SAS/IML® 9.1

User's Guide

The Power to Know®

SAS/IML® 9.1 User's Guide

Contents

What's New in SAS/IML 9 and 9.1

Contents

What's New in SAS/IML 9 and 9.1

Overview

New to SAS/IML are the following:

- A set of new iterative methods for large sparse systems of linear equations
- A set of new direct algorithms utilizing Cholesky factorization for large sparse systems of linear equations
- A new chapter on Genetic Algorithms
- A set of new functions and calls for grouping and sorting
- A set of new calls for random number generation
- A new programming feature that extends the matrix subscripting functionality

Iterative Methods for Large Sparse Systems of Linear Equations

Iterative methods create a new direction in IML: the capability to handle large, sparse matrices and systems. With sparsity exploitation and iterative approximation algorithms, IML can solve linear systems containing as many as 10^6 variables on a regular PC. The new methods that are used are Conjugate Gradient, Minimum Residual, and Biconjugate Gradient. The collection of preconditioners include fast incomplete Cholesky factorization and Jacobi.

`9.1`

Direct Algorithms for Large Sparse Systems of Linear Equations: Cholesky Factorization

The Cholesky algorithm comprises a minimum degree ordering heuristic and symbolic factorization. The implementation is based on a quotient graph model.

`9.1`

Genetic Algorithms

Genetic algorithms are a family of search algorithms that seek optimal solutions to problems using an approach that parallels the principles of natural selection and evolution in nature. There are several new experimental subroutines and functions that enable you to formulate and implement genetic algorithm optimizations. You have the flexibility either to write your own modules to specify the objective function and genetic operators or to use standard functions and operators provided by IML.

New Functions and Calls

BY Group Processing for Matrices

A new function, UNIQUEBY, has been added to IML that makes it easier to retrieve and process BY groups in a sorted matrix. You can use the SORT and SORTNDX calls to sort a matrix, and then call the UNIQUEBY function to get a vector of indices of the first row of each unique BY group in the input matrix. See the Language Reference section for full details.

SORT and SORTNDX Calls

Two new subroutines have been added to make sorting of matrices easier. The SORT call sorts a matrix in place, sorting the rows by specified columns. The SORTNDX function creates an index matrix, without rearranging the input matrix, which enables you to access the rows of the input matrix in sorted order. The SORTNDX call will normally be faster than the SORT call, and you can use it to create multiple indexes into the same input matrix. See the Language Reference section for full details.

RANDSEED and RANDGEN Calls

9.1

Two new subroutines have been added to improve the efficiency of random number generation. RANDSEED and RANDGEN use the same algorithm as the data functions STREAMINIT and RAND, with changes to maximize performance in IML. See the Language Reference section for full details.

New Programming Feature

Subscripting by Row or Column Names

Matrix subscripting functionality has been extended, so that you can specify a set of rows or columns by the names given in a MATTRIB statement or assigned by a data set statement. This is done by using a character matrix as one or both of the subscripts. IML will attempt to match character string elements to column or row names assigned to the matrix. See the Language Reference section for full details and examples.

Chapter 1
Introduction to SAS/IML Software

Chapter Contents

Chapter 1
Introduction to SAS/IML Software

Overview

SAS/IML software gives you access to a powerful and flexible programming language (**I**nteractive **M**atrix **L**anguage) in a dynamic, interactive environment. The fundamental object of the language is a data matrix. You can use SAS/IML software interactively (at the statement level) to see results immediately, or you can store statements in a module and execute them later. The programming is dynamic because necessary activities such as memory allocation and dimensioning of matrices are performed automatically.

SAS/IML software is powerful. You can access built-in operators and call routines to perform complex tasks such as matrix inversion or eigenvector generation. You can define your own functions and subroutines using SAS/IML modules. You can operate on a single value or take advantage of matrix operators to perform operations on an entire data matrix. For example, the statement

```
x=x+1;
```

can be used to add 1 to a single value X, or to add 1 to all elements of a matrix **X**.

You have access to a wide choice of data management commands. You can read, create, and update SAS data sets from inside SAS/IML software without ever using the DATA step. For example, reading a SAS data set to get phone numbers for all individuals whose last name begins with "Smith" is easy:

```
read all var{phone} where(lastname=:"Smith");
```

The result is a matrix **PHONE** of phone numbers.

SAS/IML Software: An Overview

SAS/IML software is a programming language.

You can program easily and efficiently with the many features for arithmetic and character expressions in SAS/IML software. You have access to a wide range of built-in subroutines designed to make your programming fast, easy, and efficient. Because SAS/IML software is part of the SAS System, you can access SAS data sets or external files with an extensive set of data processing commands for data input and output, and you can edit existing SAS data sets or create new ones.

SAS/IML software has a complete set of control statements, such as DO/END, START/FINISH, iterative DO, IF-THEN/ELSE, GOTO, LINK, PAUSE, and STOP, giving you all of the commands necessary for execution control and program modularization.

SAS/IML software operates on matrices.

While most programming languages deal with single data elements, the fundamental data element with SAS/IML software is the matrix, a two-dimensional (row $\times$ column) array of numeric or character values.

SAS/IML software possesses a powerful vocabulary of operators.

You can access built-in matrix operations that require calls to math-library subroutines in other languages. You have access to many operators, functions, and CALL subroutines.

SAS/IML software uses operators that apply to entire matrices.

You can add elements of the matrices $\mathbf{A}$ and $\mathbf{B}$ with the expression $\mathbf{A} + \mathbf{B}$. You can perform matrix multiplication with the expression $\mathbf{A} * \mathbf{B}$ and perform elementwise multiplication with the expression $\mathbf{A}\#\mathbf{B}$.

SAS/IML software is interactive.

You can execute a command as soon as you enter it, or you can collect commands in a module to execute later. When you execute a command, you see the results immediately. You can interact with an executing module by programming IML to pause, enabling you to enter additional statements before continuing execution.

SAS/IML software is dynamic.

You do not need to declare, dimension, and allocate storage for a data matrix. SAS/IML software does this automatically. You can change the dimension or type of a matrix at any time. You can open multiple files or access many libraries. You can reset options or replace modules at any time.

SAS/IML software processes data.

You can read all observations or read conditionally selected observations from a SAS data set into a matrix, creating either multiple vectors (one for each variable in the data set) or a matrix that contains a column for each data set variable. You can create a new SAS data set, or you can edit or append observations to an existing SAS data set.

SAS/IML software produces graphics.

You have access to a wide range of graphics commands, enabling you to visually explore relationships in data.

An Introductory Interactive Session

Here is a simple introductory session that uses SAS/IML software to estimate the square root of a number, accurate to three decimal places. In this session, you define a function module named APPROX to perform the calculations and return the approximation. You then call APPROX to estimate the square root of several numbers given in a matrix literal (enclosed in braces), and you print the results.

Throughout the session, the right angle brackets (>) indicate statements that you submit; responses from IML follow.

```
>    proc iml;                                /* begin IML session */

     IML Ready

>    start approx(x);                         /* begin module */
>       y=1;                                  /* initialize y */
>       do until(w<1e-3);                     /* begin do loop */
>          z=y;                               /* set z=y */
>          y=.5#(z+x/z);              /* estimate square root */
>          w=abs(y-z);        /* compute change in estimate */
>       end;                                  /* end do loop */
>       return(y);                 /* return approximation */
>    finish approx;                           /* end module */

     NOTE: Module APPROX defined.

>    t=approx({3,5,7,9});           /* call function APPROX  */
>    print t;                               /* print matrix */

                                    T
                            1.7320508
                             2.236068
                            2.6457513
                                    3

>    quit;

     Exiting IML
```

PROC IML Statement

PROC IML <**SYMSIZE=**$n1$ > <**WORKSIZE=**$n2$ >**;**

where $n1$ and $n2$ are specified in kilobytes.

The SYMSIZE= and WORKSIZE= options in the PROC IML statement give you control over the size of memory allocated to the symbol space and the size of each extent of workspace. If you do not specify these options, PROC IML uses host dependent defaults.

Generally, you do not need to be concerned with the details of memory usage because memory allocation is done automatically. For special situations, however, see Memory and Workspace.

Chapter 2
Understanding the Language

Chapter Contents

Chapter 2
Understanding the Language

Defining a Matrix

The fundamental data object on which all Interactive Matrix Language commands operate is a two-dimensional (row × column) numeric or character matrix. By their very nature, matrices are useful for representing data and efficient for working with data. Matrices have the following properties:

- Matrices can be either numeric or character. Elements of a numeric matrix are stored in double precision. Elements of a character matrix are character strings of equal length. The length can range from 1 to 32676 characters.

- Matrices are referred to by valid SAS names. Names can be from 1 to 32 characters long, beginning with a letter or underscore, and continuing with letters, numbers, and underscores.

- Matrices have dimension defined by the number of rows and columns.

- Matrices can contain elements that have missing values (see the section "Missing Values" on page 24 later in this chapter).

The dimension of a matrix is defined by the number of rows and columns it has. An $m \times n$ matrix has mn elements arranged in m rows and n columns. The following nomenclature is standard in this book:

- $1 \times n$ matrices are called *row vectors*.
- $m \times 1$ matrices are called *column vectors*.
- 1×1 matrices are called *scalars*.

Matrix Names and Literals

Matrix Names

A matrix is referred to by a valid SAS name. Names can be from 1 to 8 characters long, beginning with a letter or underscore and continuing with letters, numbers, and underscores. You associate a name with a matrix when you create or define the matrix. A matrix name exists independently of values. This means that at any time, you can change the values associated with a particular matrix name, change the dimension of the matrix, or even change its type (numeric or character).

Matrix Literals

A *matrix literal* is a matrix represented by its values. When you represent a matrix by a literal, you are simply specifying the values of each element of the matrix. A matrix literal can have a single element (a scalar) or have many elements arranged in a rectangular form (rows × columns). The matrix can be numeric (all elements are numeric) or character (all elements are character). The dimension of the matrix is automatically determined by the way you punctuate the values.

If there are multiple elements, use braces ({ }) to enclose the values and commas to separate the rows. Within the braces, values must be either all numeric or all character. If you use commas to create multiple rows, all rows must have the same number of elements (columns).

The values you input can be any of the following:

- a number, with or without decimal points, possibly in scientific notation (such as 1E−5)

- a character string. Character strings can be enclosed in either single quotes (') or double quotes ("), but they do not necessarily need quotes. Quotes are required when there are no enclosing braces or when you want to preserve case, special characters, or blanks in the string. If the string has embedded quotes, you must double them (for example, WORD='Can"t'). Special characters can be any of the following: ? = * : ( ).

- a period (.), representing a missing numeric value

- numbers in brackets ([ ]), representing repetition factors

Creating Matrices from Matrix Literals

Creating matrices using matrix literals is easy. You simply input the element values one at a time, usually inside of braces. Representing a matrix as a matrix literal is not the only way to create matrices. A matrix can also be created as a result of a function, a CALL statement, or an assignment statement. Below are some simple examples of matrix literals, some with a single element (scalars) and some with multiple elements.

For more information on matrix literals, see Chapter 4, "Working with Matrices."

Scalar Literals

The following examples define scalars as literals. These are examples of simple assignment statements, with the matrix name on the left-hand side of the equal sign and the value on the right. Notice that you do not need to use braces when there is only one element.

```
a=12;
a=. ;
a='hi there';
a="Hello";
```

Numeric Literals

Matrix literals with multiple elements have the elements enclosed in braces. Use commas to separate the rows of a matrix. For example, the statement

```
x={1 2 3 4 5 6};
```

assigns a row vector to the matrix **X**:

```
            X
1           2           3           4           5           6
```

The statement

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

assigns a column vector to the matrix **Y**:

```
          Y
          1
          2
          3
          4
          5
```

The statement

```
z={1 2, 3 4, 5 6};
```

assigns a 3×2 matrix literal to the matrix **Z**:

```
          Z
          1           2
          3           4
          5           6
```

The following assignment

```
w=3#z;
```

creates a matrix **W** that is three times the matrix **Z**:

```
          W
          3           6
          9          12
         15          18
```

Character Literals

You input a character matrix literal by entering character strings. If you do not use quotes, all characters are converted to uppercase. You must use either single or double quotes to preserve case or when blanks or special characters are present in the string. For character matrix literals, the length of the elements is determined from the longest element. Shorter strings are padded on the right with blanks. For example, the assignment of the literal

```
a={abc defg};
```

results in $\mathbf{A}$ being defined as a 1×2 character matrix with string length 4 (the length of the longer string).

```
       A
ABC   DEFG
```

The assignment

```
a={'abc' 'DEFG'};
```

preserves the case of the elements, resulting in the matrix

```
       A
abc   DEFG
```

Note that the string length is still 4.

Repetition Factors

A repetition factor can be placed in brackets before a literal element to have the element repeated. For example, the statement

```
answer={[2]  'Yes', [2]  'No'};
```

is equivalent to

```
answer={'Yes' 'Yes', 'No' 'No'};
```

and results in the matrix

```
       ANSWER
Yes   Yes
No    No
```

Reassigning Values

You can assign new values to elements of a matrix at any time. The following statement creates a 2×3 numeric matrix named $\mathbf{A}$.

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

The statement

```
a={'Sales' 'Marketing' 'Administration'};
```

redefines the matrix $\mathbf{A}$ as a 1×3 character matrix.

Assignment Statements

Assignment statements create matrices by evaluating expressions and assigning the results to a matrix. The expressions can be composed of operators (for example, matrix multiplication) or functions (for example, matrix inversion) operating on matrices. Because of the nature of linear algebraic expressions, the resulting matrices automatically acquire appropriate characteristics and values. Assignment statements have the general form

$$result = expression;$$

where *result* is the name of the new matrix and *expression* is an expression that is evaluated, the results of which are assigned to the new matrix.

Functions as Expressions

Matrices can be created as a result of a function call. Scalar functions such as LOG or SQRT operate on each element of a matrix, while matrix functions such as INV or RANK operate on the entire matrix. For example, the statement

```
a=sqrt(b);
```

assigns the square root of each element of $\mathbf{B}$ to the corresponding element of $\mathbf{A}$.

The statement

```
y=inv(x);
```

calls the INV function to compute the inverse matrix of $\mathbf{X}$ and assign the results to $\mathbf{Y}$.

The statement

```
r=rank(x);
```

creates a matrix $\mathbf{R}$ with elements that are the ranks of the corresponding elements of $\mathbf{X}$.

Operators within Expressions

There are three types of operators that can be used in assignment statement expressions. Be sure that the matrices on which an operator acts are conformable to the operation. For example, matrix multiplication requires that the number of columns of the left-hand matrix be equal to the number of rows of the right-hand matrix.

The three types of operators are as follows:

prefix operators   are placed in front of an operand $(-\mathbf{A})$.

infix operators   are placed between operands $(\mathbf{A} * \mathbf{B})$.

postfix operators   are placed after an operand $(\mathbf{A}')$.

All operators can work in a one-to-many or many-to-one manner; that is, they enable you to, for example, add a scalar to a matrix or divide a matrix by a scalar. The followingis an example of using operators in an assignment statement.

```
y=x#(x>0);
```

This assignment statement creates a matrix $\mathbf{Y}$ in which each negative element of the matrix $\mathbf{X}$ is replaced with zero. The statement actually has two expressions evaluated. The expression $(\mathbf{X}>0)$ is a many-to-one operation that compares each element of $\mathbf{X}$ to zero and creates a temporary matrix of results; an element of the temporary matrix is 1 when the corresponding element of $\mathbf{X}$ is positive, and 0 otherwise. The original matrix $\mathbf{X}$ is then multiplied elementwise by the temporary matrix, resulting in the matrix $\mathbf{Y}$.

For a complete listing and explanation of operators, see Chapter 20, "Language Reference."

Types of Statements

Statements in SAS/IML software can be classified into three general categories:

Control Statements
direct the flow of execution. For example, the IF-THEN/ELSE Statement conditionally controls statement execution.

Functions and **CALL Statements**
perform special tasks or user-defined operations. For example, the statement CALL: *GSTART* activates the SAS/IML graphics system.

Commands
perform special processing, such as setting options, displaying, and handling input/output. For example, the command RESET: *PRINT* turns on the automatic displaying option so that matrix results are displayed as you submit statements.

Control Statements

SAS/IML software has a set of statements for controlling program execution. Control statements direct the flow of execution of statements in IML. With them, you can define DO-groups and modules (also known as subroutines) and route execution of your program. Some control statements are described as follows.

Statements	Action
DO, END	group statements
iterative DO, END	define an iteration loop
GOTO, LINK	transfer control
IF-THEN/ELSE	routes execution conditionall
PAUSE	instructs a module to pause during execution
QUIT	ends a SAS/IML session
RESUME	instructs a module to resume execution
RETURN	returns from a LINK statement or a CALL module
RUN	executes a module
START, FINISH	define a module
STOP, ABORT	stop execution of an IML program

See Chapter 5, "Programming Statements," later in this book for more information on control statements.

Functions

The general form of a function is

$$result = FUNCTION(arguments);$$

where *arguments* can be matrix names, matrix literals, or expressions. Functions always return a single result (whereas subroutines can return multiple results or no result). If a function returns a character result, the matrix to hold the result is allocated with a string length equal to the longest element, and all shorter elements are padded with blanks.

Categories of Functions

Functions fall into the following six categories:

matrix inquiry functions

> return information about a matrix. For example, the ANY function returns a value of 1 if any of the elements of the argument matrix are nonzero.

scalar functions

> operate on each element of the matrix argument. For example, the ABS function returns a matrix with elements that are the absolute values of the corresponding elements of the argument matrix.

summary functions

> return summary statistics based on all elements of the matrix argument. For example, the SSQ function returns the sum of squares of all elements of the argument matrix.

matrix arithmetic functions

> perform matrix algebraic operations on the argument. For example, the TRACE function returns the trace of the argument matrix.

matrix reshaping functions

> manipulate the matrix argument and return a reshaped matrix. For example, the DIAG function returns a matrix with diagonal elements that are equal to the diagonal elements of a square argument matrix. All off-diagonal elements are zero.

linear algebra and statistical functions

> perform linear algebraic functions on the matrix argument. For example, the GINV function returns the matrix that is the generalized inverse of the argument matrix.

Exceptions to the SAS DATA Step

SAS/IML software supports most functions supported in the SAS DATA step. These functions all accept matrix arguments, and the result has the same dimension as the argument. (See Appendix 1 for a list of these functions.) The following functions are **not** supported by SAS/IML software:

DIFn	HBOUND	LAGn	PUT
DIM	INPUT	LBOUND	

The following functions are implemented differently in SAS/IML software.

(See Chapter 20, "Language Reference," for descriptions.)

MAX	RANK	SOUND	SUBSTR
MIN	REPEAT	SSQ	SUM

The random number functions, UNIFORM and NORMAL, are built-in and produce the same streams as the RANUNI and RANNOR functions, respectively, of the DATA step. For example, to create a 10×1 vector of random numbers, use

```
x=uniform(repeat(0,10,1));
```

Also, SAS/IML software does not support the OF clause of the SAS DATA step. For example, the statement

```
a=mean(of x1-x10); /* invalid in IML */
```

cannot be interpreted properly in IML. The term (X1-X10) would be interpreted as subtraction of the two matrix arguments rather than its DATA step meaning, "X1 through X10."

CALL Statements and Subroutines

CALL statements invoke a subroutine to perform calculations, operations, or a service. CALL statements are often used in place of functions when the operation returns multiple results or, in some cases, no result. The general form of the CALL statement is

> **CALL** *SUBROUTINE arguments* ;

where *arguments* can be matrix names, matrix literals, or expressions. If you specify several arguments, use commas to separate them. Also, when using arguments for output results, always use variable names rather than expressions or literals.

Creating Matrices with CALL Statements

Matrices are created whenever a CALL statement returns one or more result matrices. For example, the statement

```
call eigen(val,vec,t);
```

returns two matrices (vectors), **VAL** and **VEC**, containing the eigenvalues and eigenvectors, respectively, of the symmetric matrix **T**.

You can program your own subroutine using the START and FINISH statements to define a module. You can then execute the module with a CALL statement or a RUN statement. For example, the following statements define a module named MYMOD that returns matrices containing the square root and log of each element of the argument matrix:

```
start mymod(a,b,c);
   a=sqrt(c);
   b=log(c);
finish;
run mymod(s,l,x);
```

Execution of the module statements create matrices **S** and **L**, containing the square roots and logs, respectively, of the elements of **X**.

Performing Services

You can use CALL statements to perform special services, such as managing SAS data sets or accessing the graphics system. For example, the statement

```
call delete(mydata);
```

deletes the SAS data set named MYDATA.

The statements

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

activate the graphics system (CALL GSTART), open a new graphics segment (CALL GOPEN), produce a scatter plot of points (CALL GPOINT), and display the graph (CALL GSHOW).

Commands

Commands are used to perform specific system actions, such as storing and loading matrices and modules, or to perform special data processing requests. The following is a list of some commands and the actions they perform.

Command	Action
FREE	frees a matrix of its values and increases available space
LOAD	loads a matrix or module from the storage library
MATTRIB	associates printing attributes with matrices
PRINT	prints a matrix or message
RESET	sets various system options
REMOVE	removes a matrix or module from library storage
SHOW	requests that system information be displayed
STORE	stores a matrix or module in the storage library

These commands play an important role in SAS/IML software. With them, for example, you can control displayed output (with RESET PRINT, RESET NOPRINT, or MATTRIB) or get system information (with SHOW SPACE, SHOW STORAGE, or SHOW ALL).

If you are running short on available space, you can use commands to store matrices in the storage library, free the matrices of their values, and load them back later when you need them again, as shown in the following example.

Throughout this session, the right angle brackets (>) indicate statements that you submit; responses from IML follow. First, invoke the procedure by entering PROC IML at the input prompt. Then, create matrices **A** and **B** as matrix literals.

```
> proc iml;

  IML Ready

> a={1 2 3, 4 5 6, 7 8 9};
> b={2 2 2};
```

List the names and attributes of all of your matrices with the SHOW NAMES command.

```
> show names;

  A               3 rows      3 cols num      8
  B               1 row       3 cols num      8
  Number of symbols = 2  (includes those without values)
```

Store these matrices in library storage with the STORE command, and release the space with the FREE command. To list the matrices and modules in library storage, use the SHOW STORAGE command.

```
> store a b;
> free a b;
> show storage;

  Contents of storage = SASUSER.IMLSTOR
  Matrices:
   A        B

  Modules:
```

The output from the SHOW STORAGE statement indicates that you have two matrices in storage. Because you have not stored any modules in this session, there are no modules listed in storage. Return these matrices from the storage library with the LOAD command. (See Chapter 14, "Storage Features," for details about storage.)

```
> load a b;
```

End the session with the QUIT command.

```
> quit;

  Exiting IML
```

Data Management Commands

SAS/IML software has many data management commands that enable you to manage your SAS data sets from within the SAS/IML environment. These data management commands operate on SAS data sets. There are also commands for accessing external files. The following is a list of some commands and the actions they perform.

Command	Action
APPEND	adds records to an output SAS data set
CLOSE	closes a SAS data set
CREATE	creates a new SAS data set
DELETE	deletes records in an output SAS data set
EDIT	reads from or writes to an existing SAS data set
FIND	finds records that meet some condition
LIST	lists records
PURGE	purges records marked for deletion
READ	reads records from a SAS data set into IML variables
SETIN	makes a SAS data set the current input data set
SETOUT	makes a SAS data set the current output data set
SORT	sorts a SAS data set
USE	opens an existing SAS data set for read access

These commands can be used to perform any necessary data management functions. For example, you can read observations from a SAS data set into a target matrix with the USE or EDIT command. You can edit a SAS data set, appending or deleting records. If you have generated data in a matrix, you can output the data to a SAS data set with the APPEND or CREATE command. See Chapter 6, "Working with SAS Data Sets," and Chapter 7, "File Access," for more information on these commands.

Missing Values

With SAS/IML software, a numeric element can have a special value called a *missing value* that indicates that the value is unknown or unspecified. Such missing values are coded, for logical comparison purposes, in the bit pattern of very large negative numbers. A numeric matrix can have any mixture of missing and nonmissing values. A matrix with missing values should not be confused with an empty or unvalued matrix, that is, a matrix with zero rows and zero columns.

In matrix literals, a numeric missing value is specified as a single period. In data processing operations involving a SAS data set, you can append or delete missing values. All operations that move values move missing values properly.

SAS/IML software supports missing values in a limited way, however. Most matrix operators and functions do not support missing values. For example, matrix multiplication involving a matrix with missing values is not meaningful. Also, the inverse of a matrix with missing values has no meaning. Performing matrix operations such as these on matrices that have missing values can result in inconsistencies, depending on the host environment.

See Chapter 4, "Working with Matrices," and Chapter 19, "Further Notes," for more details on missing values.

Summary

In this chapter, you were introduced to the fundamentals of the SAS/IML language. The basic data element, the matrix, was defined, and you learned several ways to create matrices: the matrix literal, CALL statements that return matrix results, and assignment statements.

You were introduced to the types of statements with which you can program: commands, control statements for iterative programming and module definition, functions, and CALL subroutines.

Chapter 3, "Tutorial: A Module for Linear Regression," offers an introductory tutorial that demonstrates using SAS/IML software to build and execute a module.

Chapter 3

Tutorial: A Module for Linear Regression

Chapter Contents

Chapter 3
Tutorial: A Module for Linear Regression

Overview

SAS/IML software makes it possible for you to solve mathematical problems or implement new statistical techniques and algorithms. The language is patterned after linear algebra notation. For example, the least-squares formula familiar to statisticians

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

can be easily translated into the Interactive Matrix Language statement

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

This is an example of an assignment statement that uses a built-in function (INV) and operators (transpose and matrix multiplication).

If a statistical method has not been implemented directly in a SAS procedure, you may be able to program it using IML. Because the operations in IML deal with arrays of numbers rather than with one number at a time, and the most commonly used mathematical and matrix operations are built directly into the language, programs that take hundreds of lines of code in other languages often take only a few lines in IML.

Solving a System of Equations

Because IML is built around traditional matrix algebra notation, it is often possible to directly translate mathematical methods from matrix algebraic expressions into executable IML statements. For example, consider the problem of solving three simultaneous equations:

$$
\begin{aligned}
3x_1 - x_2 + 2x_3 &= 8 \\
2x_1 - 2x_2 + 3x_3 &= 2 \\
4x_1 + x_2 - 4x_3 &= 9
\end{aligned}
$$

These equations can be written in matrix form as

$$
\begin{bmatrix}
3 & -1 & 2 \\
2 & -2 & 3 \\
4 & 1 & -4
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
x_3
\end{bmatrix}
=
\begin{bmatrix}
8 \\
2 \\
9
\end{bmatrix}
$$

and can be expressed symbolically as

$$\mathbf{A}\mathbf{x} = \mathbf{c}$$

Because **A** is nonsingular, the system has a solution given by

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{c}$$

In the following example, you solve this system of equations using an interactive session. Submit the PROC IML statement to begin the procedure. Throughout this chapter, the right angle brackets (>) indicate statements you submit; responses from IML follow:

```
proc iml;

IML Ready
```

Enter

```
reset print;
```

The PRINT option of the RESET command causes automatic printing of results. Notice that as you submit each statement, it is executed and the results are displayed. While you are learning IML or developing modules, it is a good idea to have all results printed automatically. Once you are familiar with SAS/IML software, you will not need to use automatic printing.

Next, set up the matrices **A** and **c**. Both of these matrices are input as matrix literals; that is, input the row and column values as discussed in Chapter 2, "Understanding the Language."

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

A	3 rows	3 cols	(numeric)
	3	-1	2
	2	-2	3
	4	1	-4

```
> c={8, 2, 9};
```

C	3 rows	1 col	(numeric)
	8		
	2		
	9		

Now write the solution equation, $\mathbf{x} = \mathbf{A}^{-1}\mathbf{c}$, as an IML statement. The appropriate statement is an assignment statement that uses a built-in function and an operator (INV is a built-in function that takes the inverse of a square matrix, and * is the operator for matrix multiplication).

```
> x=inv(a)*c;

            X               3 rows      1 col      (numeric)

                                            3
                                            5
                                            2
```

After IML executes the statement, the first row of matrix $\mathbf{X}$ contains the x_1 value for which you are solving, the second row contains the x_2 value, and the third row contains the x_3 value.

Now end the session by entering the QUIT command.

```
>    quit;

     Exiting IML
```

A Module for Linear Regression

The previous method may be more familiar to statisticians when different notation is used. A linear model is usually written

$$\mathbf{y} = \mathbf{Xb} + \mathbf{e}$$

where $\mathbf{y}$ is the vector of responses, $\mathbf{X}$ is the design matrix, and $\mathbf{b}$ is a vector of unknown parameters estimated by minimizing the sum of squares of $\mathbf{e}$, the error or residual.

The following example illustrates the programming techniques involved in performing linear regression. It is not meant to replace regression procedures such as the REG procedure, which are more efficient for regressions and offer a multitude of diagnostic options.

Suppose that you have response data $\mathbf{y}$ measured at five values of the independent variable $\mathbf{x}$ and you want to perform a quadratic regression.

Submit the PROC IML statement to begin the procedure.

```
> proc iml;

    IML Ready
```

Input the design matrix $\mathbf{X}$ and the data vector $\mathbf{y}$ as matrix literals.

```
> x={1 1 1,
>    1 2 4,
>    1 3 9,
>    1 4 16,
>    1 5 25};
```

```
        X           5 rows       3 cols     (numeric)

                      1            1            1
                      1            2            4
                      1            3            9
                      1            4           16
                      1            5           25
```

```
> y={1,5,9,23,36};
```

```
        Y           5 rows       1 col      (numeric)

                      1
                      5
                      9
                     23
                     36
```

Compute the least-squares estimate of **b** using the traditional formula.

```
> b=inv(x\baccent *x)*x\baccent *y;
```

```
        B           3 rows       1 col      (numeric)

                             2.4
                            -3.2
                              2
```

The predicted values are simply the **X** matrix multiplied by the parameter estimates, and the residuals are the difference between actual and predicted **y**.

```
> yhat=x*b;
```

```
        YHAT        5 rows       1 col      (numeric)

                            1.2
                              4
                           10.8
                           21.6
                           36.4
```

```
> r=y-yhat;
```

```
        R           5 rows       1 col      (numeric)
```

```
                              -0.2
                                 1
                              -1.8
                               1.4
                              -0.4
```

To calculate the estimate of the variance of the responses, calculate the sum of squared errors (SSE), its degrees of freedom (DFE), and the mean squared error (MSE). Note that in computing the degrees, you use the function NCOL to return the number of columns of **X**.

```
> sse=ssq(r);

            SSE            1 row        1 col      (numeric)

                                        6.4

> dfe=nrow(x)-ncol(x);

            DFE            1 row        1 col      (numeric)

                                        2


> mse=sse/dfe;

            MSE            1 row        1 col      (numeric)

                                        3.2
```

Notice that each calculation has required one simple line of code.

Now suppose you want to solve the problem repeatedly on new data sets without reentering the code. To do this, define a module (or subroutine). Modules begin with a START statement and end with a FINISH statement, with the program statements in between. The following statements define a module named REGRESS to perform linear regression.

```
> start regress;                       /* begin module */
>    xpxi=inv(t(x)*x);          /* inverse of X'X        */
>    beta=xpxi*(t(x)*y);        /* parameter estimate    */
>    yhat=x*beta;               /* predicted values      */
>    resid=y-yhat;                 /* residuals          */
>    sse=ssq(resid);               /* SSE                */
>    n=nrow(x);                    /* sample size        */
>    dfe=nrow(x)-ncol(x);          /* error DF           */
>    mse=sse/dfe;                  /* MSE                */
>    cssy=ssq(y-sum(y)/n);         /* corrected total SS */
>    rsquare=(cssy-sse)/cssy;      /* RSQUARE            */
>    print,"Regression Results",
>        sse dfe mse rsquare;
>    stdb=sqrt(vecdiag(xpxi)*mse); /* std of estimates    */
```

```
>    t=beta/stdb;                    /* parameter t-tests  */
>    prob=1-probf(t#t,1,dfe);        /* p-values           */
>    print,"Parameter Estimates",,
>       beta stdb t prob;
>    print,y yhat resid;
> finish regress;                    /* end module         */
```

Submit the module REGRESS for execution.

```
> reset noprint;
> run regress;                       /* execute module     */
```

Regression Results

SSE	DFE	MSE	RSQUARE
6.4	2	3.2	0.9923518

Parameter Estimates

BETA	STDB	T	PROB
2.4	3.8366652	0.6255432	0.5954801
-3.2	2.9237940	-1.094468	0.3879690
2	0.4780914	4.1833001	0.0526691

Y	YHAT	RESID
1	1.2	-0.2
5	4	1
9	10.8	-1.8
23	21.6	1.4
36	36.4	-0.4

At this point, you still have all of the matrices defined if you want to continue calculations. Suppose that you want to correlate the estimates. First, calculate the covariance estimate of the estimates; then, scale the covariance into a correlation matrix with values of 1 on the diagonal.

```
> reset print;            /* turn on auto printing  */
> covb=xpxi*mse;          /* covariance of estimates */
```

COVB	3 rows	3 cols	(numeric)

14.72	-10.56	1.6
-10.56	8.5485714	-1.371429
1.6	-1.371429	0.2285714

```
> s=1/sqrt(vecdiag(covb));
```

S	3 rows	1 col	(numeric)

```
                              0.260643
                              0.3420214
                              2.0916501
```

```
> corrb=diag(s)*covb*diag(s);    /* correlation of estimates */
```

```
            CORRB           3 rows        3 cols      (numeric)

                           1   -0.941376   0.8722784
                   -0.941376           1   -0.981105
                   0.8722784   -0.981105           1
```

Your module REGRESS remains available to do another regression, in this case, an orthogonalized version of the last polynomial example. In general, the columns of **X** will not be orthogonal. You can use the ORPOL function to generate orthogonal polynomials for the regression. Using them provides greater computing accuracy and reduced computing times. When using orthogonal polynomial regression, you expect the statistics of fit to be the same and the estimates to be more stable and uncorrelated.

To perform an orthogonal regression on the data, you must first create a vector containing the values of the independent variable x, which is the second column of the design matrix **X**. Then, use the ORPOL function to generate orthogonal second degree polynomials.

```
> x1={1,2,3,4,5};                    /* second column of X */

            X1              5 rows       1 col      (numeric)

                              1
                              2
                              3
                              4
                              5
```

```
> x=orpol(x1,2);    /* generates orthogonal polynomials */

            X               5 rows        3 cols      (numeric)

                   0.4472136   -0.632456   0.5345225
                   0.4472136   -0.316228   -0.267261
                   0.4472136           0   -0.534522
                   0.4472136   0.3162278   -0.267261
                   0.4472136   0.6324555   0.5345225
```

```
> reset noprint;              /* turns off auto printing    */
> run regress;                /* run REGRESS */
```

```
                              Regression Results


                SSE          DFE         MSE     RSQUARE
                6.4            2          3.2 0.9923518




                           Parameter Estimates



                     BETA          STDB            T         PROB
                33.093806 1.7888544          18.5 0.0029091
                27.828043 1.7888544 15.556349 0.0041068
                7.4833148 1.7888544 4.1833001 0.0526691

                            Y          YHAT        RESID
                            1           1.2         -0.2
                            5             4            1
                            9          10.8         -1.8
                           23          21.6          1.4
                           36          36.4         -0.4
```

```
> reset print;
> covb=xpxi*mse;
```

```
           COVB              3 rows         3 cols      (numeric)

                     3.2 -2.73E-17 4.693E-16
                -2.73E-17       3.2 -2.18E-15
                4.693E-16 -2.18E-15       3.2
```

```
> s=1/sqrt(vecdiag(covb));
```

```
           S                 3 rows         1 col       (numeric)

                          0.559017
                          0.559017
                          0.559017
```

```
> corrb=diag(s)*covb*diag(s);
```

```
           CORRB             3 rows         3 cols      (numeric)

                        1  -8.54E-18  1.467E-16
                -8.54E-18          1    -6.8E-16
                1.467E-16   -6.8E-16          1
```

Note that the values on the off-diagonal are displayed in scientific notation; the values are close to zero but not exactly zero because of the imprecision of floating-point

arithmetic. To clean up the appearance of the correlation matrix, use the FUZZ option.

```
> reset fuzz;
> corrb=diag(s)*covb*diag(s);
```

```
              CORRB          3 rows        3 cols       (numeric)

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

Plotting Regression Results

You can create some simple plots by using the PGRAF subroutine. The PGRAF subroutine produces scatter plots suitable for printing on a line printer. If you want to produce better quality graphics using color, you can use the graphics capabilities of IML (see Chapter 12, "Graphics Examples," for more information).

Here is how you can plot the residuals against **x**. First, create a matrix containing the pairs of points by concatenating **X1** with **RESID** using the horizontal concatenation operator (‖).

```
> xy=x1||resid;
```

```
              XY             5 rows        2 cols       (numeric)

                               1          -0.2
                               2             1
                               3          -1.8
                               4           1.4
                               5          -0.4
```

Next, use a CALL statement to call the PGRAF subroutine to produce the desired plot. The arguments to PGRAF are, in order,

- the matrix containing the pairs of points
- a plotting symbol
- a label for the X-axis
- a label for the Y-axis
- a title for the plot

```
> call pgraf(xy,'r','x','Residuals','Plot of Residuals');
```

```
                            Plot of Residuals
    2 +
      |
      |                                               r
R     |
e     |                   r
s     |
i     |
d  0  +
u     |      r                                                   r
a     |
l     |
s     |
      |
      |                              r
  -2  +
      --------+------+------+------+------+------+------+------+------+--------
            1.0    1.5    2.0    2.5    3.0    3.5    4.0    4.5    5.0
                                       x
```
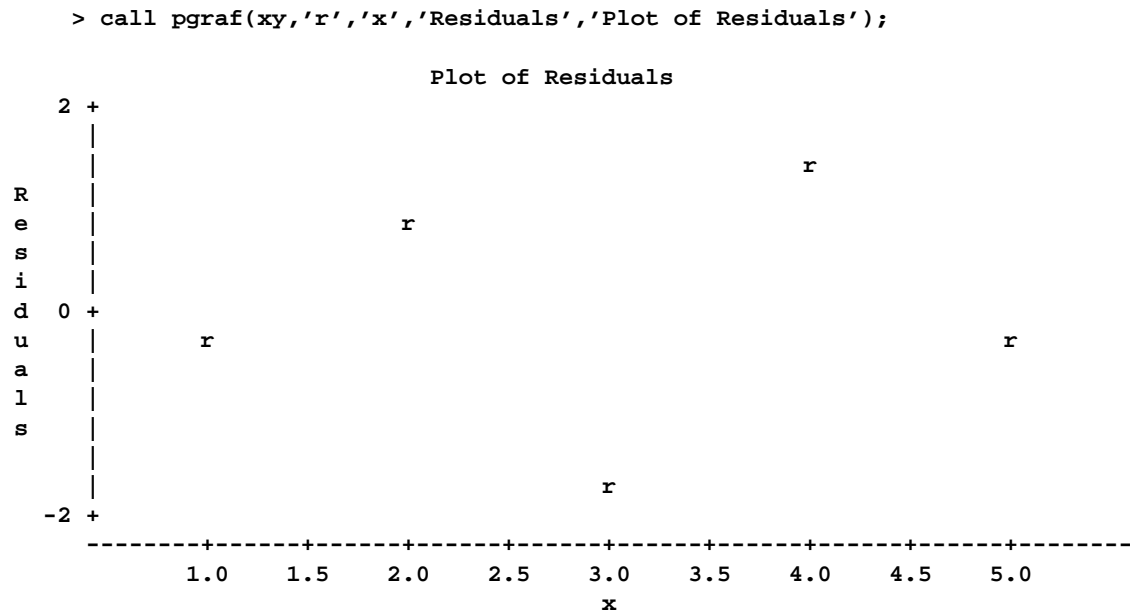
You can also plot the predicted values ŷ against **x**. You must first create a matrix, say
XYH, containing the points. Do this by concatenating **X1** with **YHAT**. Next, call the
PGRAF subroutine to plot the points.

```
> xyh=x1||yhat;

          XYH              5 rows      2 cols     (numeric)

                                 1       1.2
                                 2         4
                                 3      10.8
                                 4      21.6
                                 5      36.4


> call pgraf(xyh,'*','x','Predicted','Plot of Predicted Values');
```

```
                          Plot of Predicted Values
     40 +
        |                                                              *
        |
  P     |
  r     |
  e     |
  d     |                                              *
  i 20 +
  c     |
  t     |
  e     |                              *
  d     |
        |
        |                      *
     0 +           *
        --------+------+------+------+------+------+------+------+------+--------
              1.0    1.5    2.0    2.5    3.0    3.5    4.0    4.5    5.0
                                        x
```
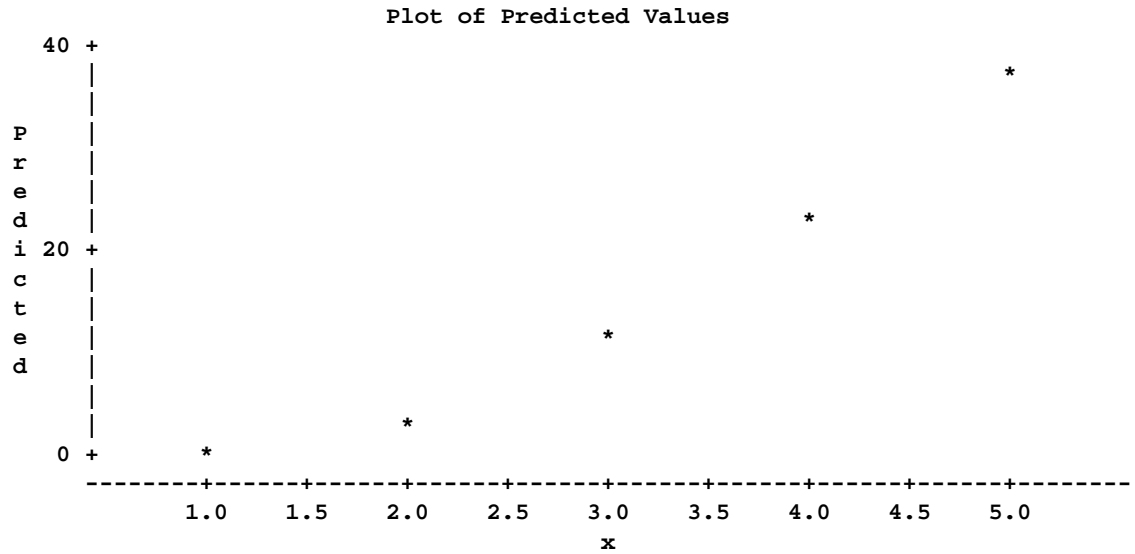
You can get a more detailed plot, denoting the observed values with a "y" and the predicted values with a "p" using the following statements. Create a matrix **NEWXY** containing the pairs of points to overlay. You need to use both the horizontal concatenation operator (‖) and the vertical concatenation operator (//). The NROW function returns the number of observations, that is, the number of rows of **X1**. The matrix **LABEL** contains the character label for each point, plotting a "y" for each observed point and a "p" for each predicted point.

```
> newxy=(x1//x1)||(y//yhat);

              NEWXY            10 rows        2 cols      (numeric)

                                      1           1
                                      2           5
                                      3           9
                                      4          23
                                      5          36
                                      1         1.2
                                      2           4
                                      3        10.8
                                      4        21.6
                                      5        36.4

> n=nrow(x1);

              N                 1 row         1 col       (numeric)

                                      5

> label=repeat('y',n,1)//repeat('p',n,1);
```

```
            LABEL          10 rows        1 col       (character, size 1)

                                           Y
                                           Y
                                           Y
                                           Y
                                           Y
                                           P
                                           P
                                           P
                                           P
                                           P


 > call pgraf(newxy,label,'x','y','Scatter Plot with Regression Line' );


                       Scatter Plot with Regression Line
y 40 +
    |                                                              Y
    |
    |
    |
    |
    |                                              Y
 20 +
    |
    |
    |                                  P
    |                                  Y
    |                      Y
    |                      P
  0 +        Y
   --------+------+------+------+------+------+------+------+------+----
          1.0    1.5    2.0    2.5    3.0    3.5    4.0    4.5    5.0
                                       x
```
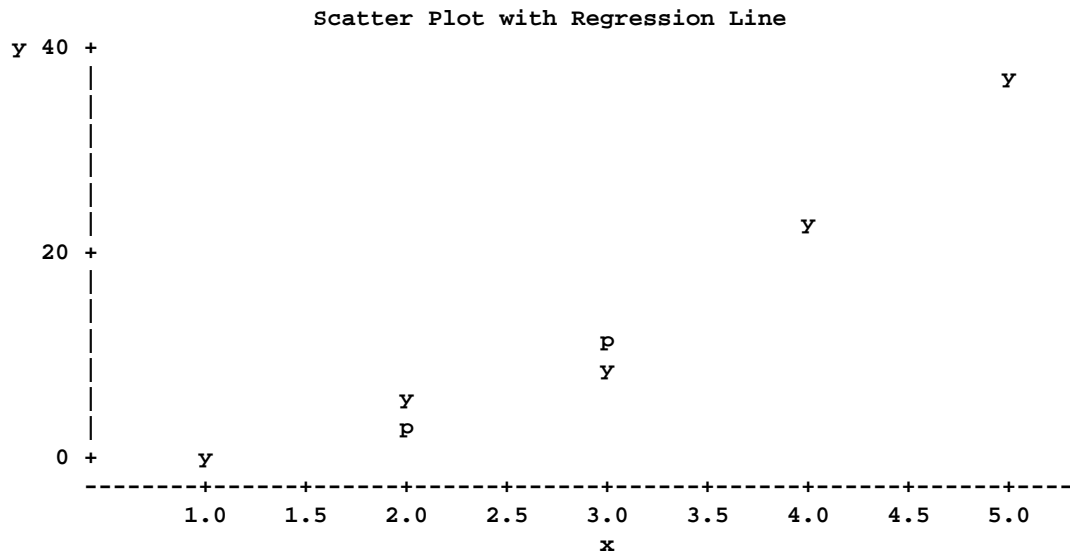
As you can see, the observed and predicted values are too close together to be able to distinguish them at all values of x.

Summary

In this chapter, you have seen the programming techniques necessary for solving systems of equations. You have seen how to define a module for performing linear regression and obtaining covariance and correlation matrices, and how to obtain some simple diagnostic plots. Many of the ideas presented in Chapter 2, "Understanding the Language," such as the use of assignment statements, functions, CALL statements, and subscripting have been demonstrated.

Chapter 4
Working with Matrices

Chapter Contents

Chapter 4
Working with Matrices

Overview

SAS/IML software provides many ways to create matrices. You can create matrices by doing any of the following:

- entering data yourself as a matrix literal
- using assignment statements
- using matrix-generating functions
- creating submatrices from existing matrices with subscripts
- using SAS data sets (see Chapter 6, "Working with SAS Data Sets," for more information)

Once you have defined matrices, you have access to many operators and functions for working on them in matrix expressions. These operators and functions facilitate programming and make referring to submatrices efficient and simple.

Finally, you have several means available for tailoring your displayed output.

Entering Data as Matrix Literals

The most basic way to create a matrix is to define a matrix literal, either numeric or character, by entering the matrix elements. A matrix literal can be a single element (called a *scalar*), a single row of data (called a *row vector*), a single column of data (called a *column vector*), or a rectangular array of data (called a *matrix*). The *dimension* of a matrix is given by its number of rows and columns. An $n \times m$ matrix has n rows and m columns.

Scalars

Scalars are matrices that have only one element. You define a scalar with the matrix name on the left-hand side of an assignment statement and its value on the right-hand side. You can use the following statements to create and display several examples of scalar literals. First, you must invoke the IML procedure.

```
>   proc iml;

    IML Ready

>   x=12;
>   y=12.34;
```

```
>   z=.;
>   a='Hello';
>   b="Hi there";
>   print x y z a b;
```

```
              X            Y         Z   A       B
             12        12.34         .   Hello   Hi there
```

Notice that, when defining a character literal, you need to use either single quotes (')
or double quotes ("). Using quotes preserves uppercase and lowercase distinctions
and embedded blanks. It is also always correct to enclose the data values inside of
braces ({ }).

Matrices with Multiple Elements

To enter a matrix having multiple elements, use braces ({ }) to enclose the data values
and, if needed, commas to separate rows. Inside of the braces, all elements must be
either numeric or character. You cannot have a mixture of data types within a matrix.
Each row must have the same number of elements.

For example, suppose that you have one week of data on daily coffee consumption
(cups per day) for your office of four people. Create a matrix **COFFEE** with each
person's consumption as a row of the matrix and each day represented by a column.
First, submit the RESET: PRINT statement so that results are displayed as you submit
statements.

```
>   reset print;
>   coffee={4 2 2 3 2,
>           3 3 1 2 1,
>           2 1 0 2 1,
>           5 4 4 3 4};
```

```
        COFFEE          4 rows       5 cols      (numeric)

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

Now create a character matrix called **NAMES** with rows containing the names of the
people in your office. Note that when you do not use quotes, characters are converted
to uppercase.

```
>   names={Jenny, Linda, Jim, Samuel};
```

```
        NAMES           4 rows       1 col      (character, size 6)

                            JENNY
                            LINDA
                            JIM
                            SAMUEL
```

Notice that the output with the RESET PRINT statement includes the dimension, type, and (when type is character) the element size of the matrix. The element size represents the length of each string, and it is determined from the length of the longest string.

Now, display the **COFFEE** matrix using **NAMES** as row labels by specifying the ROWNAME= option in the PRINT statement.

```
>   print coffee [rowname=names];

    COFFEE
    JENNY          4         2         2         3         2
    LINDA          3         3         1         2         1
    JIM            2         1         0         2         1
    SAMUEL         5         4         4         3         4
```

Using Assignment Statements

Assignment statements create matrices by evaluating expressions and assigning the results to a matrix. The expressions can be composed of operators (for example, the matrix addition operator (+)), functions (for example, the INV function), and subscripts. Assignment statements have the general form

$result = expression$;

where *result* is the name of the new matrix and *expression* is an expression that is evaluated. The resulting matrix automatically acquires the appropriate dimension, type, and value. Details on writing expressions are described in "Using Matrix Expressions" later in this chapter.

Simple Assignment Statements

Simple assignment statements involve an equation having the matrix name on the left-hand side and either an expression involving other matrices or a matrix-generating function on the right-hand side.

Suppose you want to generate some statistics for the weekly coffee data. If a cup of coffee costs 30 cents, then you can create a matrix with the daily expenses, **DAYCOST**, by multiplying the per-cup cost with the matrix **COFFEE** using the elementwise multiplication operator (#). Turn off the automatic printing so that you can tailor the output with the ROWNAME= and FORMAT= options in the PRINT statement.

```
>   reset noprint;
>   daycost=0.30#coffee;
>   print "Daily totals", daycost[rowname=names format=8.2];
```

```
    Daily totals

      DAYCOST
      JENNY        1.20       0.60       0.60       0.90       0.60
      LINDA        0.90       0.90       0.30       0.60       0.30
      JIM          0.60       0.30       0.00       0.60       0.30
      SAMUEL       1.50       1.20       1.20       0.90       1.20
```

You can calculate the weekly total cost for each person using the matrix multiplication operator (*). First create a 5×1 vector of 1s. This vector sums the daily costs for each person when multiplied with **COFFEE**. (You will see later that there is a more efficient way to do this using subscript reduction operators.)

```
>   ones={1,1,1,1,1};
>   weektot=daycost*ones;
>   print "Week total", weektot[rowname=names format=8.2];
```

```
          Week total

          WEEKTOT
          JENNY        3.90
          LINDA        3.00
          JIM          1.80
          SAMUEL       6.00
```

Finally, you can calculate the average number of cups drunk per day by dividing the grand total of cups by days. To find the grand total, use the SUM function, which returns the sum of all elements of a matrix. Next, divide the grand total by 5, the number of days (which is the number of columns) using the division operator (/) and the NCOL function. These two matrices are created separately, but the entire calculation could be done in one statement.

```
>   grandtot=sum(coffee);
>   average=grandtot/ncol(coffee);
>   print "Total number of cups", grandtot,,"Daily average",average;
```

```
              Total number of cups

                   GRANDTOT
                      49

                  Daily average

                   AVERAGE
                      9.8
```

Matrix-generating Functions

SAS/IML software has many built-in functions that generate useful matrices. For example, the J function creates a matrix with a given dimension and

element value when you supply the number of rows and columns, and an element value for the new matrix. This function is useful to initialize a matrix to a predetermined size. Several matrix-generating functions are listed below:

BLOCK	creates a block-diagonal matrix.
DESIGNF	creates a full-rank design matrix.
I	creates an identity matrix.
J	creates a matrix of a given dimension.
SHAPE	shapes a new matrix from the argument.

The sections that follow illustrate these matrix-generating functions. Again, they are shown with automatic printing of results, activated by invoking the RESET: PRINT statement.

```
reset print;
```

The BLOCK Function

The BLOCK function has the general form

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

and creates a block-diagonal matrix from the argument matrices. For example, the statements

```
> a={1 1,1 1};
```

```
          A              2 rows     2 cols    (numeric)

                           1          1
                           1          1
```

```
> b={2 2, 2 2};
```

```
          B              2 rows     2 cols    (numeric)

                           2          2
                           2          2
```

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

result in the matrix

```
          C              4 rows     4 cols    (numeric)

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

The J Function

The J function has the general form

J(*nrow<,ncol<,value> >* **);**

and creates a matrix having *nrow* rows, *ncol* columns, and all element values equal to *value*. The *ncol* and *value* arguments are optional, but you will usually want to specify them. In many statistical applications, it is helpful to be able to create a row (or column) vector of 1s (you did so to calculate coffee totals in the last section). You can do this with the J function. For example, the following statement creates a 1×5 row vector of 1s:

```
>  one=j(1,5,1);
```

```
            ONE             1 row        5 cols      (numeric)
                      1         1          1          1          1
```

The I Function

The I function creates an identity matrix of a given size. It has the general form

I(*dimension* **);**

where *dimension* gives the number of rows. For example, the following statement creates a 3×3 identity matrix:

```
>  I3=I(3);
```

```
            I3              3 rows       3 cols      (numeric)

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

The DESIGNF Function

The DESIGNF function generates a full-rank design matrix, useful in calculating ANOVA tables. It has the general form

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

For example, the following statement creates a full-rank design matrix for a one-way *ANOVA*, where the treatment factor has three levels and there are $n_1 = 3$, $n_2 = 2$, and $n_3 = 2$ observations at the factor levels:

```
>  d=designf({1,1,1,2,2,3,3});
```

```
       D              7 rows        2 cols     (numeric)

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

The SHAPE Function

The SHAPE function shapes a new matrix from an argument matrix. It has the general form

SHAPE(*matrix<,nrow<,ncol<,pad-value >>>***);**

Although the *nrow, ncol*, and *pad-value* arguments are optional, you will usually want to specify them. The following example uses the SHAPE function to create a 3×3 matrix containing the values 99 and 33. The function cycles back and repeats values to fill in when no *pad-value* is given.

```
>  aa=shape({99 33,99 33},3,3);

          AA              3 rows        3 cols     (numeric)

                        99              33              99
                        33              99              33
                        99              33              99
```

In the next example, a *pad-value* is specified for filling in the matrix:

```
>  aa=shape({99 33,99 33},3,3,0);

          AA              3 rows        3 cols     (numeric)

                        99              33              99
                        33               0               0
                         0               0               0
```

The SHAPE function cycles through the argument matrix elements in row-major order and then fills in with 0s after the first cycle through the argument matrix.

Index Vectors

You can create a vector by using the index operator (:). Several examples of statements involving index vectors are shown in the following code:

```
>  r=1:5;
            R              1 row       5 cols     (numeric)
                     1         2         3          4          5
>  s=10:6;
            S              1 row       5 cols     (numeric)
                    10         9         8          7          6
>  t='abc1':'abc5';
            T              1 row       5 cols     (character, size 4)
                          abc1 abc2 abc3 abc4 abc5
```

If you want an increment other than 1, use the DO function. For example, if you want a vector ranging from −1 to 1 by 0.5, use the following statement:

```
>  r=do(-1,1,.5);

            R              1 row       5 cols     (numeric)
                    -1      -0.5         0        0.5          1
```

Using Matrix Expressions

Matrix expressions are a sequence of names, literals, operators, and functions that perform some calculation, evaluate some condition, or manipulate values. These expressions can appear on either side of an assignment statement.

Operators

Operators used in matrix expressions fall into three general categories:

prefix operators are placed in front of operands. For example, $-\mathbf{A}$ uses the sign reverse prefix operator $(-)$ in front of the operand $\mathbf{A}$ to reverse the sign of each element of $\mathbf{A}$.

infix operators are placed between operands. For example, $\mathbf{A} + \mathbf{B}$ uses the addition infix operator $(+)$ between operands $\mathbf{A}$ and $\mathbf{B}$ to add corresponding elements of the matrices.

postfix operators are placed after an operand. For example, $\mathbf{A}$` uses the transpose postfix operator (`) after the operand $\mathbf{A}$ to transpose $\mathbf{A}$.

Matrix operators are listed in Appendix 1, "SAS/IML Quick Reference," and described in detail in Chapter 20, "Language Reference."

Table 4.1 on page 51 shows the precedence of matrix operators in Interactive Matrix Language.

Table 4.1. Operator Precedence

Priority Group	Operators					
I (highest)	ˆ	`	subscripts	−(prefix)	##	**
II	*	#	<>	><	/	@
III	+	−				
IV	|||	//	:			
V	<	<=	>	>=	=	ˆ=
VI	&					
VII (lowest)	|					

Compound Expressions

With SAS/IML software, you can write compound expressions involving several matrix operators and operands. For example, the following statements are valid matrix assignment statements:

```
a=x+y+z;
a=x+y*z\prime ;
a=(-x)#(y-z);
```

The rules for evaluating compound expressions are as follows:

- Evaluation follows the order of operator precedence, as described in Table 4.1. Group I has the highest priority; that is, Group I operators are evaluated first. Group II operators are evaluated after Group I operators, and so forth. For example, the statement

```
a=x+y*z;
```

  first multiplies matrices **Y** and **Z** since the * operator (Group II) has higher precedence than the + operator (Group III). It then adds the result of this multiplication to the matrix **X** and assigns the new matrix to **A**.

- If neighboring operators in an expression have equal precedence, the expression is evaluated from left to right, except for the Group 1 operators. For example, the statement

```
a=x/y/z;
```

  first divides each element of matrix **X** by the corresponding element of matrix **Y**. Then, using the result of this division, it divides each element of the resulting matrix by the corresponding element of matrix **Z**. The operators in Group 1, described in Table 4.1, are evaluated from right to left. For example, the expression

```
-x**2
```

is evaluated as

```
-(x**2)
```

When multiple prefix or postfix operators are juxtaposed, precedence is determined by their order from inside to outside.

For example, the expression

```
a`[i,j]
```

is evaluated as $(\mathbf{A}`)[i, j]$.

- All expressions enclosed in parentheses are evaluated first, using the two preceding rules. Thus, the IML statement

```
a=x/(y/z);
```

is evaluated by first dividing elements of $\mathbf{Y}$ by the elements of $\mathbf{Z}$, then dividing this result into $\mathbf{X}$.

Elementwise Binary Operators

Elementwise binary operators produce a result matrix from element-by-element operations on two argument matrices.

Table 4.2 on page 52 lists the elementwise binary operators.

Table 4.2. Elementwise Binary Operators

Operator	Action
$+$	addition, concatenation
$-$	subtraction
$\#$	elementwise multiplication
$\#\#$	elementwise power
$/$	division
$<>$	element maximum
$><$	element minimum
$|$	logical OR
$\&$	logical AND
$<$	less than
$<=$	less than or equal to
$>$	greater than
$>=$	greater than or equal to
$\hat{}=$	not equal to
$=$	equal to
$\text{MOD}(m, n)$	modulo (remainder)

For example, consider the following two matrices $\mathbf{A}$ and $\mathbf{B}$ given below.

Let $\mathbf{A} = \begin{bmatrix} 2 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 4 & 5 \\ 1 & 0 \end{bmatrix}$

The addition operator $(+)$ adds corresponding matrix elements:

$$\mathbf{A} + \mathbf{B} \text{ yields } \begin{bmatrix} 6 & 7 \\ 4 & 4 \end{bmatrix}$$

The elementwise multiplication operator $(\#)$ multiplies corresponding elements:

$$\mathbf{A}\#\mathbf{B} \text{ yields } \begin{bmatrix} 8 & 10 \\ 3 & 0 \end{bmatrix}$$

The elementwise power operator $(\#\#)$ raises elements to powers:

$$\mathbf{A}\#\#2 \text{ yields } \begin{bmatrix} 4 & 4 \\ 9 & 16 \end{bmatrix}$$

The element maximum operator $(<>)$ compares corresponding elements and chooses the larger:

$$\mathbf{A} <> \mathbf{B} \text{ yields } \begin{bmatrix} 4 & 5 \\ 3 & 4 \end{bmatrix}$$

The less than or equal to operator $(<=)$ returns a 1 if an element of $\mathbf{A}$ is less than or equal to the corresponding element of $\mathbf{B}$, and returns a 0 otherwise:

$$\mathbf{A} <= \mathbf{B} \text{ yields } \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

The modulo operator returns the remainder of each element divided by the argument:

$$\text{MOD}(\mathbf{A}, 3) \text{ yields } \begin{bmatrix} 2 & 2 \\ 0 & 1 \end{bmatrix}$$

All operators can also work in a one-to-many or many-to-one manner, as well as in an element-to-element manner; that is, they enable you to perform tasks such as adding a scalar to a matrix or dividing a matrix by a scalar. For example, the statement

```
x=x#(x>0);
```

replaces each negative element of the matrix $\mathbf{X}$ with 0. The expression (X>0) is a many-to-one operation that compares each element of $\mathbf{X}$ to 0 and creates a temporary matrix of results; an element in the result matrix is 1 when the expression is true and 0 when it is false. When the expression is true (the element is positive), the element is multiplied by 1. When the expression is false (the element is negative or 0), the element is multiplied by 0. To fully understand the intermediate calculations, you can use the RESET: PRINTALL command to have the temporary result matrices displayed.

Subscripts

Subscripts are special postfix operators placed in square brackets ([ ]) after a matrix operand. Subscript operations have the general form

$$operand[row, column]$$

where

operand	is usually a matrix name, but it can also be an expression or literal.
row	refers to an expression, either scalar or vector, for selecting one or more rows from the operand.
column	refers to an expression, either scalar or vector, for selecting one or more columns from the operand.

You can use subscripts to

- refer to a single element of a matrix
- refer to an entire row or column of a matrix
- refer to any submatrix contained within a matrix
- perform a reduction across rows or columns of a matrix

In expressions, subscripts have the same (high) precedence as the transpose postfix operator (`). Note that when both *row* and *column* subscripts are used, they are separated by a comma. If a matrix has row or column labels associated with it from a MATTRIB or READ statement, then the corresponding row or column subscript may be a character matrix whose elements match the labels of the rows or columns to be selected.

Selecting a Single Element

You can select a single element of a matrix in several ways. You can use two subscripts (*row, column*) to refer to its location, or you can use one subscript to look for the element down the rows. For instance, referring to the coffee example used earlier, find the element corresponding to the number of cups that Linda drank on Monday.

First, you can refer to the element by row and column location. In this case, you want the second row and first column. You can call this matrix **c21**.

