	converts FIPS codes to two-character postal codes
STFIPS	converts state postal codes to FIPS state codes
STNAME	converts state postal codes to uppercase state names
STNAMEL	converts state postal codes to mixed-case state names
ZIPCITY	returns a city name and the two-character postal code that corresponds to a zip code
ZIPCITYDISTANCE	returns the geodetic distance between two zip code locations
ZIPFIPS	converts zip codes to FIPS state codes
ZIPNAME	converts zip codes to uppercase state names
ZIPNAMEL	converts zip codes to mixed-case state names
ZIPSTATE	converts zip codes to state postal codes

Trigonometric and Hyperbolic Functions

ARCOS	returns the arccosine
ARSIN	returns the arcsine
ATAN	returns the arctangent
ATAN2	returns the arc tangent of two numeric variables
COS	returns the cosine

COSH	returns the hyperbolic cosine
SIN	returns the sine
SINH	returns the hyperbolic sine
TAN	returns the tangent
CALL TANH	returns the hyperbolic tangent of each argument
TANH	returns the hyperbolic tangent

Truncation Functions

CEIL	returns the smallest integer ≥ the argument
CEILZ	returns the smallest integer that is greater than or equal to the argument, using zero fuzzing
FLOOR	returns the largest integer ≤ the argument
FLOORZ	returns the largest integer that is less than or equal to the argument, using zero fuzzing
FUZZ	returns the nearest integer if the argument is within 1E-12
INT	returns the integer portion of a value
INTZ	returns the integer portion of the argument, using zero fuzzing
MODZ	returns the remainder from the division of the first argument by the second argument, using zero fuzzing
ROUND	rounds a value to the nearest round-off unit
ROUNDE	rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples
ROUNDZ	rounds the first argument to the nearest multiple of the second argument, with zero fuzzing
TRUNC	returns a truncated numeric value of a specified length

Web Tools

HTMLDECODE	decodes a string that contains HTML numeric character references or HTML character entity references and returns the decoded string
HTMLENCODE	encodes characters by using HTML character entity references and returns the encoded string
URLDECODE	returns a string that was decoded by using the URL escape syntax
URLENCODE	returns a string that was encoded by using the URL escape syntax

References

Abramowitz, M. and Stegun, I. A. (1972), *Handbook of Mathematical Functions*, New York: Dover Publications.

Aiken, R. C. (1985), *Stiff Computation*, New York: Oxford University Press.

Al-Baali, M. and Fletcher, R. (1985), "Variational Methods for Nonlinear Least Squares," *Journal of the Operations Research Society*, 36, 405–421.

Al-Baali, M. and Fletcher, R. (1986), "An Efficient Line Search for Nonlinear Least Squares," *Journal of Optimization Theory and Applications*, 48, 359–377.

Ansley, C. (1979), "An Algorithm for the Exact Likelihood of a Mixed Autoregressive-Moving Average Process," *Biometrika*, 66, 59–65.

Ansley, C. F. (1980), "Computation of the Theoretical Autocovariance Function for a Vector ARMA Process," *Journal of Statistical Computation and Simulation*, 12, 15–24.

Ansley, C. F. and Kohn, R. (1986), "A Note on Reparameterizing a Vector Autoregressive Moving Average Model to Enforce Stationary," *Journal of Statistical Computation and Simulation*, 24, 99–106.

Barnett, V. and Lewis, T. (1978), *Outliers in Statistical Data*, New York: John Wiley & Sons.

Barreto, H. and Maharry, D. (2006), "Least Median of Squares and Regression Through the Origin," *Computatitional Statistics and Data Analysis*, 50, 1391–1397.

Barrodale, I. and Roberts, F. D. K. (1974), "Algorithm 478: Solution of an Overdetermined System of Equations in the L_1-Norm," *Communications ACM*, 17, 319–320.

Bates, D., Lindstrom, M., Wahba, G., and Yandell, B. (1987), "GCVPACK-Routines for Generalized Cross Validation," *Communications in Statistics: Simulation and Computation*, 16, 263–297.

Beale, E. M. L. (1972), "A Derivation of Conjugate Gradients," in F. A. Lootsma, ed., *Numerical Methods for Nonlinear Optimization*, London: Academic Press.

Beaton, A. E. (1964), *The Use of Special Matrix Operations in Statistical Calculus*, Princeton: Educational Testing Service.

Bickart, T. A. and Picel, Z. (1973), "High Order Stiffly Stable Composit Multistep Methods for Numerical Integration of Stiff Differential Equations," *BIT*, 13, 272–286.

Bishop, Y. M. 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*, Revised Edition, San Francisco: Holden-Day.

Breiman, L. (1995), "Better Subset Regression Using the Nonnegative Garrote," *Technometrics*, 37, 373–384.