```
   >  print  coffee[rowname=names];

       COFFEE
       JENNY          4          2          2          3          2
       LINDA          3          3          1          2          1
       JIM            2          1          0          2          1
       SAMUEL         5          4          4          3          4
```

```
> c21=coffee[2,1];
> print c21;
```

<div align="center">

C21
3

</div>

You could also use row and column labels, which can be assigned with an MATTRIB statement:

```
> mattrib coffee rowname=names
>    colname={'MON' 'TUE' 'WED' 'THU' 'FRI'};

> c21=coffee['LINDA','MON'];
> print c21;
```

<div align="center">

C21
3

</div>

You can also look for the element down the rows. In this case, you refer to this element as the sixth element of **COFFEE** in row-major order.

```
> c6=coffee[6];
> print c6;
```

<div align="center">

C6
3

</div>

Selecting a Row or Column

To refer to an entire row or column of a matrix, write the subscript with the row or column number, omitting the other subscript but not the comma. For example, to refer to the row of **COFFEE** that corresponds to Jim, you want the submatrix consisting of the third row and all columns:

```
> jim=coffee[3,];
> print jim;
```

```
        JIM
         2         1         0         2         1
```

You could also use the row labels assigned previously with the MATTRIB statement:

```
> jim=coffee['JIM',];
> print jim;
```

```
        JIM
         2         1         0         2         1
```

If you want the data for Friday, you know that the fifth column corresponds to Friday, so you want the submatrix consisting of the fifth column and all rows:

```
> friday=coffee[,5];
> print friday;
```

```
              FRIDAY
                   2
                   1
                   1
                   4
```

You could also use the previously assigned column labels:

```
> friday=coffee[,'FRI'];
> print friday;
```

```
              FRIDAY
                   2
                   1
                   1
                   4
```

Submatrices

You refer to a submatrix by the specific rows and columns you want. Include within the brackets the rows you want, a comma, and the columns you want. For example, to create the submatrix of **COFFEE** consisting of the first and third rows and the second, third, and fifth columns, submit the following statements:

```
> submat1=coffee[{1 3},{2 3 5}];
> print submat1;
```

```
        SUBMAT1
             2         2         2
             1         0         1
```

The first vector, {1 3}, selects the rows, and the second vector, {2 3 5}, selects the columns. Alternately, you can create the vectors beforehand and supply their names as arguments.

```
> rows={1 3};
> cols={2 3 5};
> submat1=coffee[rows,cols];
```

Similarly, you can use the previously assigned row and column labels:

```
> submat1=coffee[{'JENNY' 'JIM'},{'TUE' 'WED' 'FRI'}];
> print submat1;
```

```
              SUBMAT1
                  2         2         2
                  1         0         1
```

```
> rows={'JENNY' 'JIM'};
> cols={'TUE' 'WED' 'FRI'};
> submat1=coffee[rows,cols];
```

You can use index vectors generated by the index creation operator (:) in subscripts
to refer to successive rows or columns. For example, to select the first three rows and
last three columns of **COFFEE**, use the following statements:

```
> submat2=coffee[1:3,3:5];
> print submat2;
```

```
              SUBMAT2
                  2         3         2
                  1         2         1
                  0         2         1
```

Note that, in each example, the number in the first subscript defines the number of
rows in the new matrix; the number in the second subscript defines the number of
columns.

Subscripted Assignment

You can assign values into a matrix using subscripts to refer to the element or sub-
matrix. In this type of assignment, the subscripts appear on the left-hand side of
the equal sign. For example, to change the value in the first row, second column
of **COFFEE** from 2 to 4, use subscripts to refer to the appropriate element in an
assignment statement:

```
> coffee[1,2]=4;
> print coffee;
```

```
          COFFEE
              4         4         2         3         2
              3         3         1         2         1
              2         1         0         2         1
              5         4         4         3         4
```

To change the values in the last column of **COFFEE** to 0s use the following state-
ment:

```
> coffee[,5]={0,0,0,0};
> print coffee;
```

```
          COFFEE
              4         4         2         3         0
              3         3         1         2         0
              2         1         0         2         0
              5         4         4         3         0
```

As before, you could also use previously assigned row and column labels:

```
>  coffee[,'FRI']={0,0,0,0};
```

In the next example, you first locate the positions of negative elements of a matrix and then set these elements equal to 0. This can be useful in situations where negative elements may indicate errors or be impossible values. The LOC function is useful for creating an index vector for a matrix that satisfies some condition.

In the following example, the LOC function is used to find the positions of the negative elements of the matrix **T** and then to set these elements equal to 0 using subscripted assignment:

```
>  t={ 3   2 -1,
>        6 -4   3,
>        2   2   2 };
>  print t;
```

```
             T
             3            2           -1
             6           -4            3
             2            2            2
```

```
>  i=loc(t<0);
>  print i;
```

```
             I
             3            5
```

```
>  t[i]=0;
>  print t;
```

```
             T
             3            2            0
             6            0            3
             2            2            2
```

Subscripts can also contain expressions with results that are either row or column vectors. These statements can also be written

```
>  t[loc(t<0)]=0;
```

If you use a noninteger value as a subscript, only the integer portion is used. Using a subscript value less than one or greater than the dimension of the matrix results in an error.

Subscript Reduction Operators

You can use reduction operators, which return a matrix of reduced dimension, in place of values for subscripts to get reductions across all rows and columns. Table 4.3 lists the eight operators for subscript reduction in IML.

Table 4.3. Subscript Reduction Operators

Operator	Action
+	addition
#	multiplication
<>	maximum
><	minimum
<:>	index of maximum
>:<	index of minimum
:	mean
##	sum of squares

For example, to get column sums of the matrix $\mathbf{X}$ (sum across the rows, which reduces the row dimension to 1), specify X[+,]. The first subscript (+) specifies that summation reduction take place across the rows. Omitting the second subscript, corresponding to columns, leaves the column dimension unchanged. The elements in each column are added, and the new matrix consists of one row containing the column sums.

You can use these operators to reduce either rows or columns or both. When both rows and columns are reduced, row reduction is done first.

For example, the expression A[+, <>] results in the maximum (<>) of the column sums (+).

You can repeat reduction operators. To get the sum of the row maxima, use the expression A[, <>][+,].

A subscript such as A[{23}, +] first selects the second and third rows of $\mathbf{A}$ and then finds the row sums of that matrix. The following examples demonstrate how to use the operators for subscript reduction.

$$\text{Let } \mathbf{A} = \begin{bmatrix} 0 & 1 & 2 \\ 5 & 4 & 3 \\ 7 & 6 & 8 \end{bmatrix}$$

The following statements are true:

$$\mathbf{A}[23, +] \text{ yields } \begin{bmatrix} 12 \\ 21 \end{bmatrix} \text{ (row sums for rows 2 and 3)}$$

$$\mathbf{A}[+, <>] \text{ yields } \begin{bmatrix} 13 \end{bmatrix} \text{ (maximum of column sums)}$$

$\mathbf{A}[<>, +]$ yields $\begin{bmatrix} 21 \end{bmatrix}$ (sum of column maxima)

$\mathbf{A}[, ><][+,]$ yields $\begin{bmatrix} 9 \end{bmatrix}$ (sum of row minima)

$\mathbf{A}[, <:>]$ yields $\begin{bmatrix} 3 \\ 1 \\ 3 \end{bmatrix}$ (indices of row maxima)

$\mathbf{A}[>:<,]$ yields $\begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ (indices of column minima)

$\mathbf{A}[:]$ yields $\begin{bmatrix} 4 \end{bmatrix}$ (mean of all elements)

Displaying Matrices with Row and Column Headings

You can tailor the way your matrices are displayed with the AUTONAME option, the ROWNAME= and COLNAME= options, or the MATTRIB statement.

Using the AUTONAME Option

You can use the RESET statement with the AUTONAME option to automatically display row and column headings. If your matrix has n rows and m columns, the row headings are ROW1 to ROWn and the column headings are COL1 to COLm. For example, the following statements produce the following results:

```
> reset autoname;
> print coffee;
```

COFFEE	COL1	COL2	COL3	COL4	COL5
ROW1	4	2	2	3	2
ROW2	3	3	1	2	1
ROW3	2	1	0	2	1
ROW4	5	4	4	3	4

Using the ROWNAME= and COLNAME= Options

You can specify your own row and column headings. The easiest way is to create vectors containing the headings and then display the matrix is with the ROWNAME= and COLNAME= options. For example, the following statements produce the following results:

```
> names={jenny linda jim samuel};
> days={mon tue wed thu fri};
> print coffee[rowname=names colname=days];
```

COFFEE	MON	TUE	WED	THU	FRI
JENNY	4	2	2	3	2
LINDA	3	3	1	2	1
JIM	2	1	0	2	1
SAMUEL	5	4	4	3	4

Using the MATTRIB Statement

The MATTRIB statement associates printing characteristics with matrices. You can use the MATTRIB statement to display **COFFEE** with row and column headings. In addition, you can format the displayed numeric output and assign a label to the matrix name. The following example shows how to tailor your displayed output:

```
> mattrib coffee rowname=({jenny linda jim samuel})
>         colname=({mon tue wed thu fri})
>         label='Weekly Coffee'
>         format=2.0;
> print coffee;
```

Weekly Coffee	MON	TUE	WED	THU	FRI
JENNY	4	2	2	3	2
LINDA	3	3	1	2	1
JIM	2	1	0	2	1
SAMUEL	5	4	4	3	4

More on Missing Values

Missing values in matrices are discussed in You should read that chapter and carefully so that you are aware of the way IML treats missing values. Following are several examples that show how IML handles missing values in a matrix.

$$\text{Let } \mathbf{X} = \begin{bmatrix} 1 & 2 & . \\ . & 5 & 6 \\ 7 & . & 9 \end{bmatrix} \text{ and } \mathbf{Y} = \begin{bmatrix} 4 & . & 2 \\ 2 & 1 & 3 \\ 6 & . & 5 \end{bmatrix}$$

The following statements are true:

$$\mathbf{X} + \mathbf{Y} \text{ yields } \begin{bmatrix} 5 & . & . \\ . & 6 & 9 \\ 13 & . & 14 \end{bmatrix} \text{ (matrix addition)}$$

$$\mathbf{X}\#\mathbf{Y} \text{ yields } \begin{bmatrix} 4 & . & . \\ . & 5 & 18 \\ 42 & . & 45 \end{bmatrix} \text{ (element multiplication)}$$

$$\mathbf{X}[+,] \text{ yields } \begin{bmatrix} 8 & 7 & 15 \end{bmatrix} \text{ (column sums)}$$

Chapter 5
Programming Statements

Chapter Contents

Chapter 5
Programming Statements

Overview

As a programming language, the Interactive Matrix Language has many features that enable you to control the path of execution through the statements. The control statements in IML function in a way that is similar to the corresponding statements in the SAS DATA step. This chapter presents the following control features:

- IF-THEN/ELSE statements
- DO groups
- iterative execution
- jumping (nonconsecutive execution)
- module definition and execution
- termination of execution.

IF-THEN/ELSE Statements

To perform an operation conditionally, use an IF statement to test an expression. Alternative actions appear in a THEN clause and, optionally, an ELSE statement. The general form of the IF-THEN/ELSE statement is

IF *expression* **THEN** *statement1* ;

ELSE *statement2* ;

The IF expression is evaluated first. If the expression is true, execution flows through the THEN alternative. If the expression is false, the ELSE statement, if present, is executed. Otherwise, the next statement is executed.

The expression to be evaluated is often a comparison, for example,

```
if max(a)<20 then p=0;
else p=1;
```

The IF statement results in the evaluation of the condition (MAX(A)<20). If the largest value found in matrix **A** is less than 20, P is set to 0. Otherwise, P is set to 1.

You can nest IF statements within the clauses of other IF or ELSE statements. Any number of nesting levels is allowed. The following is an example of nested IF statements:

```
if x=y then
   if abs(y)=z then w=-1;
   else w=0;
else w=1;
```

When the condition to be evaluated is a matrix expression, the result of the evaluation is a temporary matrix of 0s, 1s, and possibly missing values. If all values of the result matrix are nonzero and nonmissing, the condition is true; if any element in the result matrix is 0, the condition is false. This evaluation is equivalent to using the ALL function.

For example, the statement

```
if x<y then   statement;
```

produces the same result as the statement

```
if all(x<y) then   statement;
```

The expressions

```
if a^=b then   statement;
```

and

```
if ^(a=b) then   statement;
```

are valid, but the THEN clause in each case is executed only when all corresponding elements of **A** and **B** are unequal.

If you require that only one element in **A** not be equal to its corresponding element in **B**, use the ANY function. For example, evaluation of the expression

```
if any(a^=b) then   statement;
```

requires only one element of **A** and **B** to be unequal for the expression to be true.

DO Groups

A set of statements can be treated as a unit by putting them into a DO group, which starts with a DO statement and ends with an END statement. In this way, you can submit the entire group of statements for execution as a single unit. For some programming applications, you must use either a DO group or a module. For example, LINK and GOTO statements must be programmed inside of a DO group or a module.

The two principal uses of DO groups are

- to group a set of statements so that they are executed as a unit
- to group a set of statements for a conditional (IF-THEN/ELSE) clause

DO groups have the following general form:

DO ;

   additional statements

END ;

You can nest DO groups to any level, just like you nest IF-THEN/ELSE statements. The following is an example of nested DO groups:

```
do;
    statements;
    do;
        statements;
        do;
            statements;
        end;
    end;
end;
```

It is good practice to indent the statements in DO groups, as shown abovein the preceding statements so that their position indicates the levels of nesting.

For IF-THEN/ELSE conditionals, DO groups can be used as units for either THEN or ELSE clauses so that you can perform many statements as part of the conditional action. An example follows:

```
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;
```

Iterative Execution

The DO statement also serves the feature of iteration. With a DO statement, you can repeatedly execute a set of statements until some condition stops the execution. A DO statement is iterative if you specify it with any of the following iteration clauses. The type of clause determines when to stop the repetition.

Clause	DO Statement
DATA	DO DATA statement
variable = *start* TO *stop* < BY *increment* >	iterative DO statement
WHILE(*expression*)	DO WHILE statement
UNTIL(*expression*)	DO UNTIL statement

A DO statement can have any combination of these four iteration clauses, but a given DO statement must be specified in the order listed in the preceding table.

DO DATA Statement

The general form of the DO DATA statement is

DO DATA ;

The DATA keyword specifies that iteration is to stop when an end-of-file condition occurs. The group is exited immediately upon encountering the end-of-file condition. Other DO specifications exit after tests are performed at the top or bottom of the loop.

See Chapter 6, "Working with SAS Data Sets," and Chapter 7, "File Access," for more information about processing data.

You can use the DO DATA statement to read data from an external file or to process observations from a SAS data set. In the DATA step in base SAS software, the iteration is usually implied. The DO DATA statement simulates this iteration until the end of file is reached.

The following example reads data from an external file named MYDATA and inputs the data values into a vector. The data values are read one at a time into the dummy variable XX and collected into the vector **X** using the vertical concatenation operator (//) after each value is read.

```
infile 'mydata';                      /* infile statement    */
do data;                              /* begin read loop     */
   input xx;                          /* read a data value   */
   x=x//xx;                           /* concatenate values  */
end;                                  /* end loop            */
```

Iterative DO Statement

The general form of the iterative DO statement is

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

The *variable* sequence specification assigns the *start* value to the given variable. This value is then incremented by the *increment* value (or by 1 if *increment* is not specified) until it is greater than or equal to the *stop* value. (If *increment* is negative, then the iterations stop when the value is less than or equal to *stop*.)

For example, the following statement specifies a DO loop that executes by multiples of 10 until I is greater than 100:

```
do i=10 to 100 by 10;
```

DO WHILE Statement

The general form of the DO WHILE statement is

DO WHILE *expression***;**

With a WHILE clause, the expression is evaluated at the beginning of each loop, with repetition continuing until the expression is false (that is, until the value contains a 0 or missing value). Note that if the expression is false the first time it is evaluated, the loop is not executed.

For example, if the variable COUNT has an initial value of 1, the statements

```
do while(count<5);
   print count;
   count=count+1;
end;
```

print COUNT four times.

DO UNTIL Statement

The general form of the DO UNTIL statement is

DO UNTIL *expression***;**

The UNTIL clause is like the WHILE clause except that the expression is evaluated at the bottom of the loop. This means that the loop always executes at least once.

For example, if the variable COUNT has an initial value of 1, the statements

```
do until(count>5);
   print count;
   count=count+1;
end;
```

print COUNT five times.

Jumping

During normal execution, statements are executed one after another. The GOTO and LINK statements instruct IML to jump from one part of a program to another. The place to which execution jumps is identified by a *label*, which is a name followed by a colon placed before an executable statement. You can program a jump by using either the GOTO statement or the LINK statement:

GOTO *label***;**

LINK *label***;**

Both the GOTO and the LINK statements instruct IML to jump immediately to the labeled statement. The LINK statement, however, reminds IML where it jumped from so that execution can be returned there if a RETURN statement is encountered.

The GOTO statement does not have this feature. Thus, the LINK statement provides a way of calling sections of code as if they were subroutines. The LINK statement calls the routine. The routine begins with the label and ends with a RETURN statement. LINK statements can be nested within other LINK statements to any level.

CAUTION: The GOTO and LINK statements are limited to being inside a module or DO group. These statements must be able to resolve the referenced label within the current unit of statements. Although matrix symbols can be shared across modules, statement labels cannot. Therefore, all GOTO statement labels and LINK statement labels must be local to the DO group or module.

The GOTO and LINK statements are not often used because you can usually write more understandable programs by using other features, such as DO groups for conditionals, iterative DO groups for looping, and module invocations for subroutine calls.

Here are two DO groups that illustrate how the GOTO and LINK statements work:

```
do;                                     do;
   if x<0 then goto negative;              if x<0 then link negative;
   y=sqrt(x);                              y=sqrt(x);
   print y;                                print y;
   stop;                                   stop;
negative:                               negative:
   print "Sorry, X is negative";           print "Using Abs. value of negative X";
end;                                        x=abs(x);
                                            return;
                                        end;
```

The following is a comparable way to write the program on the left without using GOTO or LINK statements:

```
if x<0 then print "Sorry, X is negative";
else
   do;
      y=sqrt(x);
      print y;
   end;
```

Modules

Modules are used for

- creating groups of statements that can be invoked as a unit from anywhere in the program, that is, making a subroutine or function

- creating a separate (symbol-table) environment, that is, defining variables that are local to the module rather than global

A module always begins with the START statement and ends with the FINISH statement. Modules can be thought of as being either functions or subroutines. When a module returns a single parameter, it is called a function and is executed as if it were

a built-in IML function; a function is invoked by its name in an assignment statement rather than in a CALL or RUN statement. Otherwise, a module is called a subroutine, and you execute the module in either the RUN statement or the CALL statement.

Defining and Executing a Module

Modules begin with a START statement, which has the general form

> **START** $<$ *name* $>$ $<$ **(** *arguments* **)** $>$ $<$ **GLOBAL(** *arguments* **)** $>$ **;**

Modules end with a FINISH statement, which has the general form

> **FINISH** $<$ *name* $>$ **;**

If no name appears in the START statement, the name of the module defaults to MAIN.

There are two ways you can execute a module.

You can use either a RUN statement or a CALL statement. The only difference is the order of resolution.

The general forms of these statements are

> **RUN** *name* $<$ **(** *arguments* **)** $>$ **;**

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

The RUN and CALL statements must have arguments to correspond to the ones defined for the modules they invoke. A module can call other modules provided that it never recursively calls itself.

The RUN and CALL statements have orders of resolution that need to be considered only when you have given a module the same name as a built-in IML subroutine. In such cases, use the CALL statement to execute the built-in subroutine and the RUN statement to execute the user-defined module.

The RUN statement is resolved in the following order:

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

The CALL statement is resolved in the following order:

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

Nesting Modules

You can nest one module within another. You must make sure that each nested module is completely contained inside of the parent module. Each module is collected independently of the others. When you nest modules, it is a good idea to indent the statements relative to the level of nesting, as shown in the following example:

```
start a;
   reset print;
   start b;
      a=a+1;
   finish b;
   run b;
finish a;
run a;
```

In this example, IML starts collecting statements for a module called A. In the middle
of this module, it recognizes the start of a new module called B. It saves its current
work on A and collects B until encountering the first FINISH statement. It then
finishes collecting A. Thus, it behaves the same as if B were collected before A, as
shown below:

```
start b;
   a=a+1;
finish;
start a;
   reset print;
   run b;
finish;
run a;
```

Understanding Symbol Tables

Whenever a variable is defined outside of the module environment, its name is stored
in the *global symbol table*. Whenever you are programming in immediate mode out-
side of a module, you are working with symbols (variables) from the global symbol
table. For each module you define with arguments given in a START statement, a
separate symbol table called a *local symbol table* is created for that module. All sym-
bols (variables) used inside the module are stored in its local symbol table. There
can be many local symbol tables, one for each module with arguments. A symbol
can exist in the global table, one or more local tables, or in both the global table and
one or more local tables. Also, depending on how a module is defined, there can be
a one-to-one correspondence between variables across symbol tables (although there
need not be any connection between a variable, say X, in the global table and a vari-
able X in a local table). Values of symbols in a local table are temporary; that is, they
exist only while the module is executing and are lost when the module has finished
execution. Whether or not these temporary values are transferred to corresponding
global variables depends on how the module is defined.

Modules with No Arguments

When you define a module with no arguments, a local symbol table is not created. All symbols (variables) are global, that is, equally accessible inside and outside the module. The symbols referenced inside the module are the same as the symbols outside the module environment. This means that variables created inside the module are also global, and any operations done on variables inside a module affect the global variables as well.

The following example shows a module with no arguments:

```
>      /* module without arguments, all symbols are global.  */
>  proc iml;
>  a=10;                                /* A is global     */
>  b=20;                                /* B is global     */
>  c=30;                                /* C is global     */
>  start mod1;                          /* begin module    */
>     p=a+b;                            /* P is global     */
>     q=b-a;                            /* Q is global     */
>     c=40;                             /* C already global */
>  finish;                              /* end module      */

   NOTE: Module MOD1 defined.

>  run mod1;
>  print a b c p q;

                        A         B         C         P         Q
                       10        20        40        30        10
```

Note that after executing the module,

- A is still 10

- B is still 20

- C has been changed to 40

- P and Q are created, added to the global symbol table, and set to 30 and 10, respectively

Modules with Arguments

In general, the following statements are true about modules with arguments:

- You can specify arguments as variable names.

- If you specify several arguments, use commas to separate them.

- If you have both output variables and input variables, it is good practice to list the output variables first.

- When a module is invoked with either a RUN or a CALL statement, the arguments can be any name, expression, or literal. However, when using arguments for output results, use variable names rather than expressions or literals.

When a module is executed with either a RUN or a CALL statement, the value for each argument is transferred from the global symbol table to the local symbol table. For example, consider the module MOD2 defined in the following statements. The first four statements are submitted in the global environment, and they define variables (A,B,C, and D): the values of these variables are stored in the global symbol table. The START statement begins definition of MOD2 and lists two variables (X and Y) as arguments. This creates a local symbol table for MOD2. All symbols used inside the module (X, Y, P, Q, and C) are in the local symbol table. There is also a one-to-one correspondence between the arguments in the RUN statement (A and B) and the arguments in the START statement (X and Y). Also note that A, B, and D exist only in the global symbol table, whereas X, Y, P, and Q exist only in the local symbol table. The symbol C exists independently in both the local and global tables. When MOD2 is executed with the statement RUN MOD2(A,B), the value of A is transferred from the global symbol table to X in the local table. Similarly, the value of B in the global table is transferred to Y in the local table. Because C is not an argument, there is no correspondence between the value of C in the global table and the value of C in the local table. When the module finishes execution, the final values of X and Y in the local table are transferred back to A and B in the global table.

```
>   proc iml;
>   a=10;
>   b=20;
>   c=30;
>   d=90;
>   start mod2(x,y);                       /* begin module */
>       p=x+y;
>       q=y-x;
>       y=100;
>       c=25;
>   finish mod2;                           /* end module   */

    NOTE: Module MOD2 defined.

>   run mod2(a,b);
>   print a b c d;
```

A	B	C	D
10	100	30	90

The PRINT statement prints the values of variables in the global symbol table.

Notice that

- A is still 10
- B is changed to 100 since the corresponding argument Y was changed to 100 inside the module

- C is still 30. Inside the module, the local symbol C was set equal to 25, but there is no correspondence between the global symbol C and the local symbol C.

- D is still 90

Also note that, inside the module, the symbols A, B, and D do not exist. Outside the module, the symbols P, Q, X, and Y do not exist.

Defining Function Modules

Functions are special modules that return a single value. They are a special type of module because modules can, in general, return any number of values through their argument list. To write a function module, include a RETURN statement that assigns the returned value to a variable. The RETURN statement is necessary for a module to be a function. You invoke a function module in an assignment statement, as you would a standard function.

The symbol-table logic described in the preceding section also applies to function modules. The following is an example of a function module. In this module, the value of C in the local symbol table is transferred to the global symbol Z.

```
>  proc iml;
>  a=10;
>  b=20;
>  c=30;
>  d=90;
>  start mod3(x,y);
>     p=x+y;
>     q=y-x;
>     y=100;
>     c=40;
>     return (c);            /* return function value */
>  finish mod3;

   NOTE: Module MOD3 defined.

>  z = mod3(a,b);            /* call function       */
>  print a b c d z;
```

A	B	C	D	Z
10	100	30	90	40

Note the following about this example:

- A is still 10.

- B is changed to 100 because Y is set to 100 inside the module, and there is a one-to-one correspondence between B and Y.

- C is still 30. The symbol C in the global table has no connection with the symbol C in the local table.

- Z is set to 40, which is the value of C in the local table.

Again note that, inside the module, the symbols A, B, D, and Z do not exist. Outside the module, symbols P, Q, X, and Y do not exist.

In the next example, you define your own function ADD for adding two arguments:

```
>  proc iml;
>  reset print;
>  start add(x,y);
>      sum=x+y;
>      return(sum);
>  finish;

NOTE: Module ADD defined.

>    a={9 2,5 7};
```

```
                                        A
                               9                 2
                               5                 7
```

```
>    b={1 6,8 10};
```

```
                                        B
                               1                 6
                               8                10
```

```
>    c=add(a,b);
```

```
                                        C
                              10                 8
                              13                17
```

Function modules can also be called inside each other. For example, in the following statements, the ADD function is called twice from within the first ADD function:

```
>  d=add(add(6,3),add(5,5));
>  print d;
                                            D
                                           19
```

Functions are resolved in this order:

1. IML built-in function
2. user-defined function module
3. SAS DATA step function

This means that you should not use a name for a function that is already the name of an IML built-in function.

Using the GLOBAL Clause

For modules with arguments, the variables used inside the module are local and have no connection with any variables of the same name existing outside the module in the global table. However, it is possible to specify that certain variables not be placed in the local symbol table but rather be accessed from the global table. Use the GLOBAL clause to specify variables you want shared between local and global symbol tables. The following is an example of a module using a GLOBAL clause to define the symbol C as global. This defines a one-to-one correspondence between the value of C in the global table and the value of C in the local table.

```
>  proc iml;
>  a=10;
>  b=20;
>  c=30;
>  d=90;
>  start mod4(x,y) global (c);
>      p=x+y;
>      q=y-x;
>      y=100;
>      c=40;
>      d=500;
>  finish mod4;

   NOTE: Module MOD4 defined.

>  run mod4(a,b);
>  print a b c d;
```

A	B	C	D
10	100	40	90

Note the following about this example:

- A is still 10.
- B is changed to 100.
- C is changed to 40 because it was declared global. The C inside the module and outside the module are the "same."
- D is still 90 and not 500, since D independently exists in the global and local symbol tables.

Also note that every module with arguments has its own local table; thus it is possible to have a global and many local tables. A variable can independently exist in one or more of these tables. However, a variable can be commonly shared between the global and any number of local tables when the GLOBAL clause is used.

Nesting Modules with Arguments

For nested module calls, the concept of global and local symbol tables is somewhat different. Consider the following example:

```
>  start mod1 (a,b);
>      c=a+b;
>      d=a-b;
>      run mod2 (c,d);
>      print c d;
>  finish mod1;

   NOTE: Module MOD1 defined.

>  start mod2 (x,y);
>      x=y-x;
>      y=x+1;
>      run mod3(x,y);
>  finish mod2;

   NOTE: Module MOD2 defined.

>  start mod3(w,v);
>      w=w#v;
>  finish mod3;

   NOTE: Module MOD3 defined.
```

The local symbol table of MOD1 in effect becomes the global table for MOD2. The local symbol table of MOD2 is the global table for MOD3. The distinction between the global and local environments is necessary only for modules with arguments. If a module (say, A) calls another module (say, B) which has no arguments, B shares all the symbols existing in A's local symbol table.

For example, consider the following statements:

```
>  x=457;
>  start a;
>      print 'from a' x;
>      finish;
>  start b(p);
>      print 'from b' p;
>      run a;
>      finish;
>  run b(x);
```

```
                                          P
                          from b          457

     ERROR: Matrix X has not been set to a value.
     Error occurred in module A
     called    from    module B
     stmt: PRINT

     Paused in module A.
```

In this example, module A is called from module B. Therefore, the local symbol table
of module B becomes the global symbol table for module A. Module A has access to
all symbols available in module B. No X exists in the local environment of module B;
thus no X is available in module A as well. This causes the error that X is unvalued.

More about Argument Passing

You can pass expressions and subscripted matrices as arguments to a module, but you
must be careful and understand the way IML evaluates the expressions and passes
results to the module. Expressions are evaluated, and the evaluated values are stored
in temporary variables. Similarly, submatrices are created from subscripted variables
and stored in temporary variables. The temporary variables are passed to the module
while the original matrix remains intact. Notice that, in the example that follows, the
matrix X remains intact. You might expect X to contain the squared values of Y.

```
     >  proc iml;
     >  reset printall;
     >  start square(a,b);
     >     a=b##2;
     >  finish;
     >     /*  create two data matrices                        */
     >  x={5 9 };

               X                1 row      2 cols     (numeric)

                                  5         9

     >  y={10 4};

               Y                1 row      2 cols     (numeric)

                                 10         4
     >     /*  pass matrices to module element-by-element      */
     >  do i=1 to 2;
     >     run square(x[i],y[i]);
     >  end;
     >     /*  RESET PRINTALL prints all intermediate results  */
```

```
            I              1 row        1 col     (numeric)

                                          1


          #TEM1002         1 row        1 col     (numeric)

                                         10


          #TEM1001         1 row        1 col     (numeric)

                                          5


            A              1 row        1 col     (numeric)

                                        100

          #TEM1002         1 row        1 col     (numeric)

                                          4


          #TEM1001         1 row        1 col     (numeric)

                                          9

            A              1 row        1 col     (numeric)

                                         16
```

```
>      /*  show X and Y are unchanged                          */
>  print x y;


                    X                      Y
                    5          9          10          4
```

The symbol X remains unchanged because the temporary variables that you generally do not see are changed. Note that IML will properly warn you of any such instances in which your results may be lost to the temporary variables.

Module Storage

You can store and reload modules using the forms of the STORE and LOAD statements as they pertain to modules:

STORE MODULE= *name***;**

LOAD MODULE= *name***;**

You can view the names of the modules in storage with the SHOW statement:

```
show storage;
```

See Chapter 14, "Storage Features," for details on using the library storage facilities.

Stopping Execution

You can stop execution with a PAUSE, STOP, or ABORT statement. The QUIT statement is also a stopping statement, but it immediately removes you from the IML environment; the other stopping statements can be performed in the context of a program. Following are descriptions of the STOP, ABORT, and PAUSE statements.

PAUSE Statement

The general form of the PAUSE statement is

PAUSE $<$ *message* $> < * > $ **;**

The PAUSE statement

- stops execution of the module
- remembers where it stopped executing
- prints a pause *message* that you can specify
- puts you in immediate mode within the module environment using the module's local symbol table. At this point you can enter more statements.

A RESUME statement enables you to continue execution at the place where the most recent PAUSE statement was executed.

You can use a STOP statement as an alternative to the RESUME statement to remove the paused states and return to the immediate environment outside of the module.

You can specify a message in the PAUSE statement to display a message as the pause prompt. If no message is specified, IML displays the following default message:

```
paused in module \ob XXX\obe
```

where XXX is the name of the module containing the pause. To suppress the display of any messages, use the * option:

```
pause *;
```

The followingw are some examples of PAUSE statements with operands:

```
pause "Please enter an assignment for X, then enter RESUME;";
```

```
msg ="Please enter an assignment for X, then enter RESUME;";
pause msg;
```

When you use the PAUSE, RESUME, STOP, or ABORT statement, you should be aware of the following details:

- The PAUSE statement can be issued only from within a module.
- IML diagnoses an error if you execute a RESUME statement without any pauses outstanding.
- You can define and execute modules while paused from other modules.
- A PAUSE statement is automatically issued if an error occurs while executing statements inside a module. This gives you an opportunity to correct the error and resume execution of the module with a RESUME statement. Alternately, you can submit a STOP or ABORT statement to exit from the module environment.
- You cannot reenter or redefine an active (paused) module; you will get an error for recursive module execution.
- In paused mode, you can run another module that, in turn, pauses; the paused environments are stacked.
- You can put a RESUME statement inside a module. For example, suppose you are paused in module A and then run module B, which executes a RESUME statement. Execution is resumed in module A and does not return to module B.
- IML supports stopping execution while in a paused state in both subroutine and function modules.
- If you pause in a subroutine module that has its own symbol table, then the immediate mode during the pause uses this symbol table rather than the global one. You must use a RESUME or a STOP statement to return to the global symbol table environment.
- You can use the PAUSE and RESUME statements, in conjunction with the PUSH, QUEUE, and EXECUTE subroutines described in Chapter 15, "Using SAS/IML Software to Generate IML Statements," to execute IML statements that you generate within a module.

STOP Statement

The general form of the STOP statement is

STOP ;

The STOP statement stops execution and returns you to immediate mode, where new statements that you enter are executed. If execution is interrupted by a PAUSE statement, the STOP statement clears all pauses and returns to immediate mode of execution.

ABORT Statement

The general form of the ABORT statement is

ABORT ;

The ABORT statement stops execution and exits from IML much like a QUIT statement, except that the ABORT statement is executable and programmable. For example, you may want to exit IML if a certain error occurs. You can check for the error in a module and program an ABORT statement to execute if the error occurs. The ABORT statement does not execute until the module is executed, while the QUIT statement executes immediately and ends the IML session.

Summary

In this chapter you learned the basics of programming with SAS/IML software. You learned about conditional execution (IF-THEN/ELSE statements), grouping statements as a unit (DO groups), iterative execution, nonconsecutive execution, defining subroutines and functions (modules), and stopping execution. With these programming capabilities, you are able to write your own sophisticated programs and store the code as a module. You can then execute the program later with a RUN or CALL statement.

Chapter 6
Working with SAS Data Sets

Chapter Contents

Chapter 6
Working with SAS Data Sets

Overview

SAS/IML software has many statements for passing data from SAS data sets to matrices and from matrices to SAS data sets. You can create matrices from the variables and observations of a SAS data set in several ways. You can create a column vector for each data set variable, or you can create a matrix where columns correspond to data set variables. You can use all the observations in a data set or use a subset of them.

You can also create a SAS data set from a matrix. The columns correspond to data set variables and the rows correspond to observations. Data management commands enable you to edit, append, rename, or delete SAS data sets from within the SAS/IML environment.

When reading a SAS data set, you can read any number of observations into a matrix either sequentially, directly by record number, or conditionally according to conditions in a WHERE clause. You can also index a SAS data set. The indexing capability facilitates retrievals by the indexed variable.

Operations on SAS data sets are performed with straightforward, consistent, and powerful statements. For example, the LIST statement can perform the following tasks:

- list the next record
- list a specified record
- list any number of specified records
- list the whole file
- list records satisfying one or more conditions
- list specified variables or all variables

If you want to read values into a matrix, use the READ statement instead of the LIST statement with the same operands and features as the LIST statement. You can specify operands that control which records and variables are used indirectly, as matrices, so that you can dynamically program the records, variables, and conditional values you want.

In this chapter, you use the SAS data set CLASS, which contains the variables NAME, SEX, AGE, HEIGHT, and WEIGHT, to learn about

- opening a SAS data set
- examining the contents of a SAS data set

- displaying data values with the LIST statement
- reading observations from a SAS data set into matrices
- editing a SAS data set
- creating a SAS data set from a matrix
- displaying matrices with row and column headings
- producing summary statistics
- sorting a SAS data set
- indexing a SAS data set
- similarities and differences between the data set and the SAS DATA step

Throughout this chapter, the right angle brackets (>) indicate statements that you submit; responses from Interactive Matrix Language follow.

First, invoke the IML procedure:

```
> proc iml;

IML Ready
```

Opening a SAS Data Set

Before you can access a SAS data set, you must first submit a command to open it. There are three ways to open a SAS data set:

- To simply read from an existing data set, submit a USE statement to open it for read access. The general form of the USE statement is

    USE *SAS-data-set* < **VAR** *operand* > < **WHERE(***expression***)** > **;**

    With read access, you can use the FIND, INDEX, LIST, and READ statements on the data set.

- To read and write to an existing data set, use the EDIT statement. The general form of the EDIT statement is

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

    This statement enables you to use both the reading statements (LIST, READ, INDEX, and FIND) and the writing statements (REPLACE, APPEND, DELETE, and PURGE).

- To create a new data set, use the CREATE statement to open a new data set for both output and input. The general form of the CREATE statement is

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

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

Use the APPEND statement to place the matrix data into the newly created data set. If you don't use the APPEND statement, the new data set has no observations.

If you want to list observations and create matrices from the data in the SAS data set CLASS, you must first submit a statement to open the CLASS data set. Because CLASS already exists, specify the USE statement.

Making a SAS Data Set Current

IML data processing commands work on the current data set. This feature makes it unnecessary for you to specify the data set as an operand each time. There are two current data sets, one for input and one for output. IML makes a data set the current one as it is opened. You can also make a data set current by using two setting statements, SETIN and SETOUT:

- The USE and SETIN statements make a data set current for input.
- The SETOUT statement makes a data set current for output.
- The CREATE and EDIT statements make a data set current for both input and output.

If you issue a USE, EDIT, or CREATE statement for a data set that is already open, the data set is made the current data set. To find out which data sets are open and which are current input and current output data sets, use the SHOW DATASETS statement.

The current observation is set by the last operation that performed input/output (I/O). If you want to set the current observation without doing any I/O, use the SETIN (or SETOUT) statement with the POINT option. After a data set is opened, the current observation is set to 0. If you attempt to list or read the current observation, the current observation is converted to 1. You can make the data set CLASS current for input and position the pointer at the tenth observation with the statement

```
> setin class point 10;
```

Displaying SAS Data Set Information

You can use SHOW statements to display information about your SAS data sets. The SHOW DATASETS statement lists all open SAS data sets and their status. The SHOW CONTENTS statement displays the variable names and types, the size, and the number of observations in the current input data set. For example, to get information for the CLASS data set, issue the following statements:

```
> use class;
> show datasets;
```

```
          LIBNAME MEMNAME    OPEN MODE    STATUS
          ------- -------    ---------    ------
          WORK    .CLASS     Input        Current Input
```

```
> show contents;
```

```
          VAR NAME    TYPE    SIZE
          NAME        CHAR     8
          SEX         CHAR     8
          AGE         NUM      8
          HEIGHT      NUM      8
          WEIGHT      NUM      8
          Number of Variables:    5
          Number of Observations: 19
```

As you can see, CLASS is the only data set open. The USE statement opens it for input, and it is the current input data set. The full name for CLASS is WORK.CLASS. The libref is the default, WORK. The next section tells you how to change the libref to another name.

Referring to a SAS Data Set

The USE, EDIT, and CREATE statements take as their first operand the data set name. This name can have either one or two levels. If it is a two-level name, the first level refers to the name of the SAS data library; the second name is the data set name. If the libref is WORK, the data set is stored in a directory for temporary data sets; these are automatically deleted at the end of the session. Other librefs are associated with SAS data libraries using the LIBNAME statement.

If you specify only a single name, then IML supplies a default libref. At the beginning of an IML session, the default libref is SASUSER if SASUSER is defined as a libref or WORK otherwise. You can reset the default libref by using the RESET DEFLIB statement. If you want to create a permanent SAS data set, you must specify a two-level name using the RESET DEFLIB statement (refer to the chapter on SAS files in *SAS Language Reference: Concepts* for more information about permanent SAS data sets).

```
> reset deflib=name;
```

Listing Observations

You can list variables and observations in a SAS data set with the LIST statement. The general form of the LIST statement is

> **LIST** < *range* > < **VAR** *operand* > < **WHERE(***expression***)** > **;**

where

range	specifies a range of observations.
operand	selects a set of variables.
expression	is an expression that is evaluated for being true or false.

The next three sections discuss how to use each of these clauses with the CLASS data set.

Specifying a Range of Observations

You can specify a range of observations with a keyword or by record number using the POINT option. You can use the *range* operand with the data management statements DELETE, FIND, LIST, READ, and REPLACE.

You can specify *range* using any of the following keywords:

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

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

If you want to list all observations in the CLASS data set, use the keyword ALL to indicate that the range is all observations:

```
> list all;

   OBS NAME     SEX           AGE     HEIGHT     WEIGHT
   ------ -------- -------- --------- --------- ---------
        1 JOYCE    F         11.0000   51.3000    50.5000
        2 THOMAS   M         11.0000   57.5000    85.0000
        3 JAMES    M         12.0000   57.3000    83.0000
        4 JANE     F         12.0000   59.8000    84.5000
        5 JOHN     M         12.0000   59.0000    99.5000
        6 LOUISE   F         12.0000   56.3000    77.0000
        7 ROBERT   M         12.0000   64.8000   128.0000
        8 ALICE    F         13.0000   56.5000    84.0000
        9 BARBARA  F         13.0000   65.3000    98.0000
       10 JEFFREY  M         13.0000   62.5000    84.0000
       11 CAROL    F         14.0000   62.8000   102.5000
       12 HENRY    M         14.0000   63.5000   102.5000
       13 ALFRED   M         14.0000   69.0000   112.5000
       14 JUDY     F         14.0000   64.3000    90.0000
       15 JANET    F         15.0000   62.5000   112.5000
       16 MARY     F         15.0000   66.5000   112.0000
       17 RONALD   M         15.0000   67.0000   133.0000
       18 WILLIAM  M         15.0000   66.5000   112.0000
       19 PHILIP   M         16.0000   72.0000   150.0000
```

Without a *range* specification, the LIST statement lists only the current observation,
which in this example is now the last observation because of the previous LIST state-
ment:

```
> list;

   OBS NAME     SEX           AGE     HEIGHT     WEIGHT
   ------ -------- -------- --------- --------- ---------
       19 PHILIP   M         16.0000   72.0000   150.0000
```

Use the POINT keyword with record numbers to list specific observations. You can
follow the keyword POINT with a single record number or with a literal giving several
record numbers.

```
> list point 5;

   OBS NAME     SEX           AGE     HEIGHT     WEIGHT
   ------ -------- -------- --------- --------- ---------
        5 JOHN     M         12.0000   59.0000    99.5000

> list point {2 4 9};

   OBS NAME     SEX           AGE     HEIGHT     WEIGHT
   ------ -------- -------- --------- --------- ---------
        2 THOMAS   M         11.0000   57.5000    85.0000
        4 JANE     F         12.0000   59.8000    84.5000
        9 BARBARA  F         13.0000   65.3000    98.0000
```

You can also indicate the range indirectly by creating a matrix containing the records you want listed:

```
> p={2 4 9};
> list point p;
```

```
         OBS NAME      SEX            AGE     HEIGHT     WEIGHT
         ------ -------- -------- --------- --------- ---------
           2 THOMAS    M          11.0000   57.5000   85.0000
           4 JANE      F          12.0000   59.8000   84.5000
           9 BARBARA   F          13.0000   65.3000   98.0000
```

The *range* operand is usually listed first when you are using the access statements DELETE, FIND, LIST, READ, and REPLACE. Listed below are access statements and their default ranges.

Statement	Default Range
LIST	current
READ	current
FIND	all
REPLACE	current
APPEND	always at end
DELETE	current

Selecting a Set of Variables

You can use the VAR clause to select a set of variables. The general form of the VAR clause is

VAR *operand*

where *operand* can be specified using one of the following:

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

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

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

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

For example, to list students' names from the CLASS data set, use the VAR clause with a literal.

```
> list point p var{name};

          OBS NAME
        ------ --------
            2 THOMAS
            4 JANE
            9 BARBARA
```

To list AGE, HEIGHT, and WEIGHT, you can use the VAR clause with a matrix giving the variables:

```
> v={age height weight};
> list point p var v;

          OBS      AGE    HEIGHT    WEIGHT
        ------ --------- --------- ---------
            2  11.0000   57.5000   85.0000
            4  12.0000   59.8000   84.5000
            9  13.0000   65.3000   98.0000
```

The VAR clause can be used with the following statements for the tasks described:

Statement	VAR Clause Function
APPEND	specifies which IML variables contain data to append to the data set
CREATE	specifies the variables to go in the data set
EDIT	limits which variables are accessed
LIST	specifies which variables to list
READ	specifies which variables to read
REPLACE	specifies which data set variable's data values to replace with corresponding IML variable data values
USE	limits which variables are accessed

Selecting Observations

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

WHERE *variable comparison-op operand* ;

where

variable   is a variable in the SAS data set.

comparison-op   is one of the following comparison operators:

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

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

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

$$\text{^} = \quad \text{^} ? \quad < \quad <= \quad > \quad >=$$

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

$$= \quad ? \quad =: \quad =*$$

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

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

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

For example, to list the names of all males in the data set CLASS, use the following statement:

```
> list all var{name} where(sex='M');
```

```
            OBS                    NAME
          ------  ----------------------
             2                    THOMAS
             3                    JAMES
             5                    JOHN
             7                    ROBERT
            10                    JEFFREY
            12                    HENRY
            13                    ALFRED
            17                    RONALD
            18                    WILLIAM
            19                    PHILIP
```

The WHERE comparison arguments can be matrices. In the following cases using the =* operator, the comparison is made on each name to find a string that sounds like or is spelled similar to the given string or strings:

```
> n={name sex age};
> list all var n where(name=*{"ALFRED","CAROL","JUDY"});

          OBS          NAME      SEX         AGE
        ----- ---------------- -------- ---------
          11          CAROL     F       14.0000
          13          ALFRED    M       14.0000
          14          JUDY      F       14.0000

> list all var n where(name=*{"JON","JAN"});

            OBS NAME       SEX         AGE
          ------ -------- -------- ---------
              4 JANE      F       12.0000
              5 JOHN      M       12.0000
```

To list AGE, HEIGHT, and WEIGHT for all students in their teens, use the following statement:

```
> list all var v where(age>12);

            OBS       AGE    HEIGHT    WEIGHT
          ------ --------- --------- ---------
             8    13.0000   56.5000   84.0000
             9    13.0000   65.3000   98.0000
            10    13.0000   62.5000   84.0000
            11    14.0000   62.8000  102.5000
            12    14.0000   63.5000  102.5000
            13    14.0000   69.0000  112.5000
            14    14.0000   64.3000   90.0000
            15    15.0000   62.5000  112.5000
            16    15.0000   66.5000  112.0000
            17    15.0000   67.0000  133.0000
            18    15.0000   66.5000  112.0000
            19    16.0000   72.0000  150.0000
```

Note: In the WHERE clause, 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 cannot use comparisons involving more than one data set variable in a single comparison; for example, you cannot use either of the following expressions:

```
list all where(height>weight);
list all where(weight-height>0);
```

You could use the first statement if WEIGHT were a matrix name already defined rather than a variable in the SAS data set.

Reading Observations from a SAS Data Set

Transferring data from a SAS data set to a matrix is done using the READ statement. The SAS data set you want to read data from must already be open. You can open a SAS data set with either the USE or the EDIT statement. If you already have several data sets open, you can point to the one you want with the SETIN statement, making it the current input data set. The general form of the READ statement is

READ < *range* > < **VAR** *operand* > < **WHERE(***expression***)** >

    < **INTO** *name* > ;

where

range   specifies a range of observations.

operand   selects a set of variables.

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

name   names a target matrix for the data.

Using the READ Statement with the VAR Clause

Use the READ statement with the VAR clause to read variables from the current SAS data set into column vectors of the VAR clause. Each variable in the VAR clause becomes a column vector with the same name as the variable in the SAS data set. The number of rows is equal to the number of observations processed, depending on the range specification and the WHERE clause. For example, to read the numeric variables AGE, HEIGHT, and WEIGHT for all observations in the CLASS data set, use the following statements:

```
> read all var {age height weight};
```

Now submit the SHOW NAMES statement to display all the matrices you have created so far in this chapter:

```
> show names;
```

```
        AGE           19 rows      1 col   num       8
```

```
HEIGHT          19 rows      1 col  num      8
N                1 row      3 cols char     4
P                1 row      3 cols num      8
V                1 row      3 cols char     6
WEIGHT          19 rows      1 col  num      8
Number of symbols = 8  (includes those without values)
```

You see that, with the READ Statement, you have created the three numeric vectors
AGE, **HEIGHT**, and **WEIGHT**. (Notice that the matrices you created earlier, **N**, **P**,
and **V**, are also listed.) You can select the variables that you want to access with a
VAR clause in the USE statement. The two previous statements can also be written
as

```
use class var{age height weight};
read all;
```

Using the READ Statement with the VAR and INTO Clauses

Sometimes you want to have all of the numeric variables in the same matrix so that
you can determine correlations. Use the READ statement with the INTO clause and
the VAR clause to read the variables listed in the VAR clause into the single matrix
named in the INTO clause. Each variable in the VAR clause becomes a column of
the target matrix. If there are p variables in the VAR clause and n observations are
processed, the target matrix in the INTO clause is an $n \times p$ matrix.

The following statement creates a matrix **X** containing the numeric variables of the
CLASS data set. Notice the use of the keyword _NUM_ in the VAR clause to specify
that all numeric variables be read.

```
> read all var _num_ into x;
> print x;
```

```
                    X
            11      51.3      50.5
            11      57.5        85
            12      57.3        83
            12      59.8      84.5
            12        59      99.5
            12      56.3        77
            12      64.8       128
            13      56.5        84
            13      65.3        98
            13      62.5        84
            14      62.8     102.5
            14      63.5     102.5
            14        69     112.5
            14      64.3        90
            15      62.5     112.5
            15      66.5       112
            15        67       133
            15      66.5       112
            16        72       150
```

Using the READ Statement with the WHERE Clause

Use the WHERE clause as you did with the LIST statement, to conditionally se-
lect observations from within the specified range. If you want to create a matrix
FEMALE containing the variables AGE, HEIGHT, and WEIGHT for females only,
use the following statements.

```
> read all var _num_ into female where(sex="F");
> print female;
```

```
                FEMALE
            11        51.3         50.5
            12        59.8         84.5
            12        56.3           77
            13        56.5           84
            13        65.3           98
            14        62.8        102.5
            14        64.3           90
            15        62.5        112.5
            15        66.5          112
```

Now try some special features of the WHERE clause to find values that begin with
certain characters (the =: operator) or that contain certain strings (the ? operator). To
create a matrix **J** containing the students whose names begin with the letter "J", use
the following statements:

```
> read all var{name} into j where(name=:"J");
> print j;
```

```
                                J
                                JOYCE
                                JAMES
                                JANE
                                JOHN
                                JEFFREY
                                JUDY
                                JANET
```

To creat a matrix **AL** of children with names containing the string "AL", use the
statement

```
> read all var{name} into al where(name?"AL");
> print al;
```

```
                                AL
                                ALICE
                                ALFRED
                                RONALD
```

Editing a SAS Data Set

You can edit a SAS data set using the EDIT statement. You can update values of variables, mark observations for deletion, delete the marked observations, and save the changes you make. The general form of the EDIT statement is

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

where

SAS-data-set names an existing SAS data set.

operand   selects a set of variables.

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

Updating Observations

Suppose you have updated data and want to change some values in the CLASS data set. For instance, suppose that the student named HENRY has had a birthday since the data were added to the CLASS data set. You can

- make the data set CLASS current for input and output
- read the data
- change the appropriate data value
- replace the changed data in the data set

First, submit an EDIT statement to make the CLASS data set current for input and output. Then use the FIND statement, which finds observation numbers and stores them in a matrix, to find the observation number of the data for HENRY and store it in the matrix **d**.

```
> edit class;
> find all where(name={'HENRY'}) into d;
> print d;
```

```
                               D
                              12
```

List the observation containing the data for HENRY.

```
> list point d;
```

```
    OBS NAME       SEX             AGE    HEIGHT    WEIGHT
    ------ -------- -------- --------- --------- ---------
       12 HENRY     M          14.0000   63.5000  102.5000
```

As you see, the observation number is 12. Now read the value for AGE into a matrix and update its value. Finally, replace the value in the CLASS data set and list the observation containing the data for HENRY again.

```
> age=15;
> replace;

                        1 observations replaced.

> list point 12;

    OBS NAME       SEX           AGE     HEIGHT     WEIGHT
    ------ -------- -------- --------- --------- ---------
    12 HENRY      M         15.0000   63.5000   102.5000
```

Deleting Observations

Use the DELETE statement to mark an observation to be deleted. The general form
of the DELETE statement is

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

where

range   specifies a range of observations.

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

The following are examples of valid uses of the DELETE statement.

Code	Action
`delete;`	deletes the current observation
`delete point 10;`	deletes observation 10
`delete all where (age>12);`	deletes all observations where AGE is greater than 12

If a file accumulates a number of observations marked as deleted, you can clean out
these observations and renumber the remaining observations by using the PURGE
statement.

Suppose that the student named John has moved and you want to update the CLASS
data set. You can remove the observation using the EDIT and DELETE statements.
First, find the observation number of the data for JOHN and store it in the matrix **d**
using the FIND statement. Then submit a DELETE statement to mark the record for
deletion. A deleted observation is still physically in the file and still has an observa-
tion number, but it is excluded from processing. The deleted observations appear as
gaps when you list the file by observation number:

```
> find all where(name={'JOHN'}) into d;
> print d;

                          D
                          5
```

```
> delete point d;

                    1 observation deleted.
> list all;
```

```
    OBS NAME      SEX            AGE      HEIGHT     WEIGHT
    ------ -------- -------- --------- --------- ---------
        1 JOYCE     F         11.0000    51.3000    50.5000
        2 THOMAS    M         11.0000    57.5000    85.0000
        3 JAMES     M         12.0000    57.3000    83.0000
        4 JANE      F         12.0000    59.8000    84.5000
        6 LOUISE    F         12.0000    56.3000    77.0000
        7 ROBERT    M         12.0000    64.8000   128.0000
        8 ALICE     F         13.0000    56.5000    84.0000
        9 BARBARA   F         13.0000    65.3000    98.0000
       10 JEFFREY   M         13.0000    62.5000    84.0000
       11 CAROL     F         14.0000    62.8000   102.5000
       12 HENRY     M         15.0000    63.5000   102.5000
       13 ALFRED    M         14.0000    69.0000   112.5000
       14 JUDY      F         14.0000    64.3000    90.0000
       15 JANET     F         15.0000    62.5000   112.5000
       16 MARY      F         15.0000    66.5000   112.0000
       17 RONALD    M         15.0000    67.0000   133.0000
       18 WILLIAM   M         15.0000    66.5000   112.0000
       19 PHILIP    M         16.0000    72.0000   150.0000
```

Notice that there is a gap in the data where the deleted observation was (observation 5). To renumber the observations and close the gaps, submit the PURGE statement. Note that the PURGE statement deletes any indexes associated with a data set.

```
> purge;
```

Creating a SAS Data Set from a Matrix

SAS/IML software provides the ability to create a new SAS data set from a matrix. Use the CREATE and APPEND statements to create a SAS data set from a matrix, where the columns of the matrix become the data set variables and the rows of the matrix become the observations. Thus, an $n \times m$ matrix creates a SAS data set with m variables and n observations. The CREATE statement opens the new SAS data set for both input and output, and the APPEND statement writes to (outputs to) the data set.

Using the CREATE Statement with the FROM Option

You can create a SAS data set from a matrix using the CREATE statement with the FROM option. This form of the CREATE statement is

CREATE *SAS-data-set* **FROM** *matrix*

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

where

SAS-data-set	names the new data set.
matrix	names the matrix containing the data.
column-name	names the variables in the data set.
row-name	adds a variable containing row titles to the data set.

Suppose you want to create a SAS data set named RATIO containing a variable with the height-to-weight ratios for each student. You first create a matrix containing the ratios from the matrices **HEIGHT** and **WEIGHT** that you have already defined. Next, use the CREATE and APPEND statements to open a new SAS data set called RATIO and append the observations, naming the data set variable HTWT instead of COL1.

```
htwt=height/weight;
create ratio from htwt[colname='htwt'];
append from htwt;
```

Now submit the SHOW DATASETS and SHOW CONTENTS statements.

```
> show datasets;

  LIBNAME MEMNAME    OPEN MODE    STATUS
  ------- -------    ---------    ------
  WORK    .CLASS     Update
  WORK    .RATIO     Update       Current Input    Current Output

> show contents;

  VAR NAME    TYPE    SIZE
  HTWT        NUM       8
  Number of Variables:    1
  Number of Observations: 18

> close ratio;
```

As you can see, the new SAS data set RATIO has been created. It has 18 observations and 1 variable (recall that you deleted 1 observation earlier).

Using the CREATE Statement with the VAR Clause

You can use a VAR clause with the CREATE statement to select the variables you want to include in the new data set. In the previous example, the new data set RATIO had one variable. If you want to create a similar data set but include the second variable NAME, you use the VAR clause. You could not do this using the FROM option because the variable HTWT is numeric and the variable NAME is character. The following statements create a new data set RATIO2 having the variables NAME and HTWT.

```
> create ratio2 var{name htwt};
> append;
> show contents;

          VAR NAME    TYPE    SIZE
          NAME        CHAR      8
          HTWT        NUM       8
          Number of Variables:    2
          Number of Observations: 18

> close ratio2;
```

Notice that now the variable NAME is in the data set.

Understanding the End-of-File Condition

If you try to read past the end of a data set or point to an observation greater than the number of observations in the data set, you create an end-of-file condition. If an end of file occurs while inside a DO DATA iteration group, IML transfers control to the next statement outside the current DO DATA group.

The following example uses a DO DATA loop while reading the CLASS data set. It reads the variable WEIGHT in one observation at a time and accumulates the weights of the students in the IML matrix SUM. When the data are read, the total class weight is stored in the matrix SUM.

```
setin class point 0;
sum=0;
do data;
   read next var{weight};
   sum=sum+weight;
end;
print sum;
```

Producing Summary Statistics

Summary statistics on the numeric variables of a SAS data set can be obtained with the SUMMARY statement. These statistics can be based on subgroups of the data by using the CLASS clause in the SUMMARY statement. The SAVE option in the OPT

clause enables you to save the computed statistics in matrices for later perusal. For example, consider the following statement.

```
> summary var {height weight} class {sex} stat{mean std} opt{save};
```

```
        SEX       Nobs  Variable       MEAN         STD
        ------------------------------------------------
        F            9  HEIGHT       60.58889      5.01833
                        WEIGHT       90.11111     19.38391

        M            9  HEIGHT       64.45556      4.90742
                        WEIGHT      110.00000     23.84717

        All         18  HEIGHT       62.52222      5.20978
                        WEIGHT      100.05556     23.43382
        ------------------------------------------------
```

This summary statement gives the mean and standard deviation of the variables HEIGHT and WEIGHT for the two subgroups (male and female) of the data set CLASS. Since the SAVE option is set, the statistics of the variables are stored in matrices under the name of the corresponding variables, with each column corresponding to a statistic requested and each row corresponding to a subgroup. Two other vectors, SEX and _NOBS_, are created. The vector SEX contains the two distinct values of the class variable SEX used in forming the two subgroups. The vector _NOBS_ has the number of observations in each subgroup.

Note that the combined means and standard deviations of the two subgroups are displayed but are not saved.

More than one class variable can be used, in which case a subgroup is defined by the combination of the values of the class variables.

Sorting a SAS Data Set

The observations in a SAS data set can be ordered (sorted) by specific key variables. To sort a SAS data set, close the data set if it is currently open, and issue a SORT statement for the variables by which you want the observations to be ordered. Specify an output data set name if you want to keep the original data set. For example, the statement

```
> sort class out=sorted by name;
```

creates a new SAS data set named SORTED. The new data set has the observations from the data set CLASS, ordered by the variable NAME.

The statement

```
> sort class by name;
```

sorts in place the data set CLASS by the variable NAME. However, at the completion of the SORT statement, the original data set is replaced by the sorted data set.

You can specify as many key variables as needed, and, optionally, each variable can be preceded by the keyword DESCENDING, which denotes that the variable that follows is to be sorted in descending order.

Indexing a SAS Data Set

Searching through a large data set for information about one or more specific observations may take a long time because the procedure must read each record. You can reduce this search time by first indexing the data set by a variable. The INDEX statement builds a special companion file containing the values and record numbers of the indexed variables. Once the index is built, IML may use the index for queries with WHERE clauses if it decides that indexed retrieval is more efficient. Any number of variables can be indexed, but only one index is in use at a given time. Note that purging a data set with the PURGE statement results in the loss of all associated indexes.

Once you have indexed a data set, IML can use this index whenever a search is conducted with respect to the indexed variables. The indexes are updated automatically whenever you change values in indexed variables. When an index is in use, observations cannot be randomly accessed by their physical location numbers. This means that the POINT range cannot be used when an index is in effect. However, if you purge the observations marked for deletion, or sort the data set in place, the indexes become invalid and IML automatically deletes them.

For example, if you want a list of all female students in the CLASS data set, you can first index CLASS by the variable SEX. Then use the LIST statement with a WHERE clause. Of course, the CLASS data set is small, and indexing does little if anything to speed queries with the WHERE clause. If the data set had thousands of students, though, indexing could save search time.

To index the data set by the variable SEX, submit the statement

```
> index sex;

  NOTE: Variable SEX indexed.
  NOTE: Retrieval by SEX.
```

Now list all students. Notice the ordering of the special file built by indexing by the variable SEX. Retrievals by SEX will be quick.