Brockwell, P. J. and Davis, R. A. (1991), *Time Series: Theory and Methods*, Second Edition, New York: Springer-Verlag.

Brownlee, K. A. (1965), *Statistical Theory and Methodology in Science and Engineering*, New York: John Wiley & Sons.

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.

Christensen, R. (1997), *Log-Linear Models and Logistic Regression*, Second Edition, New York: Springer-Verlag.

Chung, C. F. (1996), "A Generalized Fractionally Integrated ARMA Process," *Journal of Time Series Analysis*, 2, 111–140.

Cox, D. R. and Hinkley, D. V. (1974), *Theoretical Statistics*, London: Chapman & Hall.

Daubechies, I. (1992), *Ten Lectures on Wavelets*, Volume 61, CBMS-NSF Regional Conference Series in Applied Mathematics, Philadelphia, PA: Society for Industrial and Applied Mathematics.

Davies, L. (1992), "The Asymptotics of Rousseeuw's Minimum Volume Ellipsoid Estimator," *The Annals of Statistics*, 20, 1828–1843.

De Jong, P. (1991), "Stable Algorithms for the State Space Model," *Journal of Time Series Analysis*, 12, 143–157.

DeBoor, C. (1981), *A Practical Guide to Splines*, New York: Springer-Verlag.

Dennis, J. E., Gay, D. M., and Welsch, R. E. (1981), "An Adaptive Nonlinear Least-Squares Algorithm," *ACM Transactions on Mathematical Software*, 7, 348–368.

Dennis, J. E. and Mei, H. H. W. (1979), "Two New Unconstrained Optimization Algorithms Which Use Function and Gradient Values," *Journal of Optimization Theory Applications*, 28, 453–482.

Donelson, J. and Hansen, E. (1971), "Cyclic Composite Predictor-Corrector Methods," *SIAM Journal on Numerical Analysis*, 8, 137–157.

Donoho, D. L. and Johnstone, I. M. (1994), "Ideal Spatial Adaptation via Wavelet Shrinkage," *Biometrika*, 81, 425–455.

Donoho, D. L. and Johnstone, I. M. (1995), "Adapting to Unknown Smoothness via Wavelet Shrinkage," *Journal of the American Statistical Association*, 90, 1200–1224.

Duchon, J. (1976), "Fonctions-Spline et Esperances Conditionnelles de Champs Gaussiens," *Ann. Sci. Univ. Clermont Ferrand II Math.*, 14, 19–27.

Emerson, P. L. (1968), "Numerical Construction of Orthogonal Polynomials from a General Recurrence Formula," *Biometrics*, 24, 695–701.

Eskow, E. and Schnabel, R. B. (1991), "Algorithm 695: Software for a New Modified Cholesky Factorization," *ACM Transactions on Mathematical Software*, 17, 306–312.

Fletcher, R. (1987), *Practical Methods of Optimization*, Second Edition, Chichester, UK: 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.

Furnival, G. M. and Wilson, R. W. (1974), "Regression by Leaps and Bounds," *Technometrics*, 16, 499–511.

Gaffney, P. W. (1984), "A Performance Evaluation of Some FORTRAN Subroutines for the Solution of Stiff Oscillatory Ordinary Differential Equations," *Association for Computing Machinery, Transactions on Mathematical Software*, 10, 58–72.

Gay, D. M. (1983), "Subroutines for Unconstrained Minimization," *ACM Transactions on Mathematical Software*, 9, 503–524.

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 Solutions of Large Sparse Positive Definite Systems*, Englewood Cliffs, NJ: Prentice-Hall.

Geweke, J. and Porter-Hudak, S. (1983), "The Estimation and Application of Long Memory Time Series Models," *Journal of Time Series Analysis*, 4, 221–238.

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 Transactions on Mathematical Software*, 10, 282–298.

Golub, G. H. (1969), "Matrix Decompositions and Statistical Calculations," in R. C. Milton and J. A. Nelder, eds., *Statistical Computation*, New York: Academic Press.

Golub, G. H. and Van Loan, C. F. (1989), *Matrix Computations*, Second Edition, Baltimore: Johns Hopkins University Press.

Gonin, R. and Money, A. H. (1989), *Nonlinear L_p-Norm Estimation*, New York: Marcel Dekker.

Goodnight, J. H. (1979), "A Tutorial on the Sweep Operator," *The American Statistician*, 33, 149–158.

Graybill, F. A. (1969), *Introduction to Matrices with Applications in Statistics*, Belmont, CA: Wadsworth.

Grizzle, J. E., Starmer, C. F., and Koch, G. G. (1969), "Analysis of Categorical Data by Linear Models," *Biometrics*, 25, 489–504.

Hadley, G. (1962), *Linear Programming*, Reading, MA: Addison-Wesley.

Harvey, A. C. (1989), *Forecasting, Structural Time Series Models and the Kalman Filter*, Cambridge: Cambridge University Press.

Harville, D. A. (1997), *Matrix Algebra from a Statistician's Perspective*, New York: Springer-Verlag.

Hocking, R. R. (1985), *The Analysis of Linear Models*, Belmont, CA: Brooks/Cole.

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.

Jennrich, 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*.

Kaiser, H. F. and Caffrey, J. (1965), "Alpha Factor Analysis," *Psychometrika*, 30, 1–14.

Kastenbaum, M. A. and Lamphiear, D. E. (1959), "Calculation of Chi-Square to Test the No Three-Factor Interaction Hypothesis," *Biometrics*, 15, 107–122.

Kohn, R. and Ansley, C. F. (1982), "A Note on Obtaining the Theoretical Autocovariances of an ARMA Process," *Journal of Statistical Computation and Simulation*, 15, 273–283.

Korff, F. A., Taback, M. A. M., and Beard, J. H. (1952), "A Coordinated Investigation of a Food Poisoning Outbreak," *Public Health Reports*, 67, 909–913.

Kruskal, J. B. (1964), "Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis," *Psychometrika*, 29, 1–27.

Lee, W. and Gentle, J. E. (1986), "The LAV Procedure," *SUGI Supplememtal Library User's Guide*.

Lewart, C. R. (1973), "Algorithm 463: Algorithms SCALE1, SCALE2, and SCALE3 for Determination of Scales on Computer Generated Plots," *Communications of the ACM*, 16(10), 639–640, available at http://portal.acm.org/citation.cfm?id=362375.362417.

Lindström, P. and Wedin, P. A. (1984), "A New Line-Search Algorithm for Nonlinear Least-Squares Problems," *Mathematical Programming*, 29, 268–296.

Madsen, K. and Nielsen, H. B. (1993), "A Finite Smoothing Algorithm for Linear L_1 Estimation," *SIAM Journal on Optimization*, 3, 223–235.

Mallat, S. (1989), "Multiresolution Approximation and Wavelets," *Transactions of the American Mathematical Society*, 315, 69–88.

McKean, J. W. and Schrader, R. M. (1987), "Least Absolute Errors Analysis of Variance," *Statistical Data Analysis - Based on L_1 Norm and Related Methods*.

McLeod, I. (1975), "Derivation of the Theoretical Autocovariance Function of Autoregressive-Moving Average Time Series," *Applied Statistics*, 24, 255–256.

Mittnik, S. (1990), "Computation of Theoretical Autocovariance Matrices of Multivariate Autoregressive Moving Average Time Series," *Journal of Royal Statistical Society*.

Monro, D. M. and Branch, J. L. (1976), "Algorithm AS 117. The Chirp Dis Fourier Transform and General Length," *Applied Statistics*, 351–361.

Moré, J. J. (1978), "The Levenberg-Marquardt Algorithm: Implementation and Theory," in G. A. Watson, ed., *Lecture Notes in Mathematics*, volume 30, 105–116, Berlin-Heidelberg-New York: Springer-Verlag.

Moré, J. J. and Sorensen, D. C. (1983), "Computing a Trust-Region Step," *SIAM Journal on Scientific and Statistical Computing*, 4, 553–572.

Nelder, J. A. and Wedderburn, R. W. M. (1972), "Generalized Linear Models," *Journal of the Royal Statistical Society, Series A*, 135, 370–384.

Nijenhuis, A. and Wilf, H. S. (1978), *Combinatorial Algorithms*, New York: Academic Press.

Nussbaumer, H. J. (1982), *Fast Fourier Transform and Convolution Algorithms*, Second Edition, New York: Springer-Verlag.

Ogden, R. T. (1997), *Essential Wavelets for Statistical Applications and Data Analysis*, Boston: Birkhäuser.

Osborne, M. R. (1985), *Finite Algorithms in Optimization and Data Analysis*, New York: John Wiley & Sons, Inc.

Pizer, S. M. (1975), *Numerical Computing and Mathematical Analysis*, Chicago, IL: Science Research Associates.

Pocock, S. J. (1977), "Group Sequential Methods in the Design and Analysis of Clinical Trials," *Biometrika*, 64, 191–199.

Pocock, S. J. (1982), "Interim Analyses for Randomized Clinical Trials: The Group Sequential Approach," *Biometrics*, 38, 153–162.

Powell, M. J. D. (1977), "Restart Procedures for the Conjugate Gradient Method," *Mathematical Programming*, 12, 241–254.

Powell, M. J. D. (1978), "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," in G. A. Watson, ed., *Lecture Notes in Mathematics*, volume 630, 144–175, Berlin-Heidelberg-New York: Springer-Verlag.