```
> list all;
```

OBS	NAME	SEX	AGE	HEIGHT	WEIGHT
1	JOYCE	F	11.0000	51.3000	50.5000
4	JANE	F	12.0000	59.8000	84.5000
6	LOUISE	F	12.0000	56.3000	77.0000
8	ALICE	F	13.0000	56.5000	84.0000
9	BARBARA	F	13.0000	65.3000	98.0000
11	CAROL	F	14.0000	62.8000	102.5000
14	JUDY	F	14.0000	64.3000	90.0000
15	JANET	F	15.0000	62.5000	112.5000
16	MARY	F	15.0000	66.5000	112.0000
2	THOMAS	M	11.0000	57.5000	85.0000
3	JAMES	M	12.0000	57.3000	83.0000
7	ROBERT	M	12.0000	64.8000	128.0000
10	JEFFREY	M	13.0000	62.5000	84.0000
12	HENRY	M	15.0000	63.5000	102.5000
13	ALFRED	M	14.0000	69.0000	112.5000
17	RONALD	M	15.0000	67.0000	133.0000
18	WILLIAM	M	15.0000	66.5000	112.0000
19	PHILIP	M	16.0000	72.0000	150.0000

Data Set Maintenance Functions

Two functions and two subroutines are provided to perform data set maintenance:

DATASETS function	obtains members in a data library. This function returns a character matrix containing the names of the SAS data sets in a library.
CONTENTS function	obtains variables in a member. This function returns a character matrix containing the variable names for the SAS data set specified by *libname* and *memname*. The variable list is returned in alphabetic order.
RENAME subroutine	renames a SAS data set member in a specified library.
DELETE subroutine	deletes a SAS data set member in a specified library.

See Chapter 20, "Language Reference," for details and examples of these functions and routines.

Summary of Commands

You have seen that IML has an extensive set of commands that operate on SAS data sets. The following table summarizes the data management commands you can use to perform management tasks for which you might normally use the SAS DATA step.

Table 6.1. Data Management Commands

Command	Action
APPEND	adds observations to the end of a SAS data set
CLOSE	closes a SAS data set
CREATE	creates and opens a new SAS data set for input and output
DELETE	marks observations for deletion in a SAS data set
EDIT	opens an existing SAS data set for input and output
FIND	finds observations
INDEX	indexes variables in a SAS data set
LIST	lists observations
PURGE	purges all deleted observations from a SAS data set
READ	reads observations into IML variables
REPLACE	writes observations back into a SAS data set
RESET DEFLIB	names default libname
SAVE	saves changes and reopens a SAS data set
SETIN	selects an open SAS data set for input
SETOUT	selects an open SAS data set for output
SHOW CONTENTS	shows contents of the current input SAS data set
SHOW DATASETS	shows SAS data sets currently open
SORT	sorts a SAS data set
SUMMARY	produces summary statistics for numeric variables
USE	opens an existing SAS data set for input

Similarities and Differences with the SAS DATA Step

If you want to remain in the IML environment and mimic DATA step processing, you need to learn the basic differences between IML and the DATA step:

- With SAS/IML software, you start with a CREATE statement instead of a DATA statement. You must explicitly set up all your variables with the right attributes before you create a data set. This means that you must define character variables having the desired string length beforehand. Numeric variables are the default, so any variable not defined as character is assumed to be numeric. In the DATA step, the variable attributes are determined from context across the whole step.

- With SAS/IML software, you must use an APPEND statement to output an observation; in the DATA step, you either use an OUTPUT statement or let the DATA step output it automatically.

- With SAS/IML software, you iterate with a DO DATA loop. In the DATA step, the iterations are implied.

- With SAS/IML software, you have to close the data set with a CLOSE statement unless you plan to leave the IML environment with a QUIT statement.

  The DATA step closes the data set automatically at the end of the step.

- The DATA step usually executes faster than IML.

In short, the DATA step treats the problem with greater simplicity, allowing shorter programs. However, IML has more flexibility because it is both interactive and has a powerful matrix-handling capability.

Summary

In this chapter, you have learned many ways to interact with SAS data sets from within the IML environment. You learned how to open and close a SAS data set, how to make it current for input and output, how to list observations by specifying a range of observations to process, a set of variables to use, and a condition for subsetting observations. You also learned summary statistics. You also know how to read observations and variables from a SAS data set into matrices as well as create a SAS data set from a matrix of values.

Chapter 7
File Access

Chapter Contents

Chapter 7
File Access

Overview

In this chapter you learn about external files and how to refer to an external file, whether it is a text file or a binary file. You learn how to read data from a file using the INFILE and INPUT statements and how to write data to an external file using the FILE and PUT statements.

With external files, you must know the format in which the data are stored or to be written. This is in contrast to SAS data sets, which are specialized files with a structure that is already known to the SAS System.

The Interactive Matrix Language statements used to access files are very similar to the corresponding statements in the SAS DATA step. The following table summarizes the IML statements and their functions.

Statement	Function
CLOSEFILE	closes an external file
FILE	opens an external file for output
INFILE	opens an external file for input
INPUT	reads from the current input file
PUT	writes to the current output file
SHOW: *FILES*	shows all open files, their attributes, and their status (current input and output files)

Referring to an External File

Suppose that you have data for students in a class. You have recorded the values for the variables NAME, SEX, AGE, HEIGHT, and WEIGHT for each student and have stored the data in an external text file named USER.TEXT.CLASS. If you want to read this data into IML variables, you need to indicate where the data are stored. In other words, you need to name the input file. If you want to write data from matrices to a file, you also need to name an output file.

There are two ways to refer to an input or output file: a *filepath* and a *filename*. A *filepath* is the name of the file as it is known to the operating system. A *filename* is an indirect SAS reference to the file made using the FILENAME statement. You can identify a file in either way using the FILE and INFILE statements.

For example, you can refer to the input file where the class data are stored using a literal filepath, that is, a quoted string. The statement

```
    infile 'user.text.class';
```

opens the file USER.TEXT.CLASS for input. Similarly, if you want to output data to the file USER.TEXT.NEWCLASS, you need to reference the output file with the statement

```
    file 'user.text.newclass';
```

You can also refer to external files using a *filename*. When using a filename as the operand, simply give the name. The name must be one already associated with a filepath by a previously issued FILENAME statement.

For example, suppose you want to reference the file with the class data using a FILENAME statement. First, you must associate the filepath with an alias (called a *fileref*), say INCLASS. Then you can refer to USER.TEXT.CLASS with the fileref INCLASS.

The following statements accomplish the same thing as the previous INFILE statement with the quoted filepath:

```
    filename inclass 'user.text.class';
    infile inclass;
```

You can use the same technique for output files. The following statements have the same effect as the previous file statement:

```
    filename outclass 'user.text.newclass';
    file outclass;
```

Three filenames have special meaning to IML: CARDS, LOG, and PRINT. These refer to the standard input and output streams for all SAS sessions, as described below:

CARDS   is a special filename for instream input data.

LOG   is a special filename for log output.

PRINT   is a special filename for standard print output.

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

Types of External Files

Most files that you work with are *text files*, which means that they can be edited and displayed without any special program. Text files under most host environments have special characters, called carriage-control characters or end-of-line characters, to separate one record from the next.

If your file does not adhere to these conventions, it is called a *binary file*. Typically, binary files do not have the usual record separators, and they may use any binary

codes, including unprintable control characters. If you want to read a binary file, you must specify RECFM=N in the INFILE statement and use the byte operand (<) in the INPUT statement to specify the length of each item you want read. Treating a file as binary enables you to have direct access to a file position by byte-address using the byte operand (>) in the INPUT or PUT statement.

You write data to an external file using the FILE and PUT statements. The output file can be text or binary. If your output file is binary, you must specify RECFM=N in the FILE statement. One difference between binary and text files in output is that the PUT statement does not put the record-separator characters on the end of each record written for binary files.

Reading from an External File

After you have chosen a method to refer to the external file you want to read, you need an INFILE statement to open it for input and an INPUT statement to tell IML how to read the data.

The next several sections cover how to use an INFILE statement and how to specify an INPUT statement so that you can input data from an external file.

Using the INFILE Statement

An INFILE statement identifies an external file containing data that you want to read. It opens the file for input or, if the file is already open, makes it the current input file. This means that subsequent INPUT statements are read from this file until another file is made the current input file.

The following options can be used with the INFILE statement:

FLOWOVER
enables the INPUT statement to go to the next record to obtain values for the variables.

LENGTH=*variable*
names a variable containing the length of the current record, where the value is set to the number of bytes used after each INPUT statement.

MISSOVER
prevents reading from the next input record when an INPUT statement reaches the end of the current record without finding values for all variables. It assigns missing values to all values that are expected but not found.

RECFM=N
specifies that the file is to be read in as a pure binary file rather than as a file with record-separator characters. You must use the byte operands (< and >) to get new records rather than separate INPUT statements or the new line operator (/).

STOPOVER
stops reading when an INPUT statement reaches the end of the current record without finding values for all variables in the statement. It treats going past the end of a record

as an error condition, triggering an end-of-file condition. The STOPOVER option is the default.

The FLOWOVER, MISSOVER, and STOPOVER options control how the INPUT statement works when you try to read past the end of a record. You can specify only one of these options. Read these options carefully so that you understand them completely.

Below is an example using the INFILE statement with a FILENAME statement to read the class data file. The MISSOVER option is used to prevent reading from the next record if values for all variables in the INPUT statement are not found.

```
filename inclass 'user.text.class';
infile inclass missover;
```

You can specify the filepath with a quoted literal also. The preceding statements could be written as

```
infile 'user.text.class' missover;
```

Using the INPUT Statement

Once you have referenced the data file containing your data with an INFILE statement, you need to tell IML exactly how the data are arranged:

- the number of variables and their names
- each variable's type, either numeric or character
- the format of each variable's values
- the columns that correspond to each variable

In other words, you must tell IML how to read the data.

The INPUT statement describes the arrangement of values in an input record. The INPUT statement reads records from a file specified in the previously executed INFILE statement, reading the values into IML variables.

There are two ways to describe a record's values in an IML INPUT statement:

- list (or scanning) input
- formatted input

Here are several examples of valid INPUT statements for the class data file, depending, of course, on how the data are stored.

If the data are stored with a blank or a comma between fields, then list input can be used. For example, the INPUT statement for the class data file might look as follows:

```
infile inclass;
input name $ sex $ age height weight;
```

These statements tell IML the following:

- There are five variables: NAME, SEX, AGE, HEIGHT and WEIGHT.
- Data fields are separated by commas or blanks.
- NAME and SEX are character variables, as indicated by the dollar sign ($).
- AGE, HEIGHT, and WEIGHT are numeric variables, the default.

The data must be stored in the same order in which the variables are listed in the INPUT statement. Otherwise, you can use formatted input, which is column specific. Formatted input is the most flexible and can handle any data file. Your INPUT statement for the class data file might look as follows:

```
infile inclass;
input @1 name $char8. @10 sex $char1. @15 age 2.0
   @20 height 4.1 @25 weight 5.1;
```

These statements tell IML the following:

- NAME is a character variable; its value begins in column 1 (indicated by @1) and occupies eight columns ($char8.).
- SEX is a character variable; its value is in column 10 ($char1.).
- AGE is a numeric variable; its value is found in columns 15 and 16 and has no decimal places (2.0).
- HEIGHT is a numeric variable found in columns 20 through 23 with one decimal place implied (4.1).
- WEIGHT is a numeric variable found in columns 25 through 29 with one decimal place implied (5.1).

The next sections discuss these two modes of input.

List Input

If your data are recorded with a comma or one or more blanks between data fields, you can use list input to read your data. If you have missing values, that is, unknown values, they must be represented by a period (.) rather than a blank field.

When IML looks for a value, it skips past blanks and tab characters. Then it scans for a delimiter to the value. The delimiter is a blank, a comma, or the end of the record. When the ampersand (&) format modifier is used, IML looks for two blanks, a comma, or the end of the record.

The general form of the INPUT statement for list input is

INPUT *variable* $<$ $\$$ $>$ $<$ $\&$ $>$ $<$ $\ldots$*variable* $<$ $\$$ $>$ $<$ $\&$ $>$ $>$;

where

variable　names the variable to be read by the INPUT statement.

$\$$　　　　indicates that the preceding variable is character.

$\&$　　　　indicates that a character value may have a single embedded blank. Because a blank normally indicates the end of a data value, use the ampersand format modifier to indicate the end of the value with at least two blanks or a comma.

With list input, IML scans the input lines for values. Consider using list input when

- blanks or commas separate input values
- periods rather than blanks represent missing values

List input is the default in several situations. Descriptions of these situations and the behavior of IML follow:

- If no input format is specified for a variable, IML scans for a number.
- If a single dollar sign or ampersand format modifier is specified, IML scans for a character value. The ampersand format modifier allows single embedded blanks to occur.
- If a format is given with width unspecified or 0, IML scans for the first blank or comma.

If the end of a record is encountered before IML finds a value, then the behavior is as described by the record overflow options in the INFILE statement discussed in the section "Using the INFILE Statement."

When you read with list input, the order of the variables listed in the INPUT statement must agree with the order of the values in the data file. For example, consider the following data:

```
Alice     f     10    61    97
Beth      f     11    64    105
Bill      m     12    63    110
```

You can use list input to read this data by specifying the following INPUT statement:

```
input name $ sex $ age height weight;
```

Note: This statement implies that the variables are stored in the order given. That is, each line of data contains a student's NAME, SEX, AGE, HEIGHT, and WEIGHT in that order and separated by at least one blank or by a comma.

Formatted Input

The alternative to list input is formatted input. An INPUT statement reading formatted input must have a SAS informat after each variable. An *informat* gives the data type and field width of an input value. Formatted input may be used with pointer controls and format modifiers. Note, however, that neither pointer controls nor format modifiers are necessary for formatted input.

Pointer control features

Pointer controls reset the pointer's column and line positions and tell the INPUT statement where to go to read the data value. You use pointer controls to specify the columns and lines from which you want to read:

- *Column pointer controls* move the pointer to the column you specify.
- *Line pointer controls* move the pointer to the next line.
- *Line hold controls* keep the pointer on the current input line.
- *Binary file indicator controls* indicate that the input line is from a binary file.

Column pointer controls

Column pointer controls indicate in which column an input value starts. Column pointer controls begin with either an at sign (@) or a plus sign (+).

@*n*   moves the pointer to column *n*.

@*point-variable*   moves the pointer to the column given by the current value of *point-variable*.

@(*expression*)   moves the pointer to the column given by the value of the *expression*. The *expression* must evaluate to a positive integer.

+*n*   moves the pointer *n* columns.

+*point-variable*   moves the pointer the number of columns given by the value of *point-variable*.

+(*expression*)   moves the pointer the number of columns given by the value of *expression*. The value of *expression* can be positive or negative.

Here are some examples using column pointer controls:

Example	Meaning
@12	go to column 12
@N	go to the column given by the value of N
@(N−1)	go to the column given by the value of N−1
+5	skip 5 spaces
+N	skip N spaces
+(N+1)	skip N+1 spaces

In the earlier example using formatted input, you used several pointer controls:

```
infile inclass;
input @1 name $char8. @10 sex $char1. @15 age 2.0
      @20 height 4.1 @25 weight 5.1;
```

The @1 moves the pointer to column 1, the @10 moves it to column 10, and so on. You move the pointer to the column where the data field begins and then supply an informat specifying how many columns the variable occupies. The INPUT statement could also be written as

```
input @1 name $char8. +1 sex $char1. +4 age 2. +3 height 4.1
      +1 weight 5.1;
```

In this form, you move the pointer to column 1 (@1) and read eight columns. The pointer is now at column 9. Now, move the pointer +1 columns to column 10 to read SEX. The $char1. informat says to read a character variable occupying one column. After you read the value for SEX, the pointer is at column 11, so move it to column 15 with +4 and read AGE in columns 15 and 16 (the 2. informat). The pointer is now at column 17, so move +3 columns and read HEIGHT. The same idea applies for reading WEIGHT.

Line pointer control

The line pointer control (/) directs IML to skip to the next line of input. You need a line pointer control when a record of data takes more than one line. You use the new line pointer control (/) to skip to the next line and continue reading data. In the example reading the class data, you do not need to skip a line because each line of data contains all the variables for a student.

Line hold control

The trailing at sign (@), when at the end of an INPUT statement, directs IML to hold the pointer on the current record so that you can read more data with subsequent INPUT statements. You can use it to read several records from a single line of data. Sometimes, when a record is very short, say ten columns or so, you can save space in your external file by coding several records on the same line.

Binary file indicator controls

When the external file you want to read is a binary file (RECFM=N is specified in the INFILE statement), you must tell IML how to read the values using the following binary file indicator controls:

>n	start reading the next record at the byte position n in the file.
>*point-variable*	start reading the next record at the byte position in the file given by *point-variable*.
>(*expression*)	start reading the next record at the byte position in the file given by *expression*.
<n	read the number of bytes indicated by the value of n.
<*point-variable*	read the number of bytes indicated by the value of *point-variable*.
<(*expression*)	read the number of bytes indicated by the value of *expression*.

Pattern Searching

You can have the input mechanism search for patterns of text by using the at sign
(@) positional with a character operand. IML starts searching at the current position,
advances until it finds the pattern, and leaves the pointer at the position immediately
after the found pattern in the input record. For example, the statement

```
input @ 'NAME=' name $;
```

searches for the pattern **NAME=** and then uses list input to read the value after the
found pattern.

If the pattern is not found, then the pointer is left past the end of the record,
and the rest of the INPUT statement follows the conventions based on the options
MISSOVER, STOPOVER, and FLOWOVER described in the section "Using the
INFILE Statement" earlier in this chapter. If you use pattern searching, you usually
specify the MISSOVER option so that you can control for the occurrences of the
pattern not being found.

Notice that the MISSOVER feature enables you to search for a variety of items on
the same record, even if some of them are not found. For example, the statements

```
infile in1 missover;
input @1 @ "NAME=" name $
      @1 @ "ADDR=" addr &
      @1 @ "PHONE=" phone $;
```

are able to read in the ADDR variable even if **NAME=** is not found (in which case,
NAME is unvalued).

The pattern operand can use any characters except for the following:

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

Record Directives

Each INPUT statement goes to a new record except for the following special cases:

- An at sign (@) at the end of an INPUT statement specifies that the record is to
  be held for future INPUT statements.
- Binary files (RECFM=N) always hold their records until the > directive.

As discussed in the syntax of the INPUT statement, the line pointer operator (/)
instructs the input mechanism to go immediately to the next record. For binary
(RECFM=N) files, the > directive is used instead of the /.

Blanks

For character values, the informat determines the way blanks are interpreted. For example, the $CHAR$w. format reads blanks as part of the whole value, while the BZw. format turns blanks into 0s. Refer to *SAS Language Reference: Dictionary* for more information on informats.

Missing Values

Missing values in formatted input are represented by blanks or a single period for a numeric value and by blanks for a character value.

Matrix Use

Data values are either character or numeric. Input variables always result in scalar (one row by one column) values with type (character or numeric) and length determined by the input format.

End-of-File Condition

End of file is the condition of trying to read a record when there are no more records to read from the file. The consequences of an end-of-file condition are described as follows.

- All the variables in the INPUT statement that encountered end of file are freed of their values. You can use the NROW or NCOL function to test if this has happened.
- If end of file occurs while inside a DO DATA loop, execution is passed to the statement after the END statement in the loop.

For text files, the end of file is encountered first as the end of the last record. The next time input is attempted, the end-of-file condition is raised.

For binary files, the end of file can result in the input mechanism returning a record that is shorter than the requested length. In this case IML still attempts to process the record, using the rules described in the section "Using the INFILE Statement," earlier in this chapter.

The DO DATA mechanism provides a convenient mechanism for handling end of file.

For example, to read the class data from the external file USER.TEXT.CLASS into a SAS data set, you need to perform the following steps:

1. Establish a *fileref* referencing the data file.
2. Use an INFILE statement to open the file for input.
3. Initialize any character variables by setting the length.
4. Create a new SAS data set with a CREATE statement. You want to list the variables you plan to input in a VAR clause.
5. Use a DO DATA loop to read the data one line at a time.

6. Write an INPUT statement telling IML how to read the data.

7. Use an APPEND statement to add the new data line to the end of the new SAS data set.

8. End the DO DATA loop.

9. Close the new data set.

10. Close the external file with a CLOSEFILE statement.

Your code would look as follows.

```
filename inclass 'user.text.class';
infile inclass missover;
name="12345678";
sex="1";
create class  var{name sex age height weight};
do data;
    input name $ sex $ age height weight;
    append;
end;
close class;
closefile inclass;
```

Note that the APPEND statement is not executed if the INPUT statement reads past the end of file since IML escapes the loop immediately when the condition is encountered.

Differences with the SAS DATA Step

If you are familiar with the SAS DATA step, you will notice that the following features are supported differently or are not supported in IML:

- The pound sign (#) directive supporting multiple current records is not supported.

- Grouping parentheses are not supported.

- The colon (:) format modifier is not supported.

- The byte operands ($<$ and $>$) are new features supporting binary files.

- The ampersand (&) format modifier causes IML to stop reading data if a comma is encountered. Use of the ampersand format modifier is valid with list input only.

- The RECFM=F option is not supported.

Writing to an External File

If you have data in matrices and you want to write this data to an external file, you need to reference, or point to, the file (as discussed in the section "Referring to an External File"). The FILE statement opens the file for output so that you can write data to it. You need to specify a PUT statement to direct how the data is output. These two statements are discussed in the following sections.

Using the FILE Statement

The FILE statement is used to refer to an external file. If you have values stored in matrices, you can write these values to a file. Just as with the INFILE statement, you need a fileref to point to the file you want to write to. You use a FILE statement to indicate that you want to write to rather than read from a file.

For example, if you want to output to the file USER.TEXT.NEWCLASS, you can specify the file with a quoted literal filepath.

```
> file 'user.text.newclass';
```

Otherwise, you can first establish a fileref and then refer to the file by its fileref:

```
> filename outclass 'user.text.class';
> file outclass;
```

There are two options you can use in the FILE statement:

RECFM=N   specifies that the file is to be written as a pure binary file without record-separator characters.

LRECL=*operand*   specifies the size of the buffer to hold the records.

The FILE statement opens a file for output or, if the file is already open, makes it the current output file so that subsequent PUT statements write to the file. The FILE statement is similar in syntax and operation to the INFILE statement.

Using the PUT Statement

The PUT statement writes lines to the SAS log, to the SAS output file, or to any external file specified in a FILE statement. The file associated with the most recently executed FILE statement is the *current output file*.

You can use the following arguments with the PUT statement:

variable   names the IML variable with a value that is put to the current pointer position in the record. The variable must be scalar valued. The put variable can be followed immediately by an output format.

literal   gives a literal to be put to the current pointer position in the record. The literal can be followed immediately by an output format.

(expression)   must produce a scalar-valued result. The expression can be immediately followed by an output format.

format   names the output formats for the values.

pointer-control   moves the output pointer to a line or column.

Pointer Control Features

Most PUT statements need the added flexibility obtained with pointer controls. IML keeps track of its position on each output line with a pointer. With specifications in the PUT statement, you can control pointer movement from column to column and line to line. The pointer controls available are discussed in the section "Using the INPUT statement".

Differences with the SAS DATA Step

If you are familiar with the SAS DATA step, you will notice that the following features are supported differently or are not supported:

- The pound sign (#) directive supporting multiple current records is not supported.
- Grouping parentheses are not supported.
- The byte operands (< and >) are a new feature supporting binary files.

Examples

Writing a Matrix to an External File

If you have data stored in an $n \times m$ matrix and you want to output the values to an external file, you need to write out the matrix element by element.

For example, suppose that you have a matrix **X** containing data that you want written to the file USER.MATRIX. Suppose also that **X** contains 1s and 0s so that the format for output can be one column. You need to do the following:

1. Establish a fileref, for example, OUT.
2. Use a FILE statement to open the file for output.
3. Specify DO loop for the rows of the matrix.
4. Specify DO loop for the columns of the matrix.
5. Use a PUT statement to specify how to write the element value.
6. End the inner DO loop.
7. Skip a line.
8. End the outer DO loop.
9. Close the file.

Your code should look as follows:

```
 filename out 'user.matrix';
 file out;
    do i=1 to nrow(x);
       do j=1 to ncol(x);
          put (x[i,j]) 1.0 +2 @;
       end;
       put;
    end;
 closefile out;
```

The output file contains a record for each row of the matrix. For example, if your matrix is 4×4, then the file might look as follows:

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

Quick Printing to the PRINT File

You can use the FILE PRINT statement to route output to the standard print file. The following statements generate data that are output to the PRINT file:

```
> file print;
> do a=0 to 6.28 by .2;
>    x=sin(a);
>    p=(x+1)#30;
>    put @1 a 6.4 +p x 8.4;
> end;
```

The result is shown below:

```
0.0000                                        0.0000
0.2000                                           0.1987
0.4000                                              0.3894
0.6000                                                 0.5646
0.8000                                                    0.7174
1.0000                                                       0.8415
1.2000                                                          0.9320
1.4000                                                             0.9854
1.6000                                                             0.9996
1.8000                                                             0.9738
2.0000                                                          0.9093
2.2000                                                       0.8085
2.4000                                                    0.6755
2.6000                                                 0.5155
2.8000                                              0.3350
3.0000                                           0.1411
3.2000                                      -0.0584
3.4000                                   -0.2555
3.6000                                -0.4425
3.8000                            -0.6119
4.0000                         -0.7568
4.2000                      -0.8716
4.4000                   -0.9516
4.6000                 -0.9937
4.8000                 -0.9962
5.0000                   -0.9589
5.2000                      -0.8835
5.4000                         -0.7728
5.6000                            -0.6313
5.8000                                -0.4646
6.0000                                   -0.2794
6.2000                                      -0.0831
```

Listing Your External Files

To list all open files and their current input or current output status, use the SHOW FILES statement.

Closing an External File

The CLOSEFILE statement closes files opened by an INFILE or a FILE statement. You specify the CLOSEFILE statement just as you do the INFILE or FILE statement. For example, the following statements open the external file USER.TEXT.CLASS for input and then close it:

```
filename in 'user.text.class';
infile in;
closefile in;
```

Summary

In this chapter, you learned how to refer to, or point to, an external file with a FILENAME statement. You can use the FILENAME statement whether you want to read from or write to an external file. The file can also be referenced by a quoted literal filepath. You also learned about the difference between a text file and a binary file.

You learned how to read data from an external file with the INFILE and INPUT statements, using either list or formatted input. You learned how to write your matrices to an external file using the FILE and PUT statements. Finally, you learned how to close your files.

Chapter 8
General Statistics Examples

Chapter Contents

Chapter 8
General Statistics Examples

Overview

SAS/IML software has many linear operators that perform high-level operations commonly needed in applying linear algebra techniques to data analysis. The similarity of the Interactive Matrix Language notation and matrix algebra notation makes translation from algorithm to program a straightforward task. The examples in this chapter show a variety of matrix operators at work.

You can use these examples to gain insight into the more complex problems you may need to solve. Some of the examples perform the same analyses as performed by procedures in SAS/STAT software and are not meant to replace them. The examples are included as learning tools.

General Statistics Examples

Example 8.1. Correlation

This example defines modules to compute correlation coefficients between numeric variables and standardized values for a set of data.

```
   /* Module to compute correlations  */
start corr;
   n=nrow(x);                         /* number of observations */
   sum=x[+,] ;                          /* compute column sums */
   xpx=t(x)*x-t(sum)*sum/n;        /* compute sscp matrix    */
   s=diag(1/sqrt(vecdiag(xpx)));         /* scaling matrix */
   corr=s*xpx*s;                        /* correlation matrix */
   print "Correlation Matrix",,corr[rowname=nm colname=nm] ;
finish corr;

   /* Module to standardize data */
start std;
   mean=x[+,] /n;                        /* means for columns */
   x=x-repeat(mean,n,1);          /* center x to mean zero */
   ss=x[##,] ;                   /* sum of squares for columns */
   std=sqrt(ss/(n-1));        /* standard deviation estimate*/
   x=x*diag(1/std);                      /* scaling to std dev 1 */
   print ,"Standardized Data",,X[colname=nm] ;
finish std;
```

```
      /* Sample run */
 x = { 1 2 3,
       3 2 1,
       4 2 1,
       0 4 1,
      24 1 0,
       1 3 8};
 nm={age weight height};
 run corr;
 run std;
```

The results are as follows.

```
                    Correlation Matrix


                       CORR
                  AGE      WEIGHT     HEIGHT

         AGE            1 -0.717102 -0.436558
         WEIGHT -0.717102         1 0.3508232
         HEIGHT -0.436558 0.3508232         1


                   Standardized Data


                        X
                  AGE     WEIGHT     HEIGHT

          -0.490116 -0.322749 0.2264554
          -0.272287 -0.322749 -0.452911
          -0.163372 -0.322749 -0.452911
           -0.59903 1.6137431 -0.452911
          2.0149206 -1.290994 -0.792594
          -0.490116 0.6454972  1.924871
```

Example 8.2. Newton's Method for Solving Nonlinear Systems of Equations

This example solves a nonlinear system of equations by Newton's method. Let the nonlinear system be represented by

$$F(\mathbf{x}) = 0$$

where $\mathbf{x}$ is a vector and F is a vector-valued, possibly nonlinear, function.

In order to find $\mathbf{x}$ such that F goes to 0, an initial estimate $\mathbf{x}_0$ is chosen, and Newton's iterative method for converging to the solution is used:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n)F(\mathbf{x}_n)$$

where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix of partial derivatives of F with respect to $\mathbf{x}$.

For optimization problems, the same method is used, where $F(\mathbf{x})$ is the gradient of the objective function and $\mathbf{J}(\mathbf{x})$ becomes the Hessian (Newton-Raphson).

In this example, the system to be solved is

$$x_1 + x_2 - x_1 x_2 + 2 = 0$$
$$x_1 \exp(-x_2) - 1 = 0$$

The code is organized into three modules: NEWTON, FUN, and DERIV.

```
/*      Newton's Method to Solve a Nonlinear Function      */
/* The user must supply initial values,                   */
/* and the FUN and DERIV functions.                       */
/* on entry: FUN evaluates the function f in terms of x    */
/* initial values are given to x                          */
/* DERIV evaluates jacobian j                             */
/* tuning variables: CONVERGE, MAXITER.                   */
/* on exit: solution in x, function value in f close to 0 */
/* ITER has number of iterations.                         */

start newton;
   run fun;              /* evaluate function at starting values */
   do iter=1 to maxiter  /* iterate until maxiter iterations */
   while(max(abs(f))>converge);              /* or convergence */
      run deriv;                  /* evaluate derivatives in j */
      delta=-solve(j,f);      /* solve for correction vector */
      x=x+delta;                      /* the new approximation */
      run fun;                        /* evaluate the function */
   end;
finish newton;

maxiter=15;                       /* default maximum iterations */
converge=.000001;            /* default convergence criterion */


   /* User-supplied function evaluation */
start fun;
   x1=x[1] ;
   x2=x[2] ;                              /* extract the values */
   f= (x1+x2-x1*x2+2)//
   (x1*exp(-x2)-1);                       /* evaluate the function */
finish fun;

   /* User-supplied derivatives of the function */
start deriv;
   /* evaluate jacobian */
   j=((1-x2)||(1-x1) )//(exp(-x2)||(-x1*exp(-x2)));
finish deriv;

do;
   print "Solving the system: X1+X2-X1*X2+2=0, X1*EXP(-X2)-1=0" ,;
   x={.1, -2};      /* starting values */
   run newton;
```

```
      print x f;
  end;
```

The results are as follows.

```
                        The SAS System

     Solving the system:  X1+X2-X1*X2+2=0,  X1*EXP(-X2)-1=0


                              X        F

                    0.0977731 5.3523E-9
                    -2.325106 6.1501E-8
```

Example 8.3. Regression

This example shows a regression module that calculates statistics not calculated by
the two previous examples:

```
      /*          Regression Routine            */
      /* Given X, and Y, this fits Y = X B + E    */
      /* by least squares.                        */

  start reg;
     n=nrow(x);                         /* number of observations */
     k=ncol(x);                           /* number of variables */
     xpx=x`*x;                                  /* cross-products */
     xpy=x`*y;
     xpxi=inv(xpx);                      /* inverse crossproducts */
     b=xpxi*xpy;                          /* parameter estimates */
     yhat=x*b;                             /* predicted values */
     resid=y-yhat;                               /* residuals */
     sse=resid`*resid;               /* sum of squared errors */
     dfe=n-k;                      /* degrees of freedom error */
     mse=sse/dfe;                        /* mean squared error */
     rmse=sqrt(mse);               /* root mean squared error */
     covb=xpxi#mse;               /* covariance of estimates */
     stdb=sqrt(vecdiag(covb));            /* standard errors */
     t=b/stdb;                        /* ttest for estimates=0 */
     probt=1-probf(t#t,1,dfe);     /* significance probability */
     print name b stdb t probt;
     s=diag(1/stdb);
     corrb=s*covb*s;               /* correlation of estimates */
     print ,"Covariance of Estimates", covb[r=name c=name] ,
            "Correlation of Estimates",corrb[r=name c=name] ;

     if nrow(tval)=0 then return;   /* is a t-value specified? */
     projx=x*xpxi*x`;                        /* hat matrix */
     vresid=(i(n)-projx)*mse;         /* covariance of residuals */
     vpred=projx#mse;        /* covariance of predicted values  */
     h=vecdiag(projx);                /* hat leverage values */
```

Example 8.3. Regression ⬩ 135

```
    lowerm=yhat-tval#sqrt(h*mse); /* low. conf. lim. for mean */
    upperm=yhat+tval#sqrt(h*mse);    /* upper limit for mean */
    lower=yhat-tval#sqrt(h*mse+mse); /* lower limit for indiv */
    upper=yhat+tval#sqrt(h*mse+mse); /* upper limit for indiv */
    print ,,"Predicted Values, Residuals, and Limits" ,,
    y yhat resid h lowerm upperm lower upper;
finish reg;


    /* Routine to test a linear combination of the estimates  */
    /* given L, this routine tests hypothesis that LB = 0.    */

start test;
    dfn=nrow(L);
    Lb=L*b;
    vLb=L*xpxi*L`;
    q=Lb`*inv(vLb)*Lb /dfn;
    f=q/mse;
    prob=1-probf(f,dfn,dfe);
    print ,f dfn dfe prob;
finish test;

  /* Run it on population of U.S. for decades beginning 1790 */

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};
name={"Intercept", "Decade", "Decade**2" };
tval=2.57;  /* for 5 df at 0.025 level to get 95% conf. int. */
reset fw=7;
run reg;
do;
    print ,"TEST Coef for Linear";
    L={0 1 0 };
    run test;
    print ,"TEST Coef for Linear,Quad";
    L={0 1 0,0 0 1};
    run test;
    print ,"TEST Linear+Quad = 0";
    L={0 1 1 };
    run test;
end;
```

The results are as follows.

```
          NAME            B    STDB       T   PROBT

          Intercept 5.06934 0.96559 5.24997 0.00333
          Decade    -1.1099  0.4923 -2.2546 0.07385
          Decade**2 0.53964  0.0534  10.106 0.00016


                         COVB
                Intercept  Decade Decade**2

          Intercept   0.93237 -0.4362   0.04277
          Decade      -0.4362 0.24236   -0.0257
          Decade**2   0.04277 -0.0257   0.00285


                         CORRB
                Intercept  Decade Decade**2

          Intercept         1 -0.9177    0.8295
          Decade      -0.9177       1   -0.9762
          Decade**2    0.8295 -0.9762         1


      Y    YHAT    RESID       H  LOWERM  UPPERM   LOWER    UPPER

   3.929 4.49904    -0.57 0.70833 3.00202 5.99606 2.17419 6.82389
   5.308 5.00802 0.29998 0.27976 4.06721 5.94883 2.99581 7.02023
   7.239 6.59627 0.64273 0.23214 5.73926 7.45328 4.62185 8.57069
   9.638 9.26379 0.37421 0.27976 8.32298 10.2046 7.25158  11.276
  12.866 13.0106 -0.1446 0.27976 12.0698 13.9514 10.9984 15.0228
  17.069 17.8367 -0.7677 0.23214 16.9797 18.6937 15.8622 19.8111
  23.191  23.742  -0.551 0.27976 22.8012 24.6828 21.7298 25.7542
  31.443 30.7266 0.71638 0.70833 29.2296 32.2236 28.4018 33.0515


                  F      DFN     DFE    PROB

               5.08317     1       5 0.07385


                  F      DFN     DFE    PROB

               666.511     2       5 8.54E-7


                  F      DFN     DFE    PROB

               1.67746     1       5 0.25184
```

Example 8.4. Alpha Factor Analysis

This example shows how an algorithm for computing alpha factor patterns (Kaiser and Caffrey 1965) is transcribed into IML code.

For later reference, you could store the following ALPHA subroutine in an IML catalog and load it when needed.

Example 8.4. Alpha Factor Analysis ◆ 137

```
/*              Alpha Factor Analysis                      */
/*  Ref: Kaiser et al., 1965 Psychometrika, pp. 12-13     */
/*  r correlation matrix (n.s.) already set up            */
/*  p number of variables                                 */
/*  q number of factors                                   */
/*  h communalities                                       */
/*  m eigenvalues                                         */
/*  e eigenvectors                                        */
/*  f factor pattern                                      */
/*  (IQ,H2,HI,G,MM) temporary use. freed up               */
/*                                                        */

start alpha;
   p=ncol(r);
   q=0;
   h=0;                                          /* initialize */
   h2=i(p)-diag(1/vecdiag(inv(r)));                    /* smcs */
   do while(max(abs(h-h2))>.001); /* iterate until converges */
      h=h2;
      hi=diag(sqrt(1/vecdiag(h)));
      g=hi*(r-i(p))*hi+i(p);
      call eigen(m,e,g);          /* get eigenvalues and vecs */
      if q=0 then
      do;
         q=sum(m>1);                     /* number of factors */
         iq=1:q;
      end;                                     /* index vector */
      mm=diag(sqrt(m[iq,]));            /* collapse eigvals */
      e=e[,iq] ;                        /* collapse eigvecs */
      h2=h*diag((e*mm) [,##]);          /* new communalities */
   end;
   hi=sqrt(h);
   h=vecdiag(h2);
   f=hi*e*mm;                               /* resulting pattern */
   free iq h2 hi g mm;                      /* free temporaries */
finish;

   /* Correlation Matrix from Harmon, Modern Factor Analysis, */
   /* 2nd edition, page 124, "Eight Physical Variables"       */

r={1.000 .846 .805 .859 .473 .398 .301 .382 ,
     .846 1.000 .881 .826 .376 .326 .277 .415 ,
     .805 .881 1.000 .801 .380 .319 .237 .345 ,
     .859 .826 .801 1.000 .436 .329 .327 .365 ,
     .473 .376 .380 .436 1.000 .762 .730 .629 ,
     .398 .326 .319 .329 .762 1.000 .583 .577 ,
     .301 .277 .237 .327 .730 .583 1.000 .539 ,
     .382 .415 .345 .365 .629 .577 .539 1.000};
nm = {Var1 Var2 Var3 Var4 Var5 Var6 Var7 Var8};
run alpha;
print ,"EIGENVALUES" , m;
print ,"COMMUNALITIES" , h[rowname=nm];
print ,"FACTOR PATTERN", f[rowname=nm];
```

The results are as follows.

```
                         M

                      5.937855
                      2.0621956
                      0.1390178
                      0.0821054
                       0.018097
                      -0.047487
                       -0.09148
                      -0.100304


                         H

                  VAR1 0.8381205
                  VAR2 0.8905717
                  VAR3   0.81893
                  VAR4 0.8067292
                  VAR5 0.8802149
                  VAR6 0.6391977
                  VAR7 0.5821583
                  VAR8 0.4998126


                         F

              VAR1  0.813386 -0.420147
              VAR2 0.8028363  -0.49601
              VAR3 0.7579087 -0.494474
              VAR4 0.7874461 -0.432039
              VAR5 0.8051439 0.4816205
              VAR6 0.6804127 0.4198051
              VAR7  0.620623 0.4438303
              VAR8 0.6449419 0.2895902
```

Example 8.5. Categorical Linear Models

This example fits a linear model to a function of the response probabilities

$$\mathbf{K} \log \pi = \mathbf{X}\beta + e$$

where $\mathbf{K}$ is a matrix that compares each response category to the last. Data are from Kastenbaum and Lamphiear (1959). First, the Grizzle-Starmer-Koch (1969) approach is used to obtain generalized least-squares estimates of β. These form the initial values for the Newton-Raphson solution for the maximum-likelihood estimates. The CATMOD procedure can also be used to analyze these binary data (refer to Cox 1970).

```
/* Categorical Linear Models                   */
/* by Least Squares and Maximum Likelihood     */
/*   CATLIN                                     */
/*   Input:                                     */
/*     n the s by p matrix of response counts   */
/*     x the s by r design matrix               */
```

Example 8.5. Categorical Linear Models ◆ 139

```
start catlin;

   /*---find dimensions---*/
   s=nrow(n);                            /* number of populations */
   r=ncol(n);                             /* number of responses */
   q=r-1;                         /* number of function values */
   d=ncol(x);                     /* number of design parameters */
   qd=q*d;                         /* total number of parameters */

   /*---get probability estimates---*/
   rown=n[,+];                                 /* row totals */
   pr=n/(rown*repeat(1,1,r));          /* probability estimates */
   p=shape(pr[,1:q] ,0,1);         /* cut and shaped to vector */
   print "INITIAL PROBABILITY ESTIMATES" ,pr;

      /*    estimate by the GSK method   */

      /* function of probabilities */
   f=log(p)-log(pr[,r])@repeat(1,q,1);

      /* inverse covariance of f */
   si=(diag(p)-p*p`)#(diag(rown)@repeat(1,q,q));
   z=x@i(q);                            /* expanded design matrix */
   h=z`*si*z;                             /* crossproducts matrix */
   g=z`*si*f;                                   /* cross with f */
   beta=solve(h,g);                      /* least squares solution */
   stderr=sqrt(vecdiag(inv(h)));           /* standard errors */
   run prob;
   print ,"GSK ESTIMATES" , beta stderr ,pi;

      /*    iterations for ML solution    */
   crit=1;
   do it=1 to 8 while(crit>.0005); /* iterate until converge */

      /* block diagonal weighting   */
      si=(diag(pi)-pi*pi`)#(diag(rown)@repeat(1,q,q));
      g=z`*(rown@repeat(1,q,1)#(p-pi));          /* gradient */
      h=z`*si*z;                                  /* hessian */
      delta=solve(h,g);                /* solve for correction */
      beta=beta+delta;                 /* apply the correction */
      run prob;                        /* compute prob estimates */
      crit=max(abs(delta));            /* convergence criterion */
   end;
   stderr=sqrt(vecdiag(inv(h)));           /* standard errors */
   print , "ML Estimates", beta stderr, pi;
   print , "Iterations" it "Criterion" crit;
finish catlin;

   /*    subroutine to compute new prob estimates @ parameters   */
start prob;
   la=exp(x*shape(beta,0,q));
   pi=la/((1+la[,+] )*repeat(1,1,q));
   pi=shape(pi,0,1);
```

```
finish prob;


    /*---prepare frequency data and design matrix---*/
n= { 58 11 05,
     75 19 07,
     49 14 10,
     58 17 08,
     33 18 15,
     45 22 10,
     15 13 15,
     39 22 18,
     04 12 17,
     05 15 08};      /* frequency counts*/

x= { 1 1 1 0 0 0,
     1 -1 1 0 0 0,
     1 1 0 1 0 0,
     1 -1 0 1 0 0,
     1 1 0 0 1 0,
     1 -1 0 0 1 0,
     1 1 0 0 0 1,
     1 -1 0 0 0 1,
     1 1 -1 -1 -1 -1,
     1 -1 -1 -1 -1 -1};    /* design matrix*/

run catlin;
```

The results are as follows.

Example 8.5. Categorical Linear Models ◆ 141

```
                    PR

        0.7837838 0.1486486 0.0675676
        0.7425743 0.1881188 0.0693069
        0.6712329 0.1917808 0.1369863
        0.6987952 0.2048193 0.0963855
              0.5 0.2727273 0.2272727
        0.5844156 0.2857143 0.1298701
        0.3488372 0.3023256 0.3488372
        0.4936709   0.278481 0.2278481
        0.1212121 0.3636364 0.5151515
        0.1785714 0.5357143 0.2857143


                  BETA      STDERR

        0.9454429 0.1290925
        0.4003259 0.1284867
        -0.277777 0.1164699
        -0.278472 0.1255916
        1.4146936   0.267351
         0.474136   0.294943
        0.8464701 0.2362639
        0.1526095 0.2633051
        0.1952395 0.2214436
        0.0723489 0.2366597
        -0.514488 0.2171995
        -0.400831 0.2285779
```

```
                PI

            0.7402867
            0.1674472
            0.7704057
            0.1745023
            0.6624811
            0.1917744
            0.7061615
            0.2047033
             0.516981
            0.2648871
            0.5697446
            0.2923278
            0.3988695
            0.2589096
            0.4667924
            0.3034204
            0.1320359
            0.3958019
            0.1651907
            0.4958784


            BETA     STDERR

        0.9533597 0.1286179
        0.4069338 0.1284592
        -0.279081 0.1156222
        -0.280699 0.1252816
        1.4423195 0.2669357
        0.4993123 0.2943437
        0.8411595 0.2363089
        0.1485875 0.2635159
        0.1883383 0.2202755
        0.0667313  0.236031
        -0.527163  0.216581
        -0.414965 0.2299618



                PI

            0.7431759
            0.1673155
            0.7723266
            0.1744421
            0.6627266
            0.1916645
            0.7062766
            0.2049216
            0.5170782
            0.2646857
            0.5697771
             0.292607
            0.3984205
            0.2576653
            0.4666825
            0.3027898
            0.1323243
            0.3963114
             0.165475
            0.4972044


                IT              CRIT

    Iterations        3 Criterion 0.0004092
```

Example 8.6. Regression of Subsets of Variables ◆ 143

Example 8.6. Regression of Subsets of Variables

The following example performs regression with variable selection similar to some
of the features in the REG procedure.

```
     /*       Initialization                           */
     /* C,CSAVE the crossproducts matrix               */
     /* N       number of observations                 */
     /* K       total number of variables to consider  */
     /* L       number of variables currently in model */
     /* IN      a 0-1 vector of whether variable is in */
     /* B       print collects results (L MSE RSQ BETAS ) */

start initial;
   n=nrow(x);
   k=ncol(x);
   k1=k+1;
   ik=1:k;
   bnames={nparm mse rsquare} ||varnames;

      /* Correct by mean, adjust out intercept parameter */
   y=y-y[+,]/n;                          /* correct y by mean */
   x=x-repeat(x[+,]/n,n,1);              /* correct x by mean */
   xpy=x`*y;                                  /* crossproducts */
   ypy=y`*y;
   xpx=x`*x;
   free x y;                          /* no longer need the data */

      /* Save a copy of crossproducts matrix */
   csave=(xpx || xpy) // (xpy`|| ypy);
finish initial;



    /* Forward method */
start forward;
  print / "FORWARD SELECTION METHOD";
  free bprint;
  c=csave;
  in=repeat(0,k,1);
  l=0;                                /* No variables are in */
  dfe=n-1;
  mse=ypy/dfe;
  sprob=0;
  do while(sprob<.15 & l<k);
     indx=loc(^in);       /* where are the variables not in? */
     cd=vecdiag(c)[indx,];                     /* xpx diagonals */
     cb=c[indx,k1];                            /* adjusted xpy */
     tsqr=cb#cb/(cd#mse);              /* squares of t tests */
     imax=tsqr[<:>,];        /* location of maximum in indx */
     sprob=(1-probt(sqrt(tsqr[imax,]),dfe))*2;
     if sprob<.15 then

     do;                                /* if t-test significant */
```

```
              ii=indx[,imax];                    /* pick most significant */
              run swp;                                /* routine to sweep */
              run bpr;                        /* routine to collect results */
           end;
        end;
        print  bprint[colname=bnames] ;
     finish forward;

        /* Backward method */
     start backward;
        print / "BACKWARD ELIMINATION ";
        free bprint;
        c=csave;
        in=repeat(0,k,1);
        ii=1:k;
        run swp;
        run bpr;                        /* start with all variables in */
        sprob=1;
        do while(sprob>.15 & l>0);
           indx=loc(in);                /* where are the variables in? */
           cd=vecdiag(c)[indx,];                        /* xpx diagonals */
           cb=c[indx,k1];                                      /* bvalues */
           tsqr=cb#cb/(cd#mse);                    /* squares of t tests */
           imin=tsqr[>:<,];             /* location of minimum in indx */
           sprob=(1-probt(sqrt(tsqr[imin,]),dfe))*2;
           if sprob>.15 then
           do;                          /* if t-test nonsignificant */
              ii=indx[,imin];                    /* pick least significant */
              run swp;                   /* routine to sweep in variable */
              run bpr;                        /* routine to collect results */
           end;
        end;
        print  bprint[colname=bnames] ;
     finish backward;

        /* Stepwise method */
     start stepwise;
        print /"STEPWISE METHOD";
        free bprint;
        c=csave;
        in=repeat(0,k,1);
        l=0;
        dfe=n-1;
        mse=ypy/dfe;
        sprob=0;

        do while(sprob<.15 & l<k);
           indx=loc(^in);          /* where are the variables not in? */
           nindx=loc(in);                /* where are the variables in? */
           cd=vecdiag(c)[indx,];                        /* xpx diagonals */
           cb=c[indx,k1];                                  /* adjusted xpy */
           tsqr=cb#cb/cd/mse;                    /* squares of t tests */
           imax=tsqr[<:>,];             /* location of maximum in indx */
           sprob=(1-probt(sqrt(tsqr[imax,]),dfe))*2;
```

Example 8.6. Regression of Subsets of Variables ◆ 145

```
   if sprob<.15 then
   do;                                  /* if t-test significant */
      ii=indx[,imax];                        /* find index into c */
      run swp;                                  /* routine to sweep */
      run backstep;              /* check if remove any terms */
      run bpr;                      /* routine to collect results */
      end;
   end;
   print  bprint[colname=bnames] ;
finish stepwise;

   /* Routine to backwards-eliminate for stepwise  */
start backstep;
   if nrow(nindx)=0 then return;
   bprob=1;
   do while(bprob>.15 & l<k);
      cd=vecdiag(c)[nindx,];                    /* xpx diagonals */
      cb=c[nindx,k1];                               /* bvalues */
      tsqr=cb#cb/(cd#mse);              /* squares of t tests */
      imin=tsqr[>:<,];          /* location of minimum in nindx */
      bprob=(1-probt(sqrt(tsqr[imin,]),dfe))*2;
      if bprob>.15 then
      do;
         ii=nindx[,imin];
         run swp;
         run bpr;
      end;
   end;
finish backstep;

   /* Search all possible models */
start all;

   /* Use method of schatzoff et al. for search technique */
   betak=repeat(0,k,k); /* rec. ests. for best l-param model */
   msek=repeat(1e50,k,1);     /* record best mse per # parms */
   rsqk=repeat(0,k,1);                  /* record best rsquare */
   ink=repeat(0,k,k);        /* record best set per # parms */
   limit=2##k-1;              /* number of models to examine */
   c=csave;
   in=repeat(0,k,1); /* start out with no variables in model */

   do kk=1 to limit;
      run ztrail;                   /* find which one to sweep */
      run swp;                               /* sweep it in */
      bb=bb//(l||mse||rsq||(c[ik,k1]#in)`);
      if mse<msek[l,] then
      do;                         /* was this best for l parms? */
         msek[l,]=mse;                             /* record mse */
         rsqk[l,]=rsq;                        /* record rsquare */
         ink[,l]=in;        /* record which parms in model */
         betak[l,]=(c[ik,k1]#in)`;     /* record estimates */
      end;
   end;
```

```
    print / "ALL POSSIBLE MODELS" " IN SEARCH ORDER";
    print bb[colname=bnames]; free bb;
    bprint=ik`||msek||rsqk||betak;
    print ,"THE BEST MODEL FOR EACH NUMBER OF PARAMETERS";
    print bprint[colname=bnames];
        /* Mallows CP plot */
    cp=msek#(n-ik`-1)/min(msek)-(n-2#ik`);
    cp=ik`||cp;
    cpname={"nparm" "cp"};
        /* output cp out=cp colname=cpname; */
finish all;

    /*  Subroutine to find number of          */
    /*  trailing zeros in binary number       */
    /*  on entry: kk is the number to examine  */
    /*  on exit:  ii has the result           */
start ztrail;
    ii=1;
    zz=kk;
    do while(mod(zz,2)=0);
        ii=ii+1;
        zz=zz/2;
    end;
finish ztrail;


    /* Subroutine to sweep in a pivot          */
    /* on entry: ii has the position(s) to pivot   */
    /* on exit:  in, l, dfe, mse, rsq recalculated */
start swp;
    if abs(c[ii,ii])<1e-9 then
    do;
        print , "FAILURE", c;
        stop;
    end;
    c=sweep(c,ii);
    in[ii,]=^in[ii,];
    l=sum(in);
    dfe=n-1-l;
    sse=c[k1,k1];
    mse=sse/dfe;
    rsq=1-sse/ypy;
finish swp;


    /* Subroutine to collect bprint results           */
    /* on entry: l,mse,rsq, and c set up to collect    */
    /* on exit:  bprint has another row                */

start bpr;
    bprint=bprint//(l||mse||rsq||(c [ik,k1]#in)`);
finish bpr;

    /*              Stepwise Methods                   */
    /* After a run to the initial routine, which sets up   */
```

Example 8.6. Regression of Subsets of Variables ⧫ 147

```
    /* the data, four different routines can be called       */
    /* to do four different model-selection methods.         */

start seq;
    run initial;          /* initialization                  */
    run all;              /* all possible models             */
    run forward;          /* foreward selection method       */
    run backward;         /* backward elimination method     */
    run stepwise;         /* stepwise method                 */
finish seq;


/*              Data on physical fitness                   */
/* these measurements were made on men involved in a physical*/
/* fitness course at n.c.state univ.  the variables are      */
/* age(years), weight(kg), oxygen uptake rate(ml per kg body */
/* weight per minute), time to run 1.5 miles(minutes), heart */
/* rate while resting, heart rate while running (same time   */
/* oxygen rate measured), and maximum heart rate recorded    */
/* while running. certain values of maxpulse were modified   */
/* for consistency.         data courtesy dr. a.c. linnerud  */

data=
  { 44 89.47  44.609 11.37 62 178 182  ,
    40 75.07  45.313 10.07 62 185 185  ,
    44 85.84  54.297  8.65 45 156 168  ,
    42 68.15  59.571  8.17 40 166 172  ,
    38 89.02  49.874  9.22 55 178 180  ,
    47 77.45  44.811 11.63 58 176 176  ,
    40 75.98  45.681 11.95 70 176 180  ,
    43 81.19  49.091 10.85 64 162 170  ,
    44 81.42  39.442 13.08 63 174 176  ,
    38 81.87  60.055  8.63 48 170 186  ,
    44 73.03  50.541 10.13 45 168 168  ,
    45 87.66  37.388 14.03 56 186 192  ,
    45 66.45  44.754 11.12 51 176 176  ,
    47 79.15  47.273 10.60 47 162 164  ,
    54 83.12  51.855 10.33 50 166 170  ,
    49 81.42  49.156  8.95 44 180 185  ,
    51 69.63  40.836 10.95 57 168 172  ,
    51 77.91  46.672 10.00 48 162 168  ,
    48 91.63  46.774 10.25 48 162 164  ,
    49 73.37  50.388 10.08 67 168 168  ,
    57 73.37  39.407 12.63 58 174 176  ,
    54 79.38  46.080 11.17 62 156 165  ,
    52 76.32  45.441  9.63 48 164 166  ,
    50 70.87  54.625  8.92 48 146 155  ,
    51 67.25  45.118 11.08 48 172 172  ,
    54 91.63  39.203 12.88 44 168 172  ,
    51 73.71  45.790 10.47 59 186 188  ,
    57 59.08  50.545  9.93 49 148 155  ,
    49 76.32  48.673  9.40 56 186 188  ,
    48 61.24  47.920 11.50 52 170 176  ,
    52 82.78  47.467 10.50 53 170 172
```

```
        };

x=data[,{1 2 4 5 6 7 }];
y=data[,3];
free data;
varnames={age weight runtime rstpulse runpulse maxpulse};
reset fw=8 linesize=90;
run seq;
```

The results are as follows.

Example 8.6. Regression of Subsets of Variables ◆ 149

BB

NPARM	MSE	RSQUARE	AGE	WEIGHT	RUNTIME	RSTPULSE	RUNPULSE	MAXPULSE
1	26.63425	0.092777	-0.31136	0	0	0	0	0
2	25.82619	0.150635	-0.37042	-0.15823	0	0	0	0
1	28.58034	0.026488	0	-0.1041	0	0	0	0
2	7.755636	0.744935	0	-0.02548	-3.2886	0	0	0
3	7.226318	0.770831	-0.17388	-0.05444	-3.14039	0	0	0
2	7.168422	0.764247	-0.15037	0	-3.20395	0	0	0
1	7.533843	0.74338	0	0	-3.31056	0	0	0
2	7.798261	0.743533	0	0	-3.28661	-0.00968	0	0
3	7.336089	0.767349	-0.16755	0	-3.07925	-0.04549	0	0
4	7.366649	0.775033	-0.19603	-0.05915	-2.9889	-0.05326	0	0
3	8.037314	0.745111	0	-0.02569	-3.26268	-0.01041	0	0
2	24.91487	0.180607	0	-0.09305	0	-0.27474	0	0
3	20.28031	0.356847	-0.44698	-0.15647	0	-0.32186	0	0
2	21.27632	0.30027	-0.38882	0	0	-0.32286	0	0
1	24.67582	0.159485	0	0	0	-0.27921	0	0
2	23.26003	0.235031	0	0	0	-0.20684	-0.15262	0
3	16.81799	0.466648	-0.52338	0	0	-0.22524	-0.23769	0
4	16.26146	0.503398	-0.56317	-0.12697	0	-0.22981	-0.2246	0
3	23.81815	0.244651	0	-0.06381	0	-0.20843	-0.14279	0
4	7.785151	0.762252	0	-0.01231	-3.16759	0.016669	-0.0749	0
5	6.213174	0.817556	-0.28528	-0.05184	-2.70392	-0.02711	-0.12628	0
4	6.166944	0.81167	-0.26213	0	-2.77733	-0.01981	-0.12874	0
3	7.507972	0.761898	0	0	-3.17665	0.017616	-0.07658	0
2	7.254263	0.761424	0	0	-3.14019	0	-0.07351	0
3	5.956692	0.811094	-0.2564	0	-2.82538	0	-0.13091	0
4	6.009033	0.816493	-0.27642	-0.04932	-2.77237	0	-0.12932	0
3	7.510162	0.761829	0	-0.01315	-3.13261	0	-0.07189	0
2	25.333	0.166855	0	-0.05987	0	0	-0.19797	0
3	18.63184	0.409126	-0.54408	-0.12049	0	0	-0.28248	0
2	18.97378	0.375995	-0.50665	0	0	0	-0.29382	0
1	24.70817	0.158383	0	0	0	0	-0.2068	0
2	21.60626	0.289419	0	0	0	0	-0.6818	0.571538
3	18.21725	0.422273	-0.4214	0	0	0	-0.57966	0.361557
4	17.29877	0.47172	-0.45243	-0.14944	0	0	-0.61723	0.426862
3	21.41763	0.320779	0	-0.11815	0	0	-0.71745	0.635395
4	6.030105	0.815849	0	-0.05159	-2.9255	0	-0.39529	0.38537
5	5.176338	0.848002	-0.21962	-0.0723	-2.68252	0	-0.3734	0.304908
4	5.343462	0.836818	-0.19773	0	-2.76758	0	-0.34811	0.270513
3	5.991568	0.809988	0	0	-2.97019	0	-0.37511	0.354219
4	6.208523	0.8104	0	0	-3.00426	0.016412	-0.37778	0.353998
5	5.549941	0.837031	-0.20154	0	-2.7386	-0.01208	-0.34562	0.269064
6	5.368247	0.848672	-0.22697	-0.07418	-2.62865	-0.02153	-0.36963	0.303217
5	6.263348	0.816083	0	-0.05091	-2.95182	0.01239	-0.39704	0.384793
4	20.11235	0.385797	0	-0.1194	0	-0.19092	-0.64584	0.609632
5	15.1864	0.554066	-0.47923	-0.1527	0	-0.21555	-0.53045	0.385424
4	16.29247	0.502451	-0.44717	0	0	-0.21266	-0.49323	0.319267
3	20.37729	0.353772	0	0	0	-0.18993	-0.61019	0.545236
2	25.11456	0.174039	0	0	0	-0.25219	0	-0.07364
3	19.2347	0.390007	-0.52736	0	0	-0.26492	0	-0.20024
4	18.80875	0.425607	-0.55881	-0.12604	0	-0.27056	0	-0.17799
3	25.59719	0.188232	0	-0.07874	0	-0.25524	0	-0.05502
4	8.311496	0.746179	0	-0.02053	-3.25232	-0.00393	0	-0.02064
5	7.19584	0.788701	-0.25795	-0.04936	-2.86147	-0.04121	0	-0.08153
4	7.091611	0.783432	-0.23928	0	-2.92597	-0.0339	0	-0.08777
3	8.033673	0.745227	0	0	-3.26805	-0.00193	0	-0.02526
2	7.746932	0.745221	0	0	-3.27232	0	0	-0.02561
3	6.882626	0.78173	-0.22923	0	-3.01222	0	0	-0.09094
4	7.00018	0.786224	-0.24436	-0.04525	-2.97011	0	0	-0.08585
3	8.00441	0.746155	0	-0.02027	-3.26114	0	0	-0.02139
2	28.35356	0.067516	0	-0.07074	0	0	0	-0.12159
3	22.38148	0.290212	-0.54076	-0.11605	0	0	0	-0.24445
2	22.50135	0.259982	-0.5121	0	0	0	0	-0.2637
1	27.71259	0.056046	0	0	0	0	0	-0.13762

BPRINT

NPARM	MSE	RSQUARE	AGE	WEIGHT	RUNTIME	RSTPULSE	RUNPULSE	MAXPULSE
1	7.533843	0.74338	0	0	-3.31056	0	0	0
2	7.168422	0.764247	-0.15037	0	-3.20395	0	0	0
3	5.956692	0.811094	-0.2564	0	-2.82538	0	-0.13091	0
4	5.343462	0.836818	-0.19773	0	-2.76758	0	-0.34811	0.270513
5	5.176338	0.848002	-0.21962	-0.0723	-2.68252	0	-0.3734	0.304908
6	5.368247	0.848672	-0.22697	-0.07418	-2.62865	-0.02153	-0.36963	0.303217

```
                                    BPRINT
           NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE


               1 7.533843  0.74338        0          0 -3.31056        0        0        0
               2 7.168422 0.764247 -0.15037          0 -3.20395        0        0        0
               3 5.956692 0.811094  -0.2564          0 -2.82538        0 -0.13091        0
               4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513


                                    BPRINT
           NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE


               6 5.368247 0.848672 -0.22697 -0.07418 -2.62865 -0.02153 -0.36963 0.303217
               5 5.176338 0.848002 -0.21962  -0.0723 -2.68252        0  -0.3734 0.304908
               4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513


                                    BPRINT
           NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE


               1 7.533843  0.74338        0          0 -3.31056        0        0        0
               2 7.168422 0.764247 -0.15037          0 -3.20395        0        0        0
               3 5.956692 0.811094  -0.2564          0 -2.82538        0 -0.13091        0
               4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513


               THE BEST MODEL FOR EACH NUMBER OF PARAMETERS



NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE

 1 7.533843  0.74338        0          0 -3.31056        0        0        0
 2 7.168422 0.764247 -0.15037          0 -3.20395        0        0        0
 3 5.956692 0.811094  -0.2564          0 -2.82538        0 -0.13091        0
 4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513
 5 5.176338 0.848002 -0.21962  -0.0723 -2.68252        0  -0.3734 0.304908
 6 5.368247 0.848672 -0.22697 -0.07418 -2.62865 -0.02153 -0.36963 0.303217


          FORWARD SELECTION METHOD



NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE

 1 7.533843  0.74338        0          0 -3.31056        0        0        0
 2 7.168422 0.764247 -0.15037          0 -3.20395        0        0        0
 3 5.956692 0.811094  -0.2564          0 -2.82538        0 -0.13091        0
 4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513


          BACKWARD ELIMINATION



NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE

 6 5.368247 0.848672 -0.22697 -0.07418 -2.62865 -0.02153 -0.36963 0.303217
 5 5.176338 0.848002 -0.21962  -0.0723 -2.68252        0  -0.3734 0.304908
 4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513


          STEPWISE METHOD



NPARM        MSE  RSQUARE        AGE     WEIGHT   RUNTIME RSTPULSE RUNPULSE MAXPULSE

 1 7.533843  0.74338        0          0 -3.31056        0        0        0
 2 7.168422 0.764247 -0.15037          0 -3.20395        0        0        0
 3 5.956692 0.811094  -0.2564          0 -2.82538        0 -0.13091        0
 4 5.343462 0.836818 -0.19773          0 -2.76758        0 -0.34811 0.270513
```

Example 8.7. Response Surface Methodology ⬥ 151

Example 8.7. Response Surface Methodology

A regression model with a complete quadratic set of regressions across several factors can be processed to yield the estimated critical values that may optimize a response. First, the regression is performed for two variables according to the model

$$y = c + b_1 x_1 + b_2 x_2 + a_{11} x_1^2 + a_{12} x_1 x_2 + a_{22} x_2^2 + e .$$

The estimates are then divided into a vector of linear coefficients (estimates) **b** and a matrix of quadratic coefficients **A**. The solution for critical values is

$$\mathbf{x} = -\frac{1}{2} \mathbf{A}^{-1} \mathbf{b} .$$

The following program creates a module to perform quadratic response surface regression.