Powell, M. J. D. (1982), "VMCWD: A Fortran Subroutine for Constrained Optimization," *DAMTP 1982/NA4*, cambridge, England.

Powell, M. J. D. (1992), "A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation," *DAMTP/NA5*, cambridge, England.

Ralston, A. and Rabinowitz, P. (1978), *A First Course in Numerical Analysis*, New York: McGraw-Hill.

Rao, C. R. and Mitra, S. K. (1971), *Generalized Inverse of Matrices and Its Applications*, New York: John Wiley & Sons.

Reinsch, C. H. (1967), "Smoothing by Spline Functions," *Numerische Mahematik*, 10, 177–183.

Reinsel, G. C. (1997), *Elements of Multivariate Time Series Analysis*, Second Edition, New York: Springer-Verlag.

Rice, S. O. (1973), "Efficient Evaluation of Integrals of Analytic Functions by the Trapezoidal Rule," *Bell System Technical Journal*.

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 W. Grossmann, G. Pflug, I. Vincze, and W. Wertz, eds., *Mathematical Statistics and Applications*, 283–297, Dordrecht: Reidel Publishing.

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. (1996), "Recent Development in PROGRESS," *Computational Statistics and Data Analysis*, 21, 67–85.

Rousseeuw, P. J. and Hubert, M. (1997), "Recent Developments in PROGRESS," L_1*-Statistical Procedures and Related Topics*.

Rousseeuw, P. J. and Leroy, A. M. (1987), *Robust Regression and Outlier Detection*, New York: John Wiley & Sons.

Rousseeuw, P. J. and Van Driessen, K. (1998), *Computing LTS Regression for Large Data Sets*, Technical report, University of Antwerp.

Rousseeuw, P. J. and Van Driessen, K. (1999), "A Fast Algorithm for the Minimum Covariance Determinant Estimator," *Technometrics*, 41, 212–223.

Rousseeuw, P. J. and Van Zomeren, B. C. (1990), "Unmasking Multivariate Outliers and Leverage Points," *Journal of the American Statistical Association*, 85, 633–639.

Schatzoff, M., Tsao, R., and Fienberg, S. (1968), "Efficient Calculation of All Possible Regressions," *Technometrics*, 10(4), 769–779.

Shampine, L. (1978), "Stability Properties of Adams Codes," *Association for Computing Machinery, Transactions on Mathematical Software*, 4, 323–329.

Sikorsky, K. (1982), "Optimal Quadrature Algorithms in H_P Spaces," *Numerische Mathematik*, 39, 405–410.

Sikorsky, K. and Stenger, F. (1984), "Optimal Quadratures in H_P Spaces," *Association for Computing Machinery, Transactions on Mathematical Software*, 3, 140–151.

Singleton, R. C. (1969), "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Transactions on Audio and Electoacoustics*.

Sowell, F. (1992), "Maximum Likelihood Estimation of Stationary Univariate Fractionally Integrated Time Series Models," *Journal of Econometrics*, 53, 165–188.

Squire, W. (1987), "Comparison of Gauss-Hermite and Midpoint Quadrature with the Application of the Voigt Function," *Numerical Integration: Recent Developments*.

Stenger, F. (1973a), "Integration Formulas Based on the Trapezoidal Formula," *Journal of the Institute of Mathematics and Its Applications*, 12, 103–114.

Stenger, F. (1973b), "Remarks on Integration Formulas Based on the Trapezoidal Formula," *Jornal of the Institute of Mathematics and Its Applications*, 19, 145–147.

Stenger, F. (1978), "Optimal Convergence of minimum Norm Approximations in H_P," *Numerische Mathematik*, 29, 345–362.

Stoer, J. and Bulirsch, R. (1980), *Introduction to Numerical Analysis*, New York: Springer-Verlag.

Thisted, R. A. (1988), *Elements of Statistical Computing: Numerical Computation*, London: Chapman & Hall.

Trotter, H. (1962), "Algorithm 115: PERM," *Communications of the ACM*, 5, 434–435.

Wahba, G. (1990), *Spline Models for Observational Data*, Philadelphia: Society for Industrial and Applied Mathematics.

Wang, S. and Tsiatis, A. (1987), "Approximately Optimal One Parameter Boundaries for Group Sequential Trials," *Biometrics*, 43, 193–199.

Wilkinson, J. H. and Reinsch, C. (1971), *Handbook for Automatic Computation: Linear Algebra*, Volume 2, New York: Springer-Verlag.

Woodfield, T. J. (1988), "Simulating Stationary Gaussian ARMA Time Series," *Computer Science and Statistics: Proceedings of the 20th Symposium on the Interface*.

Young, F. W. (1981), "Quantitative Analysis of Qualitative Data," *Psychometrika*, 46, 357–388.