```
/*          Quadratic Response Surface Regression          */
/* This matrix routine reads in the factor variables and   */
/* the response, forms the quadratic regression model and  */
/* estimates the parameters, then solves for the optimal   */
/* response, prints the optimal factors and response, and  */
/* then displays the eigenvalues and eigenvectors of the   */
/* matrix of quadratic parameter estimates to determine if */
/* the solution is a maximum or minimum, or saddlepoint, and */
/* which direction has the steepest and gentlest slopes.   */
/*                                                         */
/* Given that d contains the factor variables,             */
/* and y contains the response.                            */
/*                                                         */
start rsm;
   n=nrow(d);
   k=ncol(d);                                /* dimensions */
   x=j(n,1,1)||d;                    /* set up design matrix */
   do i=1 to k;
      do j=1 to i;
         x=x||d[,i] #d[,j];
      end;
   end;
   beta=solve(x`*x,x`*y);       /* solve parameter estimates */
   print "Parameter Estimates" , beta;
   c=beta[1];                          /* intercept estimate */
   b=beta[2:(k+1)];                      /* linear estimates */
   a=j(k,k,0);
   L=k+1;                   /* form quadratics into matrix */
   do i=1 to k;
      do j=1 to i;
         L=L+1;
         a[i,j]=beta [L,];
      end;
   end;
   a=(a+a`)*.5;                                /* symmetrize */
```

```
    xx=-.5*solve(a,b);                 /* solve for critical value */
    print , "Critical Factor Values" , xx;
        /* Compute response at critical value */
    yopt=c + b'*xx + xx'*a*xx;
    print , "Response at Critical Value" yopt;
    call eigen(eval,evec,a);
    print , "Eigenvalues and Eigenvectors", eval, evec;
    if min(eval)>0 then print , "Solution Was a Minimum";
    if max(eval)<0 then print , "Solution Was a Maximum";
finish rsm;
```

Running the module with the following sample data produces the results shown be-
low.

```
/* Sample Problem with Two Factors */
d={-1 -1, -1  0, -1  1,
    0 -1,  0  0,  0  1,
    1 -1,  1  0,  1  1};
y={ 71.7, 75.2, 76.3, 79.2, 81.5, 80.2, 80.1, 79.1, 75.8};
run rsm;
```

```
                            BETA

                         81.222222
                         1.9666667
                         0.2166667
                         -3.933333
                            -2.225
                         -1.383333


                             XX

                         0.2949376
                         -0.158881


                                        YOPT

              Response at Critical Value 81.495032


                            EVAL

                          -0.96621
                         -4.350457


                            EVEC

                    -0.351076 0.9363469
                     0.9363469 0.3510761
```

Example 8.8. Logistic and Probit Regression for Binary Response Models

A binary response Y is fit to a linear model according to

$$
\begin{aligned}
\Pr(Y = 1) &= F(\mathbf{X}\beta) \\
\Pr(Y = 0) &= 1 - F(\mathbf{X}\beta)
\end{aligned}
$$

where F is some smooth probability distribution function. The normal and logistic distribution functions are supported. The method is maximum likelihood via iteratively reweighted least squares (described by Charnes, Frome, and Yu 1976; Jennrich and Moore 1975; and Nelder and Wedderburn 1972). The row scaling is done by the derivative of the distribution (density). The weighting is done by $w/p(1 - p)$, where w has the counts or other weights. The following code calculates logistic and probit regression for binary response models.

```
/* routine for estimating binary response models          */
/* y is the binary response, x are regressors,            */
/* wgt are count weights,                                  */
/* model is choice of logit probit,                        */
/* parm has the names of the parameters                    */

proc iml ;

start binest;
   b=repeat(0,ncol(x),1);
   oldb=b+1;                              /* starting values */
   do iter=1 to 20 while(max(abs(b-oldb))>1e-8);
      oldb=b;
      z=x*b;
      run f;
      loglik=sum(((y=1)#log(p) + (y=0)#log(1-p))#wgt);
      btransp=b`;
      print iter loglik btransp;
      w=wgt/(p#(1-p));
      xx=f#x;
      xpxi=inv(xx`*(w#xx));
      b=b + xpxi*(xx`*(w#(y-p)));
   end;
   p0=sum((y=1)#wgt)/sum(wgt);          /* average response */
   loglik0=sum(((y=1)#log(p0) + (y=0)#log(1-p0))#wgt);
   chisq=(2#(loglik-loglik0));
   df=ncol(x)-1;
   prob=1-probchi(chisq,df);
   print ,
      'Likelihood Ratio with Intercept-only Model' chisq df prob,;
   stderr=sqrt(vecdiag(xpxi));
   tratio=b/stderr;
   print parm b stderr tratio,,;
finish;
```

```
   /*---routine to yield distribution function and density---*/
start f;
   if model='LOGIT' then
   do;
      p=1/(1+exp(-z));
      f=p#p#exp(-z);
   end;
   if model='PROBIT' then
   do;
      p=probnorm(z);
      f=exp(-z#z/2)/sqrt(8*atan(1));
   end;
finish;

     /* Ingot Data From COX (1970, pp. 67-68)*/
data={ 7 1.0 0 10, 14 1.0 0 31, 27 1.0 1 56, 51 1.0 3 13,
       7 1.7 0 17, 14 1.7 0 43, 27 1.7 4 44, 51 1.7 0 1,
       7 2.2 0 7, 14 2.2 2 33, 27 2.2 0 21, 51 2.2 0 1,
       7 2.8 0 12, 14 2.8 0 31, 27 2.8 1 22,
       7 4.0 0 9, 14 4.0 0 19, 27 4.0 1 16, 51 4.0 0 1};
nready=data[,3];
ntotal=data[,4];
n=nrow(data);
x=repeat(1,n,1)||(data[,{1 2}]);    /* intercept, heat, soak */
x=x//x;                                       /* regressors */
y=repeat(1,n,1)//repeat(0,n,1);          /* binary response */
wgt=nready//(ntotal-nready);                 /* row weights */
parm={intercept, heat, soak};       /* names of regressors */

model={logit};
run binest;                                /* run logit model */

model={probit};
run binest;                               /* run probit model */
```

The results are as follows.

```
                    ITER     LOGLIK

                     1  -268.248


                     BTRANSP

             0           0           0


                    ITER     LOGLIK

                     2 -76.29481


                     BTRANSP

        -2.159406 0.0138784 0.0037327


                    ITER     LOGLIK

                     3 -53.38033


                     BTRANSP

         -3.53344 0.0363154 0.0119734


                    ITER     LOGLIK

                     4 -48.34609


                     BTRANSP

        -4.748899 0.0640013 0.0299201


                    ITER     LOGLIK

                     5 -47.69191


                     BTRANSP

        -5.413817 0.0790272    0.04982


                    ITER     LOGLIK

                     6 -47.67283


                     BTRANSP

        -5.553931 0.0819276 0.0564395


                    ITER     LOGLIK

                     7 -47.67281


                     BTRANSP

         -5.55916 0.0820307 0.0567708
```

```
                         ITER    LOGLIK

                          8 -47.67281


                            BTRANSP

                -5.559166 0.0820308 0.0567713


                    CHISQ        DF      PROB

                  11.64282          2 0.0029634


            PARM            B     STDERR    TRATIO

            INTERCEPT -5.559166 1.1196947 -4.964895
            HEAT       0.0820308 0.0237345 3.4561866
            SOAK       0.0567713 0.3312131 0.1714042


                         ITER    LOGLIK

                          1  -268.248


                            BTRANSP

                     0         0         0


                         ITER    LOGLIK

                          2 -71.71043


                            BTRANSP

                -1.353207  0.008697 0.0023391


                         ITER    LOGLIK

                          3 -51.64122


                            BTRANSP

                -2.053504 0.0202739 0.0073888


                         ITER    LOGLIK

                          4 -47.88947


                            BTRANSP

                -2.581302  0.032626  0.018503


                         ITER    LOGLIK

                          5 -47.48924


                            BTRANSP

                -2.838938 0.0387625 0.0309099
```

Example 8.9. Linear Programming ⬥ 157

Example 8.9. Linear Programming

The two-phase method for linear programming can be used to solve the problem

$$\max \mathbf{c}'\mathbf{x}$$
$$\text{st. } \mathbf{Ax} \leq, =, \geq \mathbf{b}$$
$$\mathbf{x} \geq 0$$

A routine written in IML to solve this problem follows. The approach appends slack, surplus, and artificial variables to the model where needed. It then solves phase 1 to find a primal feasible solution. If a primal feasible solution exists and is found, the routine then goes on to phase 2 to find an optimal solution, if one exists. The routine is general enough to handle minimizations as well as maximizations.

```
/*  Subroutine to solve Linear Programs                       */
/*  names:    names of the decision variables                 */
/*  obj:      coefficients of the objective function          */
/*  maxormin: the value 'MAX' or 'MIN', upper or lowercase    */
/*  coef:     coefficients of the constraints                 */
/*  rel:      character array of values: '<=' or '>=' or '=' */
/*  rhs:      right-hand side of constraints                  */
/*  activity: returns the optimal value of decision variables*/
/*                                                            */

start linprog( names, obj, maxormin, coef, rel, rhs, activity);

   bound=1.0e10;
   m=nrow(coef);
   n=ncol(coef);

      /* Convert to maximization */
   if upcase(maxormin)='MIN' then o=-1;
   else o=1;

      /* Build logical variables */
   rev=(rhs<0);
   adj=(-1*rev)+^ rev;
   ge =(( rel = '>=' ) & ^rev) | (( rel = '<=' ) & rev);
   eq=(rel='=');
   if max(ge)=1 then
   do;
      sr=I(m);
      logicals=-sr[,loc(ge)]||I(m);
      artobj=repeat(0,1,ncol(logicals)-m)|(eq+ge)';
   end;
   else do;
      logicals=I(m);
      artobj=eq';
   end;
   nl=ncol(logicals);
   nv=n+nl+2;
```

```
   /* Build coef matrix */
a=((o*obj)||repeat(0,1,nl)||{ -1 0 })//
  (repeat(0,1,n)||-artobj||{ 0 -1 })//
  ((adj#coef)||logicals||repeat(0,m,2));

   /* rhs, lower bounds, and basis */
b={0,0}//(adj#rhs);
L=repeat(0,1,nv-2)||-bound||-bound;
basis=nv-(0:nv-1);

   /* Phase 1 - primal feasibility */
call lp(rc,x,y,a,b,nv,,l,basis);
print ( { ' ',
        '**********Primal infeasible problem************',
        ' ',
        '*********Numerically unstable problem**********',
        '*********Singular basis encountered************',
        '*******Solution is numerically unstable********',
        '***Subroutine could not obtain enough memory***',
        '**********Number of iterations exceeded********'
        }[rc+1]);
if x[nv] ^=0 then
do;
   print '**********Primal infeasible problem************';
   stop;
end;
if rc>0 then stop;

   /* phase 2 - dual feasibility */
u=repeat(.,1,nv-2)||{ . 0 };
L=repeat(0,1,nv-2)||-bound||0;
call lp(rc,x,y,a,b,nv-1,u,l,basis);

   /* Report the solution */
print ( { '*************Solution is optimal***************',
          '*********Numerically unstable problem**********',
          '*************Unbounded problem****************',
          '*******Solution is numerically unstable********',
          '*********Singular basis encountered************',
          '*******Solution is numerically unstable********',
          '***Subroutine could not obtain enough memory***',
          '**********Number of iterations exceeded********'
          }[rc+1]);
value=o*x  [nv-1];
print ,'Objective Value ' value;
activity= x [1:n] ;
print ,'Decision Variables ' activity[r=names];
lhs=coef*x[1:n];
dual=y[3:m+2];
print ,'Constraints ' lhs rel rhs dual,
       '*********************************************';

finish;
```

Example 8.9. Linear Programming ◆ 159

Consider the following product mix example (Hadley 1963). A shop with three machines, A, B, and C, turns out products 1, 2, 3, and 4. Each product must be processed on each of the three machines (for example, lathes, drills, and milling machines). The following table shows the number of hours required by each product on each machine.

Machine	Product 1	2	3	4
A	1.5	1	2.4	1
B	1	5	1	3.5
C	1.5	3	3.5	1

The weekly time available on each of the machines is 2000, 8000, and 5000 hours, respectively. The products contribute 5.24, 7.30, 8.34, and 4.18 to profit, respectively. What mixture of products can be manufactured that maximizes profit? You can solve the problem as follows:

```
names={'product 1' 'product 2' 'product 3' 'product 4'};
profit={ 5.24 7.30 8.34 4.18};
tech={ 1.5 1 2.4 1 ,
       1 5 1 3.5 ,
       1.5 3 3.5 1 };
time={ 2000, 8000, 5000};
rel={ '<=', '<=', '<=' };
run linprog(names,profit,'max',tech,rel,time,products);
```

The output from this example follows.

```
                            VALUE

              Objective Value  12737.059


                          ACTIVITY

          Decision Variables  product 1 294.11765
                              product 2      1500
                              product 3         0
                              product 4 58.823529


                     LHS REL       RHS       DUAL

          Constraints    2000 <=   2000 1.9535294
                         8000 <=   8000 0.2423529
                         5000 <=   5000 1.3782353
```

The following example shows how to find the minimum cost flow through a network using linear programming. The arcs are defined by an array of tuples; each tuple names a new arc. The elements in the arc tuples give the names of the tail and head

nodes defining the arc. The following data are needed: arcs, cost for a unit of flow across the arcs, nodes, and supply and demand at each node.

A program generates the node-arc incidence matrix and calls the linear program routine for solution:

```
arcs={ 'ab' 'bd' 'ad' 'bc' 'ce' 'de' 'ae' };
cost={ 1 2 4 3 3 2 9 };
nodes={ 'a', 'b', 'c', 'd', 'e'};
supdem={ 2, 0, 0, -1, -1 };
rel=repeat('=',nrow(nodes),1);
inode=substr(arcs,1,1);
onode=substr(arcs,2,1);
free n_a_i n_a_o;
do i=1 to ncol(arcs);
   n_a_i=n_a_i || (inode[i]=nodes);
   n_a_o=n_a_o || (onode[i]=nodes);
end;
n_a=n_a_i - n_a_o;
run linprog(arcs,cost,'min',n_a,rel,supdem,x);
```

The solution is as follows.

```
                                VALUE

                 Objective Value        8


                           ACTIVITY

         Decision Variables  ab            2
                             bd            2
                             ad            0
                             bc            0
                             ce            0
                             de            1
                             ae            0


                      LHS REL        RHS       DUAL

         Constraints      2 =          2       -2.5
                          0 =          0       -1.5
                          0 =          0       -0.5
                         -1 =         -1       -0.5
                         -1 =         -1       -2.5
```

Example 8.10. Quadratic Programming

The quadratic program

$$\min \mathbf{c}'\mathbf{x} + \mathbf{x}'\mathbf{H}\mathbf{x}/2$$
$$\text{st. } \mathbf{G}\mathbf{x} \leq, =, \geq \mathbf{b}$$
$$\mathbf{x} \geq 0$$

Example 8.10. Quadratic Programming ⬩ 161

can be solved by solving an equivalent linear complementarity problem when **H** is positive semidefinite. The approach is outlined in the discussion of the LCP subroutine in Chapter 20, "Language Reference."

The following routine solves the quadratic problem.

```
/*            Routine to solve quadratic programs          */
/* names: the names of the decision variables             */
/* c:  vector of linear coefficients of the objective function */
/* H:  matrix of quadratic terms in the objective function    */
/* G:  matrix of constraint coefficients                  */
/* rel: character array of values: '<=' or '>=' or '='      */
/* b:   right hand side of constraints                    */
/* activity: returns the optimal value of decision variables  */

start qp( names, c, H, G, rel, b, activity);
   if min(eigval(h))<0 then
   do;
      print
           'ERROR: The minimum eigenvalue of the H matrix is negative. ';
      print '      Thus it is not positive semidefinite.                ';
      print '      QP is terminating with this error.                   ';
      stop;
   end;
   nr=nrow(G);
   nc=ncol(G);

      /* Put in canonical form */
   rev=(rel='<=');
   adj=(-1 * rev) + ^rev;
   g=adj# G; b = adj # b;
   eq=( rel = '='   );
   if max(eq)=1 then
   do;
      g=g // -(diag(eq)*G)[loc(eq),];
      b=b // -(diag(eq)*b)[loc(eq)];
   end;
   m=(h || -g`) //(g || j(nrow(g),nrow(g),0));
   q=c // -b;

      /* Solve the problem */
   call lcp(rc,w,z,M,q);

      /* Report the solution */
   reset noname;
   print ( { '*************Solution is optimal***************',
             '*********No solution possible*****************',
             ' ',
             ' ',
             ' ',
             '**********Solution is numerically unstable*****',
             '***********Not enough memory******************',
             '**********Number of iterations exceeded********'}[rc+1]);
   reset name;
   activity=z[1:nc];
   objval=c`*activity + activity`*H*activity/2;
   print ,'Objective Value ' objval,
         'Decision Variables ' activity[r=names],
```

```
                   '***********************************************';

        finish qp;
```

As an example, consider the following problem in portfolio selection. Models used in selecting investment portfolios include assessment of the proposed portfolio's expected gain and its associated risk. One such model seeks to minimize the variance of the portfolio subject to a minimum expected gain. This can be modeled as a quadratic program in which the decision variables are the proportions to invest in each of the possible securities. The quadratic component of the objective function is the covariance of gain between the securities; the first constraint is a proportionality constraint; and the second constraint gives the minimum acceptable expected gain.

The following data are used to illustrate the model and its solution:

```
c = { 0, 0, 0, 0 };
h = { 1003.1 4.3 6.3 5.9 ,
         4.3 2.2 2.1 3.9 ,
         6.3 2.1 3.5 4.8 ,
         5.9 3.9 4.8 10 };
g = {  1    1    1    1   ,
      .17  .11  .10  .18 };
b = { 1 , .10 };
rel = { '=', '>='};
names = {'ibm', 'dec', 'dg', 'prime' };
run qp(names,c,h,g,rel,b,activity);
```

The following output shows that the minimum variance portfolio achieving the 0.10 expected gain is composed of DEC and DG stock in proportions of 0.933 and 0.067.

```
                              OBJVAL

                 Objective Value  1.0966667


                              ACTIVITY

          Decision Variables  ibm              0
                              dec      0.9333333
                              dg       0.0666667
                              prime            0
```

Example 8.11. Regression Quantiles

The technique of parameter estimation in linear models using the notion of regression quantiles is a generalization of the LAE or LAV least absolute value estimation technique. For a given quantile q, the estimate $\mathbf{b}^*$ of β in the model

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

Example 8.11. Regression Quantiles ♦ 163

is the value of b that minimizes

$$\sum_{t \in T} q|y_t - x_t b| - \sum_{t \in S} (1-q)|y_t - x_t b|$$

where $T = \{t | y_t \geq x_t b\}$ and $S = \{t | y_t \leq x_t\}$. For $q = 0.5$, the solution $\mathbf{b}^*$ is identical to the estimates produced by the LAE. The following routine finds this estimate using linear programming:

```
/* Routine to find regression quantiles                  */
/* yname:     name of dependent variable                 */
/* y:         dependent variable                         */
/* xname:     names of independent variables             */
/* X:         independent variables                      */
/* b:         estimates                                  */
/* predict:   predicted values                           */
/* error:     difference of y and predicted.             */
/* q:         quantile                                   */
/*                                                       */
/* notes:     This subroutine finds the estimates b      */
/* that minimize                                         */
/*                                                       */
/*          q * (y - Xb) * e  + (1-q) * (y - Xb) * ^e    */
/*                                                       */
/* where e = ( Xb <= y ).                                */
/*                                                       */
/* This subroutine follows the approach given in:        */
/*                                                       */
/* Koenker, R. and G. Bassett (1978). Regression         */
/* quantiles. Econometrica. Vol. 46. No. 1. 33-50.       */
/*                                                       */
/* Basssett, G. and R. Koenker (1982). An empirical      */
/* quantile function for linear models with iid errors.  */
/* JASA. Vol. 77. No. 378. 407-415.                      */
/*                                                       */
/* When q = .5 this is equivalent to minimizing the sum  */
/* of the absolute deviations, which is also known as    */
/* L1 regression.  Note that for L1 regression, a faster */
/* and more accurate algorithm is available in the SAS/IML */
/* routine LAV, which is based on the approach given in: */
/*                                                       */
/* Madsen, K. and Nielsen, H. B. (1993). A finite        */
/* smoothing algorithm for linear L1 estimation.         */
/* SIAM J. Optimization, Vol. 3. 223-235.                */
/*-----------------------------------------------------*/
start rq( yname, y, xname, X, b, predict, error, q);
   bound=1.0e10;
   coef = X';
   m = nrow(coef);
   n = ncol(coef);


   /*----------------build rhs and bounds-------------------*/
      e  = repeat(1,1,n)';
```

```
   r   =   {0 0} || ((1-q)*coef*e)`;
   sign = repeat(1,1,m);

   do i=1 to m;
      if r[2+i] < 0 then do;
         sign[i]   = -1;
         r[2+i]    = -r[2+i];
         coef[i,]  = -coef[i,];
      end;
   end;

l   = repeat(0,1,n) || repeat(0,1,m)  || -bound || -bound ;
u   = repeat(1,1,n) || repeat(.,1,m)  ||   { . . }  ;

/*-------------build coefficient matrix and basis----------*/
 a  = (          y`       || repeat(0,1,m)  ||   { -1 0 }   ) //
      ( repeat(0,1,n) || repeat(-1,1,m) ||   { 0 -1 }   ) //
      (         coef    ||      I(m)        || repeat(0,m,2)) ;
 basis = n+m+2 - (0:n+m+1);

/*----------------find a feasible solution-----------------*/
call lp(rc,p,d,a,r,,u,l,basis);

/*----------------find the optimal solution----------------*/
l   = repeat(0,1,n) || repeat(0,1,m)  || -bound || {0} ;
u   = repeat(1,1,n) || repeat(0,1,m)  ||   { . 0 }  ;
call lp(rc,p,d,a,r,n+m+1,u,l,basis);

/*---------------- report the solution---------------------*/
variable = xname`; b=d[3:m+2];
do i=1 to m;
   b[i] =  b[i] * sign[i];
end;
predict = X*b;
error = y - predict;
wsum  = sum ( choose(error<0 , (q-1)*error , q*error) );

print ,,'Regression Quantile Estimation' ,
        'Dependent Variable: ' yname ,
        'Regression Quantile: ' q ,
        'Number of Observations: ' n ,
        'Sum of Weighted Absolute Errors: ' wsum ,
         variable b,
         X y predict error;
finish rq;
```

Example 8.11. Regression Quantiles ⬥ 165

The following example uses data on the United States population from 1790 to 1970:

```
z = {   3.929 1790 ,
        5.308 1800 ,
        7.239 1810 ,
        9.638 1820 ,
       12.866 1830 ,
       17.069 1840 ,
       23.191 1850 ,
       31.443 1860 ,
       39.818 1870 ,
       50.155 1880 ,
       62.947 1890 ,
       75.994 1900 ,
       91.972 1910 ,
      105.710 1920 ,
      122.775 1930 ,
      131.669 1940 ,
      151.325 1950 ,
      179.323 1960 ,
      203.211 1970 };

y=z[,1];
x=repeat(1,19,1)||z[,2]||z[,2]##2;
run rq('pop',y,{'intercpt' 'year' 'yearsq'},x,b1,pred,resid,.5);
```

The output is as follows.

```
                        Dependent Variable: pop


                                              Q

               Regression Quantile:          0.5


                                              N

            Number of Observations:           19


                                                  WSUM

         Sum of Weighted Absolute Errors:  14.826429


                     VARIABLE          B

                     intercpt 21132.758
                     year      -23.52574
                     yearsq     0.006549


         X                           Y    PREDICT       ERROR

         1    1790   3204100     3.929 5.4549176 -1.525918
         1    1800   3240000     5.308     5.308 -4.54E-12
         1    1810   3276100     7.239 6.4708902 0.7681098
         1    1820   3312400     9.638 8.9435882 0.6944118
         1    1830   3348900    12.866 12.726094 0.1399059
         1    1840   3385600    17.069 17.818408 -0.749408
         1    1850   3422500    23.191 24.220529 -1.029529
         1    1860   3459600    31.443 31.932459 -0.489459
         1    1870   3496900    39.818 40.954196 -1.136196
         1    1880   3534400    50.155 51.285741 -1.130741
         1    1890   3572100    62.947 62.927094 0.0199059
         1    1900   3610000    75.994 75.878255 0.1157451
         1    1910   3648100    91.972 90.139224 1.8327765
         1    1920   3686400    105.71     105.71 8.669E-13
         1    1930   3724900   122.775 122.59058 0.1844157
         1    1940   3763600   131.669 140.78098 -9.111976
         1    1950   3802500   151.325 160.28118 -8.956176
         1    1960   3841600   179.323 181.09118 -1.768184
         1    1970   3880900   203.211   203.211 -2.96E-12
```

The L1 norm (when $q = 0.5$) tends to allow the fit to be better at more points at the expense of allowing some points to fit worse, as the plot of the residuals against the least squares residuals:

```
    /* Compare L1 residuals with least squares residuals  */
    /* Compute the least squares residuals                */
resid2=y-x*inv(x`*x)*x`*y;

    /* x axis of plot */
xx=repeat(x[,2] ,3,1);

    /* y axis of plot */
yy=resid//resid2//repeat(0,19,1);
```

Example 8.12. Simulations of a Univariate ARMA Process ◆ 167

```
        /* plot character*/
id=repeat('1',19,1)//repeat('2',19,1)//repeat('-',19,1);
call pgraf(xx||yy,id,'Year','Residual',
           '1=L(1) residuals, 2=least squares residual');
```

The output generated is as follows.

```
                    1=l(1) residuals, 2=least squares residual

        |
     5  +
        |                                          2                      2
  R     |                                                    2
  e     |         1  1                      2  2   1  2                 2
  s  0  +    -  1   -  -   1  -   -  1   -  2 1  1   -  1  1  -   -   -   1
  i     |    1                1  1     1  1                            1
  d     |
  u     |
  a  -5 +                                                2     2
  l     |
        |
        |                                                1     1
    -10 +
        |
        -+------+------+------+------+------+------+------+------+------+------+-
        1780   1800   1820   1840   1860   1880   1900   1920   1940   1960   1980

                                    Year
```
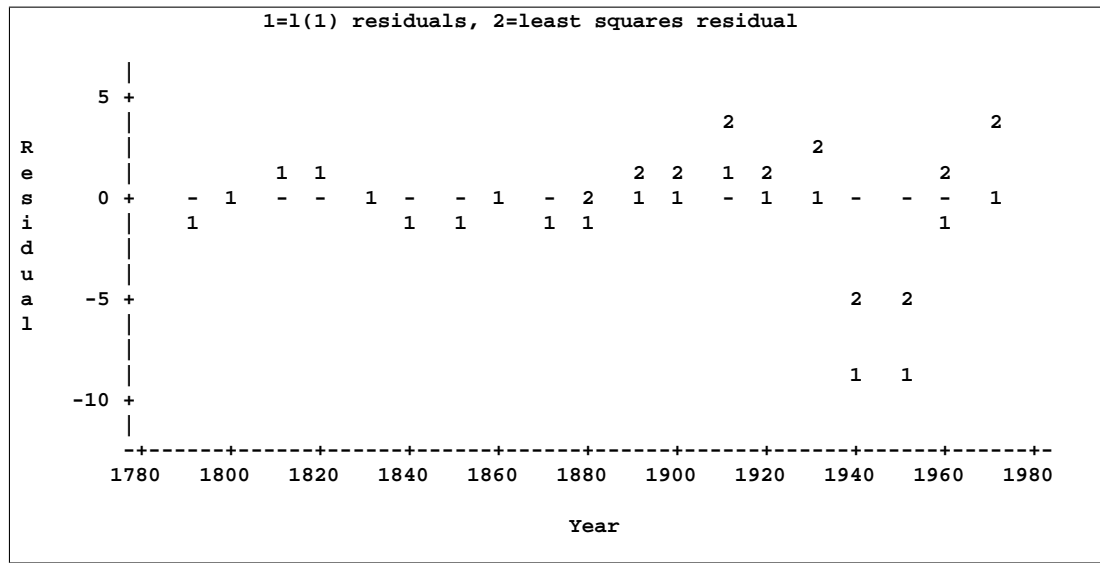
When $q = 0.5$, the results of this module can be compared with the results of the LAV routine:

```
b0 = {1 1 1};                   /* initial value              */
optn = j(4,1,.);                /* options vector             */

optn[1]= .;                     /* gamma default              */
optn[2]= 5;                     /* print all                  */
optn[3]= 0;                     /* McKean-Schradar variance   */
optn[4]= 1;                     /* convergence test           */

call LAV(rc, xr, x, y, b0, optn);
```

Example 8.12. Simulations of a Univariate ARMA Process

Simulations of time series with known ARMA structure are often needed as part of other simulations or as learning data sets for developing time series analysis skills. The following program generates a time series using the IML functions NORMAL, ARMACOV, HANKEL, PRODUCT, RATIO, TOEPLITZ, and ROOT.

```
reset noname;
start armasim(y,n,phi,theta,seed);
/*-----------------------------------------------------------*/
/* IML Module: armasim                                       */
/* Purpose: Simulate n data points from ARMA process         */
/*          exact covariance method                          */
/* Arguments:                                                */
/*                                                           */
/* Input: n    : series length                               */
/*        phi  : AR coefficients                             */
/*        theta: MA coefficients                             */
/*        seed : integer seed for normal deviate generator   */
/* Output: y: realization of ARMA process                    */
/* ----------------------------------------------------------*/

   p=ncol(phi)-1;
   q=ncol(theta)-1;
   y=normal(j(1,n+q,seed));

      /* Pure MA or white noise */
   if p=0 then y=product(theta,y)[,(q+1):(n+q)];
   else do;                   /* Pure AR or ARMA */

         /* Get the autocovariance function */
      call armacov(gamma,cov,ma,phi,theta,p);
      if gamma[1]<0 then
      do;
         print 'ARMA parameters not stable.';
         print 'Execution terminating.';
         stop;
      end;

         /* Form covariance matrix */
      gamma=toeplitz(gamma);

         /* Generate covariance between initial y and */
         /* initial innovations                       */
      if q>0 then
      do;
         psi=ratio(phi,theta,q);
         psi=hankel(psi[,-((-q):(-1))]);
         m=max(1,(q-p+1));
         psi=psi[-((-q):(-m)),];
         if p>q then psi=j(p-q,q,0)//psi;
         gamma=(gamma||psi)//(psi`||i(q));
      end;

         /* Use Cholesky root to get startup values */
      gamma=root(gamma);
      startup=y[,1:(p+q)]*gamma;
      e=y[,(p+q+1):(n+q)];

         /* Generate MA part */
      if q>0 then
```

```
        do;
           e=startup[,(p+1):(p+q)]||e;
           e=product(theta,e)[,(q+1):(n+q-p)];
        end;

        y=startup[,1:p];
        phi1=phi[,-(-(p+1):(-2))]`;

           /* Use difference equation to generate */
           /* remaining values                    */
        do ii=1 to n-p;
           y=y||(e[,ii]-y[,ii:(ii+p-1)]*phi1);
        end;
     end;
     y=y`;
  finish armasim; /* ARMASIM */

  run armasim(y,10,{1 -0.8},{1 0.5},1234321);
  print ,'Simulated Series:', y;
```

The output is as follows.

```
                              3.0764594
                              1.8931735
                              0.9527984
                              0.0892395
                              -1.811471
                                -2.8063
                               -2.52739
                              -2.865251
                              -1.332334
                              0.1049046
```

Example 8.13. Parameter Estimation for a Regression Model with ARMA Errors

Nonlinear estimation algorithms are required for obtaining estimates of the parameters of a regression model with innovations having an ARMA structure. The three estimation methods employed by the ARIMA procedure in SAS/ETS software are programmed in IML in the following example. The algorithms employed are slightly different from those used by PROC ARIMA, but the results obtained should be similar. This example combines the IML functions ARMALIK, PRODUCT, and RATIO to perform the estimation. Note the interactive nature of this example, illustrating how you can adjust the estimates when they venture outside of the stationary or invertible regions.

```
 /*-----------------------------------------------------------*/
 /*----   Grunfeld's Investment Models Fit with ARMA Errors   ----*/
 /*-----------------------------------------------------------*/
data grunfeld;
   input year gei gef gec wi wf wc;
   label gei='gross investment ge'
         gec='capital stock lagged ge'
         gef='value of outstanding shares ge lagged'
         wi ='gross investment w'
         wc ='capital stock lagged w'
         wf ='value of outstanding shares lagged w';
 /*---   GE STANDS FOR GENERAL ELECTRIC AND W FOR WESTINGHOUSE   ---*/
cards;
1935     33.1      1170.6      97.8      12.93      191.5      1.8
1936     45.0      2015.8      104.4     25.90      516.0      .8
1937     77.2      2803.3      118.0     35.05      729.0      7.4
1938     44.6      2039.7      156.2     22.89      560.4      18.1
1939     48.1      2256.2      172.6     18.84      519.9      23.5
1940     74.4      2132.2      186.6     28.57      628.5      26.5
1941     113.0     1834.1      220.9     48.51      537.1      36.2
1942     91.9      1588.0      287.8     43.34      561.2      60.8
1943     61.3      1749.4      319.9     37.02      617.2      84.4
1944     56.8      1687.2      321.3     37.81      626.7      91.2
1945     93.6      2007.7      319.6     39.27      737.2      92.4
1946     159.9     2208.3      346.0     53.46      760.5      86.0
1947     147.2     1656.7      456.4     55.56      581.4      111.1
1948     146.3     1604.4      543.4     49.56      662.3      130.6
1949     98.3      1431.8      618.3     32.04      583.8      141.8
1950     93.5      1610.5      647.4     32.24      635.2      136.7
1951     135.2     1819.4      671.3     54.38      723.8      129.7
1952     157.3     2079.7      726.1     71.78      864.1      145.5
1953     179.5     2371.6      800.3     90.08      1193.5     174.8
1954     189.6     2759.9      888.9     68.60      1188.9     213.5
;
```

```
proc iml;
reset noname;
/*------------------------------------------------------------*/
/*   name: ARMAREG Modules                                    */
/*   purpose: Perform Estimation for regression model with    */
/*   ARMA errors                                              */
/*   usage: Before invoking the command                       */
/*                                                            */
/*   run armareg;                                             */
/*                                                            */
/*   define the global parameters                             */
/*                                                            */
/*   x      - matrix of predictors.                           */
/*   y      - response vector.                                */
/*   iphi   - defines indices of nonzero AR parameters,       */
/*            omitting index 0 corresponding to the zero      */
/*            order constant one.                             */
/*   itheta - defines indices of nonzero MA parameters,       */
/*            omitting index 0 corresponding to the zero      */
/*            order constant one.                             */
/*   ml     - estimation option: -1 if Conditional Least      */
/*            Squares, 1 if Maximum Likelihood, otherwise     */
/*            Unconditional Least Squares.                     */
/*   delta  - step change in parameters (default 0.005).      */
/*   par    - initial values of parms. First ncol(iphi)       */
/*            values correspond to AR parms, next ncol(itheta)*/
/*            values correspond to MA parms, and remaining    */
/*            are regression coefficients.                     */
/*   init   - undefined or zero for first call to armareg.    */
/*   maxit  - maximum number of iterations. No other          */
/*            convergence criterion is used. You can invoke   */
/*            armareg without changing parameter values to    */
/*            continue iterations.                             */
/*   nopr   - undefined or zero implies no printing of        */
/*            intermediate results.                            */
/*                                                            */
/*   notes: Optimization using Gauss-Newton iterations        */
/*                                                            */
/*   No checking for invertibility or stationarity during     */
/*   estimation process. The parameter array par may be       */
/*   modified after running armareg to place estimates        */
/*   in the stationary and invertible regions, and then       */
/*   armareg may be run again. If a nonstationary AR operator */
/*   is employed, a PAUSE will occur after calling ARMALIK    */
/*   because of a detected singularity. Using STOP will       */
/*   permit termination of ARMAREG so that the AR             */
/*   coefficients may be modified.                            */
/*                                                            */
/*   T-ratios are only approximate and may be undependable,   */
/*   especially for small series.                             */
/*                                                            */
/*   The notation follows that of the IML function ARMALIK;   */
/*   the autoregressive and moving average coefficients have  */
/*   signs opposite those given by PROC ARIMA.                */
```

```
      /* Begin ARMA estimation modules */

      /* Generate residuals */
   start gres;
      noise=y-x*beta;
      previous=noise[:];
      if ml=-1 then do;                            /* Conditional LS */
         noise=j(nrow(y),1,previous)||noise;
         resid=product(phi,noise`) [, 1: nrow(noise)];
         resid=ratio(theta,resid,ncol(resid));
         resid=resid[,1:ncol(resid)]`;
      end;
      else do;                               /* Maximum likelihood */
         free l;
         call armalik(l,resid,std,noise,phi,theta);

            /* Nonstationary condition produces PAUSE */
         if nrow(l)=0 then
         do;
            print ,
            'In GRES: Parameter estimates outside stationary region.';
         end;
         else do;
            temp=l[3,]/(2#nrow(resid));
            if ml=1 then resid=resid#exp(temp);
         end;
      end;
   finish gres;                               /* finish module GRES  */

   start getpar;                                 /* Get Parameters  */
      if np=0 then phi=1;
      else do;
         temp=parm[,1:np];
         phi=1||j(1,p,0);
         phi[,iphi] =temp;
      end;
      if nq=0 then theta=1;
      else do;
         temp=parm[,np+1:np+nq];
         theta=1||j(1,q,0);
         theta[,itheta] =temp;
      end;
      beta=parm[,(np+nq+1):ncol(parm)]`;
   finish getpar;    /* finish module GETPAR  */


      /* Get SS Matrix - First Derivatives */
   start getss;
      parm=par;
      run getpar;
      run gres;
      s=resid;
      oldsse=ssq(resid);
      do k=1 to ncol(par);
```

```
           parm=par;
           parm[,k]=parm[,k]+delta;
           run getpar;
           run gres;
           s=s||((resid-s[,1])/delta);      /* append derivatives */
        end;
        ss=s`*s;
        if nopr^=0 then print ,'Gradient Matrix', ss;
        sssave=ss;
        do k=1 to 20;            /* Iterate if no reduction in SSE */
           do ii=2 to ncol(ss);
              ss[ii,ii]=(1+lambda)*ss[ii,ii];
           end;
           ss=sweep(ss,2:ncol(ss));       /* Gaussian elimination */
           delpar=ss[1,2:ncol(ss)];     /* update parm increments */
           parm=par+delpar;
           run getpar;
           run gres;
           sse=ssq(resid);
           if sse<oldsse then
           do;                        /* reduction, no iteration */
              lambda=lambda/10;
              k=21;
           end;

           else do;                               /* no reduction */
                               /* increase lambda and iterate */
              if nopr^=0 then print ,
                 'Lambda=' lambda 'SSE=' sse 'OLDSSE=' oldsse,
                 'Gradient Matrix', ss ;
              lambda=10*lambda;
              ss=sssave;
              if k=20 then
              do;
                 print 'In module GETSS:
                         No improvement in SSE after twenty iterations.';
                 print ' Possible Ridge Problem. ';
                 return;
              end;
           end;
        end;
        if nopr^=0 then print ,'Gradient Matrix', ss;
        finish getss;                          /* Finish module GETSS  */

   start armareg;                              /* ARMAREG main module */

        /* Initialize options and parameters */
        if nrow(delta)=0 then delta=0.005;
        if nrow(maxiter)=0 then maxiter=5;
        if nrow(nopr)=0 then nopr=0;
        if nrow(ml)=0 then ml=1;
        if nrow(init)=0 then init=0;
        if init=0 then
        do;
```

```
      p=max(iphi);
      q=max(itheta);
      np=ncol(iphi);
      nq=ncol(itheta);

          /* Make indices one-based */
      do k=1 to np;
         iphi[,k]=iphi[,k]+1;
      end;
      do k=1 to nq;
         itheta[,k]=itheta[,k]+1;
      end;

          /* Create row labels for Parameter estimates */
      if p>0 then parmname = concat("AR",char(1:p,2));
      if q>0 then parmname = parmname||concat("MA",char(1:p,2));
      parmname = parmname||concat("B",char(1:ncol(x),2));

          /* Create column labels for Parameter estimates */
      pname = {"Estimate" "Std. Error" "T-Ratio"};
      init=1;
   end;

       /* Generate starting values */
   if nrow(par)=0 then
   do;
      beta=inv(x`*x)*x`*y;
      if np+nq>0 then par=j(1,np+nq,0)||beta`;
      else par=beta`;
   end;
   print ,'Parameter Starting Values',;
   print par [colname=parmname];                /* stderr tratio */
   lambda=1e-6;                             /* Controls step size */
   do iter=1 to maxiter;             /* Do maxiter iterations */
      run getss;
      par=par+delpar;
      if nopr^=0 then
      do;
         print ,'Parameter Update',;
         print par [colname=parmname];       /* stderr tratio */
         print ,'Lambda=' lambda,;
      end;
      if abs(par[,1] )>1 then par[,1] =-.8;
   end;

   sighat=sqrt(sse/(nrow(y)-ncol(par)));
   print ,'Innovation Standard Deviation:' sighat;
   estm=par`||(sqrt(diag(ss[2:ncol(ss),2:ncol(ss)])))
        *j(ncol(par),1,sighat));
   estm=estm||(estm[,1] /estm[,2]);
   if ml=1 then print ,'Maximum Likelihood Estimation Results',;
   else if ml=-1 then print ,
      'Conditional Least Squares Estimation  Results',;
   else print ,'Unconditional Least Squares Estimation Results',;
```

```
      print estm [rowname=parmname colname=pname] ;
   finish armareg;
      /* End of ARMA Estimation modules */

      /* Begin estimation for Grunfeld's investment models */
   use grunfeld;
   read all var {gei} into y;
   read all var {gef gec} into x;
   x=j(nrow(x),1,1)||x;
   iphi=1;
   itheta=1;
   maxiter=10;
   delta=0.0005;
   ml=-1;
   /*----  To prevent overflow, specify starting values  ----*/
   par={-0.5  0.5  -9.956306  0.0265512  0.1516939};
   run armareg;   /*----  Perform CLS estimation  ----*/
```

The output is as follows.

```
                AR 1       MA 1       B 1        B 2        B 3

                  0          0 -9.956306 0.0265512 0.1516939


            Innovation Standard Deviation: 18.639064


                    Estimate Std. Error   T-Ratio

             AR 1 -0.071148  0.3248431 -0.219022
             MA 1 1.2737455  0.2319205 5.4921656
             B 1  -7.530983  20.447977   -0.3683
             B 2  0.0402554  0.0170277 2.3641054
             B 3  0.0992474  0.0354776 2.7974682
```

```
   ml=1;
   maxiter=10;
   /*----  With CLS estimates as starting values,  ----*/
   /*----   perform ML estimation.                 ----*/
   run armareg;
```

The output is as follows.

```
                AR 1      MA 1       B 1       B 2       B 3

       -0.071148 0.7850862 -7.530983 0.0402554 0.0992474


          Innovation Standard Deviation: 22.667286


               Estimate Std. Error   T-Ratio

        AR 1 -0.191675  0.3360688 -0.570345
        MA 1 0.7367182  0.2101849 3.5050966
        B 1  -19.45436  31.327362 -0.621002
        B 2   0.038099  0.0168731 2.2579666
        B 3   0.121766  0.0433174 2.8110191
```

Example 8.14. Iterative Proportional Fitting

The classical use of iterative proportional fitting is to adjust frequencies to conform to new marginal totals. Use the IPF subroutine to perform this kind of analysis. You supply a table that contains new margins and a table that contains old frequencies. The IPF subroutine returns a table of adjusted frequencies that preserves any higher order interactions appearing in the initial table.

The example is a census study that estimates a population distribution according to age and marital status (Bishop, Fienberg, and Holland 1975, pp. 97-98). Estimates of the distribution are known for the previous year, but only estimates of marginal totals are known for the current year. You want to adjust the distribution of the previous year to fit the estimated marginal totals of the current year.

```
proc iml;

   /* Stopping criteria */
mod={0.01 15};

   /* Marital status has 3 levels. age has 8 levels. */
dim={3 8};

   /* New marginal totals for age by marital status */
table={1412 0 0 ,
       1402 0 0 ,
       1174 276 0 ,
       0 1541 0 ,
       0 1681 0 ,
       0 1532 0 ,
       0 1662 0 ,
       0 5010 2634};

   /* Marginal totals are known for both */
   /* marital status and age             */
config={1 2};
```

Example 8.14. Iterative Proportional Fitting ◆ 177

```
    /* 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};

call ipf(fit,status,dim,table,config,initab,mod);

c={' SINGLE' ' MARRIED' 'WIDOWED/DIVORCED'};
r={'15 - 19' '20 - 24' '25 - 29' '30 - 34' '35 - 39' '40 - 44'
   '45 - 49' '50 OR OVER'};
print
   'POPULATION DISTRIBUTION ACCORDING TO AGE AND MARITAL STATUS',,
   'KNOWN DISTRIBUTION (PREVIOUS YEAR)',,
   initab [colname=c rowname=r format=8.0] ,,
   'ADJUSTED ESTIMATES OF DISTRIBUTION (CURRENT YEAR)',,
   fit [colname=c rowname=r format=8.2] ;
```

The output is as follows.

```
                          INITAB
                 SINGLE  MARRIED WIDOWED/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


                           FIT
                 SINGLE  MARRIED WIDOWED/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 8.15. Full-Screen Nonlinear Regression

This example shows how to build a menu system that enables you to perform nonlinear regression from a menu. Six modules are stored on an IML storage disk. After you have stored them, use this example to try out the system. First invoke IML and set up some sample data in memory, in this case the population of the U.S. from 1790 to 1970. Then invoke the module NLIN.

```
reset storage='nlin';
load module=_all_;
uspop = {3929, 5308, 7239, 9638, 12866, 17069, 23191, 31443,
         39818, 50155, 62947, 75994, 91972, 105710, 122775, 131669,
         151325, 179323, 203211}/1000;
year=do(1790,1970,10)';
time=year-1790;
print year time uspop;
run nlin;
```

A menu appears like the one below. The entry fields are shown by underscores here, but the underscores become blanks in the real session.

```
Non-Linear Regression
Response function: _____
Predictor function: _____

Parameter Value Derivative
: _____ _____ _____
: _____ _____ _____
: _____ _____ _____
: _____ _____ _____
: _____ _____ _____
: _____ _____ _____
```

Enter an exponential model and fill in the response and predictor expression fields. For each parameter, enter the name, initial value, and derivative of the predictor with respect to the parameter.

```
Non-Linear Regression
Response function: uspop_____
Predictor function: a0*exp(a1*time)_____

Parameter Value Derivative
: a0_____ _____3.9 exp(a1*time)_____
: a1_____ _____0 time*a0*exp(a1*time)_____
: _____ _____ _____
: _____ _____ _____
: _____ _____ _____
: _____ _____ _____
```

Now press the SUBMIT key. The model compiles, the iterations start blinking on the screen, and when the model has converged, the estimates are displayed along with their standard errors, t test, and significance probability.

Example 8.15. Full-Screen Nonlinear Regression ⬩ 179

To modify and rerun the model, submit the command

```
run nlrun;
```

Here is the code that defines and stores the modules of the system.

```
/*          Full Screen Nonlinear Regression           */
/* Six modules are defined, which constitute a system for    */
/* nonlinear regression. The interesting feature of this     */
/* system is that the problem is entered in a menu, and both */
/* iterations and final results are displayed on the same    */
/* menu.                                                      */
/*                                                            */
/* Run this source to get the modules stored. Examples       */
/* of use are separate.                                       */
/*                                                            */
/* Caution: this is a demonstration system only. It does not */
/* have all the necessary safeguards in it yet to            */
/* recover from user errors or rough models.                 */
/* Algorithm:                                                 */
/*    Gauss-Newton nonlinear regression with step-halving.   */
/* Notes: program variables all start with nd or _ to        */
/* minimize the problems that would occur if user variables  */
/* interfered with the program variables.                    */



/* Gauss-Newton nonlinear regression with Hartley step-halving */

/*---Routine to set up display values for new problem---*/
start nlinit;
   window nlin rows=15 columns=80 color='green'
      msgline=_msg cmndline=_cmnd
      group=title +30 'Non-Linear Regression' color='white'
      group=model / @5 'Response function:' color='white'
      +1 nddep $55. color='blue'
      / @5 'Predictor function:' color='white'
      +1 ndfun $55. color='blue'
      group=parm0 // @5 'Parameter' color='white' @15 'Value'
      @30 'Derivative'
      group=parm1 // @5 'Parameter' color='white' @15 'Value'
      group=parm2 // @5 'Parameter' color='white' @19 'Estimate'
      @33 'Std Error'
      @48 'T Ratio'
      @62 'Prob>|T|'
      group=parminit /@3 ':' color='white'
      @5 ndparm $8. color='blue'
      @15 ndbeta best12. @30 ndder $45.
      group=parmiter / @5 _parm color='white'
      @15 _beta best12. color='blue'
      group=parmest / @5 _parm color='white'
      @15 _beta best12. color='blue'
      @30 _std best12.
      @45 _t 10.4
      @60 _prob 10.4
      group=sse // @5 'Iteration =' color='white' _iter 5. color='blue'
      ' Stephalvings = ' color='white' _subit 3. color='blue'
      / @5 'Sum of Squares Error =' color='white' _sse best12.
```

```
        color='blue';
     nddep=cshape(' ',1,1,55,' ');
     ndfun=nddep;
     nd0=6;
     ndparm=repeat(' ',nd0,1);
     ndbeta=repeat(0,nd0,1);
     ndder=cshape(' ',nd0,1,55,' ');
     _msg='Enter New Nonlinear Problem';
  finish nlinit;  /* Finish module NLINIT */


     /* Main routine */
  start nlin;
     run nlinit;  /* initialization routine */
     run nlrun;    /* run routine */
  finish nlin;


     /* Routine to show each iteration */
  start nliter;
     display nlin.title noinput,
     nlin.model noinput,
     nlin.parm1 noinput,
     nlin.parmiter repeat noinput,
     nlin.sse noinput;
  finish nliter;


     /* Routine for one run */
  start nlrun;
     run nlgen; /* generate the model */
     run nlest; /* estimate the model */
  finish nlrun;


     /* Routine to generate the model */
  start nlgen;

     /* Model definition menu */
     display nlin.title, nlin.model, nlin.parm0, nlin.parminit repeat;

     /* Get number of parameters */
     t=loc(ndparm=' ');
     if nrow(t)=0 then
     do;
        print 'no parameters';
        stop;
     end;
     _k=t[1] -1;

        /* Trim extra rows, and edit '*' to '#' */
     _dep=nddep; call change(_dep,'*','#',0);
     _fun=ndfun; call change(_fun,'*','#',0);
     _parm=ndparm[1:_k,];
     _beta=ndbeta[1:_k,];
     _der=ndder [1:_k,];
     call change(_der,'*','#',0);
```

Example 8.15. Full-Screen Nonlinear Regression ◆ 181

```
   /* Construct nlresid module to split up parameters and */
   /* compute model                                        */
   call queue('start nlresid;');
   do i=1 to _k;
      call queue(_parm[i] ,"=_beta[",char(i,2),"] ;");
   end;
   call queue("_y = ",_dep,";",
              "_p = ",_fun,";",
              "_r = _y-_p;",
              "_sse = ssq(_r);",
              "finish;" );

      /* Construct nlderiv function */
   call queue('start nlderiv; _x = ');
   do i=1 to _k;
      call queue("(",_der[i] ,")#repeat(1,nobs,1)||");
   end;
   call queue(" nlnothin; finish;");

      /* Pause to compile the functions */
   call queue("resume;");
   pause *;
finish nlgen;  /* Finish module NLGEN   */

   /* Routine to do estimation */
start nlest;

   /*     Modified Gauss-Newton Nonlinear Regression     */
   /* _parm has parm names                               */
   /* _beta has initial values for parameters            */
   /* _k is the number of parameters                     */
   /* after nlresid:                                     */
   /* _y has response,                                   */
   /* _p has predictor after call                        */
   /* _r has residuals                                   */
   /* _sse has sse                                       */
   /* after nlderiv                                      */
   /* _x has jacobian                                    */
   /*                                                    */

   eps=1;
   _iter = 0;
   _subit = 0;
   _error = 0;
   run nlresid;        /* f, r, and sse for initial beta */
   run nliter;               /* print iteration zero */
   nobs = nrow(_y);
   _msg = 'Iterating';


      /* Gauss-Newton iterations */
   do _iter=1 to 30 while(eps>1e-8);
      run nlderiv;             /* subroutine for derivatives */
      _lastsse=_sse;
      _xpxi=sweep(_x`*_x);
      _delta=_xpxi*_x`*_r;              /* correction vector */
      _old = _beta;            /* save previous parameters */
      _beta=_beta+_delta;         /* apply the correction */
      run nlresid;                     /* compute residual */
```

```
      run nliter;                    /* print iteration in window */
      eps=abs((_lastsse-_sse))/(_sse+1e-6);
                                     /* convergence criterion */


        /* Hartley subiterations */
      do _subit=1 to 10 while(_sse>_lastsse);
        _delta=_delta*.5;     /* halve the correction vector */
        _beta=_old+_delta;    /* apply the halved correction */
        run nlresid;                       /* find sse et al */
        run nliter;           /* print subiteration in window */
      end;
      if _subit>10 then
      do;
        _msg = "did not improve after 10 halvings";
        eps=0; /* make it fall through iter loop */
      end;
    end;


      /* print out results  */
    _msg = ' ';
    if _iter>30 then
    do;
      _error=1;
      _msg = 'convergence failed';
    end;
    _iter=_iter-1;
    _dfe = nobs-_k;
    _mse = _sse/_dfe;
    _std = sqrt(vecdiag(_xpxi)#_mse);
    _t = _beta/_std;
    _prob= 1-probf(_t#_t,1,_dfe);
    display nlin.title noinput,
    nlin.model noinput,
    nlin.parm2 noinput,
    nlin.parmest repeat noinput,
    nlin.sse noinput;
  finish nlest;                          /* Finish module NLEST */

      /* Store the modules to run later */
   reset storage='nlin';
   store module=_all_;
```

Chapter 9
Robust Regression Examples

Chapter Contents

Chapter 9
Robust Regression Examples

Overview

SAS/IML has four subroutines that can be used for outlier detection and robust regression. The Least Median of Squares (LMS) and Least Trimmed Squares (LTS) subroutines perform *robust regression* (sometimes called *resistant regression*). These subroutines are able to detect outliers and perform a least-squares regression on the remaining observations. The Minimum Volume Ellipsoid Estimation (MVE) and Minimum Covariance Determinant Estimation (MCD) subroutines can be used to find a robust location and a robust covariance matrix that can be used for constructing confidence regions, detecting multivariate outliers and leverage points, and conducting robust canonical correlation and principle component analysis.

The LMS, LTS, MVE, and MCD methods were developed by Rousseeuw (1984) and Rousseeuw and Leroy (1987). All of these methods have the high breakdown value property. Roughly speaking, the breakdown value is a measure of the proportion of contamination that a procedure can withstand and still maintain its robustness. 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). The algorithm used in the LTS subroutine is based on the algorithm FAST-LTS of Rousseeuw and Van Driessen (2000). The MCD algorithm is based on the FAST-MCD algorithm given by Rousseeuw and Van Driessen (1999), which is similar to the FAST-LTS algorithm. The MVE algorithm is based on the algorithm used in the MINVOL program by Rousseeuw (1984). LTS estimation has higher statistical efficiency than LMS estimation. With the FAST-LTS algorithm, LTS is also faster than LMS for large data sets. Similarly, MCD is faster than MVE for large data sets.

Besides LTS estimation and LMS estimation, there are other methods for robust regression and outlier detection. You can refer to a comprehensive procedure, PROC ROBUSTREG, in SAS/STAT. A summary of these robust tools in SAS can be found in Chen (2002).

The four SAS/IML subroutines are designed for

- LMS: minimizing the hth ordered squared residual
- LTS: minimizing the sum of the h smallest squared residuals
- MCD: minimizing the determinant of the covariance of h points
- MVE: minimizing the volume of an ellipsoid containing h points

where h is defined in the range

$$\frac{N}{2} + 1 \ \leq \ h \ \leq \ \frac{3N}{4} + \frac{n+1}{4}$$

In the preceding equation, N is the number of observations and n is the number of regressors.[*] The value of h determines the *breakdown value*, which is "the smallest fraction of contamination that can cause the estimator T to take on values arbitrarily far from $T(Z)$" (Rousseeuw and Leroy 1987, p.10). Here, T denotes an estimator and $T(Z)$ applies T to a sample Z.

For each parameter vector $\mathbf{b} = (b_1, \ldots, b_n)$, the residual of observation i is $r_i = y_i - \mathbf{x}_i \mathbf{b}$. You then denote the ordered, squared residuals as

$$(r^2)_{1:N} \leq \ldots \leq (r^2)_{N:N}$$

The objective functions for the LMS, LTS, MCD, and MVE optimization problems are defined as follows:

- LMS, the objective function for the LMS optimization problem is the hth ordered squared residual,

$$F_{\text{LMS}} = (r^2)_{h:N} \longrightarrow \min$$

  Note that, for $h = N/2 + 1$, the hth quantile is the median of the squared residuals. The default h in PROGRESS is $h = \left[\frac{N+n+1}{2}\right]$, which yields the breakdown value (where $[k]$ denotes the integer part of k).

- LTS, the objective function for the LTS optimization problem is the sum of the h smallest ordered squared residuals,

$$F_{\text{LTS}} = \sqrt{\frac{1}{h} \sum_{i=1}^{h} (r^2)_{i:N}} \longrightarrow \min$$

- MCD, the objective function for the MCD optimization problem is based on the determinant of the covariance of the selected h points.

$$F_{\text{MCD}} = \det(\mathbf{C}_h) \longrightarrow \min$$

  where $\mathbf{C}_h$ is the covariance matrix of the selected h points.

- MVE, the objective function for the MVE optimization problem is based on the hth quantile $d_{h:N}$ of the Mahalanobis-type distances $\mathbf{d} = (d_1, \ldots, d_N)$,

$$F_{\text{MVE}} = \sqrt{d_{h:N}\det(\mathbf{C})} \longrightarrow \min$$

  subject to $d_{h:N} = \sqrt{\chi^2_{n,0.5}}$, where $\mathbf{C}$ is the scatter matrix estimate, and the Mahalanobis-type distances are computed as

$$\mathbf{d} = \text{diag}(\sqrt{(\mathbf{X} - T)^T \mathbf{C}^{-1}(\mathbf{X} - T)})$$

  where T is the location estimate.

---

[*]The value of h can be specified (see the "Syntax" section), but in most applications the default value works just fine and the results seem to be quite stable with different choices of h.

Because of the nonsmooth form of these objective functions, the estimates cannot be obtained with traditional optimization algorithms. For LMS and LTS, the algorithm, as in the PROGRESS program, selects a number of subsets of n observations out of the N given observations, evaluates the objective function, and saves the subset with the lowest objective function. As long as the problem size enables you to evaluate all such subsets, the result is a global optimum. If computing time does not permit you to evaluate all the different subsets, a random collection of subsets is evaluated. In such a case, you may not obtain the global optimum.

Note that the LMS, LTS, MCD, and MVE subroutines are executed only when the number N of observations is over twice the number n of explanatory variables x_j (including the intercept), that is, if $N > 2n$.

Flow Chart for LMS, LTS, MCD, and MVE
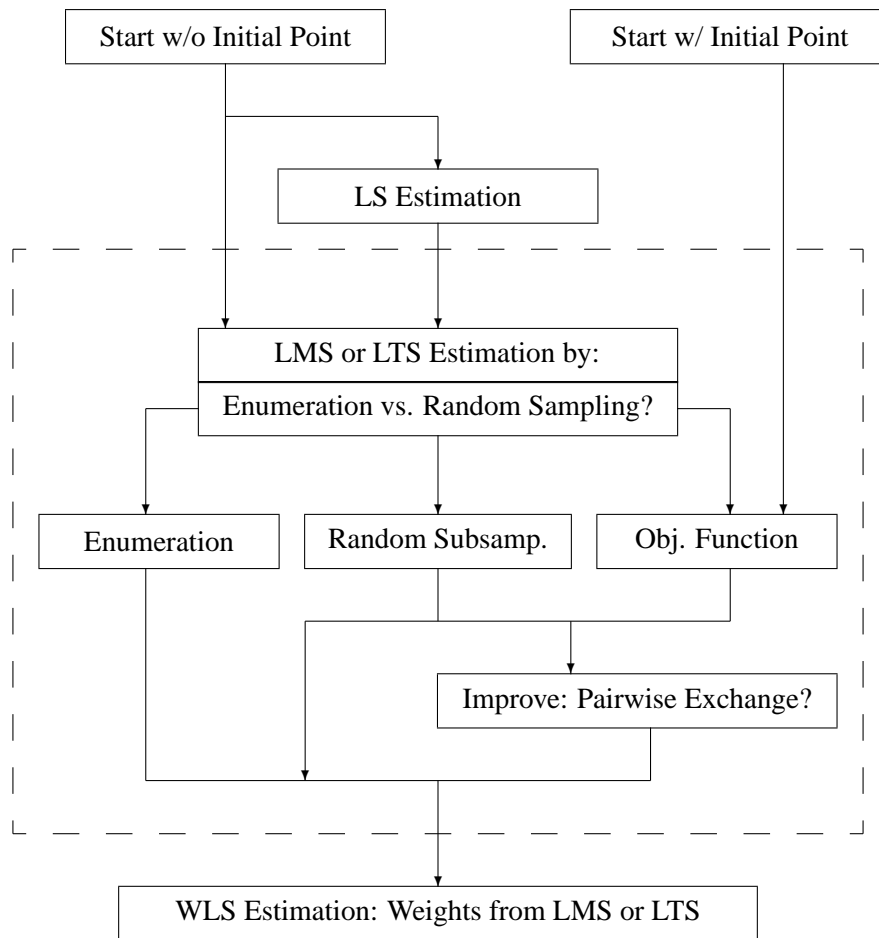
Flow Chart Indicating: LS $\longrightarrow$ [LMS or LTS] $\longrightarrow$ WLS

Separate LMS or LTS Part Inside Dashbox Corresponds to MCD, MVE

Using LMS and LTS

Because of space considerations, the output of the tables containing residuals and resistant diagnostics are not included in this document. The subroutines *prilmts*, *scatlmts*, and *lmsdiap* are used in these examples for printing and plotting the results. These routines are in the **robustmc.sas** file that is contained in the sample library.

Example 9.1. Substantial Leverage Points ⬩ 189

Example 9.1. LMS and LTS with Substantial Leverage Points: Hertzsprung-Russell Star Data

The following data are reported in Rousseeuw and Leroy (1987, p. 27) and are based on Humphreys (1978) and Vansina and De Greve (1982). The 47 observations correspond to the 47 stars of the CYG OB1 cluster in the direction of Cygnus. The regressor variable (column 2) x is the logarithm of the effective temperature at the surface of the star (T_e), and the response variable (column 3) y is the logarithm of its light intensity (L/L_0). The results for LS and LMS on page 28 of Rousseeuw and Leroy (1987) are based on a more precise (five decimal places) version of the data set. This data set is remarkable in that it contains four substantial leverage points (giant stars) corresponding to observations 11, 20, 30, and 34 that greatly affect the results of L_2 and even L_1 regression.

```
ab =  { 1   4.37  5.23,   2   4.56  5.74,   3   4.26  4.93,
        4   4.56  5.74,   5   4.30  5.19,   6   4.46  5.46,
        7   3.84  4.65,   8   4.57  5.27,   9   4.26  5.57,
       10   4.37  5.12,  11   3.49  5.73,  12   4.43  5.45,
       13   4.48  5.42,  14   4.01  4.05,  15   4.29  4.26,
       16   4.42  4.58,  17   4.23  3.94,  18   4.42  4.18,
       19   4.23  4.18,  20   3.49  5.89,  21   4.29  4.38,
       22   4.29  4.22,  23   4.42  4.42,  24   4.49  4.85,
       25   4.38  5.02,  26   4.42  4.66,  27   4.29  4.66,
       28   4.38  4.90,  29   4.22  4.39,  30   3.48  6.05,
       31   4.38  4.42,  32   4.56  5.10,  33   4.45  5.22,
       34   3.49  6.29,  35   4.23  4.34,  36   4.62  5.62,
       37   4.53  5.10,  38   4.45  5.22,  39   4.53  5.18,
       40   4.43  5.57,  41   4.38  4.62,  42   4.45  5.06,
       43   4.50  5.34,  44   4.45  5.34,  45   4.55  5.54,
       46   4.45  4.98,  47   4.42  4.50 } ;

a = ab[,2]; b = ab[,3];
```

The following code specifies that most of the output be printed.

```
print "*** Hertzsprung-Russell Star Data: Do LMS ***";
   optn = j(9,1,.);
   optn[2]= 3;    /* ipri */
   optn[3]= 3;    /* ilsq */
   optn[8]= 3;    /* icov */
call lms(sc,coef,wgt,optn,b,a);
```

Some simple statistics for the independent and response variables are shown in Output 9.1.1.

Output 9.1.1. Some Simple Statistics

```
                    Median and Mean


                            Median              Mean
        VAR1            4.420000000       4.310000000
        Intercep        1.000000000       1.000000000
        Response        5.100000000       5.012127660



            Dispersion and Standard Deviation


                        Dispersion            StdDev
        VAR1            0.1630862440      0.2908234187
        Intercep        0.0000000000      0.0000000000
        Response        0.6671709983      0.5712493409
```

Partial output for LS regression is shown in Output 9.1.2.

Output 9.1.2. Table of Unweighted LS Regression

```
                **********************************
                Unweighted Least-Squares Estimation
                **********************************


                    LS Parameter Estimates
----------------------------------------------------------------
                    Approx                      Lower     Upper
        Estimate  Std Error  T Value   Prob    Wald CI   Wald CI
----------------------------------------------------------------
VAR1      -0.4133    0.28626  -1.4438  0.156   -0.9744    0.1478
Intercep   6.7935    1.23652   5.4940 175E-8    4.3699    9.2170


            Sum of Squares = 14.346394626
               Degrees of Freedom = 45
            LS Scale Estimate = 0.5646315343


            COV Matrix of Parameter Estimates


                            VAR1            Intercep
        VAR1            0.081943343      -0.353175807
        Intercep       -0.353175807       1.528970895


               R-squared = 0.0442737441
           F(1,45) Statistic = 2.0846120667
               Probability = 0.1557164396
```

Output 9.1.3 displays the iteration history. Looking at the column *Best Crit* in the iteration history table, you see that, with complete enumeration, the optimal solution is quickly found.

Example 9.1. Substantial Leverage Points ⬥ 191

Output 9.1.3. History of the Iteration Process

```
*************************************************
***        Complete Enumeration for LMS        ***
*************************************************


        Subset  Singular        Best Crit    Pct
          271         5 0.39279108982007    25%
          541         8 0.39279108982007    50%
          811        27 0.39279108982007    75%
         1081        45 0.39279108982007   100%
              Minimum Criterion=0.3927910898


        ****************************************
        Least Median of Squares (LMS) Regression
        ****************************************


     Minimizing the 25th Ordered Squared Residual.
      Highest Possible Breakdown Value =  48.94 %
  Selection of All 1081 Subsets of 2 Cases Out of 47
          Among 1081 subsets 45 are singular.
```

The results of the optimization for LMS estimation are displayed in Output 9.1.4.

Output 9.1.4. Results of Optimization

```
                Observations of Best Subset

                     2              29


              Estimated Coefficients

                     VAR1            Intercep
                3.97058824        -12.62794118

      LMS Objective Function = 0.2620588235
       Preliminary LMS Scale = 0.3987301586
          Robust R Squared = 0.5813148789
    Final LMS Scale Estimate = 0.3645644492
```

Output 9.1.5 displays the results for WLS regression. Due to the size of the scaled residuals, six observations (with numbers 7, 9, 11, 20, 30, 34) were assigned zero weights in the following WLS analysis.

The LTS regression implements the FAST-LTS algorithm, which improves the algorithm (used in SAS/IML Version 7 and before, denoted as V7 LTS in this chapter) in Rousseeuw and Leroy (1987) using techniques called "selective iteration" and "nested extensions." These techniques are used in the C-steps of the algorithm. See Rousseeuw and Van Driessen (2000) for details. The FAST-LTS algorithm significantly improves the speed of computation.

Output 9.1.5. Table of Weighted LS Regression Based on LMS

```
              *******************************
                 Weighted Least-Squares Estimation
              *******************************


                RLS Parameter Estimates Based on LMS
     -----------------------------------------------------------------
                       Approx                     Lower     Upper
             Estimate  Std Error  T Value   Prob   Wald CI   Wald CI
     -----------------------------------------------------------------
     VAR1       3.0462   0.43734   6.9652  24E-9    2.1890    3.9033
     Intercep  -8.5001   1.92631  -4.4126 0.0001  -12.2755   -4.7246


                  Weighted Sum of Squares = 4.52819451
                      Degrees of Freedom = 39
                  RLS Scale Estimate = 0.3407455818



                  COV Matrix of Parameter Estimates


                                 VAR1             Intercep
             VAR1            0.191265604        -0.842128459
             Intercep       -0.842128459         3.710661875


               Weighted R-squared = 0.5543573521
                F(1,39) Statistic = 48.514065776
                     Probability = 2.3923178E-8
             There are 41 points with nonzero weight.
                  Average Weight = 0.8723404255
```

The following code implements the LTS regression on the Hertzsprung-Russell Star Data.

```
print "*** Hertzsprung-Russell Star Data: Do LTS ***";
   optn = j(9,1,.);
   optn[2]= 3;     /* ipri */
   optn[3]= 3;     /* ilsq */
   optn[8]= 3;     /* icov */
call lts(sc,coef,wgt,optn,b,a);
```

Output 9.1.6 summarizes the information for the LTS optimization.

Output 9.1.6. Summary of Optimization

```
                    Least Trimmed Squares (LTS) Method

            Minimizing Sum of 25 Smallest Squared Residuals.
              Highest Possible Breakdown Value =  48.94 %
            Selection of All 1081 Subsets of 2 Cases Out of 47
                  Among 1081 subsets 45 is/are singular.


The best half of the entire data set obtained after full iteration consists
                          of the cases:


  2     4     6    10    13    15    17    19    21    22    25    27    28

     29    33    35    36    38    39    41    42    43    44    45    46
```

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS ♦ 193

Output 9.1.7 displays the optimization results and Output 9.1.8 displays the weighted LS regression based on LTS estimates.

Output 9.1.7. Results of Optimization

```
                    Estimated Coefficients

                  VAR1            Intercep

             4.219182102        -13.6239903


          LTS Objective Function = 0.1829636959


         Preliminary LTS Scale = 0.4524915298


           Robust R Squared = 0.4210129994


            Final LTS Scale = 0.3731970408
```

Output 9.1.8. Table of Weighted LS Regression Based on LTS

```
              *******************************
              Weighted Least-Squares Estimation
              *******************************

              RLS Parameter Estimates Based on LTS

                          Approx          Pr >
Variable      Estimate    Std Err  t Value    |t|    Lower WCI    Upper WCI

VAR1         3.04615694  0.43733923    6.97  <.0001  2.18898779  3.90332608
Intercep     -8.5000549  1.92630783   -4.41  <.0001  -12.275549  -4.7245609

              Weighted Sum of Squares = 4.52819451

                  Degrees of Freedom = 39
              RLS Scale Estimate = 0.3407455818

              Cov Matrix of Parameter Estimates

                          VAR1            Intercep

          VAR1          0.1912656038      -0.842128459
          Intercep      -0.842128459      3.7106618752

            Weighted R-squared = 0.5543573521
            F(1,39) Statistic = 48.514065776
                Probability = 2.3923178E-8
          There are 41 points with nonzero weight.
                Average Weight = 0.8723404255
```

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS

This example presents comparisons of LMS, V7 LTS, and FAST-LTS. We use the stackloss data of Brownlee (1965), which is also used for documenting the L1 regression module. The three explanatory variables correspond to measurements for a plant oxidizing ammonia to nitric acid on 21 consecutive days:

- x_1 air flow to the plant

- x_2 cooling water inlet temperature

- x_3 acid concentration

The response variable y_i gives the permillage of ammonia lost (stackloss). These data are also given in Rousseeuw and Leroy (1987, p. 76) and Osborne (1985, p. 267):

```
print "Stackloss Data";
   aa = { 1   80   27   89   42,
          1   80   27   88   37,
          1   75   25   90   37,
          1   62   24   87   28,
          1   62   22   87   18,
          1   62   23   87   18,
          1   62   24   93   19,
          1   62   24   93   20,
          1   58   23   87   15,
          1   58   18   80   14,
          1   58   18   89   14,
          1   58   17   88   13,
          1   58   18   82   11,
          1   58   19   93   12,
          1   50   18   89    8,
          1   50   18   86    7,
          1   50   19   72    8,
          1   50   19   79    8,
          1   50   20   80    9,
          1   56   20   82   15,
          1   70   20   91   15 };
```

Rousseeuw and Leroy (1987, p. 76) cite a large number of papers in which this data set was analyzed before. They 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.

Consider 2,000 Random Subsets for LMS

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5985 different subsets of 4 observations out of 21. If you do not specify *optn*[5], the LMS algorithms draw $N_{rep} = 2000$ random sample subsets. Since there is a large number of subsets with singular linear systems that you do not want to print, you can choose *optn*[2]=2 for reduced printed output.

```
title2 "***Use 2000 Random Subsets for LMS***";
     a = aa[,2:4]; b = aa[,5];
     optn = j(9,1,.);
     optn[2]= 2;    /* ipri */
     optn[3]= 3;    /* ilsq */
     optn[8]= 3;    /* icov */

call lms(sc,coef,wgt,optn,b,a);
```

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS ◆ 195

Summary statistics are shown in Output 9.2.1.

Output 9.2.1. Some Simple Statistics

```
                        Median and Mean

                              Median                Mean
           VAR1           58.00000000         60.42857143
           VAR2           20.00000000         21.09523810
           VAR3           87.00000000         86.28571429
           Intercep        1.00000000          1.00000000
           Response       15.00000000         17.52380952


                Dispersion and Standard Deviation

                            Dispersion              StdDev
           VAR1            5.93040887          9.16826826
           VAR2            2.96520444          3.16077145
           VAR3            4.44780666          5.35857124
           Intercep       0.00000000          0.00000000
           Response       5.93040887         10.17162252
```

Output 9.2.2 displays the results of LS regression.

Output 9.2.2. Table of Unweighted LS Regression

```
          **********************************
          Unweighted Least-Squares Estimation
          **********************************


                  LS Parameter Estimates
-----------------------------------------------------------------
                  Approx                      Lower     Upper
        Estimate  Std Error  T Value   Prob   Wald CI   Wald CI
-----------------------------------------------------------------
VAR1      0.7156    0.13486   5.3066  58E-6    0.4513    0.9800
VAR2      1.2953    0.36802   3.5196  0.0026   0.5740    2.0166
VAR3     -0.1521    0.15629  -0.9733  0.344   -0.4585    0.1542
Intercep -39.9197  11.89600  -3.3557 0.0038  -63.2354  -16.6039


                Sum of Squares = 178.8299616
                 Degrees of Freedom = 17
              LS Scale Estimate = 3.2433639182



                COV Matrix of Parameter Estimates

                  VAR1           VAR2           VAR3        Intercep
VAR1         0.0181867     -0.0365107     -0.0071435      0.2875871
VAR2        -0.0365107      0.1354419      0.0000105     -0.6517944
VAR3        -0.0071435      0.0000105      0.0244278     -1.6763208
Intercep     0.2875871     -0.6517944     -1.6763208    141.5147411


                 R-squared = 0.9135769045
               F(3,17) Statistic = 59.9022259
                  Probability = 3.0163272E-9
```

Output 9.2.3 displays the LMS results for the 2000 random subsets.

Output 9.2.3. Iteration History and Optimization Results

```
*********************************************
***         Random Subsampling for LMS          ***
*********************************************

       Subset Singular        Best Crit    Pct
          500         23  0.1632616086096    25%
         1000         55 0.14051869795752    50%
         1500         79 0.14051869795752    75%
         2000        103 0.12646682816177   100%
         Minimum Criterion=0.1264668282


      ****************************************
      Least Median of Squares (LMS) Regression
      ****************************************


   Minimizing the 13th Ordered Squared Residual.
   Highest Possible Breakdown Value =   42.86 %
         Random Selection of 2103 Subsets
        Among 2103 subsets 103 are singular.



              Observations of Best Subset

          15              11              19              10


              Estimated Coefficients

      VAR1              VAR2              VAR3         Intercep
  0.75000000        0.50000000        0.00000000    -39.25000000

          LMS Objective Function = 0.75
        Preliminary LMS Scale = 1.0478510755
           Robust R Squared = 0.96484375
        Final LMS Scale Estimate = 1.2076147288
```

For LMS, observations 1, 3, 4, and 21 have scaled residuals larger than 2.5 (output not shown), and they are considered outliers. Output 9.2.4 displays the corresponding WLS results.

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS ◆ 197

Output 9.2.4. Table of Weighted LS Regression

```
*******************************
  Weighted Least-Squares Estimation
*******************************

     RLS Parameter Estimates Based on LMS
-----------------------------------------------------------------
                  Approx                    Lower     Upper
        Estimate  Std Error  T Value   Prob  Wald CI   Wald CI
-----------------------------------------------------------------
VAR1       0.7977   0.06744  11.8282   25E-9   0.6655    0.9299
VAR2       0.5773   0.16597   3.4786  0.0041   0.2520    0.9026
VAR3      -0.0671   0.06160  -1.0886   0.296  -0.1878    0.0537
Intercep -37.6525   4.73205  -7.9569  237E-8 -46.9271  -28.3778

         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     0.00454803    -0.00792141    -0.00119869     0.00156817
VAR2    -0.00792141     0.02754569    -0.00046339    -0.06501751
VAR3    -0.00119869    -0.00046339     0.00379495    -0.24610225
Intercep 0.00156817    -0.06501751    -0.24610225    22.39230535

           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
```

The subroutine, *prilmts()*, which is in the **robustmc.sas** file that is contained in the sample library, can be called to print the output summary:

```
call prilmts(3,sc,coef,wgt);
```

Output 9.2.5, Output 9.2.6, and Output 9.2.7 are the three parts of the output.

Output 9.2.5. First Part of Output Generated by prilmts()

```
          Results of Least Median Squares Estimation

     Quantile. . . . . . . . . . .        13
     Number of Subsets. . . . . .       2103
     Number of Singular Subsets .        103
     Number of Nonzero Weights. .         17
     Objective Function. . . . . .       0.75
     Preliminary Scale Estimate. . 1.0478511
     Final Scale Estimate. . . . . 1.2076147
     Robust R Squared. . . . . . . 0.9648438
     Asymptotic Consistency Factor 1.1413664
      RLS Scale Estimate. . . . .  1.252714
      Weighted Sum of Squares . .   20.4008
      Weighted R-squared. . . . . 0.9750062
      F Statistic . . . . . . . . 169.04318
```

Output 9.2.6. Second Part of Output Generated by prilmts()

```
                  Estimated LMS Coefficients
             0.75        0.5          0    -39.25

                   Indices of Best Sample
                  15          11         19          10

                  Estimated WLS Coefficients
          0.7976856 0.5773405   -0.06706 -37.65246

                       Standard Errors
          0.0674391 0.1659689 0.0616031 4.7320509

                          T Values
          11.828242 3.4786054 -1.088584 -7.956901

                        Probabilities
          2.4838E-8  0.004078 0.2961071 2.3723E-6

                       Lower Wald CI
          0.6655074 0.2520473   -0.1878 -46.92711

                       Upper Wald CI
          0.9298637 0.9026336 0.0536798 -28.37781
```

Output 9.2.7. Third Part of Output Generated by prilmts()

```
                       LMS Residuals

    6.4176097 2.2772163   6.21059 7.2456884  -0.20702 -0.621059
  :  -0.20702  0.621059 -0.621059  0.621059  0.621059 0.2070197
  : -1.863177 -1.449138  0.621059  -0.20702 0.2070197 0.2070197
  :  0.621059  1.863177 -6.831649

                        Diagnostics
    10.448052 7.9317507        10 11.666667 2.7297297 3.4864865
  : 4.7297297 4.2432432 3.6486486 3.7598351 4.6057675 4.9251688
  : 3.8888889 4.5864209 5.2970297  4.009901  6.679576 4.3053404
  : 4.0199755         3        11

                       WLS Residuals
    4.9634454 0.9185794 5.1312163 6.5250478 -0.535877 -0.996749
  : -0.338162 0.4601047 -0.844485  0.286883 0.7686702 0.3777432
  : -2.000854 -1.074607 1.0731966 0.1143341 -0.297718 0.0770058
  : 0.4679328  1.544002 -6.888934
```

Consider 2,000 Random Subsets for V7 LTS

The V7 LTS algorithm is similar to the LMS algorithm.

```
  title2 "***Use 2000 Random Subsets for LTS***";
    a = aa[,2:4]; b = aa[,5];
    optn = j(9,1,.);
    optn[2]= 2;    /* ipri */
    optn[3]= 3;    /* ilsq */
    optn[8]= 3;    /* icov */
    optn[9]= 1;    /* V7 LTS */

  call lts(sc,coef,wgt,optn,b,a);
```

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS ✦ 199

Output 9.2.8 displays the iteration history and optimization results of V7 LTS.

Output 9.2.8. Iteration History and Optimization Results

```
************************************************
***      Random Subsampling for V7 LTS       ***
************************************************


           Subset Singular        Best Crit   Pct
              500        23 0.09950690229748   25%
             1000        55 0.08781379221356   50%
             1500        79 0.08406140720682   75%
             2000       103 0.08406140720682  100%
            Minimum Criterion=0.0840614072


        ****************************************
        Least Trimmed Squares (V7 LTS) Regression
        ****************************************


     Minimizing Sum of 13 Smallest Squared Residuals.
      Highest Possible Breakdown Value =  42.86 %
             Random Selection of 2103 Subsets
           Among 2103 subsets 103 are singular.


              Observations of Best Subset

            10              11              7              15


               Estimated Coefficients
          VAR1              VAR2             VAR3           Intercep

       0.75000000       0.33333333       0.00000000    -35.70512821


          LTS Objective Function = 0.4985185153
           Preliminary LTS Scale = 1.0379336739
              Robust R Squared = 0.9719626168
        Final LTS Scale Estimate = 1.0407755737
```

In addition to observations 1, 3, 4, and 21, which were considered outliers in LMS, observations 2 and 13 for LTS have absolute scaled residuals that are larger (but not as significantly as observations 1, 3, 4, and 21) than 2.5 (output not shown). Therefore, the WLS results based on LTS are different from those based on LMS.

Output 9.2.9 displays the results for the weighted LS regression.

Output 9.2.9. Table of Weighted LS Regression

```
                    ********************************
                    Weighted Least-Squares Estimation
                    ********************************


                  RLS Parameter Estimates Based on LTS
         -----------------------------------------------------------------
                           Approx                        Lower     Upper
                Estimate  Std Error  T Value    Prob    Wald CI   Wald CI
         -----------------------------------------------------------------
         VAR1       0.7569    0.07861    9.6293 108E-8     0.6029    0.9110
         VAR2       0.4535    0.13605    3.3335 0.0067     0.1869    0.7202
         VAR3      -0.0521    0.05464   -0.9537  0.361    -0.1592    0.0550
         Intercep -34.0575    3.82882   -8.8950 235E-8   -41.5619  -26.5532


                    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       0.00617916     -0.00577686     -0.00230059     -0.03429007
         VAR2      -0.00577686      0.01850969      0.00025825     -0.06974088
         VAR3      -0.00230059      0.00025825      0.00298523     -0.13148741
         Intercep  -0.03429007     -0.06974088     -0.13148741     14.65985290


                    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
```

Consider 500 Random Subsets for FAST-LTS

The FAST-LTS algorithm uses only 500 random subsets and gets better optimization results.

```
title2 "***Use 500 Random Subsets for FAST-LTS***";
   a = aa[,2:4]; b = aa[,5];
   optn = j(9,1,.);
   optn[2]= 2;    /* ipri */
   optn[3]= 3;    /* ilsq */
   optn[8]= 3;    /* icov */
   optn[9]= 0;    /* FAST-LTS */


call lts(sc,coef,wgt,optn,b,a);
```

For this example, the two LTS algorithms identify the same outliers; however, the FAST-LTS algorithm uses only 500 random subsets and gets a smaller objective function, as seen in Output 9.2.10. For large data sets, the advantages of the FAST-LTS algorithm are more obvious.

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS ◆ 201

Output 9.2.10.   Optimization Results for FAST-LTS

```
*******************************************
        Least Trimmed Squares (FAST-LTS) Regression
*******************************************

        Minimizing Sum of 13 Smallest Squared Residuals.
          Highest Possible Breakdown Value =  42.86 %
                 Random Selection of 523 Subsets
                Among 523 subsets 23 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
```

Consider All 5,985 Subsets

You now report the results of LMS for all different subsets:

```
title2 "*** Use All 5985 Subsets***";
   a = aa[,2:4]; b = aa[,5];
   optn = j(9,1,.);
   optn[2]= 2;    /* ipri */
   optn[3]= 3;    /* ilsq */
   optn[5]= -1;   /* nrep: all 5985 subsets */
   optn[8]= 3;    /* icov */

call lms(sc,coef,wgt,optn,b,a);
```

Output 9.2.11 displays the results for LMS.

Output 9.2.11. Iteration History and Optimization Results for LMS

```
*********************************************
***      Complete Enumeration for LMS       ***
*********************************************

         Subset Singular        Best Crit   Pct
           1497        36 0.18589932664216   25%
           2993        87 0.15826842822584   50%
           4489       149 0.14051869795752   75%
           5985       266 0.12646682816177  100%
         Minimum Criterion=0.1264668282


    ****************************************
    Least Median of Squares (LMS) Regression
    ****************************************

   Minimizing the 13th Ordered Squared Residual.
    Highest Possible Breakdown Value =  42.86 %
  Selection of All 5985 Subsets of 4 Cases Out of 21
        Among 5985 subsets 266 are singular.

                Observations of Best Subset

            8             10             15             19

              Estimated Coefficients

      VAR1            VAR2            VAR3          Intercep
  0.75000000      0.50000000      0.00000000    -39.25000000

           LMS Objective Function = 0.75
         Preliminary LMS Scale = 1.0478510755
            Robust R Squared = 0.96484375
        Final LMS Scale Estimate = 1.2076147288
```

Next, report the results of LTS for all different subsets:

```
title2 "*** Use All 5985 Subsets***";
   a = aa[,2:4]; b = aa[,5];
   optn = j(9,1,.);
   optn[2]= 2;    /* ipri */
   optn[3]= 3;    /* ilsq */
   optn[5]= -1;   /* nrep: all 5985 subsets */
   optn[8]= 3;    /* icov */
   optn[9]= 1;    /* V7 LTS */

call lts(sc,coef,wgt,optn,b,a);
```

Output 9.2.12 displays the results for LTS.

Example 9.2. Comparison of LMS, V7 LTS, and FAST-LTS ◆ 203

Output 9.2.12. Iteration History and Optimization Results for LTS

```
************************************************
***      Complete Enumeration for LTS      ***
************************************************

      Subset Singular        Best Crit   Pct
       1497        36 0.13544860556893    25%
       2993        87 0.10708384510403    50%
       4489       149 0.08153552986986    75%
       5985       266 0.08153552986986   100%
       Minimum Criterion=0.0815355299


      **************************************
      Least Trimmed Squares (LTS) Regression
      **************************************


    Minimizing Sum of 13 Smallest Squared Residuals.
     Highest Possible Breakdown Value =  42.86 %
   Selection of All 5985 Subsets of 4 Cases Out of 21
        Among 5985 subsets 266 are singular.


             Observations of Best Subset

          5            12           17           18

              Estimated Coefficients

        VAR1           VAR2          VAR3        Intercep
   0.72916667     0.41666667    0.00000000   -36.22115385


        LTS Objective Function = 0.4835390299
         Preliminary LTS Scale = 1.0067458407
           Robust R Squared = 0.9736222371
         Final LTS Scale Estimate = 1.009470149
```

Example 9.3. LMS and LTS Univariate (Location) Problem: Barnett and Lewis Data

If you do not specify matrix X of the last input argument, the regression problem is reduced to the estimation problem of the location parameter a. The following example is described in Rousseeuw and Leroy (1987, p. 175):

```
print "*** Barnett and Lewis (1978) ***";
   b = { 3, 4, 7, 8, 10, 949, 951 };

   optn = j(9,1,.);
   optn[2]= 3;     /* ipri */
   optn[3]= 3;     /* ilsq */
   optn[8]= 3;     /* icov */

call lms(sc,coef,wgt,optn,b);
```

Output 9.3.1 shows the results of the unweighted LS regression.

Output 9.3.1. Table of Unweighted LS Regression

```
            Robust Estimation of Location and Scale
            *************************************


               ***********************************
               Unweighted Least-Squares Estimation
               ***********************************


        Median = 8   MAD ( * 1.4826) = 5.930408874
        Mean = 276 Standard Deviation = 460.43602523


                        LS Residuals
        -------------------------------------------------
                  Observed        Residual         Res / S
        -------------------------------------------------
            1       3.000000     -273.000000      -0.592916
            2       4.000000     -272.000000      -0.590744
            3       7.000000     -269.000000      -0.584229
            4       8.000000     -268.000000      -0.582057
            5      10.000000     -266.000000      -0.577713
            6     949.000000      673.000000       1.461658
            7     951.000000      675.000000       1.466002


                     Distribution of Residuals
        MinRes    1st Qu.    Median      Mean    3rd Qu.    MaxRes
         -273       -272      -268         0       -266       675
```

Output 9.3.2 shows the results for LMS regression.

Example 9.3. Univariate (Location) Problem ◆ 205

Output 9.3.2. Table of LMS Results

```
          **********************************
             Least Median of Squares (LMS) Method
          **********************************


             Minimizing 4th Ordered Squared Residual.
          Highest Possible Breakdown Value =  57.14 %
                LMS Objective Function = 2.5
                    LMS Location = 5.5
             Preliminary LMS Scale = 5.4137257125
                Final LMS Scale = 3.0516389039


                     LMS Residuals
          ------------------------------------------------
                 Observed       Residual        Res / S
          ------------------------------------------------
            1      3.000000      -2.500000      -0.819232
            2      4.000000      -1.500000      -0.491539
            3      7.000000       1.500000       0.491539
            4      8.000000       2.500000       0.819232
            5     10.000000       4.500000       1.474617
            6    949.000000     943.500000     309.178127
            7    951.000000     945.500000     309.833512


                    Distribution of Residuals
        MinRes    1st Qu.    Median      Mean    3rd Qu.    MaxRes
         -2.5       -1.5       2.5      270.5       4.5      945.5
```

You obtain the LMS location estimate 6.5 compared with the mean 276 (which is the LS estimate of the location parameter) and the median 8. The scale estimate in the univariate problem is a resistant (high breakdown) estimator for the dispersion of the data (refer to Rousseeuw and Leroy 1987, p. 178).

For weighted LS regression, the last two observations are ignored (that is, given zero weights), as shown in Output 9.3.3.

Output 9.3.3. Table of Weighted LS Regression

```
          ******************************
             Weighted Least-Squares Estimation
          ******************************


                    Weighted Mean = 6.4
          Weighted Standard Deviation = 2.8809720582
           There are 5 points with nonzero weight.
                Average Weight = 0.7142857143


                   Weighted LS Residuals
       ------------------------------------------------------------------
              Observed       Residual       Res / S          Weight
       ------------------------------------------------------------------
         1      3.000000      -3.400000      -1.180157       1.000000
         2      4.000000      -2.400000      -0.833052       1.000000
         3      7.000000       0.600000       0.208263       1.000000
         4      8.000000       1.600000       0.555368       1.000000
         5     10.000000       3.600000       1.249578       1.000000
         6    949.000000     942.600000     327.181236              0
         7    951.000000     944.600000     327.875447              0


                    Distribution of Residuals
        MinRes    1st Qu.    Median      Mean    3rd Qu.    MaxRes
         -3.4       -2.4       1.6      269.6       3.6      944.6
```

```
        optn = j(9,1,.);
        optn[2]= 3;    /* ipri */
        optn[3]= 3;    /* ilsq */
        optn[8]= 3;    /* icov */

    call lts(sc,coef,wgt,optn,b);
```

The results for LTS are similar to those reported for LMS in Rousseeuw and Leroy (1987), as shown in Output 9.3.4.

Output 9.3.4. Table of LTS Results

```
            *********************************
                Least Trimmed Squares (LTS) Method
                *********************************

        Minimizing Sum of 4 Smallest Squared Residuals.
          Highest Possible Breakdown Value =  57.14 %
             LTS Objective Function = 2.0615528128
                      LTS Location = 5.5
             Preliminary LTS Scale = 4.7050421234
                 Final LTS Scale = 3.0516389039

                        LTS Residuals
        -------------------------------------------------
                 Observed        Residual         Res / S
        -------------------------------------------------
            1       3.000000      -2.500000      -0.819232
            2       4.000000      -1.500000      -0.491539
            3       7.000000       1.500000       0.491539
            4       8.000000       2.500000       0.819232
            5      10.000000       4.500000       1.474617
            6     949.000000     943.500000     309.178127
            7     951.000000     945.500000     309.833512

                    Distribution of Residuals
        MinRes     1st Qu.     Median       Mean     3rd Qu.      MaxRes
        -2.5          -1.5        2.5      270.5         4.5       945.5
```

Since nonzero weights are chosen for the same observations as with LMS, the WLS results based on LTS agree with those based on LMS (shown previously in Output 9.3.3).

In summary, you obtain the following estimates for the location parameter:

- LS estimate (unweighted mean) = 276
- Median = 8
- LMS estimate = 5.5
- LTS estimate = 5.5
- WLS estimate (weighted mean based on LMS or LTS) = 6.4

Using MVE and MCD

The routines *primve*, *scatmve*, and *scatmcd* are used in these examples for printing and plotting the results. These routines are in the **robustmc.sas** file that is contained in the sample library.

Example 9.4. Brainlog Data ◆ 207

Example 9.4. Brainlog Data

The following data, consisting of the body weights (in kilograms) and brain weights (in grams) of $N = 28$ animals, are reported by Jerison (1973) and can be found also in Rousseeuw and Leroy (1987, p. 57). Instead of the original data, this example uses the logarithms of the measurements of the two variables.

```
title "*** Brainlog Data: Do MCD, MVE ***";
   aa={ 1.303338E-001  9.084851E-001 ,
        2.6674530          2.6263400  ,
        1.5602650          2.0773680  ,
        1.4418520          2.0606980  ,
      1.703332E-002  7.403627E-001    ,
        4.0681860          1.6989700  ,
        3.4060290          3.6630410  ,
        2.2720740          2.6222140  ,
        2.7168380          2.8162410  ,
        1.0000000          2.0606980  ,
      5.185139E-001          1.4082400 ,
        2.7234560          2.8325090  ,
        2.3159700          2.6085260  ,
        1.7923920          3.1205740  ,
        3.8230830          3.7567880  ,
        3.9731280          1.8450980  ,
      8.325089E-001          2.2528530 ,
        1.5440680          1.7481880  ,
     -9.208187E-001          .0000000 ,
        -1.6382720 -3.979400E-001     ,
      3.979400E-001          1.0827850 ,
        1.7442930          2.2430380  ,
        2.0000000          2.1959000  ,
        1.7173380          2.6434530  ,
        4.9395190          2.1889280  ,
     -5.528420E-001  2.787536E-001    ,
     -9.136401E-001  4.771213E-001    ,
        2.2833010          2.2552720  };
```

By default, the MVE subroutine uses only 1500 randomly selected subsets rather than all subsets. The following specification of the options vector requires that all 3,276 subsets of 3 cases out of 28 cases are generated and evaluated:

```
title2 "***MVE for BrainLog Data***";
title3 "*** Use All Subsets***";
   optn = j(9,1,.);
   optn[1]= 3;                  /* ipri */
   optn[2]= 1;                  /* pcov: print COV */
   optn[3]= 1;                  /* pcor: print CORR */
   optn[5]= -1;                 /* nrep: all subsets */
call mve(sc,xmve,dist,optn,aa);
```

Specifying *optn*[1]=3, *optn*[2]=1, and *optn*[3]=1 requests that all output be printed. shows the classical scatter and correlation matrix.

Output 9.4.1. Some Simple Statistics

```
****************************************
Minimum Volume Ellipsoid (MVE) Estimation
****************************************

 Consider Ellipsoids Containing 15 Cases.

        Classical Covariance Matrix

                    VAR1              VAR2
    VAR1       2.681651236       1.330084693
    VAR2       1.330084693       1.085753755


        Classical Correlation Matrix

                    VAR1              VAR2
    VAR1       1.000000000       0.779493464
    VAR2       0.779493464       1.000000000


           Classical Mean

           VAR1      1.637857
           VAR2      1.921947
```

Output 9.4.2 shows the results of the combinatoric optimization (complete subset sampling).

Output 9.4.2. Iteration History for MVE

```
                ***MVE for BrainLog Data***

                                 Best
       Subset     Singular      Criterion     Percent

         819          0         0.439709         25
        1638          0         0.439709         50
        2457          0         0.439709         75
        3276          0         0.439709        100

             Observations of Best Subset

             1              22              28

               Initial MVE Location
                     Estimates

               VAR1      1.3859759333
               VAR2      1.8022650333

             Initial MVE Scatter Matrix

                       VAR1              VAR2

       VAR1       4.9018525125       3.2937139101
       VAR2       3.2937139101       2.3400650932
```

Output 9.4.3 shows the optimization results after local improvement.

Example 9.4. Brainlog Data ◆ 209

Output 9.4.3. Table of MVE Results

```
                    ***MVE for BrainLog Data***

                       Robust MVE Location
                             Estimates

                    VAR1        1.29528238
                    VAR2      1.8733722792

                  Robust MVE Scatter Matrix

                          VAR1              VAR2

          VAR1      2.0566592937      1.5290250167
          VAR2      1.5290250167      1.2041353589

                    Eigenvalues of Robust
                         Scatter Matrix

                    VAR1       3.2177274012
                    VAR2       0.0430672514

                  Robust Correlation Matrix

                          VAR1              VAR2

          VAR1                 1      0.9716184659
          VAR2      0.9716184659                 1
```

Output 9.4.4 presents a table containing the classical Mahalanobis distances, the ro-
bust distances, and the weights identifying the outlier observations.

Output 9.4.4. Mahalanobis and Robust Distances

```
                    ***MVE for BrainLog Data***

        Classical Distances and Robust (Rousseeuw) Distances
                 Unsquared Mahalanobis Distance and
         Unsquared Rousseeuw Distance of Each Observation
              Mahalanobis              Robust
         N     Distances             Distances            Weight

         1      1.006591              0.897076           1.000000
         2      0.695261              1.405302           1.000000
         3      0.300831              0.186726           1.000000
         4      0.380817              0.318701           1.000000
         5      1.146485              1.135697           1.000000
         6      2.644176              8.828036                  0
         7      1.708334              1.699233           1.000000
         8      0.706522              0.686680           1.000000
         9      0.858404              1.084163           1.000000
        10      0.798698              1.580835           1.000000
        11      0.686485              0.693346           1.000000
        12      0.874349              1.071492           1.000000
        13      0.677791              0.717545           1.000000
        14      1.721526              3.398698                  0
        15      1.761947              1.762703           1.000000
        16      2.369473              7.999472                  0
        17      1.222253              2.805954                  0
        18      0.203178              1.207332           1.000000
        19      1.855201              1.773317           1.000000
        20      2.266268              2.074971           1.000000
        21      0.831416              0.785954           1.000000
        22      0.416158              0.342200           1.000000
        23      0.264182              0.918383           1.000000
        24      1.046120              1.782334           1.000000
        25      2.911101              9.565443                  0
        26      1.586458              1.543748           1.000000
        27      1.582124              1.808423           1.000000
        28      0.394664              1.523235           1.000000
```

Again, you can call the subroutine *scatmve()*, which is included in the sample library
in the file `robustmc.sas`, to plot the classical and robust confidence ellipsoids:

```
    optn = j(9,1,.); optn[5]= -1;
    vnam = { "Log Body Wgt","Log Brain Wgt" };
    filn = "brlmve";
    titl = "BrainLog Data: MVE Use All Subsets";
  call scatmve(2,optn,.9,aa,vnam,titl,1,filn);
```

The plot is shown in Output 9.4.5.

Example 9.4. Brainlog Data ◆ 211

Output 9.4.5. BrainLog Data: Classical and Robust Ellipsoid(MVE)
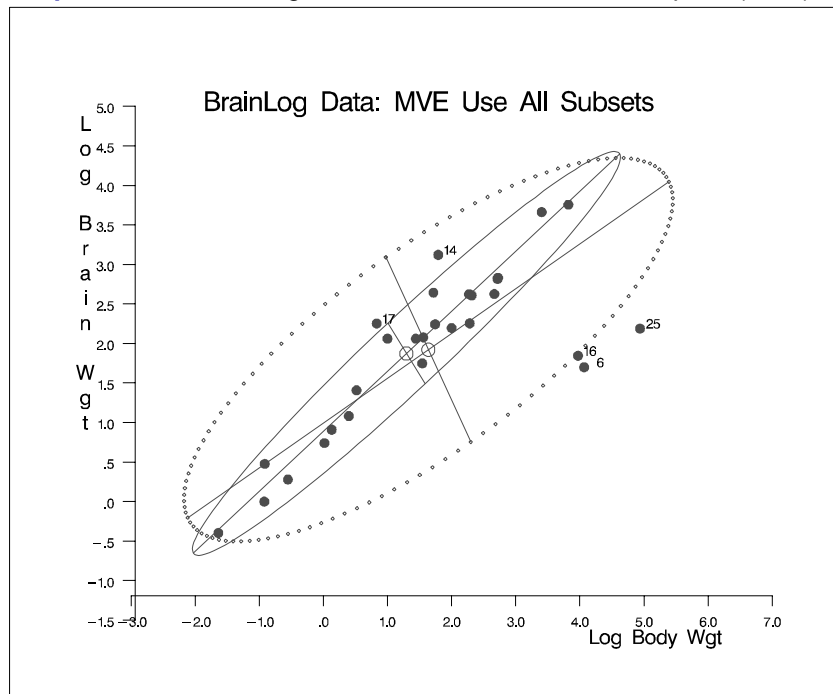
MCD is another subroutine that can be used to compute the robust location and the robust covariance of multivariate data sets.

```
title2 "***MCD for BrainLog Data***";
title3 "*** Use 500 Random Subsets***";
   optn = j(9,1,.);
   optn[1]= 3;                    /* ipri */
   optn[2]= 1;                    /* pcov: print COV  */
   optn[3]= 1;                    /* pcor: print CORR */
call mcd(sc,xmve,dist,optn,aa);
```

Similarly, specifying *optn*[1]=3, *optn*[2]=1, and *optn*[3]=1 requests that all output be printed.

Output 9.4.6 shows the results of the optimization.

Output 9.4.6. Results of the Optimization

```
   1    2    3    4    5    8    9   11   12   13   18   21   22   23   28

                        MCD Location Estimate

                         VAR1                VAR2

                    1.622226068        2.0150777867

                     MCD Scatter Matrix Estimate

                              VAR1                 VAR2

              VAR1      0.8973945995        0.6424456706
              VAR2      0.6424456706        0.4793505736
```

Output 9.4.7 shows the reweighted results after removing outliers.

Output 9.4.7. Final Reweighted MCD Results

```
                     Reweighted Location Estimate

                         VAR1                VAR2

                    1.3154029661        1.8568731174

                       Reweighted Scatter Matrix

                              VAR1                 VAR2

              VAR1        2.139986054        1.6068556533
              VAR2        1.6068556533       1.2520384784

                            Eigenvalues

                         VAR1                VAR2

                    3.363074897        0.0289496354

                     Reweighted Correlation Matrix

                              VAR1                 VAR2

              VAR1                 1        0.9816633012
              VAR2        0.9816633012                 1
```

Output 9.4.8 presents a table containing the classical Mahalanobis distances, the robust distances, and the weights identifying the outlier observations.

Example 9.4. Brainlog Data ◆ 213

Output 9.4.8. Mahalanobis and Robust Distances (MCD)

```
           Classical Distances and Robust (Rousseeuw) Distances
                    Unsquared Mahalanobis Distance and
             Unsquared Rousseeuw Distance of Each Observation
                  Mahalanobis          Robust
           N        Distances         Distances          Weight

           1         1.006591          0.855347         1.000000
           2         0.695261          1.477050         1.000000
           3         0.300831          0.239828         1.000000
           4         0.380817          0.517719         1.000000
           5         1.146485          1.108362         1.000000
           6         2.644176         10.599341                0
           7         1.708334          1.808455         1.000000
           8         0.706522          0.690099         1.000000
           9         0.858404          1.052423         1.000000
          10         0.798698          2.077131         1.000000
          11         0.686485          0.888545         1.000000
          12         0.874349          1.035824         1.000000
          13         0.677791          0.683978         1.000000
          14         1.721526          4.257963                0
          15         1.761947          1.716065         1.000000
          16         2.369473          9.584992                0
          17         1.222253          3.571700                0
          18         0.203178          1.323783         1.000000
          19         1.855201          1.741064         1.000000
          20         2.266268          2.026528         1.000000
          21         0.831416          0.743545         1.000000
          22         0.416158          0.419923         1.000000
          23         0.264182          0.944610         1.000000
          24         1.046120          2.289334         1.000000
          25         2.911101         11.471953                0
          26         1.586458          1.518721         1.000000
          27         1.582124          2.054593         1.000000
          28         0.394664          1.675651         1.000000
```

You can call the subroutine *scatmcd()*, which is included in the sample library in file **robustmc.sas**, to plot the classical and robust confidence ellipsoids:

```
    optn = j(9,1,.); optn[5]= -1;
    vnam = { "Log Body Wgt","Log Brain Wgt" };
    filn = "brlmcd";
    titl = "BrainLog Data: MCD";
  call scatmcd(2,optn,.9,aa,vnam,titl,1,filn);
```

The plot is shown in Output 9.4.9.

Output 9.4.9. BrainLog Data: Classical and Robust Ellipsoid (MCD)
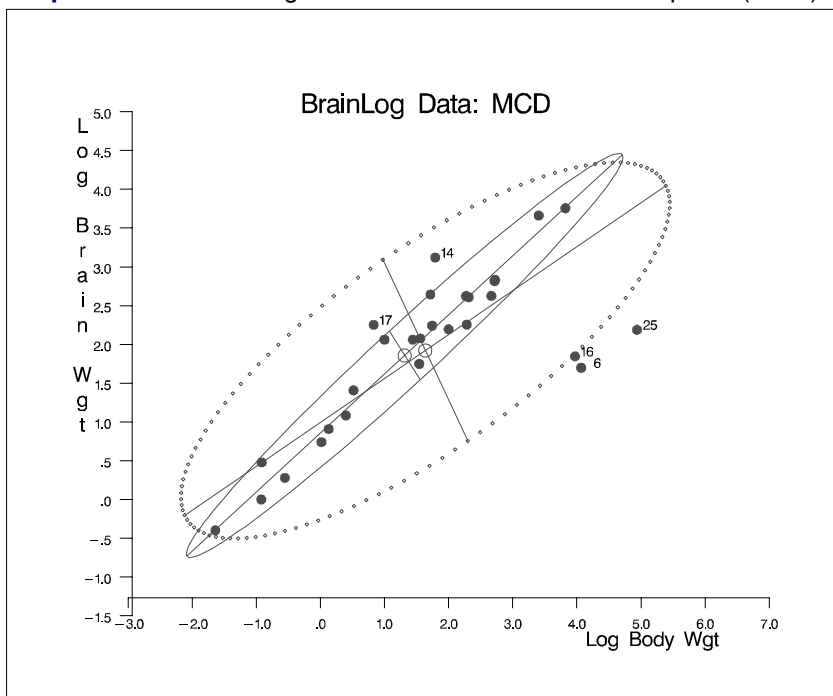
Example 9.5. Stackloss Data

This example analyzes the three regressors of Brownlee (1965) stackloss data. By default, the MVE subroutine tries only 2,000 randomly selected subsets in its search. There are, in total, 5,985 subsets of 4 cases out of 21 cases.

```
title2 "***MVE for Stackloss Data***";
title3 "*** Use All Subsets***";
   a = aa[,2:4];
   optn = j(9,1,.);
   optn[1]= 2;                /* ipri */
   optn[2]= 1;                /* pcov: print COV */
   optn[3]= 1;                /* pcor: print CORR */
   optn[5]= -1;               /* nrep: use all subsets */
call mve(sc,xmve,dist,optn,a);
```

Output 9.5.1 of the output shows the classical scatter and correlation matrix.

Example 9.5. Stackloss Data ◆ 215

Output 9.5.1. Some Simple Statistics

```
****************************************
        Minimum Volume Ellipsoid (MVE) Estimation
****************************************


        Consider Ellipsoids Containing 12 Cases.


             Classical Covariance Matrix


                   VAR1                VAR2                VAR3
    VAR1       84.05714286         22.65714286         24.57142857
    VAR2       22.65714286          9.99047619          6.62142857
    VAR3       24.57142857          6.62142857         28.71428571



             Classical Correlation Matrix


                   VAR1                VAR2                VAR3
    VAR1       1.000000000         0.781852333         0.500142875
    VAR2       0.781852333         1.000000000         0.390939538
    VAR3       0.500142875         0.390939538         1.000000000



                   Classical Mean


                   VAR1        60.42857
                   VAR2        21.09524
                   VAR3        86.28571
```

Output 9.5.2 shows the results of the optimization (complete subset sampling).

Output 9.5.2. Iteration History

```
                    ***MVE for Stackloss Data***

                                      Best
          Subset      Singular      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
```

Output 9.5.3 shows the optimization results after local improvement.

Output 9.5.3.   Table of MVE Results

```
                       ***MVE for Stackloss Data***

                          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
```

Output 9.5.4 presents a table containing the classical Mahalanobis distances, the robust distances, and the weights identifying the outlying observations (that is, the leverage points when explaining y with these three regressor variables).

Example 9.5. Stackloss Data ♦ 217

Output 9.5.4. Mahalanobis and Robust Distances

```
                    ***MVE for Stackloss Data***

        Classical Distances and Robust (Rousseeuw) Distances
                 Unsquared Mahalanobis Distance and
          Unsquared Rousseeuw Distance of Each Observation
                 Mahalanobis              Robust
           N       Distances            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
```

The following specification generates three bivariate plots of the classical and robust tolerance ellipsoids. They are shown in Output 9.5.5, Output 9.5.6, and Output 9.5.7, one plot for each pair of variables.

```
    optn = j(9,1,.); optn[5]= -1;
    vnam = { "Rate", "Temperature", "AcidConcent" };
    filn = "stlmve";
    titl = "Stackloss Data: Use All Subsets";
  call scatmve(2,optn,.9,a,vnam,titl,1,filn);
```

Output 9.5.5. Stackloss Data: Rate vs. Temperature (MVE)

Stackloss Data: MVE Use All Subsets
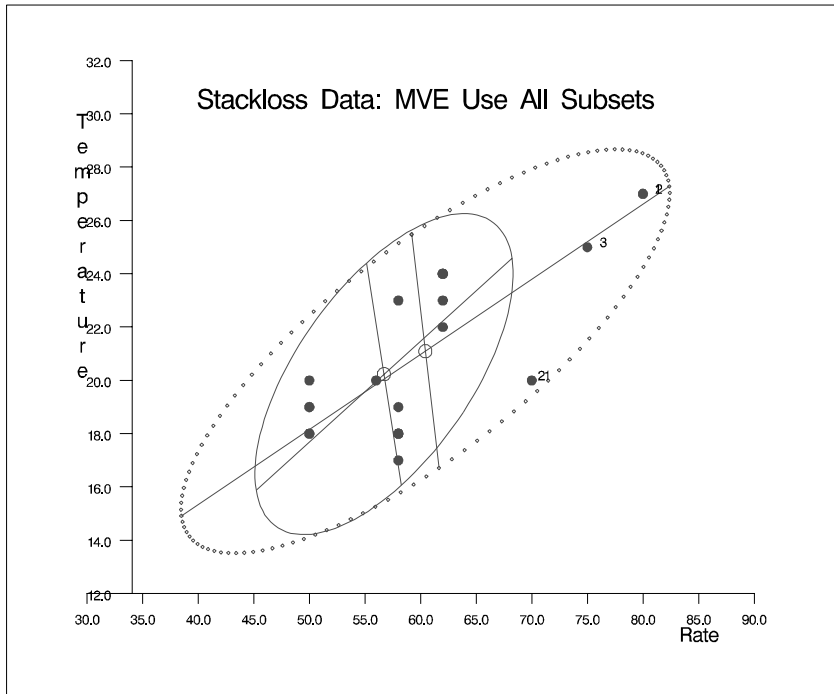
Output 9.5.6. Stackloss Data: Rate vs. Acid Concent (MVE)

Stackloss Data: MVE Use All Subsets

Example 9.5. Stackloss Data ♦ 219
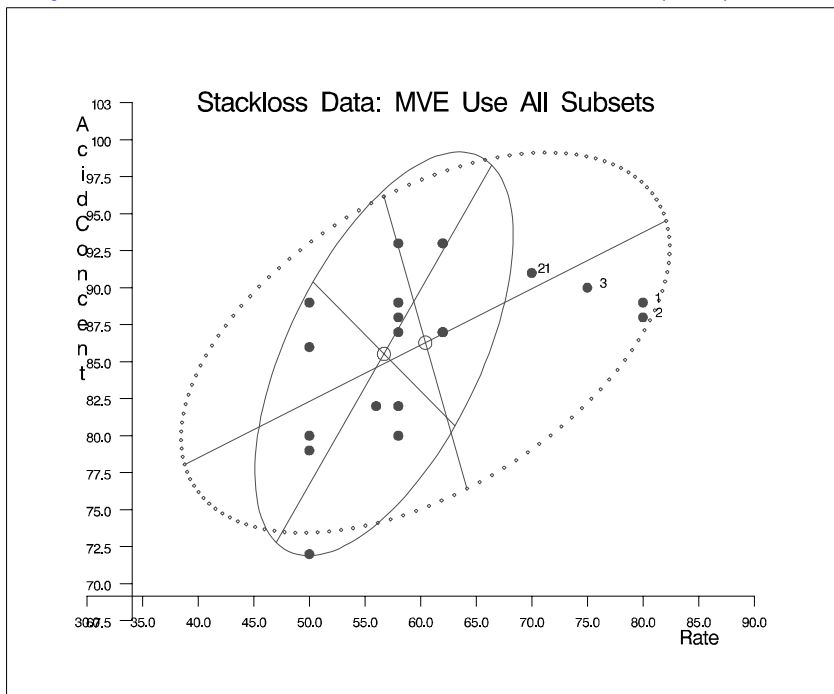
Output 9.5.7. Stackloss Data: Temperature vs. Acid Concent (MVE)

Stackloss Data: MVE Use All Subsets
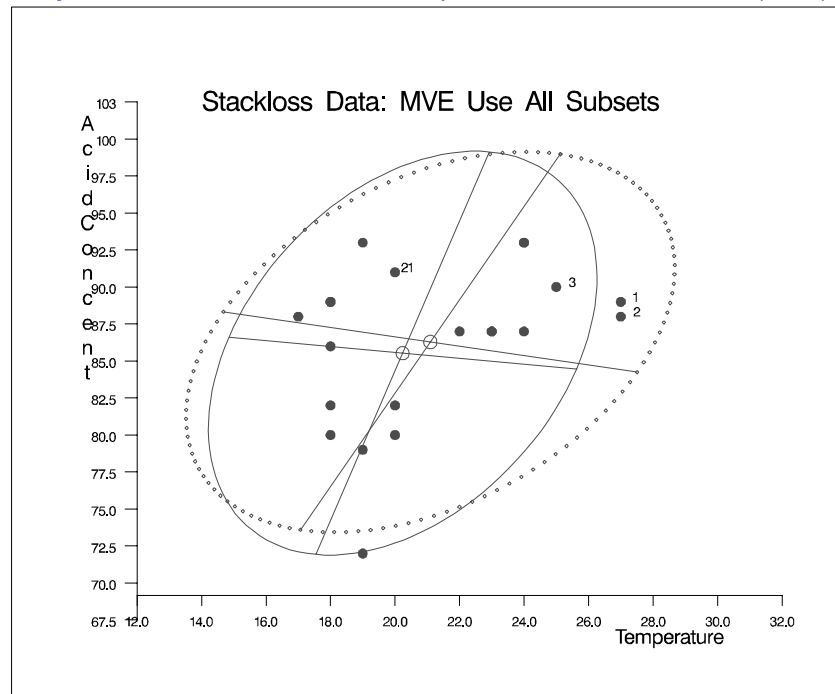
You can also use the MCD method for the stackloss data:

```
title2 "***MCD for Stackloss Data***";
title3 "*** Use 500 Random Subsets***";
a = aa[,2:4];
optn = j(8,1,.);
optn[1]= 2;                  /* ipri */
optn[2]= 1;                  /* pcov: print COV */
optn[3]= 1;                  /* pcor: print CORR */
optn[5]= -1 ;                /* nrep: use all subsets */
CALL MCD(sc,xmcd,dist,optn,a);
```

The optimization results are displayed in Output 9.5.8. The reweighted results are
displayed in Output 9.5.9.

Output 9.5.8. MCD Results of Optimization

```
                         ***MCD for Stackloss Data***
                        *** Use 500 Random Subsets***

4     5     6     7     8     9     10    11    12    13    14    20

                          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

                          MCD Correlation Matrix

                            VAR1                VAR2                VAR3

            VAR1                 1       0.7674714142       0.4745347313
            VAR2      0.7674714142                  1       0.4192963398
            VAR3      0.4745347313       0.4192963398                  1

                         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
```

Example 9.5. Stackloss Data ⬧ 221

Output 9.5.9. Final Reweighted MCD Results

```
                    ***MCD for Stackloss Data***
                   *** Use 500 Random Subsets***

                   Reweighted Location Estimate

              VAR1              VAR2              VAR3

              59.5       20.833333333      87.333333333

                   Reweighted Scatter Matrix

                    VAR1              VAR2              VAR3

      VAR1      5.1818181818      4.8181818182      4.7272727273
      VAR2      4.8181818182      7.6060606061      5.0606060606
      VAR3      4.7272727273      5.0606060606      19.151515152

                         Eigenvalues

              VAR1              VAR2              VAR3

         23.191069268       7.3520037086      1.3963209628

                   Reweighted Correlation Matrix

                    VAR1              VAR2              VAR3

      VAR1               1      0.7674714142      0.4745347313
      VAR2      0.7674714142               1      0.4192963398
      VAR3      0.4745347313      0.4192963398               1
```

The MCD robust distances and outlying diagnostic are displayed in Output 9.5.10.
MCD identifies more leverage points than MVE.

Output 9.5.10. MCD Robust Distances

```
                  ***MCD for Stackloss Data***
                  *** Use 500 Random Subsets***

         Classical Distances and Robust (Rousseeuw) Distances
                 Unsquared Mahalanobis Distance and
              Unsquared Rousseeuw Distance of Each Observation
                Mahalanobis            Robust
         N       Distances           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
```

Similarly, you can use the macro *scatmcd* to generate three bivariate plots of the classical and robust tolerance ellipsoids, one plot for each pair of variables:

```
    optn = j(9,1,.); optn[5]= -1;
    vnam = { "Rate", "Temperature", "AcidConcent" };
    filn = "stlmcd";
    titl = "Stackloss Data: Use All Subsets";
call scatmcd(2,optn,.9,a,vnam,titl,1,filn);
```

Output 9.5.11, Output 9.5.12, and Output 9.5.13 display these plots.

Example 9.5. Stackloss Data ♦ 223

Output 9.5.11. Stackloss Data: Rate vs. Temperature(MCD)
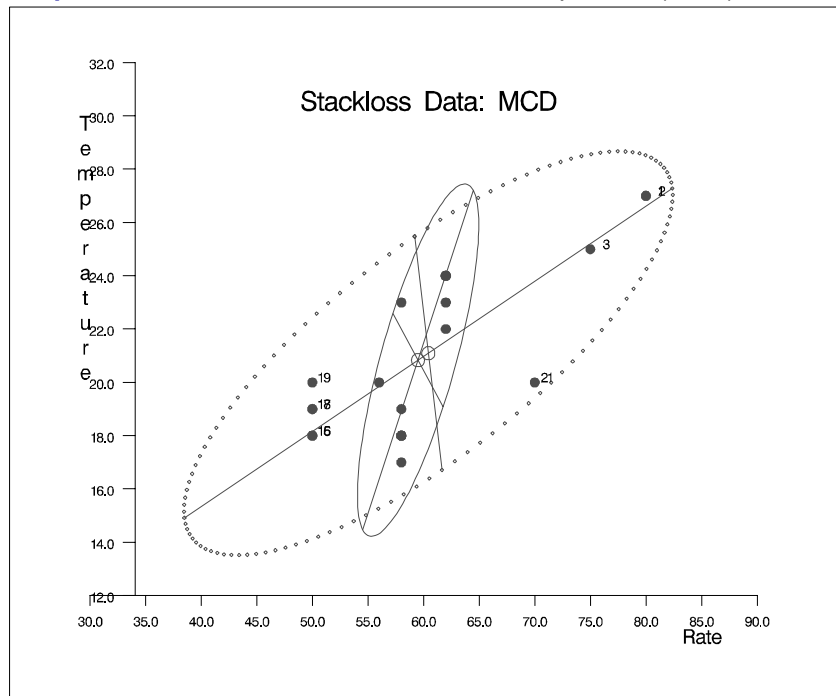
Output 9.5.12. Stackloss Data: Rate vs. Acid Concent(MCD)
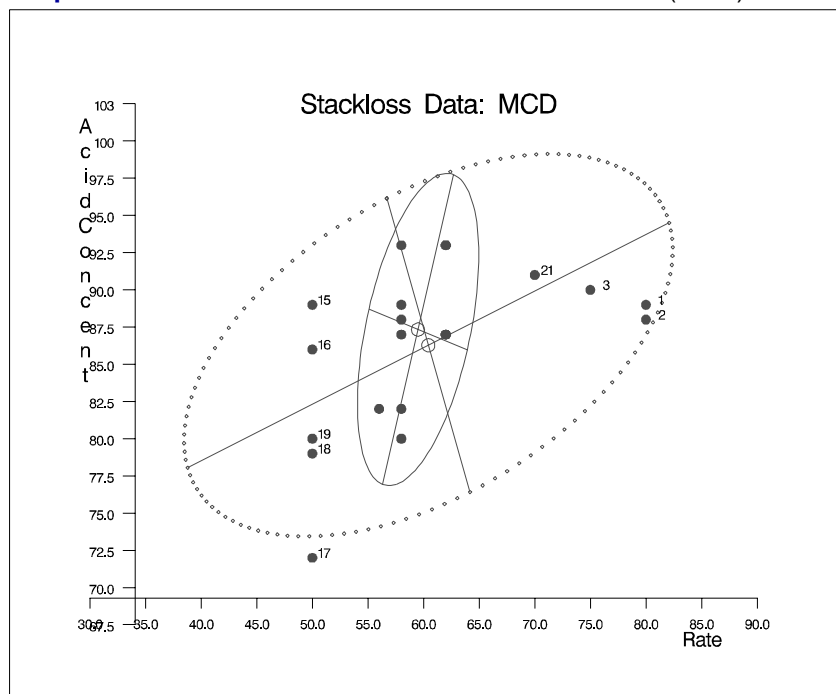
Output 9.5.13. Stackloss Data: Temperature vs. Acid Concent(MCD)
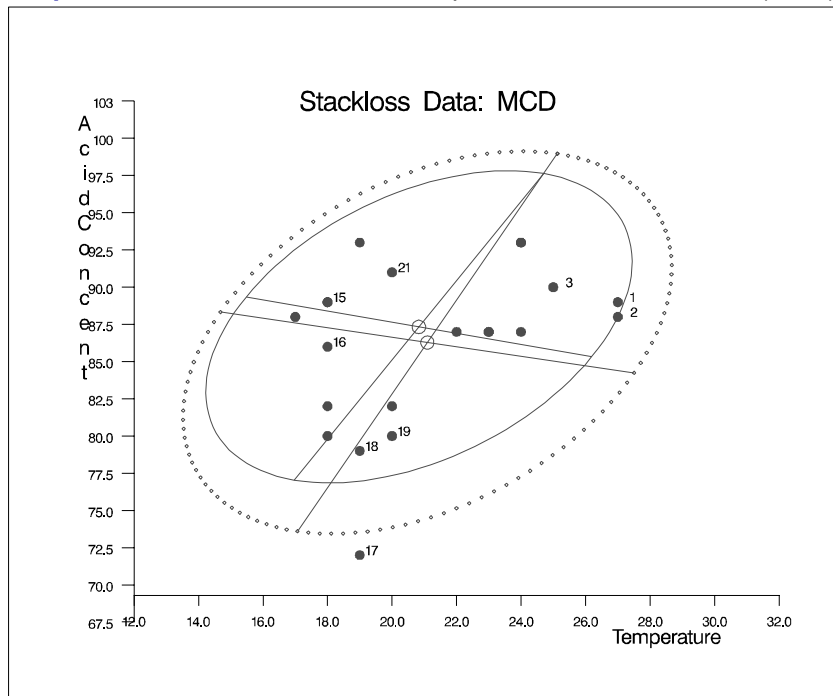
Combining Robust Residual and Robust Distance

This section is based entirely on Rousseeuw and Van Zomeren (1990). Observations $\mathbf{x}_i$, which are far away from most of the other observations, are called *leverage points*. One classical method inspects the Mahalanobis distances MD_i to find outliers $\mathbf{x}_i$:

$$MD_i = \sqrt{(\mathbf{x}_i - \mu)\mathbf{C}^{-1}(\mathbf{x}_i - \mu)^T}$$

where $\mathbf{C}$ is the classical sample covariance matrix.

Note that the MVE subroutine prints the classical Mahalanobis distances MD_i together with the robust distances RD_i. In classical linear regression, the diagonal elements h_{ii} of the *hat* matrix

$$\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

are used to identify leverage points. Rousseeuw and Van Zomeren (1990) report the following monotone relationship between the h_{ii} and MD_i:

$$h_{ii} = \frac{(MD_i)^2}{N-1} + \frac{1}{n}$$

They point out that neither the MD_i nor the h_{ii} are entirely safe for detecting leverage points reliably. Multiple outliers do not necessarily have large MD_i values because of the masking effect.

Example 9.6. Hawkins-Bradu-Kass Data ◆ 225

The definition of a *leverage point* is, therefore, based entirely on the outlyingness of $\mathbf{x}_i$ and is not related to the response value y_i. By including the y_i value in the definition, Rousseeuw and Van Zomeren (1990) distinguish between the following:

- *Good leverage points* are points $(\mathbf{x}_i, y_i)$ that are close to the regression plane; that is, good leverage points improve the precision of the regression coefficients.

- *Bad leverage points* are points $(\mathbf{x}_i, y_i)$ that are far from the regression plane; that is, bad leverage points reduce the precision of the regression coefficients.

Rousseeuw and Van Zomeren (1990) propose to plot the standardized residuals of robust regression (LMS or LTS) versus the robust distances RD_i obtained from MVE. Two horizontal lines corresponding to residual values of $+2.5$ and -2.5 are useful to distinguish between small and large residuals, and one vertical line corresponding to the $\sqrt{\chi^2_{n,.975}}$ is used to distinguish between small and large distances.

Example 9.6. Hawkins-Bradu-Kass Data

The first 14 observations of this data set (refer to Hawkins, Bradu, and Kass 1984) are leverage points; however, only observations 12, 13, and 14 have large h_{ii}, and only observations 12 and 14 have large MD_i values.

```
title "Hawkins, Bradu, Kass (1984) Data";
  aa = { 1   10.1   19.6   28.3    9.7,
         2    9.5   20.5   28.9   10.1,
         3   10.7   20.2   31.0   10.3,
         4    9.9   21.5   31.7    9.5,
         5   10.3   21.1   31.1   10.0,
         6   10.8   20.4   29.2   10.0,
         7   10.5   20.9   29.1   10.8,
         8    9.9   19.6   28.8   10.3,
         9    9.7   20.7   31.0    9.6,
        10    9.3   19.7   30.3    9.9,
        11   11.0   24.0   35.0   -0.2,
        12   12.0   23.0   37.0   -0.4,
        13   12.0   26.0   34.0    0.7,
        14   11.0   34.0   34.0    0.1,
        15    3.4    2.9    2.1   -0.4,
        16    3.1    2.2    0.3    0.6,
        17    0.0    1.6    0.2   -0.2,
        18    2.3    1.6    2.0    0.0,
        19    0.8    2.9    1.6    0.1,
        20    3.1    3.4    2.2    0.4,
        21    2.6    2.2    1.9    0.9,
        22    0.4    3.2    1.9    0.3,
        23    2.0    2.3    0.8   -0.8,
        24    1.3    2.3    0.5    0.7,
        25    1.0    0.0    0.4   -0.3,
        26    0.9    3.3    2.5   -0.8,
        27    3.3    2.5    2.9   -0.7,
```

```
             28    1.8    0.8    2.0    0.3,
             29    1.2    0.9    0.8    0.3,
             30    1.2    0.7    3.4   -0.3,
             31    3.1    1.4    1.0    0.0,
             32    0.5    2.4    0.3   -0.4,
             33    1.5    3.1    1.5   -0.6,
             34    0.4    0.0    0.7   -0.7,
             35    3.1    2.4    3.0    0.3,
             36    1.1    2.2    2.7   -1.0,
             37    0.1    3.0    2.6   -0.6,
             38    1.5    1.2    0.2    0.9,
             39    2.1    0.0    1.2   -0.7,
             40    0.5    2.0    1.2   -0.5,
             41    3.4    1.6    2.9   -0.1,
             42    0.3    1.0    2.7   -0.7,
             43    0.1    3.3    0.9    0.6,
             44    1.8    0.5    3.2   -0.7,
             45    1.9    0.1    0.6   -0.5,
             46    1.8    0.5    3.0   -0.4,
             47    3.0    0.1    0.8   -0.9,
             48    3.1    1.6    3.0    0.1,
             49    3.1    2.5    1.9    0.9,
             50    2.1    2.8    2.9   -0.4,
             51    2.3    1.5    0.4    0.7,
             52    3.3    0.6    1.2   -0.5,
             53    0.3    0.4    3.3    0.7,
             54    1.1    3.0    0.3    0.7,
             55    0.5    2.4    0.9    0.0,
             56    1.8    3.2    0.9    0.1,
             57    1.8    0.7    0.7    0.7,
             58    2.4    3.4    1.5   -0.1,
             59    1.6    2.1    3.0   -0.3,
             60    0.3    1.5    3.3   -0.9,
             61    0.4    3.4    3.0   -0.3,
             62    0.9    0.1    0.3    0.6,
             63    1.1    2.7    0.2   -0.3,
             64    2.8    3.0    2.9   -0.5,
             65    2.0    0.7    2.7    0.6,
             66    0.2    1.8    0.8   -0.9,
             67    1.6    2.0    1.2   -0.7,
             68    0.1    0.0    1.1    0.6,
             69    2.0    0.6    0.3    0.2,
             70    1.0    2.2    2.9    0.7,
             71    2.2    2.5    2.3    0.2,
             72    0.6    2.0    1.5   -0.2,
             73    0.3    1.7    2.2    0.4,
             74    0.0    2.2    1.6   -0.9,
             75    0.3    0.4    2.6    0.2 };

        a = aa[,2:4]; b = aa[,5];
```

The data are listed also in Rousseeuw and Leroy (1987, p. 94).

The complete enumeration must inspect 1,215,450 subsets.

Example 9.6. Hawkins-Bradu-Kass Data ⋄ 227

Output 9.6.1 displays the iteration history for MVE.

Output 9.6.1. Iteration History for MVE

```
***********************************************
***        Complete Enumeration for MVE       ***
***********************************************


          Subset Singular       Best Crit   Pct
          121545         0 51.1042755960104   10%
          243090         2 51.1042755960104   20%
          364635         4 51.1042755960104   30%
          486180         7 51.1042755960104   40%
          607725         9 51.1042755960104   50%
          729270        22 6.27172477029496   60%
          850815        67 6.27172477029496   70%
          972360       104 5.91230765636768   80%
         1093905       135 5.91230765636768   90%
         1215450       185 5.91230765636768  100%
             Minimum Criterion=5.9123076564
        Among 1215450 subsets 185 are singular.
```

Output 9.6.2 reports the robust parameter estimates for MVE.

Output 9.6.2. Robust Location Estimates

```
                Robust MVE Location Estimates

                   VAR1       1.513333333
                   VAR2       1.808333333
                   VAR3       1.701666667


              Robust MVE Scatter Matrix

                   VAR1            VAR2            VAR3
      VAR1    1.114395480     0.093954802     0.141672316
      VAR2    0.093954802     1.123149718     0.117443503
      VAR3    0.141672316     0.117443503     1.074742938
```

Output 9.6.3 reports the eigenvalues of the robust scatter matrix and the robust correlation matrix.

Output 9.6.3. MVE Scatter Matrix

```
              Eigenvalues of Robust Scatter Matrix

                   VAR1       1.339637154
                   VAR2       1.028124757
                   VAR3       0.944526224


              Robust Correlation Matrix

                   VAR1            VAR2            VAR3
      VAR1    1.000000000     0.083980892     0.129453270
      VAR2    0.083980892     1.000000000     0.106895118
      VAR3    0.129453270     0.106895118     1.000000000
```

Output 9.6.4 shows the classical Mahalanobis and robust distances obtained by complete enumeration. The first 14 observations are recognized as outliers (leverage points).

Output 9.6.4. Mahalanobis and Robust Distances

```
                Classical and Robust Distances
       -------------------------------------------------
          Mahalanobis        Robust
           Distance         Distance          Weight
       -------------------------------------------------
        1     1.916821      29.541649                0
        2     1.855757      30.344481                0
        3     2.313658      31.985694                0
        4     2.229655      33.011768                0
        5     2.100114      32.404938                0
        6     2.146169      30.683153                0
        7     2.010511      30.794838                0
        8     1.919277      29.905756                0
        9     2.221249      32.092048                0
       10     2.333543      31.072200                0
       11     2.446542      36.808021                0
       12     3.108335      38.071382                0
       13     2.662380      37.094539                0
       14     6.381624      41.472255                0
       15     1.815487       1.994672         1.000000
       16     2.151357       2.202278         1.000000
       17     1.384915       1.918208         1.000000
       18     0.848155       0.819163         1.000000
       19     1.148941       1.288387         1.000000
       20     1.591431       2.046703         1.000000
       21     1.089981       1.068327         1.000000
       22     1.548776       1.768905         1.000000
       23     1.085421       1.166951         1.000000
       24     0.971195       1.304648         1.000000
       25     0.799268       2.030417         1.000000
       26     1.168373       1.727131         1.000000
       27     1.449625       1.983831         1.000000
       28     0.867789       1.073856         1.000000
       29     0.576399       1.168060         1.000000
```

Example 9.6. Hawkins-Bradu-Kass Data ◆ 229

Output 9.6.4. (continued)

```
              Classical and Robust Distances
       --------------------------------------------------
            Mahalanobis        Robust
             Distance        Distance        Weight
       --------------------------------------------------
         30      1.568868        2.091386        1.000000
         31      1.838496        1.793386        1.000000
         32      1.307230        1.743558        1.000000
         33      0.981988        1.264121        1.000000
         34      1.175014        2.052641        1.000000
         35      1.243636        1.872695        1.000000
         36      0.850804        1.136658        1.000000
         37      1.832378        2.050041        1.000000
         38      0.752061        1.522734        1.000000
         39      1.265041        1.885970        1.000000
         40      1.112038        1.068841        1.000000
         41      1.699757        2.063398        1.000000
         42      1.765040        1.785637        1.000000
         43      1.870090        2.166100        1.000000
         44      1.420448        2.018610        1.000000
         45      1.075973        1.944449        1.000000
         46      1.344171        1.872483        1.000000
         47      1.966328        2.408721        1.000000
         48      1.424238        1.892539        1.000000
         49      1.569756        1.594109        1.000000
         50      0.423972        1.458595        1.000000
         51      1.302651        1.569843        1.000000
         51      1.302651        1.569843        1.000000
         52      2.076055        2.205601        1.000000
         53      2.210443        2.492631        1.000000
         54      1.414288        1.884937        1.000000
         55      1.230455        1.360622        1.000000
         56      1.331101        1.626276        1.000000
         57      0.832744        1.432408        1.000000
         58      1.404401        1.723091        1.000000
         59      0.591235        1.263700        1.000000
         60      1.889737        2.087849        1.000000
         61      1.674945        2.286045        1.000000
         62      0.759533        2.024702        1.000000
         63      1.292259        1.783035        1.000000
         64      0.973868        1.835207        1.000000
         65      1.148208        1.562278        1.000000
         66      1.296746        1.444491        1.000000
         67      0.629827        0.552899        1.000000
         68      1.549548        2.101580        1.000000
         69      1.070511        1.827919        1.000000
         70      0.997761        1.354151        1.000000
         71      0.642927        0.988770        1.000000
         72      1.053395        0.908316        1.000000
         73      1.472178        1.314779        1.000000
         74      1.646461        1.516083        1.000000
         75      1.899178        2.042560        1.000000


              Distribution of Robust Distances
    MinRes    1st Qu.     Median       Mean     3rd Qu.      MaxRes
 0.55289874 1.44449066 1.88493749 7.56960939 2.16610046 41.4722551


              Cutoff Value = 3.0575159206
The cutoff value is the square root of the 0.975 quantile of the
      chi square distribution with 3 degrees of freedom
There are 14 points with larger distances receiving zero weights.
            These may include boundary cases.
Only points whose robust distances are substantially larger than
      the cutoff value should be considered outliers.
```

The following two graphs in Output 9.6.5 and Output 9.6.6 show

- the plot of standardized LMS residuals vs. robust distances RD_i
- the plot of standardized LS residuals vs. Mahalanobis distances MD_i

The graph identifies the four good leverage points 11, 12, 13, and 14, which have small standardized LMS residuals but large robust distances, and the 10 bad leverage points $1, \ldots, 10$, which have large standardized LMS residuals and large robust distances.

Output 9.6.5. Hawkins-Bradu-Kass Data: LMS Residuals vs. Robust Distances

Example 9.7. Stackloss Data ◆ 231
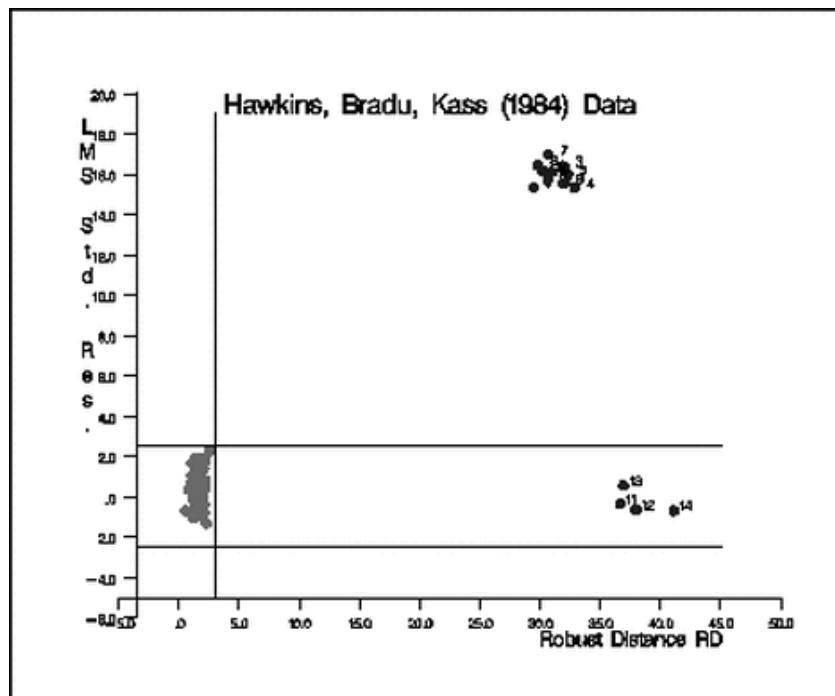
Output 9.6.6. Hawkins-Bradu-Kass Data: LS Residuals vs. Mahalanobis
Distances
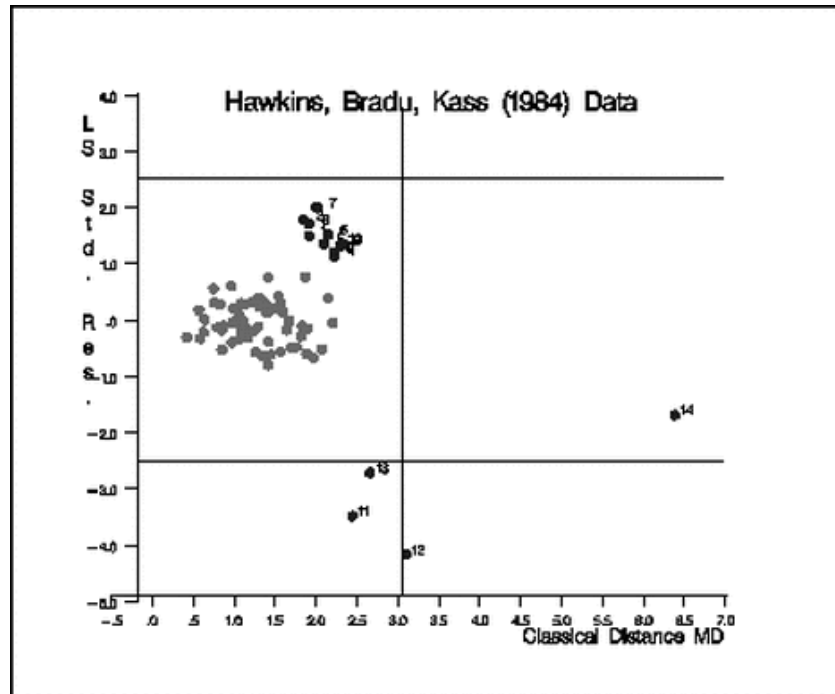
Example 9.7. Stackloss Data

The following two graphs in Output 9.7.1 and Output 9.7.2 show

- the plot of standardized LMS residuals vs. robust distances RD_i
- the plot of standardized LS residuals vs. Mahalanobis distances MD_i

In the first plot, you see that case 4 is a regression outlier but not a leverage point, so
it is a vertical outlier. Cases 1, 3, and 21 are bad leverage points, whereas case 2 is
a good leverage point. Note that case 21 lies near the boundary line between vertical
outliers and bad leverage points and that case 2 is very close to the boundary between
good and bad leverage points.

Output 9.7.1. Stackloss Data: LMS Residuals vs. Robust Distances
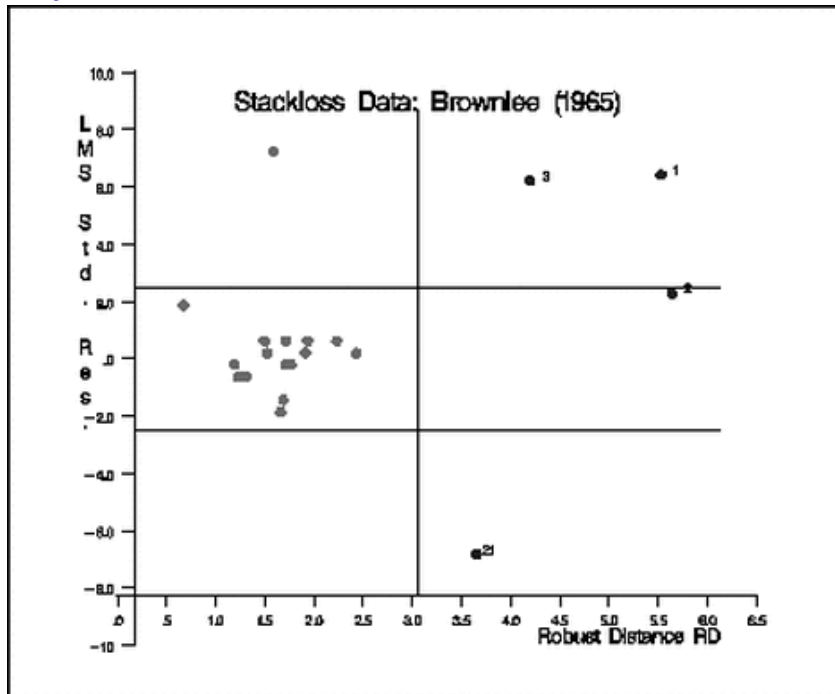
Output 9.7.2. Stackloss Data: LS Residuals vs. Mahalanobis Distances
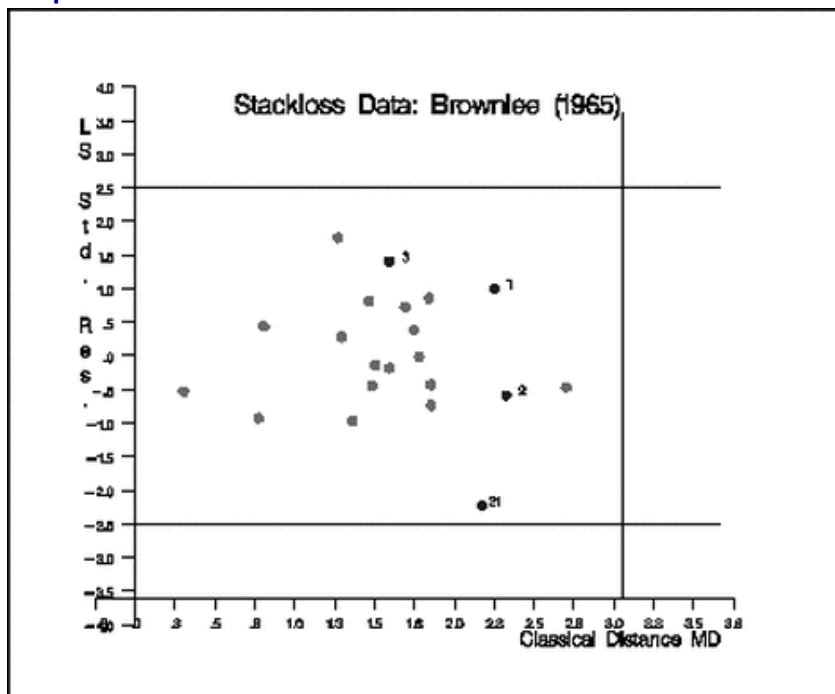
References

Afifi, A.A. and Azen, S.P. (1979), *Statistical Analysis: A Computer Oriented Approach,* New York: Academic Press.

Barnett, V. and Lewis, T. (1978), *Outliers in Statistical Data*, New York: John Wiley & Sons, Inc.

Brownlee, K.A. (1965), *Statistical Theory and Methodology in Science and Engineering*, New York: John Wiley & Sons, Inc.

Chen C. (2002), "Robust Tools in SAS," in *Developments in Robust Statistics: International Conference on Robust Statistics 2001,* (ed. by R. Dutter, P. Filzmoser, U. Gather, and P.J. Rousseeuw), Springer-Verlag, Heidelberg.

Ezekiel, M. and Fox, K.A. (1959), *Methods of Correlation and Regression Analysis*, New York: John Wiley & Sons, Inc.

Hawkins, D.M., Bradu, D., and Kass, G.V. (1984), "Location of Several Outliers in Multiple Regression Data Using Elemental Sets," *Technometrics*, 26, 197–208.

Humphreys, R.M. (1978), "Studies of Luminous Stars in Nearby Galaxies, I. Supergiants and O Stars in the Milky Way," *Astrophys. J. Suppl. Ser.*, 38, 309-350.

Jerison, H.J. (1973), *Evolution of the Brain and Intelligence*, New York: Academic Press.

Osborne, M.R. (1985), *Finite Algorithms in Optimization and Data Analysis*, New York: John Wiley & Sons, Inc.

Prescott, P. (1975), "An Approximate Test for Outliers in Linear Models," *Technometrics*, 17, 129–132.

Rousseeuw, P.J. (1984), "Least Median of Squares Regression," *Journal of the American Statistical Association*, 79, 871–880.

Rousseeuw, P.J. (1985), "Multivariate Estimation with High Breakdown Point," in *Mathematical Statistics and Applications*, (ed. by W. Grossmann, G. Pflug, I. Vincze, and W. Wertz), Dordrecht, Netherlands: Reidel Publishing Company, 283–297.

Rousseeuw, P.J. and Hubert, M. (1996), "Recent Developments in PROGRESS," Technical Report, University of Antwerp.

Rousseeuw, P.J. and Leroy, A.M. (1987), *Robust Regression and Outlier Detection*, New York: John Wiley & Sons, Inc.

Rousseeuw, P.J. and Van Driessen, K. (1999), "A Fast Algorithm for the Minimum Covariance Determinant Estimator," *Technometrics* , 41, 212–223.

Rousseeuw, P.J. and Van Driessen, K. (2000), "An Algorithm for Positive-Breakdown Regression Based on Concentration Steps," *Data Analysis: Scientific Modeling and Practical Application*, (ed. by W. Gaul, O. Opitz, and M. Schader), New York: Springer Verlag, 335–346.

Rousseeuw, P.J. and Van Zomeren, B.C. (1990), "Unmasking Multivariate Outliers and Leverage Points," *Journal of the American Statistical Association*, 85, 633 –639.

Vansina, F., and De Greve, J. P. (1982), "Close Binary Systems Before and After Mass Transfer," *Astrophys. Space Sci.*, 87, 377-401.

Chapter 10
Time Series Analysis and Examples

Chapter Contents

Chapter 10
Time Series Analysis and Examples

Overview

This chapter describes SAS/IML subroutines related to univariate, multivariate, and fractional time series analysis, and subroutines for Kalman filtering and smoothing. These subroutines can be used in analyzing economic and financial time series to model not only correlated its past values but also contemporaneously correlated each other and each other's past values. You can develop a model of the univariate time series and a model of the relationships between the vector time series. The Kalman filter subroutines provide analysis of various time series and are presented as a tool for dealing with state space models.

The subroutines offer the following functions:

- generating univariate, multivariate, and fractional time series
- computing likelihood function of ARMA, VARMA, and ARFIMA models
- computing an autocovariance function of ARMA, VARMA, and ARFIMA models
- checking the stationarity of ARMA and VARMA models
- filtering and smoothing of time series models using Kalman method
- fitting AR, periodic AR, time-varying coefficient AR, VAR, and ARFIMA models
- handling Bayesian seasonal adjustment model

In addition, decomposition analysis, forecast of an ARMA model, and fractionally differencing of the series are provided.

This chapter consists of five sections. The first section includes the ARMACOV and ARMALIK subroutines and ARMASIM function. The second section includes the TSBAYSEA, TSDECOMP, TSMLOCAR, TSMLOMAR, TSMULMAR, TSPERARS, TSPRED, TSROOT, TSTVCAR, and TSUNIMAR subroutines. The third section includes the KALCVF, KALCVS, KALDFF, and KALDFS subroutines. The four section includes the VARMACOV, VARMALIK, VARMASIM, VNORMAL, and VTSROOT subroutines. The last section includes the FARMACOV, FARMAFIT, FARMALIK, FARMASIM, and FDIF subroutines.

Basic Time Series Subroutines

In classical linear regression analysis, the underlying process often can be represented simply an intercept and slope parameters. A time series can be modeled by a type of regression analysis.

The ARMASIM function generates various time series from the underlying AR, MA, and ARMA models. Simulations of time series with known ARMA structure are often needed as part of other simulations or as learning data sets for developing time series analysis skills.

The ARMACOV subroutine provides the pattern of the autocovariance function of AR, MA, and ARMA models and helps to identify and fit a proper model.

The ARMALIK subroutine provides the log-likelihood of an ARMA model and helps to obtain estimates of the parameters of a regression model with innovations having an ARMA structure.

The following subroutines are supported:

ARMACOV   computes an autocovariance sequence for an ARMA model

ARMALIK   computes the log-likelihood and residuals for an ARMA model

ARMASIM   simulates an ARMA series

See the examples of the use of ARMACOV and ARMALIK subroutines in Chapter 8, "General Statistics Examples," .

Getting Started

Consider a time series of length 100 from the ARMA(2,1) model

$$y_t = 0.5y_{t-1} - 0.04y_{t-2} + e_t + 0.25e_{t-1}$$

where the error series follows a normal with mean 10 and variance 2.

The following statements show generating, computing 10 lags of the autocovariance functions and calculating the log-likelihood function and residuals of the ARMA(2,1) model:

```
proc iml;
   /* ARMA(2,1) model */
   phi = {1 -0.5 0.04};
   theta = {1 0.25};
   mu = 10;
   sigma = 2;
   nobs = 100;
   seed = 3456;
   lag = 10;
   yt = armasim(phi, theta, mu, sigma, nobs, seed);
   print yt;
```

```
call armacov(autocov, cross, convol, phi, theta, lag);
autocov = autocov`;
cross = cross`;
convol = convol`;
lag = (0:lag-1)`;
print autocov cross convol;
call armalik(lnl, resid, std, yt, phi, theta);
print lnl resid std;
```
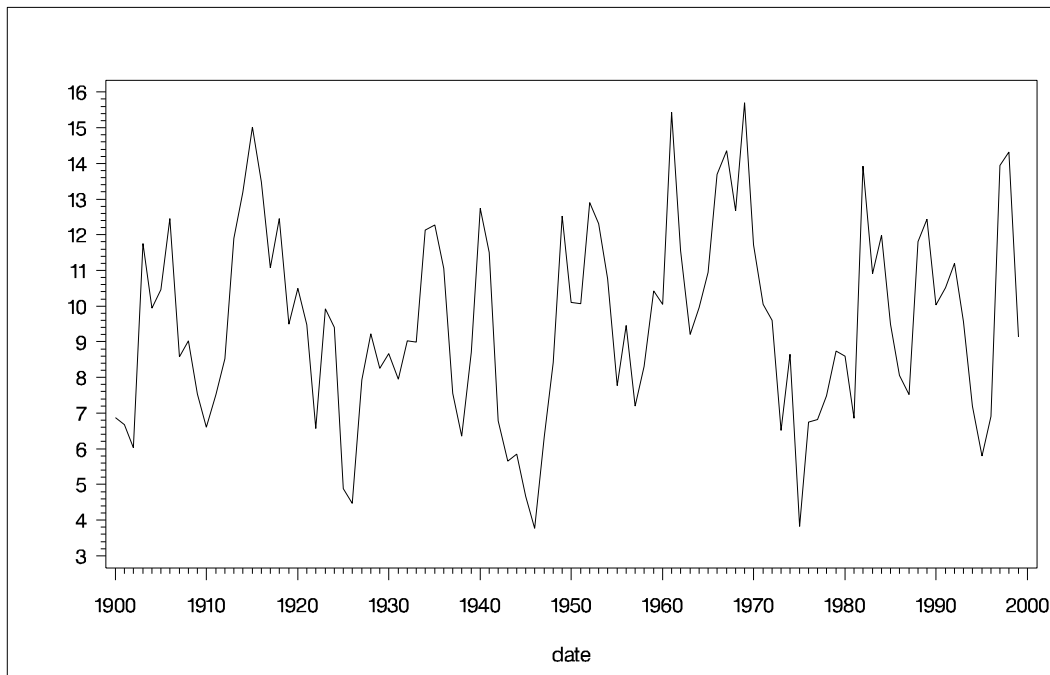

Figure 10.1.   Plot of Generated ARMA(2,1) Process (ARMASIM)

The ARMASIM function generates the data shown in Figure 10.1.

LAG	AUTOCOV	CROSS	CONVOL
0	1.6972803	1.1875	1.0625
1	1.0563848	0.25	0.25
2	0.4603012		
3	0.1878952		
4	0.0755356		
5	0.030252		
6	0.0121046		
7	0.0048422		
8	0.0019369		
9	0.0007748		

Figure 10.2.   Autocovariance functions of ARMA(2,1) Model (ARMACOV)

In Figure 10.2, the ARMACOV subroutine prints the autocovariance functions of the ARMA(2,1) Model and the covariance functions of the moving-average term with lagged values of the process and the autocovariance functions of the moving-average term.

```
              LNL      RESID       STD

      -154.9148 5.2779797 1.3027971
      22.034073 2.3491607    1.0197
      0.5705918 2.3893996 1.0011951
                8.4086892 1.0000746
                 2.200401 1.0000047
                5.4127254 1.0000003
                6.2756004         1
                1.1944693         1
                4.9425372         1
                        .         .
                        .         .
```

Figure 10.3. Log-likelihood function of ARMA(2,1) Model (ARMALIK)

The first column in Figure 10.3 shows the log-likelihood function, the estimate of the innovation variance, and the log of the determinant of the variance matrix. The next two columns are the part of results in the standardized residuals and the scale factors used to standardize the residuals.

Syntax

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

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

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

Time Series Analysis and Control Subroutines

This section describes an adaptation of parts of the **TIM**e **S**eries **A**nalysis and **C**ontrol (TIMSAC) package developed by the Institute of Statistical Mathematics (ISM) in Japan.

Selected routines from the TIMSAC package from ISM were converted by SAS Institute staff into SAS/IML routines under an agreement between SAS Institute and ISM. Credit for authorship of these TIMSAC SAS/IML routines goes to ISM, which has agreed to make them available to SAS users without charge.

There are four packages of TIMSAC programs. See the section "ISM TIMSAC Packages" on page 288 for more information on the TIMSAC package produced by ISM. Since these SAS/IML time series analysis subroutines are adapted from the corresponding FORTRAN subroutines in the TIMSAC package produced by ISM, they are collectively referred to as "the TIMSAC subroutines" in this chapter.

The subroutines analyze and forecast univariate and multivariate time series data. The nonstationary time series and seasonal adjustment models can also be analyzed by using the Interactive Matrix Language TIMSAC subroutines. These subroutines contain the Bayesian modeling of seasonal adjustment and changing spectrum estimation.

Discrete time series modeling has been widely used to analyze dynamic systems in economics, engineering, and statistics. The Box-Jenkins and Box-Tiao approaches

are classical examples of unified time series analysis through identification, estimation, and forecasting (or control). The ARIMA procedure in the SAS/ETS product uses these approaches. Bayesian methods are being increasingly applied despite the controversial issues involved in choosing a prior distribution.

The fundamental idea of the Bayesian method is that uncertainties can be explained by probabilities. If there is a class model (Ω) consisting of sets of member models (ω), you can describe the uncertainty of Ω using a prior distribution of ω. The member model ω is directly related to model parameters. Let the prior probability density function be $p(\omega)$. When you observe the data **y** that is generated from the model Ω, the data distribution is described as $p(Y|\omega)$ given the unknown ω with a prior probability density $p(\omega)$, where the function $p(Y|\omega)$ is the usual likelihood function. Then the posterior distribution is the updated prior distribution given the sample information. The posterior probability density function is proportional to *observed likelihood function $\times$ prior density function*.

The TIMSAC subroutines contain various time series analysis and Bayesian models. Most of the subroutines are based on the minimum Akaike Information Criterion (AIC) or on the minimum Akaike Bayesian Information Criterion (ABIC) method to determine the best model among alternative models. The TSBAYSEA subroutine is a typical example of Bayesian modeling. The following subroutines are supported:

TSBAYSEA	Bayesian seasonal adjustment modeling
TSDECOMP	time series decomposition analysis
TSMLOCAR	locally stationary univariate AR model fitting
TSMLOMAR	locally stationary multivariate AR model fitting
TSMULMAR	multivariate AR model fitting
TSPERARS	periodic AR model fitting
TSPRED	ARMA model forecasting and forecast error variance
TSROOT	polynomial roots or ARMA coefficients computation
TSTVCAR	time-varying coefficient AR model estimation
TSUNIMAR	univariate AR model fitting

For univariate and multivariate autoregressive model estimation, the least-squares method is used. The least-squares estimate is an approximate maximum likelihood estimate if error disturbances are assumed to be Gaussian. The least-squares least-squares performed by using the Householder transformation method. See the section "Least-Squares and Householder Transformation" for details.

The TSUNIMAR and TSMULMAR subroutines estimate the autoregressive models and select the appropriate AR order automatically by using the minimum AIC method. The TSMLOCAR and TSMLOMAR subroutines analyze the nonstationary time series data. The Bayesian time-varying AR coefficient model (TSTVCAR) offers another nonstationary time series analysis method. The state space and Kalman filter method is systematically applied to the smoothness priors models (TSDECOMP

and TSTVCAR), which have stochastically perturbed difference equation constraints. The TSBAYSEA subroutine provides a way of handling Bayesian seasonal adjustment, and it can be an alternative to the SAS/ETS X-11 procedure. The TSBAYSEA subroutine employs the smoothness priors idea through constrained least-squares estimation, while the TSDECOMP and TSTVCAR subroutines estimate the smoothness tradeoff parameters using the state space model and Kalman filter recursive computation. The TSPRED subroutine computes the one-step or multi-step predicted values of the ARMA time series model. In addition, the TSPRED subroutine computes forecast error variances and impulse response functions. The TSROOT subroutine computes the AR and MA coefficients given the characteristic roots of the polynomial equation and the characteristic roots for the AR or MA model.

Getting Started

Minimum AIC Model Selection

The time series model is automatically selected using the AIC. The TSUNIMAR call estimates the univariate autoregressive model and computes the AIC. You need to specify the maximum lag or order of the AR process with the MAXLAG= option or put the maximum lag as the sixth argument of the TSUNIMAR call.

```
proc iml;
   y = { 2.430 2.506 2.767 2.940 3.169 3.450 3.594 3.774 3.695 3.411
         2.718 1.991 2.265 2.446 2.612 3.359 3.429 3.533 3.261 2.612
         2.179 1.653 1.832 2.328 2.737 3.014 3.328 3.404 2.981 2.557
         2.576 2.352 2.556 2.864 3.214 3.435 3.458 3.326 2.835 2.476
         2.373 2.389 2.742 3.210 3.520 3.828 3.628 2.837 2.406 2.675
         2.554 2.894 3.202 3.224 3.352 3.154 2.878 2.476 2.303 2.360
         2.671 2.867 3.310 3.449 3.646 3.400 2.590 1.863 1.581 1.690
         1.771 2.274 2.576 3.111 3.605 3.543 2.769 2.021 2.185 2.588
         2.880 3.115 3.540 3.845 3.800 3.579 3.264 2.538 2.582 2.907
         3.142 3.433 3.580 3.490 3.475 3.579 2.829 1.909 1.903 2.033
         2.360 2.601 3.054 3.386 3.553 3.468 3.187 2.723 2.686 2.821
         3.000 3.201 3.424 3.531 };
   call tsunimar(arcoef,ev,nar,aic) data=y opt={-1 1} print=1
        maxlag=20;
```

You can also invoke the TSUNIMAR call as follows:

```
   call tsunimar(arcoef,ev,nar,aic,y,20,{-1 1},,1);
```

The optional arguments can be omitted. In this example, the argument MISSING is omitted, and thus the default option (MISSING=0) is used. The summary table of the minimum AIC method is displayed in Figure 10.4 and Figure 10.5. The final estimates are given in Figure 10.6.

```
ORDER   INNOVATION VARIANCE
  M           V(M)           AIC(M)
  0         0.31607294    -108.26753229
  1         0.11481982    -201.45277331
  2         0.04847420    -280.51201122
  3         0.04828185    -278.88576251
  4         0.04656506    -280.28905616
  5         0.04615922    -279.11190502
  6         0.04511943    -279.25356641
  7         0.04312403    -281.50543541
  8         0.04201118    -281.96304075
  9         0.04128036    -281.61262868
 10         0.03829179    -286.67686828
 11         0.03318558    -298.13013264
 12         0.03255171    -297.94298716
 13         0.03247784    -296.15655602
 14         0.03237083    -294.46677874
 15         0.03234790    -292.53337704
 16         0.03187416    -291.92021487
 17         0.03183282    -290.04220196
 18         0.03126946    -289.72064823
 19         0.03087893    -288.90203735
 20         0.02998019    -289.67854830
```

Figure 10.4. Minimum AIC Table - I

```
                        AIC(M)-AICMIN (truncated at 40.0)
                    0         10        20        30        40
  M    AIC(M)-AICMIN   +---------+---------+---------+---------+
  0    189.862600      |                                      .
  1     96.677359      |                                      .
  2     17.618121      |                   *                  |
  3     19.244370      |                    *                 |
  4     17.841076      |                   *                  |
  5     19.018228      |                    *                 |
  6     18.876566      |                    *                 |
  7     16.624697      |                  *                   |
  8     16.167092      |                 *                    |
  9     16.517504      |                  *                   |
 10     11.453264      |           *                          |
 11             0      *                                      |
 12      0.187145      *                                      |
 13      1.973577      | *                                    |
 14      3.663354      |   *                                  |
 15      5.596756      |      *                               |
 16      6.209918      |      *                               |
 17      8.087931      |        *                             |
 18      8.409484      |        *                             |
 19      9.228095      |         *                            |
 20      8.451584      |        *                             |
                       +---------+---------+---------+---------+
```
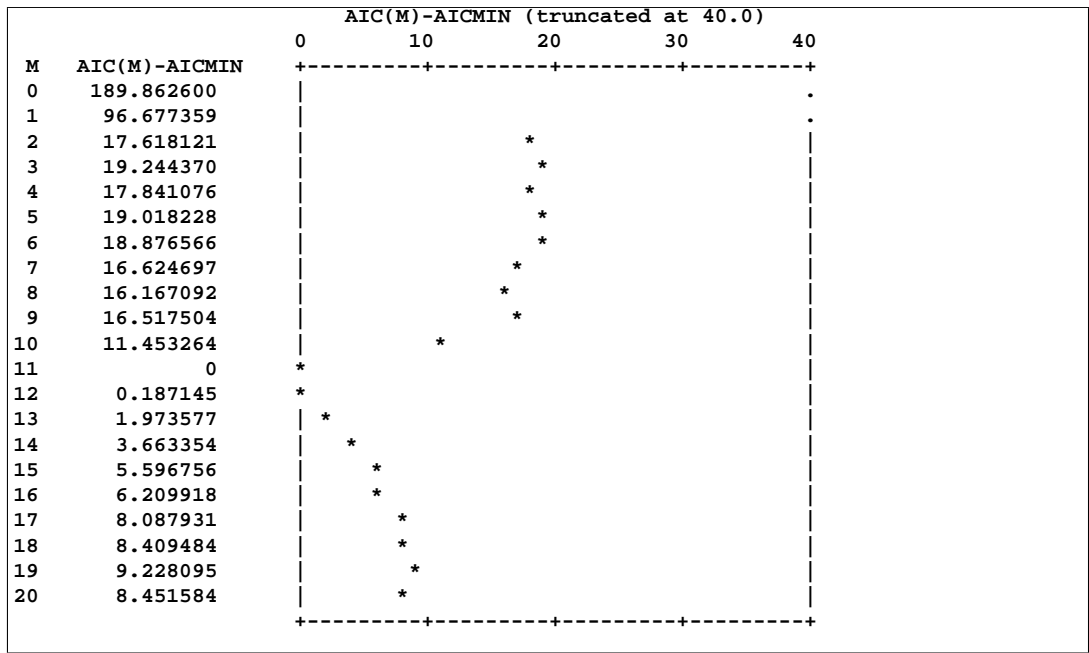
Figure 10.5. Minimum AIC Table - II

The minimum AIC order is selected as 11. Then the coefficients are estimated as in Figure 10.6. Note that the first 20 observations are used as presample values.

```
...........................M  A  I  C  E.........................
.                                                                .
.                                                                .
.                                                                .
.      M      AR Coefficients: AR(M)                             .
.                                                                .
.      1     1.181322                                            .
.      2    -0.551571                                            .
.      3     0.231372                                            .
.      4    -0.178040                                            .
.      5     0.019874                                            .
.      6    -0.062573                                            .
.      7     0.028569                                            .
.      8    -0.050710                                            .
.      9     0.199896                                            .
.     10     0.161819                                            .
.     11    -0.339086                                            .
.                                                                .
.                                                                .
.      AIC =   -298.1301326                                      .
.      Innovation Variance =    0.033186                         .
.                                                                .
.                                                                .
.            INPUT DATA   START =    21  FINISH =    114         .
.................................................................
```

Figure 10.6.   Minimum AIC Estimation

You can estimate the AR(11) model directly by specifying OPT={−1 0} and using the first 11 observations as presample values. The AR(11) estimates shown in Figure 10.7 are different from the minimum AIC estimates in Figure 10.6 because the samples are slightly different.

```
call tsunimar(arcoef,ev,nar,aic,y,11,{-1 0},,1);
```

```
........................M  A  I  C  E........................
.                                                            .
.                                                            .
.                                                            .
.        M      AR Coefficients: AR(M)                       .
.                                                            .
.        1       1.149416                                    .
.        2      -0.533719                                    .
.        3       0.276312                                    .
.        4      -0.326420                                    .
.        5       0.169336                                    .
.        6      -0.164108                                    .
.        7       0.073123                                    .
.        8      -0.030428                                    .
.        9       0.151227                                    .
.       10       0.192808                                    .
.       11      -0.340200                                    .
.                                                            .
.                                                            .
.        AIC =   -318.7984105                                .
.        Innovation Variance =     0.036563                  .
.                                                            .
.                                                            .
.               INPUT DATA   START =    12  FINISH =   114   .
............................................................
```

Figure 10.7. AR(11) Estimation

The minimum AIC procedure can also be applied to the vector autoregressive (VAR) model using the TSMULMAR subroutine. See the section "Multivariate Time Series Analysis" on page 277 for details. Three variables are used as input. The maximum lag is specified as 10.

```
data one;
   input invest income consum @@;
datalines;
   . . . data lines omitted . . .
   ;
proc iml;
   use one;
   read all into y var{invest income consum};
   mdel  = 1;
   maice = 2;
   misw  = 0;  /* instantaneous modeling ? */
   opt   = mdel ||  maice || misw;
   maxlag = 10;
   miss   = 0;
   print  = 1;
   call tsmulmar(arcoef,ev,nar,aic,y,maxlag,opt,miss,print);
```

The VAR(3) model minimizes the AIC and was selected as an appropriate model (see Figure 10.8). However, AICs of the VAR(4) and VAR(5) models show little difference from VAR(3). You can also choose VAR(4) or VAR(5) as an appropriate model in the context of minimum AIC since this AIC difference is much less than 1.

```
ORDER   INNOVATION VARIANCE
  M        LOG(|V(M)|)          AIC(M)
  0        25.98001095    2136.36089828
  1        15.70406486    1311.73331883
  2        15.48896746    1312.09533158
  3        15.18567834    1305.22562428
  4        14.96865183    1305.42944974
  5        14.74838535    1305.36759889
  6        14.60269347    1311.42086432
  7        14.54981887    1325.08514729
  8        14.38596333    1329.64899297
  9        14.16383772    1329.43469312
 10        13.85377849    1322.00983656


                         AIC(M)-AICMIN (truncated at 40.0)
                         0         10        20        30        40
  M    AIC(M)-AICMIN     +---------+---------+---------+---------+
  0     831.135274       |                                       .
  1       6.507695       |      *                                |
  2       6.869707       |      *                                |
  3              0       *                                       |
  4       0.203825       *                                       |
  5       0.141975       *                                       |
  6       6.195240       |      *                                |
  7      19.859523       |                    *                  |
  8      24.423369       |                        *              |
  9      24.209069       |                        *              |
 10      16.784212       |                 *                     |
                         +---------+---------+---------+---------+
```

Figure 10.8.   VAR Model Selection

The TSMULMAR subroutine estimates the instantaneous response model with diagonal error variance. See the section "Multivariate Time Series Analysis" on page 277 for details on the instantaneous response model. Therefore, it is possible to select the minimum AIC model independently for each equation. The best model is selected by specifying MAXLAG=5.

```
call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=5
     opt={1 1 0} print=1;
```

```
                          VARIANCE

                256.64375 29.803549 76.846777
                29.803549 228.97341 119.60387
                76.846777 119.60387 134.21764


                       AR_COEFFICIENT

                13.312109 1.5459098 15.963897
                0.8257397 0.2514803         0
                0.0958916 1.0057088         0
                0.0320985 0.3544346 0.4698934
                 0.044719 -0.201035         0
                0.0051931 -0.023346         0
                0.1169858 -0.060196 0.0483318
                0.1867829         0         0
                0.0216907         0         0
                -0.117786         0 0.3500366
                0.1541108         0         0
                0.0178966         0         0
                0.0461454         0 -0.191437
                -0.389644         0         0
                -0.045249         0         0
                -0.116671         0         0


                          AIC

                       1347.6198
```

Figure 10.9. Model Selection via Instantaneous Response Model

You can print the intermediate results of the minimum AIC procedure using the PRINT=2 option.

Note that the AIC value depends on the MAXLAG=*lag* option and the number of parameters estimated. The minimum AIC VAR estimation procedure (MAICE=2) uses the following AIC formula:

$$(T - lag)\log(|\hat{\Sigma}|) + 2(p \times n^2 + n \times \textit{intercept})$$

where p is the order of the n-variate VAR process, and *intercept*=1 if the intercept is specified; otherwise, *intercept*=0. When you use the MAICE=1 or MAICE=0 option, AIC is computed as the sum of AIC for each response equation. Therefore, there is an AIC difference of $n(n - 1)$ since the instantaneous response model contains the additional $n(n - 1)/2$ response variables as regressors.

The following code estimates the instantaneous response model. The results are shown in Figure 10.10.

```
call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=3
     opt={1 0 0};
print aic nar;
print arcoef;
```

```
                      AIC         NAR

              1403.0762            3


                      ARCOEF

          4.8245814 5.3559216 17.066894
          0.8855926 0.3401741 -0.014398
          0.1684523 1.0502619  0.107064
          0.0891034 0.4591573 0.4473672
          -0.059195 -0.298777 0.1629818
          0.1128625 -0.044039 -0.088186
          0.1684932 -0.025847 -0.025671
          0.0637227 -0.196504 0.0695746
          -0.226559 0.0532467 -0.099808
          -0.303697 -0.139022 0.2576405
```

Figure 10.10. AIC from Instantaneous Response Model

The following code estimates the VAR model. The results are shown in Figure 10.11.

```
call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=3
     opt={1 2 0};
print aic nar;
print arcoef;
```

```
                      AIC         NAR

              1397.0762            3


                      ARCOEF

          4.8245814 5.3559216 17.066894
          0.8855926 0.3401741 -0.014398
          0.1684523 1.0502619  0.107064
          0.0891034 0.4591573 0.4473672
          -0.059195 -0.298777 0.1629818
          0.1128625 -0.044039 -0.088186
          0.1684932 -0.025847 -0.025671
          0.0637227 -0.196504 0.0695746
          -0.226559 0.0532467 -0.099808
          -0.303697 -0.139022 0.2576405
```

Figure 10.11. AIC from VAR Model

The AIC computed from the instantaneous response model is greater than that obtained from the VAR model estimation by 6. There is a discrepancy between Figure 10.11 and Figure 10.8 because different observations are used for estimation.

Nonstationary Data Analysis

This example shows how to manage nonstationary data using TIMSAC calls. In practice, time series are considered to be stationary when the expected values of first and second moments of the series do not change over time. This weak or covariance stationarity can be modeled using the TSMLOCAR, TSMLOMAR, TSDECOMP, and TSTVCAR subroutines.

First, the locally stationary model is estimated. The whole series (1000 observations) is divided into three blocks of size 300 and one block of size 90, and the minimum AIC procedure is applied to each block of the data set. See the "Nonstationary Time Series" section on page 274 for more details.

```
data one;
   input y @@;
datalines;
   . . . data lines omitted . . .
   ;

proc iml;
   use one;
   read all var{y};

   mdel = -1;
   lspan = 300; /* local span of data */
   maice = 1;
   opt = mdel || lspan || maice;
   call tsmlocar(arcoef,ev,nar,aic,first,last)
                data=y maxlag=10 opt=opt print=2;
```

Estimation results are displayed with the graphs of power spectrum $(\log 10(f_{YY}(g)))$, where $f_{YY}(g)$ is a rational spectral density function. See the "Spectral Analysis" section on page 279. As the first block and the second block do not have any sizable difference, the pooled model (AIC=45.892) is selected instead of the moving model (AIC=46.957) in Figure 10.13. However, you can notice a slight change in the shape of the spectrum of the third block of the data (observations 611 through 910). See Figure 10.14 on page 252 and Figure 10.16 on page 254 for comparison. The moving model is selected since the AIC (106.830) of the moving model is smaller than that of the pooled model (108.867).

```
INITIAL LOCAL MODEL: N_CURR =    300
                     NAR_CURR =   8
                          AIC =    37.583203



.........................CURRENT MODEL.........................
.                                                             .
.                                                             .
.                                                             .
.     M      AR Coefficients: AR(M)                           .
.                                                             .
.     1      1.605717                                         .
.     2     -1.245350                                         .
.     3      1.014847                                         .
.     4     -0.931554                                         .
.     5      0.394230                                         .
.     6     -0.004344                                         .
.     7      0.111608                                         .
.     8     -0.124992                                         .
.                                                             .
.                                                             .
.      AIC =      37.5832030                                  .
.      Innovation Variance =    1.067455                      .
.                                                             .
.                                                             .
.              INPUT DATA   START =     11  FINISH =    310   .
..............................................................
```

Figure 10.12. Locally Stationary Model for First Block

```
   --- THE FOLLOWING TWO MODELS ARE COMPARED ---

   MOVING MODEL:    (N_PREV =  300, N_CURR = 300)
                 NAR_CURR =  7
                      AIC =  46.957398
  CONSTANT MODEL: N_POOLED =  600
                NAR_POOLED =  8
                      AIC =  45.892350

   *****  CONSTANT MODEL ADOPTED  *****




.........................CURRENT MODEL.........................
.                                                            .
.                                                            .
.                                                            .
.                                                            .
.    M     AR Coefficients: AR(M)                            .
.                                                            .
.    1     1.593890                                          .
.    2    -1.262379                                          .
.    3     1.013733                                          .
.    4    -0.926052                                          .
.    5     0.314480                                          .
.    6     0.193973                                          .
.    7    -0.058043                                          .
.    8    -0.078508                                          .
.                                                            .
.                                                            .
.     AIC =     45.8923501                                   .
.     Innovation Variance =    1.047585                      .
.                                                            .
.                                                            .
.          INPUT DATA   START =    11  FINISH =   610        .
..............................................................
```

Figure 10.13. Locally Stationary Model Comparison

```
              POWER SPECTRAL DENSITY
  20.00+
       |
       |
       |
       |
       |   XXXX
     XXX      XX     XXX
       |        XXXX
       |            X
       |
  10.00+
       |            X
       |
       |             X
       |
       |              X                XX
       |                                X
       |               X               X
       |
       |              X          X    X
    0+                X
       |                X       X      X
       |                 XX    XX
       |                  XXXX            X
       |
       |                                X
       |                                 X
       |
       |                                  X
       |                                   X
  -10.0+                                   X
       |                                   XX
       |                                    XX
       |                                     XX
       |                                      XXX
       |                                       XXXXXX
       |
       |
       |
  -20.0+----------+----------+----------+----------+----------+
       0.0        0.1        0.2        0.3        0.4        0.5
                              FREQUENCY
```
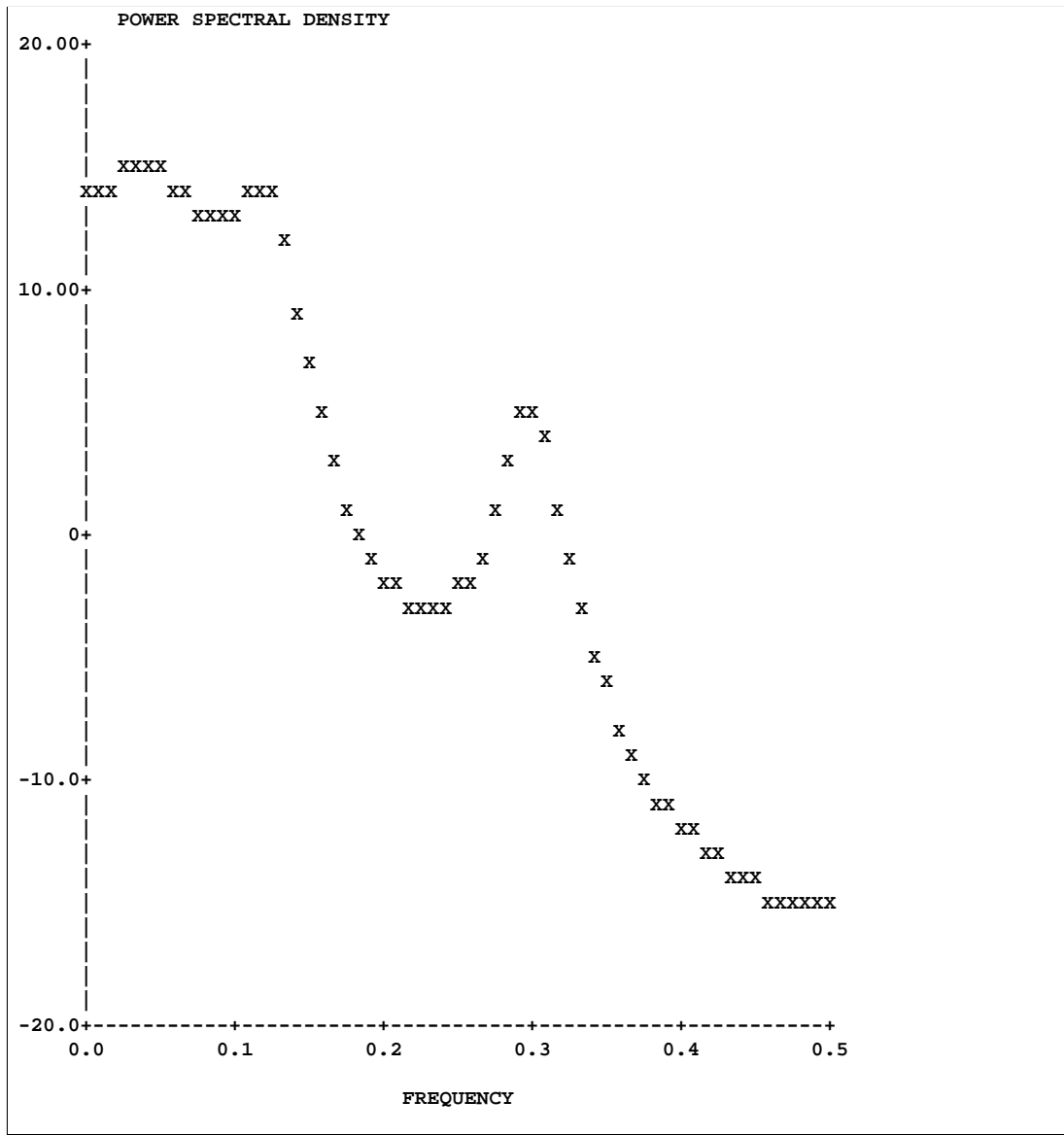
Figure 10.14. Power Spectrum for First and Second Blocks

```
--- THE FOLLOWING TWO MODELS ARE COMPARED ---

     MOVING MODEL: (N_PREV =  600, N_CURR = 300)
                 NAR_CURR =    7
                     AIC =  106.829869
     CONSTANT MODEL: N_POOLED =  900
                 NAR_POOLED =    8
                       AIC =  108.867091


***********************************
*****                       *****
*****     NEW MODEL ADOPTED     *****
*****                       *****
***********************************



.........................CURRENT MODEL.........................
.                                                             .
.                                                             .
.                                                             .
.     M      AR Coefficients: AR(M)                           .
.                                                             .
.     1     1.648544                                          .
.     2    -1.201812                                          .
.     3     0.674933                                          .
.     4    -0.567576                                          .
.     5    -0.018924                                          .
.     6     0.516627                                          .
.     7    -0.283410                                          .
.                                                             .
.                                                             .
.      AIC =      60.9375188                                  .
.      Innovation Variance =    1.161592                      .
.                                                             .
.                                                             .
.            INPUT DATA   START =   611  FINISH =    910      .
...............................................................
```

Figure 10.15. Locally Stationary Model for Third Block
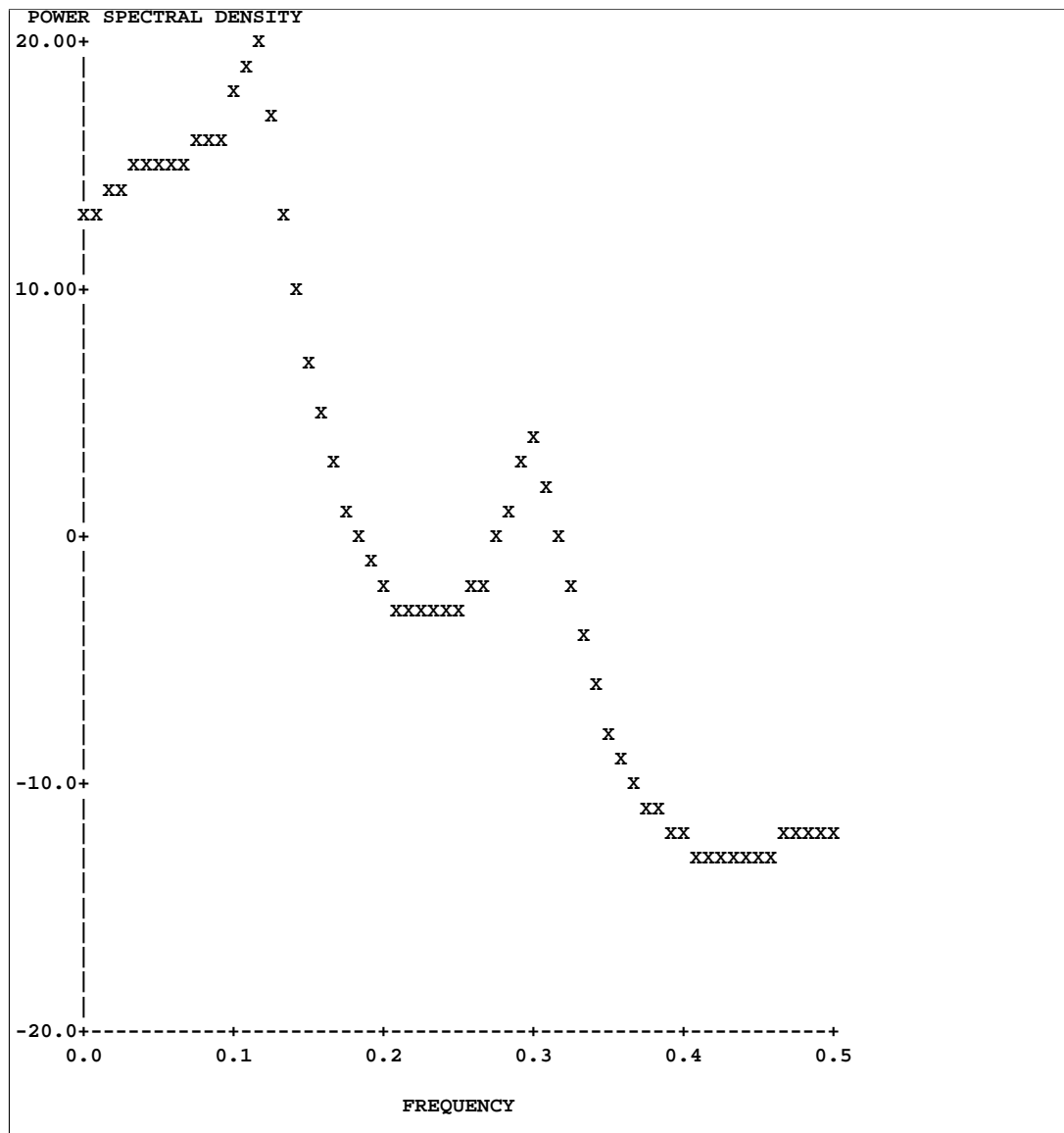
Figure 10.16. Power Spectrum for Third Block

Finally, the moving model is selected since there is a structural change in the last block of data (observations 911 through 1000). The final estimates are stored in variables ARCOEF, EV, NAR, AIC, FIRST, and LAST. The final estimates and spectrum are given in Figure 10.17 and Figure 10.18, respectively. The power spectrum of the final model (Figure 10.18) is significantly different from that of the first and second blocks (see Figure 10.14).

```
   --- THE FOLLOWING TWO MODELS ARE COMPARED ---

      MOVING MODEL:    (N_PREV =  300, N_CURR =  90)
                    NAR_CURR =    6
                         AIC =  139.579012
      CONSTANT MODEL: N_POOLED = 390
                  NAR_POOLED =   9
                         AIC =  167.783711


************************************
*****                        *****
*****    NEW MODEL ADOPTED    *****
*****                        *****
************************************




.........................CURRENT MODEL.........................
.                                                            .
.                                                            .
.                                                            .
.     M     AR Coefficients: AR(M)                           .
.                                                            .
.     1    1.181022                                          .
.     2   -0.321178                                          .
.     3   -0.113001                                          .
.     4   -0.137846                                          .
.     5   -0.141799                                          .
.     6    0.260728                                          .
.                                                            .
.                                                            .
.     AIC =     78.6414932                                   .
.     Innovation Variance =   2.050818                       .
.                                                            .
.                                                            .
.           INPUT DATA   START =   911  FINISH =  1000       .
..............................................................
```

Figure 10.17. Locally Stationary Model for Last Block
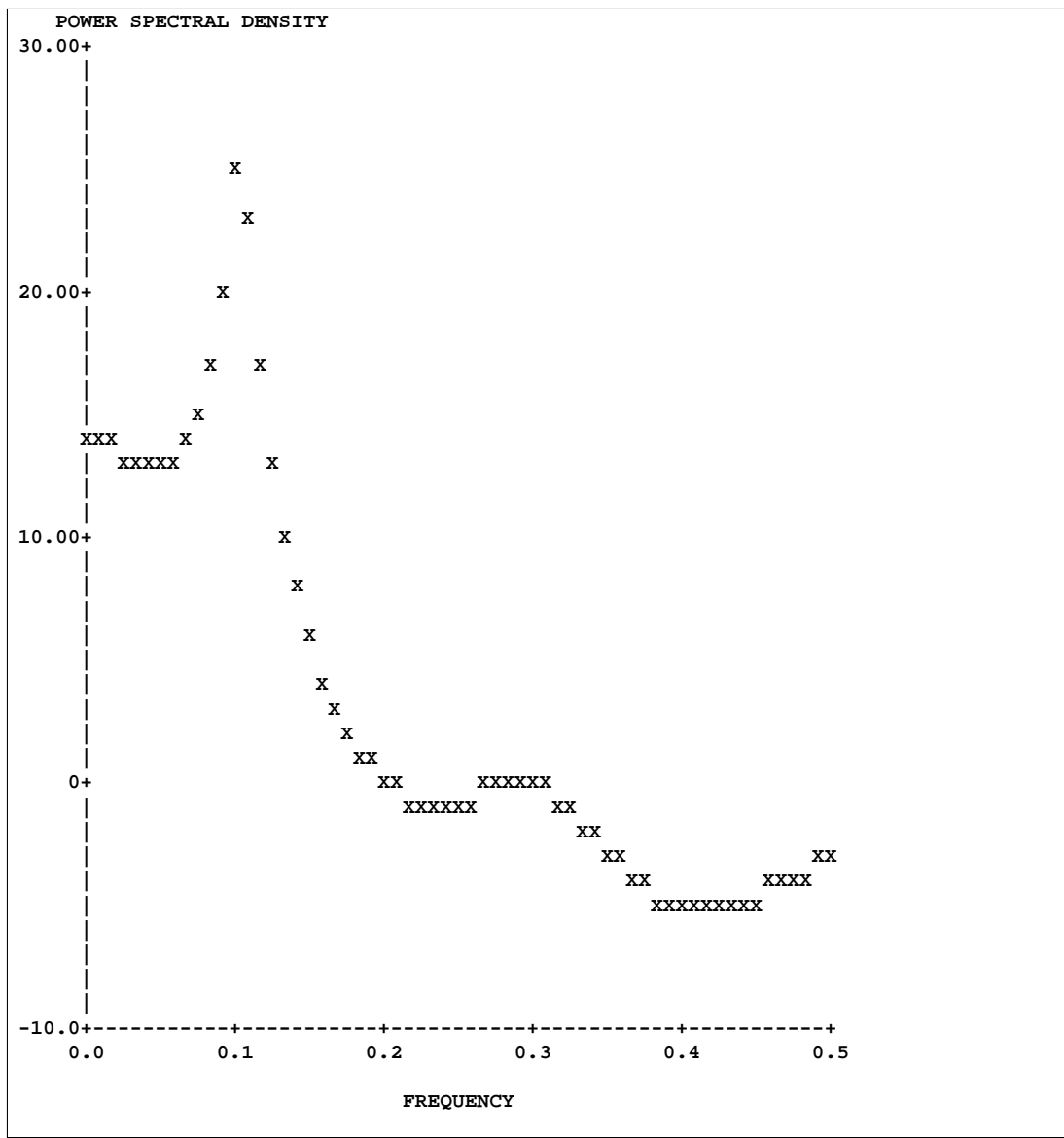
Figure 10.18.   Power Spectrum for Last Block

The multivariate analysis for locally stationary data is a straightforward extension of
the univariate analysis. The bivariate locally stationary VAR models are estimated.
The selected model is the VAR(7) process with some zero coefficients over the last
block of data. There seems to be a structural difference between observations from
11 to 610 and those from 611 to 896.

```
proc iml;
   rudder = {. . . data lines omitted . . .};
   yawing = {. . . data lines omitted . . .};

   y = rudder` || yawing`;
   c = {0.01795 0.02419};
```

```
/*-- calibration of data --*/
 y = y # (c @ j(n,1,1));
 mdel = -1;
 lspan = 300; /* local span of data */
 maice = 1;
 call tsmlomar(arcoef,ev,nar,aic,first,last) data=y maxlag=10
       opt = (mdel || lspan || maice) print=1;
```

```
--- THE FOLLOWING TWO MODELS ARE COMPARED ---

MOVING MODEL:    (N_PREV =  600, N_CURR = 286)
             NAR_CURR =    7
                  AIC = -823.845234
CONSTANT MODEL: N_POOLED =  886
           NAR_POOLED =   10
                  AIC = -716.818588


************************************
*****                        *****
*****     NEW MODEL ADOPTED     *****
*****                        *****
************************************



.........................CURRENT MODEL.........................
.                                                            .
.                                                            .
.                                                            .
.                                                            .
.    M      AR Coefficients                                  .
.                                                            .
.    1    0.932904    -0.130964                              .
.        -0.024401     0.599483                              .
.    2    0.163141     0.266876                              .
.        -0.135605     0.377923                              .
.    3   -0.322283     0.178194                              .
.         0.188603    -0.081245                              .
.    4    0.166094    -0.304755                              .
.        -0.084626    -0.180638                              .
.    5         0            0                                .
.               0    -0.036958                               .
.    6         0            0                                .
.               0     0.034578                               .
.    7         0            0                                .
.               0     0.268414                               .
.                                                            .
.                                                            .
.     AIC =   -114.6911872                                   .
.                                                            .
.          Innovation Variance                               .
.                                                            .
.          1.069929     0.145558                             .
.          0.145558     0.563985                             .
.                                                            .
.                                                            .
.            INPUT DATA   START =   611  FINISH =   896       .
.............................................................
```

Figure 10.19. Locally Stationary VAR Model Analysis

Consider the time series decomposition

$$y_t = T_t + S_t + u_t + \epsilon_t$$

where T_t and S_t are trend and seasonal components, respectively, and u_t is a stationary AR(p) process. The annual real GNP series is analyzed under second difference stochastic constraints on the trend component and the stationary AR(2) process.

$$T_t = 2T_{t-1} - T_{t-2} + w_{1t}$$
$$u_t = \alpha_1 u_{t-1} + \alpha_2 u_{t-2} + w_{2t}$$

The seasonal component is ignored if you specify SORDER=0. Therefore, the following state space model is estimated:

$$y_t = \mathbf{H}\mathbf{z}_t + \epsilon_t$$
$$\mathbf{z}_t = \mathbf{F}\mathbf{z}_{t-1} + \mathbf{w}_t$$

where

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{F} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & \alpha_1 & \alpha_2 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{z}_t = (T_t, T_{t-1}, u_t, u_{t-1})'$$

$$\mathbf{w}_t = (w_{1t}, 0, w_{2t}, 0)' \sim \left(0, \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_2^2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right)$$

The parameters of this state space model are σ_1^2, σ_2^2, α_1, and α_2.

```
proc iml;
   y = { 116.8 120.1 123.2 130.2 131.4 125.6 124.5 134.3
         135.2 151.8 146.4 139.0 127.8 147.0 165.9 165.5
         179.4 190.0 189.8 190.9 203.6 183.5 169.3 144.2
         141.5 154.3 169.5 193.0 203.2 192.9 209.4 227.2
         263.7 297.8 337.1 361.3 355.2 312.6 309.9 323.7
         324.1 355.3 383.4 395.1 412.8 406.0 438.0 446.1
         452.5 447.3 475.9 487.7 497.2 529.8 551.0 581.1
         617.8 658.1 675.2 706.6 724.7 };
   y = y'; /*-- convert to column vector --*/
   mdel  = 0;
   trade = 0;
   tvreg = 0;
   year  = 0;
```

```
    period= 0;
    log   = 0;
    maxit = 100;
    update = .; /* use default update method      */
    line   = .; /* use default line search method */
    sigmax = 0; /* no upper bound for variances   */
    back = 100;
    opt = mdel || trade || year || period || log || maxit ||
          update || line || sigmax || back;
    call tsdecomp(cmp,coef,aic) data=y order=2 sorder=0 nar=2
         npred=5 opt=opt icmp={1 3} print=1;
    y = y[52:61];
    cmp = cmp[52:66,];
    print y cmp;
```

The estimated parameters are printed when you specify the PRINT= option. In Figure 10.20, the estimated variances are printed under the title of TAU2(I), showing that $\hat{\sigma}_1^2 = 2.915$ and $\hat{\sigma}_2^2 = 113.9577$. The AR coefficient estimates are $\hat{\alpha}_1 = 1.397$ and $\hat{\alpha}_2 = -0.595$. These estimates are also stored in the output matrix COEF.

```
                  <<<  Final Estimates  >>>

              ---  PARAMETER VECTOR  ---

 1.607426E-01 6.283837E+00 8.761628E-01 -5.94879E-01

                  ---  GRADIENT  ---

 3.385021E-04 5.760929E-06 3.029534E-04 -1.18396E-04


 LIKELIHOOD = -249.937193    SIG2 =       18.135052
  AIC        =  509.874385


 I   TAU2(I)       AR(I)       PARCOR(I)
 1     2.915075     1.397374      0.876163
 2   113.957712    -0.594879     -0.594879
```

Figure 10.20. Nonstationary Time Series and State Space Modeling

The trend and stationary AR components are estimated using the smoothing method, and out-of-sample forecasts are computed using a Kalman filter prediction algorithm. The trend and AR components are stored in the matrix CMP since the ICMP={1 3} option is specified. The last 10 observations of the original series Y and the last 15 observations of two components are shown in Figure 10.21. Note that the first column of CMP is the trend component and the second column is the AR component. The last 5 observations of the CMP matrix are out-of-sample forecasts.

```
                         Y        CMP

                     487.7 514.01142 -26.94343
                     497.2 532.62744 -32.48674
                     529.8 552.02403 -24.46593
                       551 571.90122 -20.15113
                     581.1 592.31944 -10.58647
                     617.8 613.21855 5.2504378
                     658.1 634.43665 20.799213
                     675.2 655.70431 22.161602
                     706.6 677.21249 27.927985
                     724.7 698.72363   25.95797
                           720.23477 19.659209
                           741.74591 12.029407
                           763.25705 5.1147239
                           784.76819 -0.008863
                           806.27933 -3.055027
```

Figure 10.21. Smoothed and Predicted Values of Two Components

Seasonal Adjustment

Consider the simple time series decomposition

$$y_t = T_t + S_t + \epsilon_t$$

The TSBAYSEA subroutine computes seasonally adjusted series by estimating the seasonal component. The seasonally adjusted series is computed as $y_t^* = y_t - \hat{S}_t$. The details of the adjustment procedure are given in the section "Bayesian Seasonal Adjustment" on page 271.

The monthly labor force series (1972–1978) are analyzed. You do not need to specify the options vector if you want to use the default options. However, you should change OPT[2] when the data frequency is not monthly (OPT[2]=12). The NPRED= option produces the multistep forecasts for the trend and seasonal components. The stochastic constraints are specified as ORDER=2 and SORDER=1.

$$\begin{aligned} T_t &= 2T_{t-1} - T_{t-2} + w_{1t} \\ S_t &= -S_{t-1} - \cdots - S_{t-11} + w_{2t} \end{aligned}$$

In Figure 10.22, the first column shows the trend components; the second column shows the seasonal components; the third column shows the forcasts; the forth column shows the seasonally adjusted series; the last shows the value of ABIC. The last 12 rows are the forcasts.

```
proc iml;
   y = { 5447 5412 5215 4697 4344 5426
         5173 4857 4658 4470 4268 4116
         4675 4845 4512 4174 3799 4847
         4550 4208 4165 3763 4056 4058
```

```
        5008 5140 4755 4301 4144 5380
        5260 4885 5202 5044 5685 6106
        8180 8309 8359 7820 7623 8569
        8209 7696 7522 7244 7231 7195
        8174 8033 7525 6890 6304 7655
        7577 7322 7026 6833 7095 7022
        7848 8109 7556 6568 6151 7453
        6941 6757 6437 6221 6346 5880 };
Y = Y`;

call tsbaysea(trend,season,series,adj,abic)
     data=y order=2 sorder=1 npred=12 print=2;
print trend season series adj abic;
```

```
          OBS     TREND     SEASON     SERIES      ADJ      ABIC

           1 4843.2502 576.86675 5420.1169 4870.1332 874.04585
           2 4848.6664 612.79607 5461.4624 4799.2039
           3 4871.2876 324.02004 5195.3077   4890.98
           4 4896.6633 -198.7601 4697.9032 4895.7601
           5 4922.9458 -572.5562 4350.3896 4916.5562
           .       .         .         .         .
          71 6551.6017 -266.2162 6285.3855 6612.2162
          72 6388.9012 -440.3472 5948.5539 6320.3472
          73 6226.2006   650.7707 6876.9713
          74 6063.5001 800.93733 6864.4374
          75 5900.7995 396.19866 6296.9982
          76  5738.099 -340.2852 5397.8137
          77 5575.3984 -719.1146 4856.2838
          78 5412.6979 553.19764 5965.8955
          79 5249.9973 202.06582 5452.0631
          80 5087.2968 -54.44768 5032.8491
          81 4924.5962 -295.2747 4629.3215
          82 4761.8957 -487.6621 4274.2336
          83 4599.1951 -266.1917 4333.0034
          84 4436.4946 -440.3354 3996.1591
```

Figure 10.22. Trend and Seasonal Component Estimates and Forecasts

The estimated spectral density function of the irregular series $\hat{\epsilon}_t$ is shown in Figure

```
     I    Rational   0.0       10.0      20.0      30.0      40.0      50.0      60.0
          Spectrum  +---------+---------+---------+---------+---------+---------+
     0 1.366798E+00  |*                                                          ===>X
     1 1.571261E+00  |*
     2 2.414836E+00  |   *
     3 5.151906E+00  |       *
     4 1.634887E+01  |           *
     5 8.085674E+01  |              *
     6 3.805530E+02  |                   *
     7 8.082536E+02  |                     *
     8 6.366350E+02  |                     *
     9 3.479435E+02  |                  *
    10 3.872650E+02  |                  *                                         ===>X
    11 1.264805E+03  |                       *
    12 1.726138E+04  |                             *
    13 1.559041E+03  |                    *
    14 1.276516E+03  |                    *
    15 3.861089E+03  |                        *
    16 9.593184E+03  |                          *
    17 3.662145E+03  |                       *
    18 5.499783E+03  |                         *
    19 4.443303E+03  |                         *
    20 1.238135E+03  |                     *                                      ===>X
    21 8.392131E+02  |                    *
    22 1.258933E+03  |                     *
    23 2.932003E+03  |                       *
    24 1.857923E+03  |                      *
    25 1.171437E+03  |                      *
    26 1.611958E+03  |                      *
    27 4.822498E+03  |                         *
    28 4.464961E+03  |                         *
    29 1.951547E+03  |                     *
    30 1.653182E+03  |                     *                                      ===>X
    31 2.308152E+03  |                      *
    32 5.475758E+03  |                        *
    33 2.349584E+04  |                           *
    34 5.266969E+03  |                        *
    35 2.058667E+03  |                    *
    36 2.215595E+03  |                    *
    37 8.181540E+03  |                        *
    38 3.077329E+03  |                      *
    39 7.577961E+02  |                 *
    40 5.057636E+02  |                   *                                        ===>X
    41 7.312090E+02  |                   *
    42 3.131377E+03  |                      *                                     ===>T
    43 8.173276E+03  |                        *
    44 1.958359E+03  |                     *
    45 2.216458E+03  |                     *
    46 4.215465E+03  |                       *
    47 9.659340E+02  |                  *
    48 3.758466E+02  |                *
    49 2.849326E+02  |                *
    50 3.617848E+02  |                 *                                          ===>X
    51 7.659839E+02  |                *
    52 3.191969E+03  |                       *
    53 1.768107E+04  |                            *
    54 5.281385E+03  |                         *
    55 2.959704E+03  |                      *
    56 3.783522E+03  |                      *
    57 1.896625E+04  |                           *
    58 1.041753E+04  |                          *
    59 2.038940E+03  |                    *
    60 1.347568E+03  |                     *                                      ===>X

 X: If peaks (troughs) appear at these frequencies, try lower (higher) values
    of rigid and watch ABIC
 T: If a peaks appears here try trading-day adjustment
```

10.23.

Figure 10.23. Spectrum of Irregular Component

Miscellaneous Time Series Analysis Tools

The forecast values of multivariate time series are computed using the TSPRED call. In this example, the multistep ahead forecasts are produced from the VARMA(2,1) estimates. Since the VARMA model is estimated using the mean deleted series, you should specify the CONSTANT=-1 option. You need to provide the original series instead of the mean deleted series to get the correct predictions. The forecast variance MSE and the impulse response function IMPULSE are also produced.

The VARMA(p, q) model is written

$$\mathbf{y}_t + \sum_{i=1}^{p} \mathbf{A}_i \mathbf{y}_{t-i} = \epsilon_t + \sum_{i=1}^{q} \mathbf{M}_i \epsilon_{t-i}$$

Then the COEF matrix is constructed by stacking matrices $\mathbf{A}_1, \ldots, \mathbf{A}_p, \mathbf{M}_1, \ldots, \mathbf{M}_q$.

```
proc iml;
  c = { 264 235 239 239 275 277 274 334 334 306
        308 309 295 271 277 221 223 227 215 223
        241 250 270 303 311 307 322 335 335 334
        309 262 228 191 188 215 215 249 291 296 };
  f = { 690 690 688 690 694 702 702 702 700 702
        702 694 708 702 702 708 700 700 702 694
        698 694 700 702 700 702 708 708 710 704
        704 700 700 694 702 694 710 710 710 708 };
  t = { 1152 1288 1288 1288 1368 1456 1656 1496 1744 1464
        1560 1376 1336 1336 1296 1296 1280 1264 1280 1272
        1344 1328 1352 1480 1472 1600 1512 1456 1368 1280
        1224 1112 1112 1048 1176 1064 1168 1280 1336 1248 };
  p = { 254.14 253.12 251.85 250.41 249.09 249.19 249.52 250.19
        248.74 248.41 249.95 250.64 250.87 250.94 250.96 251.33
        251.18 251.05 251.00 250.99 250.79 250.44 250.12 250.19
        249.77 250.27 250.74 250.90 252.21 253.68 254.47 254.80
        254.92 254.96 254.96 254.96 254.96 254.54 253.21 252.08 };

  y = c` || f` || t` || p`;
  ar = {   .82028    -.97167      .079386    -5.4382,
          -.39983     .94448      .027938    -1.7477,
          -.42278   -2.3314      1.4682     -70.996,
           .031038   -.019231    -.0004904    1.3677,
          -.029811    .89262     -.047579     4.7873,
           .31476     .0061959   -.012221     1.4921,
           .3813     2.7182       -.52993     67.711,
          -.020818    .01764       .00037981  -.38154 };
  ma = {   .083035  -1.0509       .055898    -3.9778,
          -.40452     .36876      .026369     -.81146,
           .062379  -2.6506       .80784     -76.952,
           .03273    -.031555    -.00019776   -.025205 };
  coef = ar // ma;
  ev = { 188.55    6.8082      42.385     .042942,
          6.8082   32.169      37.995    -.062341,
```

```
        42.385    37.995     5138.8  -.10757,
        .042942  -.062341   -.10757   .34313 };

nar = 2; nma = 1;
call tspred(forecast,impulse,mse,y,coef,nar,nma,ev,
            5,nrow(y),-1);
```

OBSERVED		PREDICTED	
Y1	Y2	P1	P2
264	690	269.950	700.750
235	690	256.764	691.925
239	688	239.996	693.467
239	690	242.320	690.951
275	694	247.169	693.214
277	702	279.024	696.157
274	702	284.041	700.449
334	702	286.890	701.580
334	700	321.798	699.851
306	702	330.355	702.383
308	702	297.239	700.421
309	694	302.651	701.928
295	708	294.570	696.261
271	702	283.254	703.936
277	702	269.600	703.110
221	708	270.349	701.557
223	700	231.523	705.438
227	700	233.856	701.785
215	702	234.883	700.185
223	694	229.156	701.837
241	698	235.054	697.060
250	694	249.288	698.181
270	700	257.644	696.665
303	702	272.549	699.281
311	700	301.947	701.667
307	702	306.422	700.708
322	708	304.120	701.204
335	708	311.590	704.654
335	710	320.570	706.389
334	704	315.127	706.439
309	704	311.083	703.735
262	700	288.159	702.801
228	700	251.352	700.805
191	694	226.749	700.247
188	702	199.775	696.570
215	694	202.305	700.242
215	710	222.951	696.451
249	710	226.553	704.483
291	710	259.927	707.610
296	708	291.446	707.861
		293.899	707.430
		293.477	706.933
		292.564	706.190
		290.313	705.384
		286.559	704.618

Figure 10.24. Multivariate ARMA Prediction

The first 40 forecasts are one-step predictions. The last observation is the five-step forecast values of variables C and F. You can construct the confidence interval for these forecasts using the mean square error matrix, MSE. See the "Multivariate Time

Series Analysis" section on page 277 for more details on impulse response functions and the mean square error matrix.

The TSROOT call computes the polynomial roots of the AR and MA equations. When the AR(p) process is written

$$y_t = \sum_{i=1}^{p} \alpha_i y_{t-i} + \epsilon_t$$

you can specify the following polynomial equation:

$$z^p - \sum_{i=1}^{p} \alpha_i z^{p-i} = 0$$

When all p roots of the preceding equation are inside the unit circle, the AR(p) process is stationary. The MA(q) process is invertible if the following polynomial equation has all roots inside the unit circle:

$$z^q + \sum_{i=1}^{q} \theta_i z^{q-i} = 0$$

where θ_i are the MA coefficients. For example, the best AR model is selected and estimated by the TSUNIMAR subroutine (see Figure 10.25). You can obtain the roots of the preceding equation by calling the TSROOT call. Since the TSROOT call can handle the complex AR or MA coefficients, note that you should add zero imaginary coefficients for the second column of the MATIN matrix for real coefficients.

```
proc iml;
   y = { 2.430 2.506 2.767 2.940 3.169 3.450 3.594 3.774 3.695 3.411
         2.718 1.991 2.265 2.446 2.612 3.359 3.429 3.533 3.261 2.612
         2.179 1.653 1.832 2.328 2.737 3.014 3.328 3.404 2.981 2.557
         2.576 2.352 2.556 2.864 3.214 3.435 3.458 3.326 2.835 2.476
         2.373 2.389 2.742 3.210 3.520 3.828 3.628 2.837 2.406 2.675
         2.554 2.894 3.202 3.224 3.352 3.154 2.878 2.476 2.303 2.360
         2.671 2.867 3.310 3.449 3.646 3.400 2.590 1.863 1.581 1.690
         1.771 2.274 2.576 3.111 3.605 3.543 2.769 2.021 2.185 2.588
         2.880 3.115 3.540 3.845 3.800 3.579 3.264 2.538 2.582 2.907
         3.142 3.433 3.580 3.490 3.475 3.579 2.829 1.909 1.903 2.033
         2.360 2.601 3.054 3.386 3.553 3.468 3.187 2.723 2.686 2.821
         3.000 3.201 3.424 3.531 };

   call tsunimar(ar,v,nar,aic) data=y maxlag=5
        opt=({-1 1}) print=1;
   /*-- set up complex coefficient matrix --*/
   ar_cx = ar || j(nrow(ar),1,0);
   call tsroot(root) matin=ar_cx nar=nar nma=0 print=1;
```

In Figure 10.26, the roots and their lengths from the origin are shown. The roots are also stored in the matrix ROOT. All roots are within the unit circle, while the mod values of the fourth and fifth roots appear to be sizable (0.9194).

```
                        LAG    AR_COEF

                        1 1.3003068
                        2  -0.72328
                        3 0.2421928
                        4 -0.378757
                        5 0.1377273
                  AIC INNOVATION_VARINACE

              -318.6138          0.0490554
```

Figure 10.25. Minimum AIC AR Estimation

```
                Roots of AR Characteristic Polynomial

    I          Real      Imaginary      MOD(Z)     ATAN(I/R)        Degree

    1       -0.29755        0.55991      0.6341        2.0593      117.9869
    2       -0.29755       -0.55991      0.6341       -2.0593     -117.9869
    3        0.40529              0      0.4053             0             0
    4        0.74505        0.53866      0.9194        0.6260       35.8660
    5        0.74505       -0.53866      0.9194       -0.6260      -35.8660

       Z**5-AR(1)*Z**4-AR(2)*Z**3-AR(3)*Z**2-AR(4)*Z**1-AR(5)=0
```

Figure 10.26. Roots of AR Characteristic Polynomial Equation

The TSROOT call can also recover the polynomial coefficients if the roots are given as an input. You should specify the QCOEF=1 option when you want to compute the polynomial coefficients instead of polynomial roots. You can compare the result with the preceding output of the TSUNIMAR call.

```
    call tsroot(ar_cx) matin=root nar=nar qcoef=1
                    nma=0 print=1;
```

```
                  Polynomial Coefficents

            I        AR(real)        AR(imag)

            1         1.30031               0
            2        -0.72328      5.55112E-17
            3         0.24219      1.61885E-16
            4        -0.37876               0
            5         0.13773     -4.1674E-18
```

Figure 10.27. Polynomial Coefficients

Syntax

TIMSAC routines are controlled by the following statements:

CALL TSBAYSEA(*trend, season, series, adjust, abic, data*

 $<$,*order, sorder, rigid, npred, opt, cntl, print*$>$**)**;

CALL TSDECOMP(*comp, est, aic, data* $<$,*xdata, order, sorder,*

 nar, npred, init, opt, icmp, print$>$**)**;

CALL TSMLOCAR(*arcoef, ev, nar, aic, start, finish, data*

 $<$,*maxlag, opt, missing, print*$>$**)**;

CALL TSMLOMAR(*arcoef, ev, nar, aic, start, finish, data*

 $<$,*maxlag, opt, missing, print*$>$**)**;

CALL TSMULMAR(*arcoef, ev, nar, aic, data*

 $<$,*maxlag, opt, missing, print*$>$**)**;

CALL TSPEARS(*arcoef, ev, nar, aic, data*

 $<$,*maxlag, opt, missing, print*$>$**)**;

CALL TSPRED(*forecast, impulse, mse, data, coef, nar, nma*

 $<$,*ev, npred, start, constant*$>$**)**;

CALL TSROOT(*matout, matin, nar, nma* $<$,*qcoef, print*$>$**)**;

CALL TSTVCAR(*arcoef, variance, est, aic, data*

 $<$,*nar, init, opt, outlier, print*$>$**)**;

CALL TSUNIMAR(*arcoef, ev, nar, aic, data*

 $<$,*maxlag, opt, missing, print*$>$**)**;

Details

This section presents an introductory description of the important topics that are directly related to TIMSAC IML subroutines. The computational details, including algorithms, are described in the "Computational Details" section on page 282. A detailed explanation of each subroutine is not given; instead, basic ideas and common methodologies for all subroutines are described first and are followed by more technical details. Finally, missing values are discussed in the section "Missing Values" on page 288.

Minimum AIC Procedure

The AIC statistic is widely used to select the best model among alternative parametric models. The minimum AIC model selection procedure can be interpreted as a max-

imization of the expected entropy (Akaike 1981). The entropy of a true probability density function (PDF) φ with respect to the fitted PDF f is written as

$$B(\varphi, f) = -I(\varphi, f)$$

where $I(\varphi, f)$ is a Kullback-Leibler information measure, which is defined as

$$I(\varphi, f) = \int \left[\log \left[\frac{\varphi(z)}{f(z)} \right] \right] \varphi(z) dz$$

where the random variable Z is assumed to be continuous. Therefore,

$$B(\varphi, f) = \mathrm{E}_Z \log f(Z) - \mathrm{E}_Z \log \varphi(Z)$$

where $B(\varphi, f) \leq 0$ and E_Z denotes the expectation concerning the random variable Z. $B(\varphi, f) = 0$ if and only if $\varphi = f$ (a.s.). The larger the quantity $\mathrm{E}_Z \log f(Z)$, the closer the function f is to the true PDF φ. Given the data $\mathbf{y} = (y_1, \ldots, y_T)'$ that has the same distribution as the random variable Z, let the likelihood function of the parameter vector θ be $\prod_{t=1}^{T} f(y_t|\theta)$. Then the average of the log-likelihood function $\frac{1}{T} \sum_{t=1}^{T} \log f(y_t|\theta)$ is an estimate of the expected value of $\log f(Z)$. Akaike (1981) derived the alternative estimate of $\mathrm{E}_Z \log f(Z)$ by using the Bayesian predictive likelihood. The AIC is the bias-corrected estimate of $-2T\mathrm{E}_Z \log f(Z|\hat{\theta})$, where $\hat{\theta}$ is the maximum likelihood estimate.

$$\mathrm{AIC} = -2(\textit{maximum log-likelihood}) + 2(\textit{number of free parameters})$$

Let $\theta = (\theta_1, \ldots, \theta_K)'$ be a $K \times 1$ parameter vector that is contained in the parameter space Θ_K. Given the data $\mathbf{y}$, the log-likelihood function is

$$\ell(\theta) = \sum_{t=1}^{T} \log f(y_t|\theta)$$

Suppose the probability density function $f(y|\theta)$ has the true PDF $\varphi(y) = f(y|\theta^0)$, where the true parameter vector θ^0 is contained in Θ_K. Let $\hat{\theta}_K$ be a maximum likelihood estimate. The maximum of the log-likelihood function is denoted as $\ell(\hat{\theta}_K) = \max_{\theta \in \Theta_K} \ell(\theta)$. The expected log-likelihood function is defined by

$$\ell^*(\theta) = T\mathrm{E}_Z \log f(Z|\theta)$$

The Taylor series expansion of the expected log likelihood function around the true parameter θ^0 gives the following asymptotic relationship:

$$\ell^*(\theta) \overset{A}{=} \ell^*(\theta^0) + T(\theta - \theta^0)'\mathrm{E}_Z \frac{\partial \log f(Z|\theta^0)}{\partial \theta} - \frac{T}{2}(\theta - \theta^0)'I(\theta^0)(\theta - \theta^0)$$

where $I(\theta^0)$ is the information matrix and $\stackrel{A}{=}$ stands for asymptotic equality. Note that $\frac{\partial \log f(z|\theta^0)}{\partial \theta} = 0$ since $\log f(z|\theta)$ is maximized at θ^0. By substituting $\hat{\theta}_K$, the expected log-likelihood function can be written as

$$\ell^*(\hat{\theta}_K) \stackrel{A}{=} \ell^*(\theta^0) - \frac{T}{2}(\hat{\theta}_K - \theta^0)' I(\theta^0)(\hat{\theta}_K - \theta^0)$$

The maximum likelihood estimator is asymptotically normally distributed under the regularity conditions

$$\sqrt{T} I(\theta^0)^{1/2}(\hat{\theta}_K - \theta^0) \stackrel{d}{\to} N(0, I_K)$$

Therefore,

$$T(\hat{\theta}_K - \theta^0)' I(\theta^0)(\hat{\theta}_K - \theta^0) \stackrel{a}{\sim} \chi_K^2$$

The mean expected log-likelihood function, $\ell^*(K) = \mathrm{E}_Y \ell^*(\hat{\theta}_K)$, becomes

$$\ell^*(K) \stackrel{A}{=} \ell^*(\theta^0) - \frac{K}{2}$$

When the Taylor series expansion of the log likelihood function around $\hat{\theta}_K$ is used, the log-likelihood function $\ell(\theta)$ is written

$$\ell(\theta) \stackrel{A}{=} \ell(\hat{\theta}_K) + (\theta - \hat{\theta}_K)' \left.\frac{\partial \ell(\theta)}{\partial \theta}\right|_{\hat{\theta}_K} + \frac{1}{2}(\theta - \hat{\theta}_K)' \left.\frac{\partial^2 \ell(\theta)}{\partial \theta \partial \theta'}\right|_{\hat{\theta}_K} (\theta - \hat{\theta}_K)$$

Since $\ell(\hat{\theta}_K)$ is the maximum log-likelihood function, $\left.\frac{\partial \ell(\theta)}{\partial \theta}\right|_{\hat{\theta}_K} = 0$. Note that $\mathrm{plim}\left[-\frac{1}{T}\left.\frac{\partial^2 \ell(\theta)}{\partial \theta \partial \theta'}\right|_{\hat{\theta}_K}\right] = I(\theta^0)$ if the maximum likelihood estimator $\hat{\theta}_K$ is a consistent estimator of θ. Replacing θ with the true parameter θ^0 and taking expectations with respect to the random variable Y,

$$\mathrm{E}_Y \ell(\theta^0) \stackrel{A}{=} \mathrm{E}_Y \ell(\hat{\theta}_K) - \frac{K}{2}$$

Consider the following relationship:

$$
\begin{aligned}
\ell^*(\theta^0) &= T \mathrm{E}_Z \log f(Z|\theta^0) \\
&= \mathrm{E}_Y \sum_{t=1}^{T} \log f(Y_t|\theta^0) \\
&= \mathrm{E}_Y \ell(\theta^0)
\end{aligned}
$$

From the previous derivation,

$$\ell^*(K) \overset{A}{=} \ell^*(\theta^0) - \frac{K}{2}$$

Therefore,

$$\ell^*(K) \overset{A}{=} \mathrm{E}_Y \ell(\hat{\theta}_K) - K$$

The natural estimator for $\mathrm{E}_Y \ell(\hat{\theta}_K)$ is $\ell(\hat{\theta}_K)$. Using this estimator, you can write the mean expected log-likelihood function as

$$\ell^*(K) \overset{A}{=} \ell(\hat{\theta}_K) - K$$

Consequently, the AIC is defined as an asymptotically unbiased estimator of $-2(\textit{mean expected log-likelihood})$

$$\mathrm{AIC}(K) = -2\ell(\hat{\theta}_K) + 2K$$

In practice, the previous asymptotic result is expected to be valid in finite samples if the number of free parameters does not exceed $2\sqrt{T}$ and the upper bound of the number of free parameters is $\frac{T}{2}$. It is worth noting that the amount of AIC is not meaningful in itself, since this value is not the Kullback-Leibler information measure. The difference of AIC values can be used to select the model. The difference of the two AIC values is considered insignificant if it is far less than 1. It is possible to find a better model when the minimum AIC model contains many free parameters.

Smoothness Priors Modeling

Consider the time series y_t:

$$y_t = f(t) + \epsilon_t$$

where $f(t)$ is an unknown smooth function and ϵ_t is an iid random variable with zero mean and positive variance σ^2. Whittaker (1923) provides the solution, which balances a tradeoff between closeness to the data and the kth order difference equation. For a fixed value of λ and k, the solution $\hat{f}$ satisfies

$$\min_f \sum_{t=1}^{T} \left\{ [y_t - f(t)]^2 + \lambda^2 [\nabla^k f(t)]^2 \right\}$$

where ∇^k denotes the kth order difference operator. The value of λ can be viewed as the smoothness tradeoff measure. Akaike (1980a) proposed the Bayesian posterior PDF to solve this problem.

$$\ell(f) = \exp\left\{ -\frac{1}{2\sigma^2} \sum_{t=1}^{T} [y_t - f(t)]^2 \right\} \exp\left\{ -\frac{\lambda^2}{2\sigma^2} \sum_{t=1}^{T} [\nabla^k f(t)]^2 \right\}$$

Therefore, the solution can be obtained when the function $\ell(f)$ is maximized.

Assume that time series is decomposed as follows:

$$y_t = T_t + S_t + \epsilon_t$$

where T_t denotes the trend component and S_t is the seasonal component. The trend component follows the kth order stochastically perturbed difference equation.

$$\nabla^k T_t = w_{1t}, \qquad w_{1t} \sim N(0, \tau_1^2)$$

For example, the polynomial trend component for $k = 2$ is written as

$$T_t = 2T_{t-1} - T_{t-2} + w_{1t}$$

To accommodate regular seasonal effects, the stochastic seasonal relationship is used.

$$\sum_{i=0}^{L-1} S_{t-i} = w_{2t}, \qquad w_{2t} \sim N(0, \tau_2^2)$$

where L is the number of seasons within a period. In the context of Whittaker and Akaike, the smoothness priors problem can be solved by the maximization of

$$
\begin{aligned}
\ell(f) &= \exp\left[-\frac{1}{2\sigma^2}\sum_{t=1}^{T}(y_t - T_t - S_t)^2\right] \exp\left[-\frac{\tau_1^2}{2\sigma^2}\sum_{t=1}^{T}(\nabla^k T_t)^2\right] \\
&\quad \times \exp\left[-\frac{\tau_2^2}{2\sigma^2}\sum_{t=1}^{T}\left(\sum_{i=0}^{L-1} S_{t-i}\right)^2\right]
\end{aligned}
$$

The values of hyperparameters, τ_1^2 and τ_2^2, refer to a measure of uncertainty of prior information. For example, the large value of τ_1^2 implies a relatively smooth trend component. The ratio $\frac{\tau_i^2}{\sigma^2}$ $(i = 1, 2)$ can be considered as a signal-to-noise ratio.

Kitagawa and Gersch (1984) use the Kalman filter recursive computation for the likelihood of the tradeoff parameters. The hyperparameters are estimated by combining the grid search and optimization method. The state space model and Kalman filter recursive computation are discussed in the section, State Space and Kalman Filter Method.

Bayesian Seasonal Adjustment

Seasonal phenomena are frequently observed in many economic and business time series. For example, consumption expenditure might have strong seasonal variations because of Christmas spending. The seasonal phenomena are repeatedly observed after a regular period of time. The number of seasons within a period is defined as the smallest time span for this repetitive observation. Monthly consumption expenditure shows a strong increase during the Christmas season, with 12 seasons per period.

There are three major approaches to seasonal time series: the regression model, the moving average model, and the seasonal ARIMA model.

Regression Model

Let the trend component be $T_t = \sum_{i=1}^{m_\alpha} \alpha_i U_{it}$ and the seasonal component be $S_t = \sum_{j=1}^{m_\beta} \beta_j V_{jt}$. Then the additive time series can be written as the regression model

$$y_t = \sum_{i=1}^{m_\alpha} \alpha_i U_{it} + \sum_{j=1}^{m_\beta} \beta_j V_{jt} + \epsilon_t$$

In practice, the trend component can be written as the m_αth order polynomial, such as

$$T_t = \sum_{i=0}^{m_\alpha} \alpha_i t^i$$

The seasonal component can be approximated by the seasonal dummies (D_{jt})

$$S_t = \sum_{j=1}^{L-1} \beta_j D_{jt}$$

where L is the number of seasons within a period. The least-squares method is applied to estimate parameters α_i and β_j.

The seasonally adjusted series is obtained by subtracting the estimated seasonal component from the original series. Usually, the error term ϵ_t is assumed to be white noise, while sometimes the autocorrelation of the regression residuals needs to be allowed. However, the regression method is not robust to the regression function type, especially at the beginning and end of the series.

Moving Average Model

If you assume that the annual sum of a seasonal time series has small seasonal fluctuations, the nonseasonal component $N_t = T_t + \epsilon_t$ can be estimated by using the moving average method.

$$\hat{N}_t = \sum_{i=-m}^{m} \lambda_i y_{t-i}$$

where m is the positive integer and λ_i is the symmetric constant such that $\lambda_i = \lambda_{-i}$ and $\sum_{i=-m}^{m} \lambda_i = 1$.

When the data are not available, either an asymmetric moving average is used, or the forecast data is augmented to use the symmetric weight. The X-11 procedure is a complex modification of this moving average method.

Seasonal ARIMA Model

The regression and moving average approaches assume that the seasonal component is deterministic and independent of other nonseasonal components. The time series approach is used to handle the stochastic trend and seasonal components.

The general ARIMA model can be written

$$\prod_{j=1}^{m} \phi_j(B) \prod_{i=1}^{k} (1 - B^{s_i})^{d_i} \tilde{y}_t = \theta_0 + \prod_{i=1}^{q} \theta_i(B)\epsilon_t$$

where B is the backshift operator and

$$
\begin{aligned}
\phi_j(B) &= 1 - \phi_1 B - \cdots - \phi_j B^{p_j} \\
\theta_i(B) &= 1 - \theta_1 B - \cdots - \theta_i B^{q_i}
\end{aligned}
$$

and $\tilde{y}_t = y_t - \mathrm{E}(Y_t)$ if $d_i = 0$; otherwise, $\tilde{y}_t = y_t$. The power of B, s_i, can be considered as a seasonal factor. Specifically, the Box-Jenkins multiplicative seasonal ARIMA$(p, d, q)(P, D, Q)_s$ model is written as

$$\phi_p(B)\Phi_P(B^s)(1 - B)^d(1 - B^s)^D \tilde{y}_t = \theta_q(B)\Theta_Q(B^s)\epsilon_t$$

ARIMA modeling is appropriate for particular time series and requires burdensome computation.

The TSBAYSEA subroutine combines the simple characteristics of the regression approach and time series modeling. The TSBAYSEA and X-11 procedures use the model-based seasonal adjustment. The symmetric weights of the standard X-11 option can be approximated by using the integrated MA form

$$(1 - B)(1 - B^{12})y_t = \theta(B)\epsilon_t$$

With a fixed value ϕ, the TSBAYSEA subroutine is approximated as

$$(1 - \phi B)(1 - B)(1 - B^{12})y_t = \theta(B)\epsilon_t$$

The subroutine is flexible enough to handle trading day or leap year effects, the shift of the base observation, and missing values. The TSBAYSEA-type modeling approach has some advantages: it clearly defines the statistical model of the time series; modification of the basic model can be an efficient method of choosing a particular procedure for the seasonal adjustment of a given time series; and the use of the concept of the likelihood provides a minimum AIC model selection approach.

Nonstationary Time Series

The subroutines TSMLOCAR, TSMLOMAR, and TSTVCAR are used to analyze nonstationary time series models. The AIC statistic is extensively used to analyze the locally stationary model.

Locally Stationary AR Model

When the time series is nonstationary, the TSMLOCAR (univariate) and TSMLOMAR (multivariate) subroutines can be employed. The whole span of the series is divided into locally stationary blocks of data, and then the TSMLOCAR and TSMLOMAR subroutines estimate a stationary AR model by using the least-squares method on this stationary block. The homogeneity of two different blocks of data is tested using the AIC.

Given a set of data $\{y_1, \ldots, y_T\}$, the data can be divided into k blocks of sizes $t_1, \ldots, t_k$, where $t_1 + \cdots + t_k = T$, and k and t_i are unknown. The locally stationary model is fitted to the data

$$y_t = \alpha_0^i + \sum_{j=1}^{p_i} \alpha_j^i y_{t-j} + \epsilon_t^i$$

where

$$T_{i-1} = \sum_{j=1}^{i-1} t_j < t \le T_i = \sum_{j=1}^{i} t_j, \qquad \text{for } i = 1, \ldots, k$$

where ϵ_t^i is a Gaussian white noise with $\mathrm{E}\epsilon_t^i = 0$ and $\mathrm{E}(\epsilon_t^i)^2 = \sigma_i^2$. Therefore, the log-likelihood function of the locally stationary series is

$$\ell = -\frac{1}{2} \sum_{i=1}^{k} \left[t_i \log(2\pi\sigma_i^2) + \frac{1}{\sigma_i^2} \sum_{t=T_{i-1}+1}^{T_i} \left(y_t - \alpha_0^i - \sum_{j=1}^{p_i} \alpha_j^i y_{t-j} \right)^2 \right]$$

Given α_j^i, $j = 0, \ldots, p_i$, the maximum of the log-likelihood function is attained at

$$\hat{\sigma}_i^2 = \frac{1}{t_i} \sum_{t=T_{i-1}+1}^{T_i} \left(y_t - \hat{\alpha}_0^i - \sum_{j=1}^{p_i} \hat{\alpha}_j^i y_{t-j} \right)^2$$

The concentrated log-likelihood function is given by

$$\ell^* = -\frac{T}{2} [1 + \log(2\pi)] - \frac{1}{2} \sum_{i=1}^{k} t_i \log(\hat{\sigma}_i^2)$$

Therefore, the maximum likelihood estimates, $\hat{\alpha}_j^i$ and $\hat{\sigma}_i^2$, are obtained by minimizing the following local SSE:

$$\text{SSE} = \sum_{t=T_{i-1}+1}^{T_i} \left(y_t - \hat{\alpha}_0^i - \sum_{j=1}^{p_i} \hat{\alpha}_j^i y_{t-j} \right)^2$$

The least-squares estimation of the stationary model is explained in the section, Least Squares and Householder Transformation.

The AIC for the locally stationary model over the pooled data is written as

$$\sum_{i=1}^{k} t_i \log(\hat{\sigma}_i^2) + 2 \sum_{i=1}^{k} (p_i + \textit{intercept} + 1)$$

where *intercept* = 1 if the intercept term (α_0^i) is estimated; otherwise, *intercept* = 0. The number of stationary blocks (k), the size of each block (t_i), and the order of the locally stationary model is determined by the AIC. Consider the autoregressive model fitted over the block of data, $\{y_1, \ldots, y_T\}$, and let this model M_1 be an AR(p_1) process. When additional data, $\{y_{T+1}, \ldots, y_{T+T_1}\}$, are available, a new model M_2, an AR(p_2) process, is fitted over this new data set, assuming that these data are independent of the previous data. Then AICs for models M_1 and M_2 are defined as

$$\begin{aligned} \text{AIC}_1 &= T \log(\sigma_1^2) + 2(p_1 + \textit{intercept} + 1) \\ \text{AIC}_2 &= T_1 \log(\sigma_2^2) + 2(p_2 + \textit{intercept} + 1) \end{aligned}$$

The joint model AIC for M_1 and M_2 is obtained by summation

$$\text{AIC}_J = \text{AIC}_1 + \text{AIC}_2$$

When the two data sets are pooled and estimated over the pooled data set, $\{y_1, \ldots, y_{T+T_1}\}$, the AIC of the pooled model is

$$\text{AIC}_A = (T + T_1) \log(\hat{\sigma}_A^2) + 2(p_A + \textit{intercept} + 1)$$

where σ_A^2 is the pooled error variance and p_A is the order chosen to fit the pooled data set.

Decision

- If $\text{AIC}_J < \text{AIC}_A$, switch to the new model, since there is a change in the structure of the time series.

- If $\text{AIC}_J \geq \text{AIC}_A$, pool the two data sets, since two data sets are considered to be homogeneous.

If new observations are available, repeat the preceding steps to determine the homogeneity of the data. The basic idea of locally stationary AR modeling is that, if the structure of the time series is not changed, you should use the additional information to improve the model fitting, but you need to follow the new structure of the time series if there is any change.

Time-Varying AR Coefficient Model

Another approach to nonstationary time series, especially those that are nonstationary in the covariance, is time-varying AR coefficient modeling. When the time series is nonstationary in the covariance, the problem in modeling this series is related to an efficient parameterization. It is possible for a Bayesian approach to estimate the model with a large number of implicit parameters of the complex structure by using a relatively small number of hyperparameters.

The TSTVCAR subroutine uses smoothness priors by imposing stochastically perturbed difference equation constraints on each AR coefficient and frequency response function. The variance of each AR coefficient distribution constitutes a hyperparameter included in the state space model. The likelihood of these hyperparameters is computed by the Kalman filter recursive algorithm.

The time-varying AR coefficient model is written

$$
y_t = \sum_{i=1}^{m} \alpha_{it} y_{t-i} + \epsilon_t
$$

where time-varying coefficients α_{it} are assumed to change gradually with time. The following simple stochastic difference equation constraint is imposed on each coefficient:

$$
\nabla^k \alpha_{it} = w_{it}, \qquad w_{it} \sim N(0, \tau^2), \qquad i = 1, \dots, m
$$

The frequency response function of the AR process is written

$$
A(f) = 1 - \sum_{j=1}^{m} \alpha_{jt} \exp(-2\pi jif)
$$

The smoothness of this function can be measured by the kth derivative smoothness constraint,

$$
R_k = \int_{-1/2}^{1/2} \left| \frac{d^k A(f)}{df^k} \right|^2 df = (2\pi)^{2k} \sum_{j=1}^{m} j^{2k} \alpha_{jt}^2
$$

Then the TSTVCAR call imposes zero and second derivative smoothness constraints. The time-varying AR coefficients are the solution of the following constrained least-squares:

$$
\sum_{t=1}^{T} \left(y_t - \sum_{i=1}^{m} \alpha_{it} y_{t-i} \right)^2 + \tau^2 \sum_{t=1}^{T} \sum_{i=1}^{m} \left(\nabla^k \alpha_{it} \right)^2 + \lambda^2 \sum_{t=1}^{T} \sum_{i=1}^{m} i^2 \alpha_{it}^2 + \nu^2 \sum_{t=1}^{T} \sum_{i=1}^{m} \alpha_{it}^2
$$

where τ^2, λ^2, and ν^2 are hyperparameters of the prior distribution.

Using a state space representation, the model is

$$\begin{aligned}
\mathbf{x}_t &= \mathbf{F}\mathbf{x}_{t-1} + \mathbf{G}\mathbf{w}_t \\
y_t &= \mathbf{H}_t\mathbf{x}_t + \epsilon_t
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{x}_t &= (\alpha_{1t}, \ldots, \alpha_{mt}, \ldots, \alpha_{1,t-k+1}, \ldots, \alpha_{m,t-k+1})' \\
\mathbf{H}_t &= (y_{t-1}, \ldots, y_{t-m}, \ldots, 0, \ldots, 0) \\
\mathbf{w}_t &= (w_{1t}, \ldots, w_{mt})' \\
k &= 1 : \mathbf{F} = \mathbf{I}_m \quad \mathbf{G} = \mathbf{I}_m \\
k &= 2 : \mathbf{F} = \begin{bmatrix} 2\mathbf{I}_m & -\mathbf{I}_m \\ \mathbf{I}_m & 0 \end{bmatrix} \quad \mathbf{G} = \begin{bmatrix} \mathbf{I}_m \\ 0 \end{bmatrix} \\
k &= 3 : \mathbf{F} = \begin{bmatrix} 3\mathbf{I}_m & -3\mathbf{I}_m & \mathbf{I}_m \\ \mathbf{I}_m & 0 & 0 \\ 0 & \mathbf{I}_m & 0 \end{bmatrix} \quad \mathbf{G} = \begin{bmatrix} \mathbf{I}_m \\ 0 \\ 0 \end{bmatrix}
\end{aligned}$$

$$\begin{bmatrix} \mathbf{w}_t \\ \epsilon_t \end{bmatrix} \sim N\left(\mathbf{0}, \begin{bmatrix} \tau^2\mathbf{I} & 0 \\ 0 & \sigma^2 \end{bmatrix}\right)$$

The computation of the likelihood function is straightforward. See the section, State Space and Kalman Filter Method for the computation method.

Multivariate Time Series Analysis

The subroutines TSMULMAR, TSMLOMAR, and TSPRED analyze multivariate time series. The periodic AR model, TSPEARS, can also be estimated by using a vector AR procedure, since the periodic AR series can be represented as the covariance-stationary vector autoregressive model.

The stationary vector AR model is estimated and the order of the model (or each variable) is automatically determined by the minimum AIC procedure. The stationary vector AR model is written

$$\begin{aligned}
\mathbf{y}_t &= \mathbf{A}_0 + \mathbf{A}_1\mathbf{y}_{t-1} + \cdots + \mathbf{A}_p\mathbf{y}_{t-p} + \epsilon_t \\
\epsilon_t &\sim N(\mathbf{0}, \Sigma)
\end{aligned}$$

Using the $\mathbf{L}\mathbf{D}\mathbf{L}'$ factorization method, the error covariance is decomposed as

$$\Sigma = \mathbf{L}\mathbf{D}\mathbf{L}'$$

where $\mathbf{L}$ is a unit lower triangular matrix and $\mathbf{D}$ is a diagonal matrix. Then the instantaneous response model is defined as

$$\mathbf{C}\mathbf{y}_t = \mathbf{A}_0^* + \mathbf{A}_1^*\mathbf{y}_{t-1} + \cdots + \mathbf{A}_p^*\mathbf{y}_{t-p} + \epsilon_t^*$$

where $\mathbf{C} = \mathbf{L}^{-1}$, $\mathbf{A}_i^* = \mathbf{L}^{-1}\mathbf{A}_i$ for $i = 0, 1, \ldots, p$, and $\epsilon_t^* = \mathbf{L}^{-1}\epsilon_t$. Each equation of the instantaneous response model can be estimated independently, since its

error covariance matrix has a diagonal covariance matrix $\mathbf{D}$. Maximum likelihood estimates are obtained through the least-squares method when the disturbances are normally distributed and the presample values are fixed.

The TSMULMAR call estimates the instantaneous response model. The VAR coefficients are computed using the relationship between the VAR and instantaneous models.

The general VARMA model can be transformed as an infinite order MA process under certain conditions.

$$\mathbf{y}_t = \mu + \epsilon_t + \sum_{m=1}^{\infty} \Psi_m \epsilon_{t-m}$$

In the context of the VAR(p) model, the coefficient Ψ_m can be interpreted as the m-lagged response of a unit increase in the disturbances at time t.

$$\Psi_m = \frac{\partial \mathbf{y}_{t+m}}{\partial \epsilon_t'}$$

The lagged response on the one-unit increase in the orthogonalized disturbances ϵ_t^* is denoted

$$\frac{\partial \mathbf{y}_{t+m}}{\partial \epsilon_{jt}^*} = \frac{\partial \mathrm{E}(\mathbf{y}_{t+m}|y_{jt}, y_{j-1,t}, \ldots, \mathbf{X}_t)}{\partial y_{jt}} = \Psi_m \mathbf{L}_j$$

where $\mathbf{L}_j$ is the jth column of the unit triangular matrix $\mathbf{L}$ and $\mathbf{X}_t = [\mathbf{y}_{t-1}, \ldots, \mathbf{y}_{t-p}]$. When you estimate the VAR model using the TSMULMAR call, it is easy to compute this impulse response function.

The MSE of the m-step prediction is computed as

$$\mathrm{E}(\mathbf{y}_{t+m} - \mathbf{y}_{t+m|t})(\mathbf{y}_{t+m} - \mathbf{y}_{t+m|t})' = \Sigma + \Psi_1 \Sigma \Psi_1' + \cdots + \Psi_{m-1} \Sigma \Psi_{m-1}'$$

Note that $\epsilon_t = \mathbf{L}\epsilon_t^*$. Then the covariance matrix of ϵ_t is decomposed

$$\Sigma = \sum_{i=1}^{n} \mathbf{L}_i \mathbf{L}_i' d_{ii}$$

where d_{ii} is the ith diagonal element of the matrix $\mathbf{D}$ and n is the number of variables. The MSE matrix can be written

$$\sum_{i=1}^{n} d_{ii} \left[\mathbf{L}_i \mathbf{L}_i' + \Psi_1 \mathbf{L}_i \mathbf{L}_i' \Psi_1' + \cdots + \Psi_{m-1} \mathbf{L}_i \mathbf{L}_i' \Psi_{m-1}' \right]$$

Therefore, the contribution of the ith orthogonalized innovation to the MSE is

$$\mathbf{V}_i = d_{ii} \left[\mathbf{L}_i \mathbf{L}_i' + \Psi_1 \mathbf{L}_i \mathbf{L}_i' \Psi_1' + \cdots + \Psi_{m-1} \mathbf{L}_i \mathbf{L}_i' \Psi_{m-1}' \right]$$

The ith forecast error variance decomposition is obtained from diagonal elements of the matrix $\mathbf{V}_i$.

The nonstationary multivariate series can be analyzed by the TSMLOMAR subroutine. The estimation and model identification procedure is analogous to the univariate nonstationary procedure, which is explained in the "Nonstationary Time Series" section on page 274.

A time series y_t is periodically correlated with period d if $Ey_t = Ey_{t+d}$ and $Ey_s y_t = Ey_{s+d} y_{t+d}$. Let y_t be autoregressive of period d with AR orders $(p_1, \ldots, p_d)$, that is,

$$y_t = \sum_{j=1}^{p_t} \alpha_{jt} y_{t-j} + \epsilon_t$$

where ϵ_t is uncorrelated with mean zero and $E\epsilon_t^2 = \sigma_t^2$, $p_t = p_{t+d}$, $\sigma_t^2 = \sigma_{t+d}^2$, and $\alpha_{jt} = \alpha_{j,t+d}(j = 1, \ldots, p_t)$. Define the new variable such that $x_{jt} = y_{j+d(t-1)}$. The vector series, $\mathbf{x}_t = (x_{1t}, \ldots, x_{dt})'$, is autoregressive of order p, where $p = \max_j \text{int}((p_j - j)/d) + 1$. The TSPEARS subroutine estimates the periodic autoregressive model using minimum AIC vector AR modeling.

The TSPRED subroutine computes the one-step or multistep forecast of the multivariate ARMA model if the ARMA parameter estimates are provided. In addition, the subroutine TSPRED produces the (intermediate and permanent) impulse response function and performs forecast error variance decomposition for the vector AR model.

Spectral Analysis

The autocovariance function of the random variable Y_t is defined as

$$C_{YY}(k) = E(Y_{t+k} Y_t)$$

where $EY_t = 0$. When the real valued process Y_t is stationary and its autocovariance is absolutely summable, the population spectral density function is obtained using the Fourier transform of the autocovariance function

$$f(g) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} C_{YY}(k) \exp(-igk) \quad -\pi \le g \le \pi$$

where $i = \sqrt{-1}$ and $C_{YY}(k)$ is the autocovariance function such that $\sum_{k=-\infty}^{\infty} |C_{YY}(k)| < \infty$.

Consider the autocovariance generating function

$$\gamma(z) = \sum_{k=-\infty}^{\infty} C_{YY}(k) z^k$$

where $C_{YY}(k) = C_{YY}(-k)$ and z is a complex scalar. The spectral density function can be represented as

$$f(g) = \frac{1}{2\pi}\gamma(\exp(-ig))$$

The stationary ARMA(p, q) process is denoted:

$$\phi(B)y_t = \theta(B)\epsilon_t \quad \epsilon_t \sim (0, \sigma^2)$$

where $\phi(B)$ and $\theta(B)$ do not have common roots. Note that the autocovariance generating function of the linear process $y_t = \psi(B)\epsilon_t$ is given by

$$\gamma(B) = \sigma^2\psi(B)\psi(B^{-1})$$

For the ARMA(p, q) process, $\psi(B) = \frac{\theta(B)}{\phi(B)}$. Therefore, the spectral density function of the stationary ARMA(p, q) process becomes

$$f(g) = \frac{\sigma^2}{2\pi}\left|\frac{\theta(\exp(-ig))\theta(\exp(ig))}{\phi(\exp(-ig))\phi(\exp(ig))}\right|^2$$

The spectral density function of a white noise is a constant.

$$f(g) = \frac{\sigma^2}{2\pi}$$

The spectral density function of the AR(1) process ($\phi(B) = 1 - \phi_1 B$) is given by

$$f(g) = \frac{\sigma^2}{2\pi(1 - \phi_1\cos(g) + \phi_1^2)}$$

The spectrum of the AR(1) process has its minimum at $g = 0$ and its maximum at $g = \pm\pi$ if $\phi_1 < 0$, while the spectral density function attains its maximum at $g = 0$ and its minimum at $g = \pm\pi$, if $\phi_1 > 0$. When the series is positively autocorrelated, its spectral density function is dominated by low frequencies. It is interesting to observe that the spectrum approaches $\frac{\sigma^2}{4\pi}\frac{1}{1-\cos(g)}$ as $\phi_1 \to 1$. This relationship shows that the series is difference-stationary if its spectral density function has a remarkable peak near 0.

The spectrum of AR(2) process ($\phi(B) = 1 - \phi_1 B - \phi_2 B^2$) equals

$$f(g) = \frac{\sigma^2}{2\pi}\frac{1}{\left\{-4\phi_2\left[\cos(g) + \frac{\phi_1(1-\phi_2)}{4\phi_2}\right]^2 + \frac{(1+\phi_2)^2(4\phi_2+\phi_1^2)}{4\phi_2}\right\}}$$

Refer to Anderson (1971) for details of the characteristics of this spectral density function of the AR(2) process.

In practice, the population spectral density function cannot be computed. There are many ways of computing the sample spectral density function. The TSBAYSEA and TSMLOCAR calls compute the power spectrum using AR coefficients and the white noise variance.

The power spectral density function of Y_t is derived using the Fourier transformation of $C_{YY}(k)$.

$$f_{YY}(g) = \sum_{k=-\infty}^{\infty} \exp(-2\pi igk)C_{YY}(k), \qquad -\frac{1}{2} \le g \le \frac{1}{2}$$

where $i = \sqrt{-1}$ and g denotes frequency. The autocovariance function can also be written as

$$C_{YY}(k) = \int_{-1/2}^{1/2} \exp(2\pi igk)f_{YY}(g)dg$$

Consider the following stationary AR(p) process:

$$y_t - \sum_{i=1}^{p} \phi_i y_{t-i} = \epsilon_t$$

where ϵ_t is a white noise with mean zero and constant variance σ^2.

The autocovariance function of white noise ϵ_t equals

$$C_{\epsilon\epsilon}(k) = \delta_{k0}\sigma^2$$

where $\delta_{k0} = 1$ if $k = 0$; otherwise, $\delta_{k0} = 0$. Therefore, the power spectral density of the white noise is $f_{\epsilon\epsilon}(g) = \sigma^2$, $-\frac{1}{2} \le g \le \frac{1}{2}$. Note that, with $\phi_0 = -1$,

$$C_{\epsilon\epsilon}(k) = \sum_{m=0}^{p}\sum_{n=0}^{p} \phi_m\phi_n C_{YY}(k - m + n)$$

Using the following autocovariance function of Y_t,

$$C_{YY}(k) = \int_{-1/2}^{1/2} \exp(2\pi igk)f_{YY}(g)dg$$

the autocovariance function of the white noise is denoted as

$$\begin{aligned}
C_{\epsilon\epsilon}(k) &= \sum_{m=0}^{p}\sum_{n=0}^{p} \phi_m\phi_n \int_{-1/2}^{1/2} \exp(2\pi ig(k - m + n))f_{YY}(g)dg \\
&= \int_{-1/2}^{1/2} \exp(2\pi igk) \left|1 - \sum_{m=1}^{p} \phi_m \exp(-2\pi igm)\right|^2 f_{YY}(g)dg
\end{aligned}$$

On the other hand, another formula of the $C_{\epsilon\epsilon}(k)$ gives

$$C_{\epsilon\epsilon}(k) = \int_{-1/2}^{1/2} \exp(2\pi igk) f_{\epsilon\epsilon}(g) dg$$

Therefore,

$$f_{\epsilon\epsilon}(g) = \left| 1 - \sum_{m=1}^{p} \phi_m \exp(-2\pi igm) \right|^2 f_{YY}(g)$$

Since $f_{\epsilon\epsilon}(g) = \sigma^2$, the rational spectrum of Y_t is

$$f_{YY}(g) = \frac{\sigma^2}{|1 - \sum_{m=1}^{p} \phi_m \exp(-2\pi igm)|^2}$$

To compute the power spectrum, estimated values of white noise variance $\hat{\sigma}^2$ and AR coefficients $\hat{\phi}_m$ are used. The order of the AR process can be determined by using the minimum AIC procedure.

Computational Details

Least Squares and Householder Transformation

Consider the univariate AR(p) process

$$y_t = \alpha_0 + \sum_{i=1}^{p} \alpha_i y_{t-i} + \epsilon_t$$

Define the design matrix $\mathbf{X}$.

$$\mathbf{X} = \begin{bmatrix} 1 & y_p & \cdots & y_1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & y_{T-1} & \cdots & y_{T-p} \end{bmatrix}$$

Let $\mathbf{y} = (y_{p+1}, \ldots, y_n)'$. The least-squares estimate, $\hat{\mathbf{a}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, is the approximation to the maximum likelihood estimate of $\mathbf{a} = (\alpha_0, \alpha_1, \ldots, \alpha_p)$ if ϵ_t is assumed to be Gaussian error disturbances. Combining $\mathbf{X}$ and $\mathbf{y}$ as

$$\mathbf{Z} = [\mathbf{X} \vdots \mathbf{y}]$$

the $\mathbf{Z}$ matrix can be decomposed as

$$\mathbf{Z} = \mathbf{QU} = \mathbf{Q} \begin{bmatrix} \mathbf{R} & \mathbf{w}_1 \\ \mathbf{0} & \mathbf{w}_2 \end{bmatrix}$$

where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix, $\mathbf{w}_1 = (w_1, \ldots, w_{p+1})'$, and $\mathbf{w}_2 = (w_{p+2}, 0, \ldots, 0)'$.

$$
\mathbf{Q}'\mathbf{y} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{T-p} \end{bmatrix}
$$

The least-squares estimate using Householder transformation is computed by solving the linear system

$$
\mathbf{Ra} = \mathbf{w}_1
$$

The unbiased residual variance estimate is

$$
\hat{\sigma}^2 = \frac{1}{T-p} \sum_{i=p+2}^{T-p} w_i^2 = \frac{w_{p+2}^2}{T-p}
$$

and

$$
\text{AIC} = (T-p)\log(\hat{\sigma}^2) + 2(p+1)
$$

In practice, least-squares estimation does not require the orthogonal matrix $\mathbf{Q}$. The TIMSAC subroutines compute the upper triangular matrix without computing the matrix $\mathbf{Q}$.

Bayesian Constrained Least Squares

Consider the additive time series model

$$
y_t = T_t + S_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma^2)
$$

Practically, it is not possible to estimate parameters $\mathbf{a} = (T_1, \ldots, T_T, S_1, \ldots, S_T)'$, since the number of parameters exceeds the number of available observations. Let ∇_L^m denote the seasonal difference operator with L seasons and degree of m; that is, $\nabla_L^m = (1 - B^L)^m$. Suppose that $T = L * n$. Some constraints on the trend and seasonal components need to be imposed such that the sum of squares of $\nabla^k T_t$, $\nabla_L^m S_t$, and $(\sum_{i=0}^{L-1} S_{t-i})$ is small. The constrained least-squares estimates are obtained by minimizing

$$
\sum_{t=1}^{T} \left\{ (y_t - T_t - S_t)^2 + d^2 \left[s^2 (\nabla^k T_t)^2 + (\nabla_L^m S_t)^2 + z^2 (S_t + \cdots + S_{t-L+1})^2 \right] \right\}
$$

Using matrix notation,

$$
(\mathbf{y} - \mathbf{Ma})'(\mathbf{y} - \mathbf{Ma}) + (\mathbf{a} - \mathbf{a}_0)'\mathbf{D}'\mathbf{D}(\mathbf{a} - \mathbf{a}_0)
$$

where $\mathbf{M} = [\mathbf{I}_T \vdots \mathbf{I}_T]$, $\mathbf{y} = (y_1, \ldots, y_T)'$, and $\mathbf{a}_0$ is the initial guess of $\mathbf{a}$. The matrix $\mathbf{D}$ is a $3T \times 2T$ control matrix in which structure varies according to the order of differencing in trend and season.

$$
\mathbf{D} = d \begin{bmatrix} \mathbf{E}_m & \mathbf{0} \\ z\mathbf{F} & \mathbf{0} \\ \mathbf{0} & s\mathbf{G}_k \end{bmatrix}
$$

where

$$
\mathbf{E}_m = \mathbf{C}_m \otimes \mathbf{I}_L, \qquad m = 1, 2, 3
$$

$$
\mathbf{F} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 1 & \cdots & 1 & 1 \end{bmatrix}_{T \times T}
$$

$$
\mathbf{G}_1 = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -1 & 1 \end{bmatrix}_{T \times T}
$$

$$
\mathbf{G}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & -2 & 1 \end{bmatrix}_{T \times T}
$$

$$
\mathbf{G}_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -3 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 3 & -3 & 1 & 0 & 0 & \cdots & 0 \\ -1 & 3 & -3 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 3 & -3 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -1 & 3 & -3 & 1 \end{bmatrix}_{T \times T}
$$

The $n \times n$ matrix $\mathbf{C}_m$ has the same structure as the matrix $\mathbf{G}_m$, and $\mathbf{I}_L$ is the $L \times L$ identity matrix. The solution of the constrained least-squares method is equivalent to that of maximizing the following function

$$
L(\mathbf{a}) = \exp\left\{ -\frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{Ma})'(\mathbf{y} - \mathbf{Ma}) \right\} \exp\left\{ -\frac{1}{2\sigma^2}(\mathbf{a} - \mathbf{a}_0)'\mathbf{D}'\mathbf{D}(\mathbf{a} - \mathbf{a}_0) \right\}
$$

Therefore, the PDF of the data $\mathbf{y}$ is

$$f(\mathbf{y}|\sigma^2, \mathbf{a}) = \left(\frac{1}{2\pi}\right)^{T/2} \left(\frac{1}{\sigma}\right)^T \exp\left\{-\frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{Ma})'(\mathbf{y} - \mathbf{Ma})\right\}$$

The prior PDF of the parameter vector $\mathbf{a}$ is

$$\pi(\mathbf{a}|\mathbf{D}, \sigma^2, \mathbf{a}_0) = \left(\frac{1}{2\pi}\right)^T \left(\frac{1}{\sigma}\right)^{2T} |\mathbf{D}'\mathbf{D}| \exp\left\{-\frac{1}{2\sigma^2}(\mathbf{a} - \mathbf{a}_0)'\mathbf{D}'\mathbf{D}(\mathbf{a} - \mathbf{a}_0)\right\}$$

When the constant d is known, the estimate $\hat{\mathbf{a}}$ of $\mathbf{a}$ is the mean of the posterior distribution, where the posterior PDF of the parameter $\mathbf{a}$ is proportional to the function $L(\mathbf{a})$. It is obvious that $\hat{\mathbf{a}}$ is the minimizer of $\|\mathbf{g}(\mathbf{a}|d)\|^2 = (\tilde{\mathbf{y}} - \tilde{\mathbf{D}}\mathbf{a})'(\tilde{\mathbf{y}} - \tilde{\mathbf{D}}\mathbf{a})$, where

$$\tilde{\mathbf{y}} = \left[\begin{array}{c} \mathbf{y} \\ \mathbf{D}\mathbf{a}_0 \end{array}\right]$$

$$\tilde{\mathbf{D}} = \left[\begin{array}{c} \mathbf{M} \\ \mathbf{D} \end{array}\right]$$

The value of d is determined by the minimum ABIC procedure. The ABIC is defined as

$$\text{ABIC} = T\log\left[\frac{1}{T}\|\mathbf{g}(\mathbf{a}|d)\|^2\right] + 2\{\log[\det(\mathbf{D}'\mathbf{D} + \mathbf{M}'\mathbf{M})] - \log[\det(\mathbf{D}'\mathbf{D})]\}$$

State Space and Kalman Filter Method

In this section, the mathematical formulas for state space modeling are introduced. The Kalman filter algorithms are derived from the state space model. As an example, the state space model of the TSDECOMP subroutine is formulated.

Define the following state space model:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{F}\mathbf{x}_{t-1} + \mathbf{G}\mathbf{w}_t \\ y_t &= \mathbf{H}_t\mathbf{x}_t + \epsilon_t \end{aligned}$$

where $\epsilon_t \sim N(0, \sigma^2)$ and $\mathbf{w}_t \sim N(\mathbf{0}, \mathbf{Q})$. If the observations, $(y_1, \ldots, y_T)$, and the initial conditions, $\mathbf{x}_{0|0}$ and $\mathbf{P}_{0|0}$, are available, the one-step predictor $(\mathbf{x}_{t|t-1})$ of the state vector $\mathbf{x}_t$ and its mean square error (MSE) matrix $\mathbf{P}_{t|t-1}$ are written as

$$\mathbf{x}_{t|t-1} = \mathbf{F}\mathbf{x}_{t-1|t-1}$$

$$\mathbf{P}_{t|t-1} = \mathbf{F}\mathbf{P}_{t-1|t-1}\mathbf{F}' + \mathbf{G}\mathbf{Q}\mathbf{G}'$$

Using the current observation, the filtered value of $\mathbf{x}_t$ and its variance $\mathbf{P}_{t|t}$ are updated.

$$\mathbf{x}_{t|t} = \mathbf{x}_{t|t-1} + \mathbf{K}_t e_t$$

$$\mathbf{P}_{t|t} = (\mathbf{I} - \mathbf{K}_t \mathbf{H}_t) \mathbf{P}_{t|t-1}$$

where $e_t = y_t - \mathbf{H}_t \mathbf{x}_{t|t-1}$ and $\mathbf{K}_t = \mathbf{P}_{t|t-1} \mathbf{H}_t' [\mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}_t' + \sigma^2 \mathbf{I}]^{-1}$. The log-likelihood function is computed as

$$\ell = -\frac{1}{2} \sum_{t=1}^{T} \log(2\pi v_{t|t-1}) - \sum_{t=1}^{T} \frac{e_t^2}{2v_{t|t-1}}$$

where $v_{t|t-1}$ is the conditional variance of the one-step prediction error e_t.

Consider the additive time series decomposition

$$y_t = T_t + S_t + TD_t + u_t + \mathbf{x}_t' \beta_t + \epsilon_t$$

where $\mathbf{x}_t$ is a $(K \times 1)$ regressor vector and β_t is a $(K \times 1)$ time-varying coefficient vector. Each component has the following constraints:

$$
\begin{aligned}
\nabla^k T_t &= w_{1t}, & w_{1t} &\sim N(0, \tau_1^2) \\
\nabla_L^m S_t &= w_{2t}, & w_{2t} &\sim N(0, \tau_2^2) \\
u_t &= \sum_{i=1}^{p} \alpha_i u_{t-i} + w_{3t}, & w_{3t} &\sim N(0, \tau_3^2) \\
\beta_{jt} &= \beta_{j,t-1} + w_{3+j,t}, & w_{3+j,t} &\sim N(0, \tau_{3+j}^2), \quad j = 1, \cdots, K \\
\sum_{i=1}^{7} \gamma_{it} TD_t(i) &= \sum_{i=1}^{6} \gamma_{it}(TD_t(i) - TD_t(7)) \\
\gamma_{it} &= \gamma_{i,t-1}
\end{aligned}
$$

where $\nabla^k = (1 - B)^k$ and $\nabla_L^m = (1 - B^L)^m$. The AR component u_t is assumed to be stationary. The trading day component $TD_t(i)$ represents the number of the ith day of the week in time t. If $k = 3, p = 3, m = 1$, and $L = 12$ (monthly data),

$$
\begin{aligned}
T_t &= 3T_{t-1} - 3T_{t-2} + T_{t-3} + w_{1t} \\
\sum_{i=0}^{11} S_{t-i} &= w_{2t} \\
u_t &= \sum_{i=1}^{3} \alpha_i u_{t-i} + w_{3t}
\end{aligned}
$$

The state vector is defined as

$$\mathbf{x}_t = (T_t, T_{t-1}, T_{t-2}, S_t, \ldots, S_{t-11}, u_t, u_{t-1}, u_{t-2}, \gamma_{1t}, \ldots, \gamma_{6t})'$$

The matrix **F** is

$$
\mathbf{F} = \begin{bmatrix} \mathbf{F}_1 & 0 & 0 & 0 \\ 0 & \mathbf{F}_2 & 0 & 0 \\ 0 & 0 & \mathbf{F}_3 & 0 \\ 0 & 0 & 0 & \mathbf{F}_4 \end{bmatrix}
$$

where

$$
\mathbf{F}_1 = \begin{bmatrix} 3 & -3 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
$$

$$
\mathbf{F}_2 = \begin{bmatrix} -\mathbf{1}' & -1 \\ \mathbf{I}_{10} & 0 \end{bmatrix}
$$

$$
\mathbf{F}_3 = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
$$

$$
\mathbf{F}_4 = \mathbf{I}_6
$$

$$
\mathbf{1}' = (1, 1, \ldots, 1)
$$

The matrix G can be denoted as

$$
G = \begin{bmatrix} \mathbf{g}_1 & 0 & 0 \\ 0 & \mathbf{g}_2 & 0 \\ 0 & 0 & \mathbf{g}_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
$$

where

$$
\mathbf{g}_1 = \mathbf{g}_3 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}'
$$

$$
\mathbf{g}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}'
$$

Finally, the matrix $\mathbf{H}_t$ is time-varying,

$$
\mathbf{H}_t = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \mathbf{h}'_t \end{bmatrix}
$$

where

$$
\begin{aligned} \mathbf{h}_t &= \begin{bmatrix} D_t(1) & D_t(2) & D_t(3) & D_t(4) & D_t(5) & D_t(6) \end{bmatrix}' \\ D_t(i) &= TD_t(i) - TD_t(7), \qquad i = 1, \ldots, 6 \end{aligned}
$$

Missing Values

The TIMSAC subroutines skip any missing values at the beginning of the data set. When the univariate and multivariate AR models are estimated via least-squares (TSMLOCAR, TSMLOMAR, TSUNIMAR, TSMULMAR, and TSPEARS), there are three options available; that is, MISSING=0, MISSING=1, or MISSING=2. When the MISSING=0 (default) option is specified, the first contiguous observations with no missing values are used. The MISSING=1 option specifies that only nonmissing observations should be used by ignoring the observations with missing values. If the MISSING=2 option is specified, the missing values are filled with the sample mean. The least-squares estimator with the MISSING=2 option is biased in general.

The BAYSEA subroutine assumes the same prior distribution of the trend and seasonal components that correspond to the missing observations. A modification is made to skip the components of the vector $\mathbf{g}(\mathbf{a}|d)$ that correspond to the missing observations. The vector $\mathbf{g}(\mathbf{a}|d)$ is defined in the section, Bayesian Constrained Least Squares. In addition, the TSBAYSEA subroutine considers outliers as missing values. The TSDECOMP and TSTVCAR subroutines skip the Kalman filter updating equation when the current observation is missing.

ISM TIMSAC Packages

A description of each TIMSAC package follows. Each description includes a list of the programs provided in the TIMSAC version.

TIMSAC-72

analyzes and controls the feedback systems (for example, cement kiln process). Univariate- and multivariate-AR models are employed in this original TIMSAC package. The final prediction error (FPE) criterion is used for model selection.

- AUSPEC estimates the power spectrum by the Blackman-Tukey procedure.
- AUTCOR computes autocovariance and autocorrelation.
- DECONV computes the impulse response function.
- FFTCOR computes autocorrelation and crosscorrelation via the fast Fourier transform.
- FPEAUT computes AR coefficients and FPE for the univariate AR model.
- FPEC computes AR coefficients and FPE for the control system or multivariate AR model.
- MULCOR computes multiple covariance and correlation.
- MULNOS computes relative power contribution.
- MULRSP estimates the rational spectrum for multivariate data.
- MULSPE estimates the cross spectrum by Blackman-Tukey procedure.
- OPTDES performs optimal controller design.
- OPTSIM performs optimal controller simulation.
- RASPEC estimates the rational spectrum for univariate data.

- SGLFRE computes the frequency response function.
- WNOISE performs white noise simulation.

TIMSAC-74

estimates and forecasts the univariate and multivariate ARMA models by fitting the canonical Markovian model. A locally stationary autoregressive model is also analyzed. Akaike's information criterion (AIC) is used for model selection.

- AUTARM performs automatic univariate ARMA model fitting.
- BISPEC computes bispectrum.
- CANARM performs univariate canonical correlation analysis.
- CANOCA performs multivariate canonical correlation analysis.
- COVGEN computes the covariance from gain function.
- FRDPLY plots the frequency response function.
- MARKOV performs automatic multivariate ARMA model fitting.
- NONST estimates the locally stationary AR model.
- PRDCTR performs ARMA model prediction.
- PWDPLY plots the power spectrum.
- SIMCON performs optimal controller design and simulation.
- THIRMO computes the third-order moment.

TIMSAC-78

uses the Householder transformation to estimate the time series models. This package also contains Bayesian modeling and the exact maximum likelihood estimation of the ARMA model. Minimum AIC or Akaike Bayesian Information Criterion (ABIC) modeling is extensively used.

- BLOCAR estimates the locally stationary univariate AR model using the Bayesian method.
- BLOMAR estimates the locally stationary multivariate AR model using the Bayesian method.
- BSUBST estimates the univariate subset regression model using the Bayesian method.
- EXSAR estimates the univariate AR model using the exact maximum likelihood method.
- MLOCAR estimates the locally stationary univariate AR model using the minimum AIC method.
- MLOMAR estimates the locally stationary multivariate AR model using the minimum AIC method.
- MULBAR estimates the multivariate AR model using the Bayesian method.

- MULMAR estimates the multivariate AR model using the minimum AIC method.

- NADCON performs noise adaptive control.

- PERARS estimates the periodic AR model using the minimum AIC method.

- UNIBAR estimates the univariate AR model using the Bayesian method.

- UNIMAR estimates the univariate AR model using the minimum AIC method.

- XSARMA estimates the univariate ARMA model using the exact maximum likelihood method.

In addition, the following test subroutines are available: TSSBST, TSWIND, TSROOT, TSTIMS, and TSCANC.

TIMSAC-84

contains the Bayesian time series modeling procedure, the point process data analysis, and the seasonal adjustment procedure.

- ADAR estimates the amplitude dependent AR model.

- BAYSEA performs Bayesian seasonal adjustments.

- BAYTAP performs Bayesian tidal analysis.

- DECOMP performs time series decomposition analysis using state space modeling.

- EPTREN estimates intensity rates of either the exponential polynomial or exponential Fourier series of the nonstationary Poisson process model.

- LINLIN estimates linear intensity models of the self-exciting point process with another process input and with cyclic and trend components.

- LINSIM performs simulation of the point process estimated by the subroutine LINLIN.

- LOCCAR estimates the locally constant AR model.

- MULCON performs simulation, control, and prediction of the multivariate AR model.

- NONSPA performs nonstationary spectrum analysis using the minimum Bayesian AIC procedure.

- PGRAPH performs graphical analysis for point process data.

- PTSPEC computes periodograms of point process data with significant bands.

- SIMBVH performs simulation of bivariate Hawkes' mutually exciting point process.

- SNDE estimates the stochastic nonlinear differential equation model.

- TVCAR estimates the time-varying AR coefficient model using state space modeling.

Refer to Kitagawa and Akaike (1981) and Ishiguro (1987) for more information about TIMSAC programs.

Example 10.1. VAR Estimation and Variance Decomposition ◆ 291

Example 10.1. VAR Estimation and Variance Decomposition

In this example, a VAR model is estimated and forecast. The VAR(3) model is estimated using investment, durable consumption, and consumption expenditures. The data are found in the appendix to Lütkepohl (1991). The stationary VAR(3) process is specified as

$$\mathbf{y}_t = \mathbf{A}_0 + \mathbf{A}_1 \mathbf{y}_{t-1} + \mathbf{A}_2 \mathbf{y}_{t-2} + \mathbf{A}_3 \mathbf{y}_{t-3} + \epsilon_t$$

The matrix ARCOEF contains the AR coefficients ($\mathbf{A}_1, \mathbf{A}_2$, and $\mathbf{A}_3$), and the matrix EV contains error covariance estimates. An intercept vector $\mathbf{A}_0$ is included in the first row of the matrix ARCOEF if OPT[1]=1 is specified.

```
data one;
   input invest income consum @@;
datalines;
   180 451   415  179 465   421    185 485   434    192 493   448
   211 509   459  202 520   458    207 521   479    214 540   487
   231 548   497  229 558   510    234 574   516    237 583   525
   206 591   529  250 599   538    259 610   546    263 627   555
   264 642   574  280 653   574    282 660   586    292 694   602
   286 709   617  302 734   639    304 751   653    307 763   668
   317 766   679  314 779   686    306 808   697    304 785   688
   292 794   704  275 799   699    273 799   709    301 812   715
   280 837   724  289 853   746    303 876   758    322 897   779
   315 922   798  339 949   816    364 979   837    371 988   858
   375 1025 881   432 1063 905     453 1104 934     460 1131 968
   475 1137 983   496 1178 1013    494 1211 1034    498 1256 1064
   526 1290 1101 519 1314 1102     516 1346 1145    531 1385 1173
   573 1416 1216 551 1436 1229     538 1462 1242    532 1493 1267
   558 1516 1295 524 1557 1317     525 1613 1355    519 1642 1371
   526 1690 1402 510 1759 1452     519 1756 1485    538 1780 1516
   549 1807 1549 570 1831 1567     559 1873 1588    584 1897 1631
   611 1910 1650 597 1943 1685     603 1976 1722    619 2018 1752
   635 2040 1774 658 2070 1807     675 2121 1831    700 2132 1842
   692 2199 1890 759 2253 1958     782 2276 1948    816 2318 1994
   844 2369 2061 830 2423 2056     853 2457 2102    852 2470 2121
   833 2521 2145 860 2545 2164     870 2580 2206    830 2620 2225
   801 2639 2235 824 2618 2237     831 2628 2250    830 2651 2271
   ;
proc iml;
   use one;
   read all into y var{invest income consum};
   mdel  = 1;
   maice = 0;
   misw  = 0;   /*-- instantaneous modeling ? --*/
   call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=3
        opt=(mdel || maice || misw) print=1;
```

To obtain the unit triangular matrix $\mathbf{L}^{-1}$ and diagonal matrix $\mathbf{D}_t$, you need to estimate the instantaneous response model. When you specify the OPT[3]=1 option, the first

row of the output matrix EV contains error variances of the instantaneous response model, while the unit triangular matrix is in the second through the fifth rows. See Output 10.1.1 on page 292.

```
misw = 1;
call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=3
      opt=(mdel || maice || misw) print=1;
print ev;
```

Output 10.1.1. Error Variance and Unit Triangular Matrix

```
              VAR Estimation and Variance Decomposition

                            EV

                  295.21042 190.94664 59.361516
                          1          0          0
                   -0.02239          1          0
                  -0.256341 -0.500803          1
```

In Output 10.1.2 on page 292 and Output 10.1.3 on page 292, you can see the relationship between the instantaneous response model and the VAR model. The VAR coefficients are computed as $\mathbf{A}_i = \mathbf{L}\mathbf{A}_i^* (i = 0, 1, 2, 3)$, where $\mathbf{A}_i^*$ is a coefficient matrix of the instantaneous model. For example, you can verify this result using the first lag coefficient matrix $(\mathbf{A}_1)$.

$$
\begin{bmatrix}
0.886 & 0.340 & -0.014 \\
0.168 & 1.050 & 0.107 \\
0.089 & 0.459 & 0.447
\end{bmatrix}
=
\begin{bmatrix}
1.000 & 0 & 0 \\
-0.022 & 1.000 & 0 \\
-0.256 & -0.501 & 1.000
\end{bmatrix}^{-1}
\begin{bmatrix}
0.886 & 0.340 & -0.014 \\
0.149 & 1.043 & 0.107 \\
-0.222 & -0.154 & 0.397
\end{bmatrix}
$$

Output 10.1.2. VAR Estimates

```
                          ARCOEF

                  0.8855926 0.3401741 -0.014398
                  0.1684523 1.0502619  0.107064
                  0.0891034 0.4591573 0.4473672
                  -0.059195 -0.298777 0.1629818
                  0.1128625 -0.044039 -0.088186
                  0.1684932 -0.025847 -0.025671
                  0.0637227 -0.196504 0.0695746
                  -0.226559 0.0532467 -0.099808
                  -0.303697 -0.139022 0.2576405
```

Example 10.1. VAR Estimation and Variance Decomposition ◆ 293

Output 10.1.3. Instantaneous Response Model Estimates

```
                              ARCOEF

                     0.885593   0.340174  -0.014398
                     0.148624   1.042645   0.107386
                    -0.222272  -0.154018   0.39744
                    -0.059195  -0.298777   0.162982
                     0.114188  -0.037349  -0.091835
                     0.127145   0.072796  -0.023287
                     0.063723  -0.196504   0.069575
                    -0.227986   0.057646  -0.101366
                     -0.20657  -0.115316   0.28979
```

When the VAR estimates are available, you can forecast the future values using the TSPRED call. As a default, the one-step predictions are produced until the START= point is reached. The NPRED=h option specifies how far you want to predict. The prediction error covariance matrix MSE contains h mean square error matrices. The output matrix IMPULSE contains the estimate of the coefficients (Ψ_i) of the infinite MA process. The following IML code estimates the VAR(3) model and performs 10-step-ahead prediction.

```
mdel  = 1;
maice = 0;
misw  = 0;
call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=3
     opt=(mdel || maice || misw);
call tspred(forecast,impulse,mse,y,arcoef,nar,0,ev)
     npred=10 start=nrow(y) constant=mdel;
print impulse;
```

The lagged effects of a unit increase in the error disturbances are included in the matrix IMPULSE. For example,

$$\frac{\partial \mathbf{y}_{t+2}}{\partial \epsilon_t'} = \begin{bmatrix} 0.781100 & 0.353140 & 0.180211 \\ 0.448501 & 1.165474 & 0.069731 \\ 0.364611 & 0.692111 & 0.222342 \end{bmatrix}$$

Output 10.1.4 on page 293 displays the first 15 rows of the matrix IMPULSE.

Output 10.1.4. Moving Average Coefficients: MA(0)−MA(4)

```
                           IMPULSE

                      1          0          0
                      0          1          0
                      0          0          1
              0.8855926 0.3401741 -0.014398
              0.1684523 1.0502619  0.107064
              0.0891034 0.4591573 0.4473672
              0.7810999 0.3531397 0.1802109
              0.4485013 1.1654737 0.0697311
              0.3646106 0.6921108 0.2223425
              0.8145483  0.243637 0.2914643
              0.4997732 1.3625363 -0.018202
              0.2775237 0.7555914 0.3885065
              0.7960884 0.2593068  0.260239
              0.5275069 1.4134792 0.0335483
               0.267452 0.8659426 0.3190203
```

In addition, you can compute the lagged response on the one-unit increase in the orthogonalized disturbances ϵ_t^*.

$$\frac{\partial \mathbf{y}_{t+m}}{\partial \epsilon_{jt}^*} = \frac{\partial \mathrm{E}(\mathbf{y}_{t+m}|y_{jt}, y_{j-1,t}, \ldots, \mathbf{X}_t)}{\partial y_{jt}} = \Psi_m \mathbf{L}_j$$

When the error matrix EV is obtained from the instantaneous response model, you need to convert the matrix IMPULSE. The first 15 rows of the matrix ORTH_IMP are shown in Output 10.1.5 on page 294. Note that the matrix constructed from the last three rows of EV become the matrix $\mathbf{L}^{-1}$.

```
call tsmulmar(arcoef,ev,nar,aic) data=y maxlag=3
     opt={1 0 1};
lmtx = inv(ev[2:nrow(ev),]);
orth_imp = impulse * lmtx;
print orth_imp;
```

Output 10.1.5. Transformed Moving Average Coefficients

```
                          ORTH_IMP

                      1          0          0
              0.0223902          1          0
               0.267554 0.5008031          1
               0.889357 0.3329638 -0.014398
              0.2206132 1.1038799  0.107064
               0.219079 0.6832001 0.4473672
              0.8372229 0.4433899 0.1802109
              0.4932533 1.2003953 0.0697311
              0.4395957 0.8034606 0.2223425
              0.8979858 0.3896033 0.2914643
              0.5254106 1.3534206 -0.018202
               0.398388 0.9501566 0.3885065
              0.8715223 0.3896353  0.260239
              0.5681309 1.4302804 0.0335483
              0.3721958  1.025709 0.3190203
```

Example 10.1. VAR Estimation and Variance Decomposition ◆ 295

You can verify the result for the case of

$$\frac{\partial \mathbf{y}_{t+2}}{\partial \epsilon_{2t}^*} = \frac{\partial \mathrm{E}(\mathbf{y}_{t+2}|y_{2t}, y_{1t}, \ldots, \mathbf{X}_t)}{\partial y_{2t}} = \Psi_2 \mathbf{L}_2$$

using the simple computation

$$\begin{bmatrix} 0.443390 \\ 1.200395 \\ 0.803461 \end{bmatrix} = \begin{bmatrix} 0.781100 & 0.353140 & 0.180211 \\ 0.448501 & 1.165474 & 0.069731 \\ 0.364611 & 0.692111 & 0.222342 \end{bmatrix} \begin{bmatrix} 0.000000 \\ 1.000000 \\ 0.500803 \end{bmatrix}$$

The contribution of the ith orthogonalized innovation to the mean square error matrix of the 10-step forecast is computed using the formula

$$d_{ii}[\mathbf{L}_i\mathbf{L}_i' + \Psi_1\mathbf{L}_i\mathbf{L}_i'\Psi_1' + \cdots + \Psi_9\mathbf{L}_i\mathbf{L}_i'\Psi_9']$$

In Output 10.1.6 on page 295, diagonal elements of each decomposed MSE matrix are displayed as the matrix CONTRIB as well as those of the MSE matrix (VAR).

```
mse1  = j(3,3,0);
mse2  = j(3,3,0);
mse3  = j(3,3,0);
do i = 1 to 10;
   psi = impulse[(i-1)*3+1:3*i,];
   mse1 = mse1 + psi*lmtx[,1]*lmtx[,1]'*psi';
   mse2 = mse2 + psi*lmtx[,2]*lmtx[,2]'*psi';
   mse3 = mse3 + psi*lmtx[,3]*lmtx[,3]'*psi';
end;
mse1 = ev[1,1]#mse1;
mse2 = ev[1,2]#mse2;
mse3 = ev[1,3]#mse3;
contrib = vecdiag(mse1) || vecdiag(mse2) || vecdiag(mse3);
var  = vecdiag(mse[28:30,]);
print contrib var;
```

Output 10.1.6. Orthogonal Innovation Contribution

```
          CONTRIB                          VAR

     1197.9131 116.68096 11.003194  2163.7104
     263.12088 1439.1551 1.0555626  4573.9809
     180.09836 633.55931 89.177905   2466.506
```

The investment innovation contribution to its own variable is 1879.3774, and the income innovation contribution to the consumption expenditure is 1916.1676. It is easy to understand the contribution of innovations in the ith variable to MSE when you compute the innovation account. In Output 10.1.7 on page 296, innovations in the first variable (investment) explain 20.45% of the error variance of the second variable (income), while the innovations in the second variable explain 79.5% of its own error variance. It is straightforward to construct the general multistep forecast error variance decomposition.

```
account = contrib * 100 / (var@j(1,3,1));
print account;
```

Output 10.1.7. Innovation Account

```
                              ACCOUNT

                   55.363835 5.3926331 0.5085336
                   5.7525574 31.463951 0.0230775
                   7.3017604  25.68651  3.615556
```

Kalman Filter Subroutines

This section describes a collection of Kalman filtering and smoothing subroutines for time series analysis; immediately following are three examples using Kalman filtering subroutines. The state space model is a method for analyzing a wide range of time series models. When the time series is represented by the state space model (SSM), the Kalman filter is used for filtering, prediction, and smoothing of the state vector. The state space model is composed of the measurement and transition equations.

The following Kalman filtering and smoothing subroutines are supported:

KALCVF   performs covariance filtering and prediction

KALCVS   performs fixed-interval smoothing

KALDFF   performs diffuse covariance filtering and prediction

KALDFS   performs diffuse fixed-interval smoothing

Getting Started

The measurement (or observation) equation can be written

$$\mathbf{y}_t = \mathbf{b}_t + \mathbf{H}_t \mathbf{z}_t + \epsilon_t$$

where $\mathbf{b}_t$ is an $N_y \times 1$ vector, $\mathbf{H}_t$ is an $N_y \times N_z$ matrix, the sequence of observation noise ϵ_t is independent, $\mathbf{z}_t$ is an $N_z \times 1$ state vector, and $\mathbf{y}_t$ is an $N_y \times 1$ observed vector.

The transition (or state) equation is denoted as a first-order Markov process of the state vector.

$$\mathbf{z}_{t+1} = \mathbf{a}_t + \mathbf{F}_t \mathbf{z}_t + \eta_t$$

where $\mathbf{a}_t$ is an $N_z \times 1$ vector, $\mathbf{F}_t$ is an $N_z \times N_z$ transition matrix, and the sequence of transition noise η_t is independent. This equation is often called a *shifted transition equation* because the state vector is shifted forward one time period. The transition equation can also be denoted using an alternative specification

$$\mathbf{z}_t = \mathbf{a}_t + \mathbf{F}_t \mathbf{z}_{t-1} + \eta_t$$

There is no real difference between the shifted transition equation and this alternative equation if the observation noise and transition equation noise are uncorrelated, that is, $E(\eta_t \epsilon_t') = 0$. It is assumed that

$$
\begin{aligned}
E(\eta_t \eta_s') &= \mathbf{V}_t \delta_{ts} \\
E(\epsilon_t \epsilon_s') &= \mathbf{R}_t \delta_{ts} \\
E(\eta_t \epsilon_s') &= \mathbf{G}_t \delta_{ts}
\end{aligned}
$$

where

$$
\delta_{ts} = \left\{ \begin{array}{ll} 1 & \text{if } t = s \\ 0 & \text{if } t \neq s \end{array} \right.
$$

De Jong (1991a) proposed a diffuse Kalman filter that can handle an arbitrarily large initial state covariance matrix. The diffuse initial state assumption is reasonable if you encounter the case of parameter uncertainty or SSM nonstationarity. The SSM of the diffuse Kalman filter is written

$$
\begin{aligned}
\mathbf{y}_t &= \mathbf{X}_t \beta + \mathbf{H}_t \mathbf{z}_t + \epsilon_t \\
\mathbf{z}_{t+1} &= \mathbf{W}_t \beta + \mathbf{F}_t \mathbf{z}_t + \eta_t \\
\mathbf{z}_0 &= \mathbf{a} + \mathbf{A}\delta \\
\beta &= \mathbf{b} + \mathbf{B}\delta
\end{aligned}
$$

where δ is a random variable with a mean of μ and a variance of $\sigma^2 \Sigma$. When $\Sigma \to \infty$, the SSM is said to be diffuse.

The KALCVF call computes the one-step prediction $\mathbf{z}_{t+1|t}$ and the filtered estimate $\mathbf{z}_{t|t}$, together with their covariance matrices $\mathbf{P}_{t+1|t}$ and $\mathbf{P}_{t|t}$, using forward recursions. You can obtain the k-step prediction $\mathbf{z}_{t+k|t}$ and its covariance matrix $\mathbf{P}_{t+k|t}$ with the KALCVF call. The KALCVS call uses backward recursions to compute the smoothed estimate $\mathbf{z}_{t|T}$ and its covariance matrix $\mathbf{P}_{t|T}$ when there are T observations in the complete data.

The KALDFF call produces one-step prediction of the state and the unobserved random vector δ as well as their covariance matrices. The KALDFS call computes the smoothed estimate $\mathbf{z}_{t|T}$ and its covariance matrix $\mathbf{P}_{t|T}$.

Syntax

CALL KALCVF(*pred, vpred, filt, vfilt, data, lead,* a, f, b, h,

var $<$, *z0, vz0*$>$**);**

CALL KALCVS(*sm, vsm, data,* a*,* f*,* b*,* h*, var, pred, vpred* $<$*,un, vun>***);**

CALL KALDFF(*pred, vpred, initial, s2, data, lead, int, coef, var,*

 intd, coefd $<$*, n0, at, mt, qt>***);**

CALL KALDFS(*sm, vsm, data, int, coef, var, bvec, bmat, initial, at,*

 mt, $s2$ $<$*, un, vun>***);**

Example 10.2. Kalman   Filtering: Likelihood   Function
Evaluation

In this example, the log-likelihood function of the SSM is computed using prediction error decomposition. The annual real GNP series, y_t, can be decomposed as

$$y_t = \mu_t + \epsilon_t$$

where μ_t is a trend component and ϵ_t is a white noise error with $\epsilon_t \sim (0, \sigma_\epsilon^2)$. Refer to Nelson and Plosser (1982) for more details on these data. The trend component is assumed to be generated from the following stochastic equations:

$$
\begin{aligned}
\mu_t &= \mu_{t-1} + \beta_{t-1} + \eta_{1t} \\
\beta_t &= \beta_{t-1} + \eta_{2t}
\end{aligned}
$$

where η_{1t} and η_{2t} are independent white noise disturbances with $\eta_{1t} \sim (0, \sigma_{\eta_1}^2)$ and $\eta_{2t} \sim (0, \sigma_{\eta_2}^2)$.

It is straightforward to construct the SSM of the real GNP series.

$$
\begin{aligned}
y_t &= \mathbf{H}\mathbf{z}_t + \epsilon_t \\
\mathbf{z}_t &= \mathbf{F}\mathbf{z}_{t-1} + \eta_t
\end{aligned}
$$

where

$$\mathbf{H} = (1, 0)$$

$$\mathbf{F} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$
\begin{aligned}
\mathbf{z}_t &= (\mu_t, \beta_t)' \\
\eta_t &= (\eta_{1t}, \eta_{2t})'
\end{aligned}
$$

$$
\text{Var}\left(\begin{bmatrix} \eta_t \\ \epsilon_t \end{bmatrix}\right) = \begin{bmatrix} \sigma_{\eta 1}^2 & 0 & 0 \\ 0 & \sigma_{\eta 2}^2 & 0 \\ 0 & 0 & \sigma_\epsilon^2 \end{bmatrix}
$$

Example 10.2. Kalman Filtering: Likelihood Function Evaluation ♦ 299

When the observation noise ϵ_t is normally distributed, the average log-likelihood function of the SSM is

$$\ell = \frac{1}{T}\sum_{t=1}^{T}\ell_t$$

$$\ell_t = -\frac{N_y}{2}\log(2\pi) - \frac{1}{2}\log(|\mathbf{C}_t|) - \frac{1}{2}\hat{\epsilon}_t'\mathbf{C}_t^{-1}\hat{\epsilon}_t$$

where $\mathbf{C}_t$ is the mean square error matrix of the prediction error $\hat{\epsilon}_t$, such that $\mathbf{C}_t = \mathbf{HP}_{t|t-1}\mathbf{H}' + \mathbf{R}_t$.

The LIK module computes the average log-likelihood function. First, the average log-likelihood function is computed using the default initial values: Z0=0 and VZ0=10^6I. The second call of module LIK produces the average log-likelihood function with the given initial conditions: Z0=0 and VZ0=10^{-3}I. You can notice a sizable difference between the uncertain initial condition (VZ0=10^6I) and the almost deterministic initial condition (VZ0=10^{-3}I) in Output 10.2.1.

Finally, the first 15 observations of one-step predictions, filtered values, and real GNP series are produced under the moderate initial condition (VZ0=10I). The data are the annual real GNP for the years 1909 to 1969.

```
title 'Likelihood Evaluation of SSM';
title2 'DATA: Annual Real GNP 1909-1969';
data gnp;
   input y @@;
datalines;
   116.8 120.1 123.2 130.2 131.4 125.6 124.5 134.3
   135.2 151.8 146.4 139.0 127.8 147.0 165.9 165.5
   179.4 190.0 189.8 190.9 203.6 183.5 169.3 144.2
   141.5 154.3 169.5 193.0 203.2 192.9 209.4 227.2
   263.7 297.8 337.1 361.3 355.2 312.6 309.9 323.7
   324.1 355.3 383.4 395.1 412.8 406.0 438.0 446.1
   452.5 447.3 475.9 487.7 497.2 529.8 551.0 581.1
   617.8 658.1 675.2 706.6 724.7
   ;

 proc iml;
   start lik(y,a,b,f,h,var,z0,vz0);
       nz = nrow(f);
       n = nrow(y);
       k = ncol(y);
       const = k*log(8*atan(1));
       if ( sum(z0 = .) | sum(vz0 = .) ) then
          call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var);
       else
          call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var,z0,vz0);
       et = y - pred*h';
       sum1 = 0;
       sum2 = 0;
       do i = 1 to n;
```

```
            vpred_i = vpred[(i-1)*nz+1:i*nz,];
            et_i = et[i,];
            ft = h*vpred_i*h' + var[nz+1:nz+k,nz+1:nz+k];
            sum1 = sum1 + log(det(ft));
            sum2 = sum2 + et_i*inv(ft)*et_i';
        end;
        return(-const-.5*(sum1+sum2)/n);
    finish;

start main;
    use gnp;
    read all var {y};

    f = {1 1, 0 1};
    h = {1 0};
    a = j(nrow(f),1,0);
    b = j(nrow(h),1,0);
    var = diag(j(1,nrow(f)+ncol(y),1e-3));
    /*-- initial values are computed --*/
    z0 = j(1,nrow(f),.);
    vz0 = j(nrow(f),nrow(f),.);
    logl = lik(y,a,b,f,h,var,z0,vz0);
    print 'No initial values are given', logl;
    /*-- initial values are given --*/
    z0 = j(1,nrow(f),0);
    vz0 = 1e-3#i(nrow(f));
    logl = lik(y,a,b,f,h,var,z0,vz0);
    print 'Initial values are given', logl;
    z0 = j(1,nrow(f),0);
    vz0 = 10#i(nrow(f));
    call kalcvf(pred,vpred,filt,vfilt,y,1,a,f,b,h,var,z0,vz0);
    print y pred filt;
finish;
run;
```

Output 10.2.1. Average Log Likelihood of SSM

```
                Likelihood Evaluation of SSM
               DATA: Annual Real GNP 1909-1969


                No initial values are given
                          LOGL


                          -26314.66



                Initial values are given
                          LOGL


                          -91884.41
```

Output 10.2.2 shows the observed data, the predicted state vectors, and the filtered state vectors for the first 16 observations.

Output 10.2.2. Filtering and One-Step Prediction

```
            Y         PRED               FILT

          116.8          0          0 116.78832          0
          120.1 116.78832          0 120.09967 3.3106857
          123.2 123.41035 3.3106857 123.22338 3.1938303
          130.2 126.41721 3.1938303 129.59203 4.8825531
          131.4 134.47459 4.8825531 131.93806 3.5758561
          125.6 135.51391 3.5758561 127.36247 -0.610017
          124.5 126.75246 -0.610017 124.90123 -1.560708
          134.3 123.34052 -1.560708 132.34754 3.0651076
          135.2 135.41265 3.0651076 135.23788 2.9753526
          151.8 138.21324 2.9753526 149.37947 8.7100967
          146.4 158.08957 8.7100967 148.48254 3.7761324
            139 152.25867 3.7761324 141.36208  -1.82012
          127.8 139.54196  -1.82012 129.89187 -6.776195
            147 123.11568 -6.776195 142.74492 3.3049584
          165.9 146.04988 3.3049584 162.36363 11.683345
          165.5 174.04698 11.683345 167.02267  8.075817
```

Example 10.3. Kalman Filtering: Estimating an SSM Using the EM Algorithm

This example estimates the normal SSM of the mink-muskrat data using the EM algorithm. The mink-muskrat series are detrended. Refer to Harvey (1989) for details of this data set. Since this EM algorithm uses filtering and smoothing, you can learn how to use the KALCVF and KALCVS calls to analyze the data. Consider the bivariate SSM:

$$\mathbf{y}_t = \mathbf{H}\mathbf{z}_t + \epsilon_t$$
$$\mathbf{z}_t = \mathbf{F}\mathbf{z}_{t-1} + \eta_t$$

where $\mathbf{H}$ is a 2×2 identity matrix, the observation noise has a time invariant covariance matrix $\mathbf{R}$, and the covariance matrix of the transition equation is also assumed to be time invariant. The initial state $\mathbf{z}_0$ has mean μ and covariance Σ. For estimation, the Σ matrix is fixed as

$$\begin{bmatrix} 0.1 & 0.0 \\ 0.0 & 0.1 \end{bmatrix}$$

while the mean vector μ is updated by the smoothing procedure such that $\hat{\mu} = \mathbf{z}_{0|T}$. Note that this estimation requires an extra smoothing step since the usual smoothing procedure does not produce $\mathbf{z}_{T|0}$.

The EM algorithm maximizes the expected log-likelihood function given the current parameter estimates. In practice, the log-likelihood function of the normal SSM is evaluated while the parameters are updated using the M-step of the EM maximization

$$\mathbf{F}^{i+1} = \mathbf{S}_t(1)[\mathbf{S}_{t-1}(0)]^{-1}$$
$$\mathbf{V}^{i+1} = \frac{1}{T}\left(\mathbf{S}_t(0) - \mathbf{S}_t(0)[\mathbf{S}_{t-1}(0)]^{-1}\mathbf{S}'_t(1)\right)$$

$$\mathbf{R}^{i+1} = \frac{1}{T}\sum_{t=1}^{T}\left[(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|T})(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|T})' + \mathbf{H}\mathbf{P}_{t|T}\mathbf{H}'\right]$$

$$\mu^{i+1} = \mathbf{z}_{0|T}$$

where the index i represents the current iteration number, and

$$\mathbf{S}_t(0) = \sum_{t=1}^{T}(\mathbf{P}_{t|T} + \mathbf{z}_{t|T}\mathbf{z}'_{t|T}),$$

$$\mathbf{S}_t(1) = \sum_{t=1}^{T}(\mathbf{P}_{t,t-1|T} + \mathbf{z}_{t|T}\mathbf{z}'_{t-1|T})$$

It is necessary to compute the value of $\mathbf{P}_{t,t-1|T}$ recursively such that

$$\mathbf{P}_{t-1,t-2|T} = \mathbf{P}_{t-1|t-1}\mathbf{P}^{*\prime}_{t-2} + \mathbf{P}^{*}_{t-1}(\mathbf{P}_{t,t-1|T} - \mathbf{F}\mathbf{P}_{t-1|t-1})\mathbf{P}^{*\prime}_{t-2}$$

where $\mathbf{P}^{*}_t = \mathbf{P}_{t|t}\mathbf{F}'\mathbf{P}^{-}_{t+1|t}$ and the initial value $\mathbf{P}_{T,T-1|T}$ is derived using the formula

$$\mathbf{P}_{T,T-1|T} = \left[\mathbf{I} - \mathbf{P}_{t|t-1}\mathbf{H}'(\mathbf{H}\mathbf{P}_{t|t-1}\mathbf{H}' + \mathbf{R})\mathbf{H}\right]\mathbf{F}\mathbf{P}_{T-1|T-1}$$

Note that the initial value of the state vector is updated for each iteration

$$\mathbf{z}_{1|0} = \mathbf{F}\mu^{i}$$

$$\mathbf{P}_{1|0} = \mathbf{F}^{i}\Sigma\mathbf{F}^{i\prime} + \mathbf{V}^{i}$$

The objective function value is computed as -2ℓ in the IML module LIK. The log-likelihood function is written

$$\ell = -\frac{1}{2}\sum_{t=1}^{T}\log(|\mathbf{C}_t|) - \frac{1}{2}\sum_{t=1}^{T}(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|t-1})\mathbf{C}_t^{-1}(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|t-1})'$$

where $\mathbf{C}_t = \mathbf{H}\mathbf{P}_{t|t-1}\mathbf{H}' + \mathbf{R}$.

The iteration history is shown in Output 10.3.1. As shown in Output 10.3.2, the eigenvalues of $\mathbf{F}$ are within the unit circle, which indicates that the SSM is stationary. However, the muskrat series (Y1) is reported to be difference stationary. The estimated parameters are almost identical to those of the VAR(1) estimates. Refer to Harvey (1989, p. 469). Finally, multistep forecasts of $\mathbf{y}_t$ are computed using the KALCVF call.

```
call kalcvf(pred,vpred,filt,vfilt,y,15,a,f,b,h,var,z0,vz0);
```

The predicted values of the state vector $\mathbf{z}_t$ and their standard errors are shown in Output 10.3.3.

```
      title 'SSM Estimation using EM Algorithm';
      data one;
         input y1 y2 @@;
      datalines;
         . . . data lines omitted . . .
         ;

      proc iml;
         start lik(y,pred,vpred,h,rt);
             n = nrow(y);
             nz = ncol(h);
             et = y - pred*h';
             sum1 = 0;
             sum2 = 0;
             do i = 1 to n;
                vpred_i = vpred[(i-1)*nz+1:i*nz,];
                et_i = et[i,];
                ft = h*vpred_i*h' + rt;
                sum1 = sum1 + log(det(ft));
                sum2 = sum2 + et_i*inv(ft)*et_i';
             end;
             return(sum1+sum2);
         finish;

         start main;
           use one;
           read all into y var {y1 y2};
          /*-- mean adjust series --*/
           t = nrow(y);
           ny = ncol(y);
           nz = ny;
           f = i(nz);
           h = i(ny);

           /*-- observation noise variance is diagonal --*/
           rt = 1e-5#i(ny);

           /*-- transition noise variance --*/
           vt = .1#i(nz);
           a = j(nz,1,0);
           b = j(ny,1,0);
           myu = j(nz,1,0);
           sigma = .1#i(nz);
           converge = 0;
           do iter = 1 to 100 while( converge = 0 );


               /*--- construct big cov matrix --*/
               var = ( vt || j(nz,ny,0) ) //
                     ( j(ny,nz,0) || rt );

               /*-- initial values are changed --*/
               z0  = myu' * f';
               vz0 = f * sigma * f' + vt;

               /*-- filtering to get one-step prediction and filtered value --*/
               call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var,z0,vz0);

               /*-- smoothing using one-step prediction values --*/
```

```
              call kalcvs(sm,vsm,y,a,f,b,h,var,pred,vpred);

              /*-- compute likelihood values --*/
              logl = lik(y,pred,vpred,h,rt);

              /*-- store old parameters and function values --*/
              myu0 = myu;
              f0 = f;
              vt0 = vt;
              rt0 = rt;
              logl0 = logl;
              itermat = itermat // ( iter || logl0 || shape(f0,1) || myu0` );

              /*-- obtain P*(t) to get P_T_0 and Z_T_0   --*/
              /*-- these values are not usually needed    --*/
              /*-- See Harvey (1989 p154) or Shumway (1988, p177) --*/
              jt1 = sigma * f` * inv(vpred[1:nz,]);
              p_t_0  = sigma + jt1*(vsm[1:nz,] - vpred[1:nz,])*jt1`;
              z_t_0  = myu + jt1*(sm[1,]` - pred[1,]`);
              p_t1_t  = vpred[(t-1)*nz+1:t*nz,];
              p_t1_t1 = vfilt[(t-2)*nz+1:(t-1)*nz,];
              kt = p_t1_t*h`*inv(h*p_t1_t*h`+rt);


              /*-- obtain P_T_TT1. See Shumway (1988, p180) --*/
              p_t_ii1 = (i(nz)-kt*h)*f*p_t1_t1;
              st0 = vsm[(t-1)*nz+1:t*nz,] + sm[t,]`*sm[t,];
              st1 = p_t_ii1 + sm[t,]`*sm[t-1,];
              st00 = p_t_0 + z_t_0 * z_t_0`;
              cov = (y[t,]` - h*sm[t,]`) * (y[t,]` - h*sm[t,]`)` +
                    h*vsm[(t-1)*nz+1:t*nz,]*h`;
              do i = t to 2 by -1;
                 p_i1_i1 = vfilt[(i-2)*nz+1:(i-1)*nz,];
                 p_i1_i  = vpred[(i-1)*nz+1:i*nz,];
                 jt1 = p_i1_i1 * f` * inv(p_i1_i);
                 p_i1_i  = vpred[(i-2)*nz+1:(i-1)*nz,];
                 if ( i > 2 ) then
                    p_i2_i2 = vfilt[(i-3)*nz+1:(i-2)*nz,];
                 else
                    p_i2_i2 = sigma;
                 jt2 = p_i2_i2 * f` * inv(p_i1_i);
                 p_t_i1i2 = p_i1_i1*jt2` + jt1*(p_t_ii1 - f*p_i1_i1)*jt2`;
                 p_t_ii1 = p_t_i1i2;
                 temp = vsm[(i-2)*nz+1:(i-1)*nz,];
                 sm1 = sm[i-1,]`;
                 st0 = st0 + ( temp + sm1 * sm1` );
                 if ( i > 2 ) then
                    st1 = st1 + ( p_t_ii1 + sm1 * sm[i-2,]);
                 else st1 = st1 + ( p_t_ii1 + sm1 * z_t_0`);
                 st00 = st00 + ( temp + sm1 * sm1` );
                 cov = cov + ( h * temp * h` +
                       (y[i-1,]` - h * sm1)*(y[i-1,]` - h * sm1)` );
              end;

              /*-- M-step: update the parameters --*/
              myu = z_t_0;
              f = st1 * inv(st00);
              vt = (st0 - st1 * inv(st00) * st1`)/t;
              rt = cov / t;
```

```
                 /*-- check convergence --*/
                    if ( max(abs((myu - myu0)/(myu0+1e-6))) < 1e-2 &
                         max(abs((f - f0)/(f0+1e-6))) < 1e-2 &
                         max(abs((vt - vt0)/(vt0+1e-6))) < 1e-2 &
                         max(abs((rt - rt0)/(rt0+1e-6))) < 1e-2 &
                         abs((logl-logl0)/(logl0+1e-6)) < 1e-3 ) then
                       converge = 1;
                 end;


                 reset noname;
                 colnm = {'Iter' '-2*log L' 'F11' 'F12' 'F21' 'F22'
                          'MYU11' 'MYU22'};
                 print itermat[colname=colnm format=8.4];
                 eval = teigval(f0);
                 colnm = {'Real' 'Imag' 'MOD'};
                 eval = eval || sqrt((eval#eval)[,+]);
                 print eval[colname=colnm];
                 var = ( vt || j(nz,ny,0) ) //
                       ( j(ny,nz,0) || rt );

                 /*-- initial values are changed --*/
                 z0  = myu' * f';
                 vz0 = f * sigma * f' + vt;
                 free itermat;

                 /*-- multistep prediction --*/
                 call kalcvf(pred,vpred,filt,vfilt,y,15,a,f,b,h,var,z0,vz0);
                 do i = 1 to 15;
                    itermat = itermat // ( i || pred[t+i,] ||
                               sqrt(vecdiag(vpred[(t+i-1)*nz+1:(t+i)*nz,]))' );
                 end;
                 colnm = {'n-Step' 'Z1_T_n' 'Z2_T_n' 'SE_Z1' 'SE_Z2'};
                 print itermat[colname=colnm];
              finish;
              run;
```

Output 10.3.1.   Iteration History

```
                  SSM Estimation using EM Algorithm

      Iter -2*log L      F11      F12      F21      F22     MYU11     MYU22

     1.0000 -154.010   1.0000   0.0000   0.0000   1.0000   0.0000    0.0000
     2.0000 -237.962   0.7952  -0.6473   0.3263   0.5143   0.0530    0.0840
     3.0000 -238.083   0.7967  -0.6514   0.3259   0.5142   0.1372    0.0977
     4.0000 -238.126   0.7966  -0.6517   0.3259   0.5139   0.1853    0.1159
     5.0000 -238.143   0.7964  -0.6519   0.3257   0.5138   0.2143    0.1304
     6.0000 -238.151   0.7963  -0.6520   0.3255   0.5136   0.2324    0.1405
     7.0000 -238.153   0.7962  -0.6520   0.3254   0.5135   0.2438    0.1473
     8.0000 -238.155   0.7962  -0.6521   0.3253   0.5135   0.2511    0.1518
     9.0000 -238.155   0.7962  -0.6521   0.3253   0.5134   0.2558    0.1546
    10.0000 -238.155   0.7961  -0.6521   0.3253   0.5134   0.2588    0.1565
```

Output 10.3.2. Eigenvalues of F Matrix

```
              Real      Imag       MOD

          0.6547534   0.438317 0.7879237
          0.6547534  -0.438317 0.7879237
```

Output 10.3.3. Multistep Prediction

```
         n-Step    Z1_T_n    Z2_T_n      SE_Z1      SE_Z2

             1 -0.055792 -0.587049 0.2437666  0.237074
             2 0.3384325 -0.319505 0.3140478  0.290662
             3 0.4778022 -0.053949 0.3669731 0.3104052
             4 0.4155731 0.1276996 0.4021048 0.3218256
             5 0.2475671 0.2007098  0.419699 0.3319293
             6 0.0661993 0.1835492 0.4268943 0.3396153
             7 -0.067001 0.1157541  0.430752 0.3438409
             8 -0.128831 0.0376316 0.4341532 0.3456312
             9 -0.127107 -0.022581 0.4369411 0.3465325
            10 -0.086466 -0.052931 0.4385978 0.3473038
            11 -0.034319 -0.055293 0.4393282 0.3479612
            12 0.0087379 -0.039546 0.4396666 0.3483717
            13 0.0327466 -0.017459  0.439936 0.3485586
            14 0.0374564 0.0016876 0.4401753 0.3486415
            15 0.0287193 0.0130482  0.440335 0.3487034
```

Example 10.4. Diffuse Kalman Filtering

The nonstationary SSM is simulated to analyze the diffuse Kalman filter call KALDFF. The transition equation is generated using the following formula:

$$\begin{bmatrix} z_{1t} \\ z_{2t} \end{bmatrix} = \begin{bmatrix} 1.5 & -0.5 \\ 1.0 & 0.0 \end{bmatrix} \begin{bmatrix} z_{1t-1} \\ z_{2t-1} \end{bmatrix} + \begin{bmatrix} \eta_{1t} \\ 0 \end{bmatrix}$$

where $\eta_{1t} \sim N(0,1)$. The transition equation is nonstationary since the transition matrix **F** has one unit root.

```
proc iml;
   z_1 = 0; z_2 = 0;
   do i = 1 to 30;
      z = 1.5*z_1 - .5*z_2 + rannor(1234567);
      z_2 = z_1;
      z_1 = z;
      x =  z + .8*rannor(1234578);
      if ( i > 10 ) then y = y // x;
   end;
```

The KALDFF and KALCVF calls produce one-step prediction, and the result shows that two predictions coincide after the fifth observation (Output 10.4.1).

```
      t = nrow(y);
      h = { 1 0 };
```

Example 10.4. Diffuse Kalman Filtering ◆ 307

```
f = { 1.5 -.5, 1 0 };
rt = .64;
vt = diag({1 0});
ny = nrow(h);
nz = ncol(h);
nb = nz;
nd = nz;
a   = j(nz,1,0);
b   = j(ny,1,0);
int = j(ny+nz,nb,0);
coef = f // h;
var = ( vt || j(nz,ny,0) ) //
       ( j(ny,nz,0) || rt );
intd = j(nz+nb,1,0);
coefd = i(nz) // j(nb,nd,0);
at = j(t*nz,nd+1,0);
mt = j(t*nz,nz,0);
qt = j(t*(nd+1),nd+1,0);
n0 = -1;
call kaldff(kaldff_p,dvpred,initial,s2,y,0,int,
            coef,var,intd,coefd,n0,at,mt,qt);
call kalcvf(kalcvf_p,vpred,filt,vfilt,y,0,a,f,b,h,var);
print kalcvf_p kaldff_p;
```

Output 10.4.1.   Diffuse Kalman Filtering

```
                   Diffuse Kalman Filtering

        KALCVF_P                KALDFF_P


              0         0         0         0
       1.441911  0.961274 1.1214871 0.9612746
      -0.882128 -0.267663 -0.882138 -0.267667
      -0.723156 -0.527704 -0.723158 -0.527706
       1.2964969  0.871659 1.2964968 0.8716585
      -0.035692 0.1379633 -0.035692 0.1379633
      -2.698135 -1.967344 -2.698135 -1.967344
      -5.010039 -4.158022 -5.010039 -4.158022
      -9.048134 -7.719107 -9.048134 -7.719107
      -8.993153 -8.508513 -8.993153 -8.508513
      -11.16619 -10.44119 -11.16619 -10.44119
      -10.42932 -10.34166 -10.42932 -10.34166
      -8.331091 -8.822777 -8.331091 -8.822777
      -9.578258 -9.450848 -9.578258 -9.450848
      -6.526855 -7.241927 -6.526855 -7.241927
      -5.218651 -5.813854 -5.218651 -5.813854
       -5.01855 -5.291777  -5.01855 -5.291777
        -6.5699 -6.284522   -6.5699 -6.284522
      -4.613301 -4.995434 -4.613301 -4.995434
      -5.057926  -5.09007 -5.057926  -5.09007
```

The likelihood function for the diffuse Kalman filter under the finite initial covariance
matrix Σ_δ is written

$$\lambda(\mathbf{y}) = -\frac{1}{2}[\mathbf{y}^{\#}\log(\hat{\sigma}^2) + \sum_{t=1}^{T}\log(|\mathbf{D}_t|)]$$

where $\mathbf{y}^{(\#)}$ is the dimension of the matrix $(\mathbf{y}'_1, \cdots, \mathbf{y}'_T)'$. The likelihood function for the diffuse Kalman filter under the diffuse initial covariance matrix $(\Sigma_\delta \to \infty)$ is computed as $\lambda(\mathbf{y}) - \frac{1}{2}\log(|\mathbf{S}|)$, where the $\mathbf{S}$ matrix is the upper $N_\delta \times N_\delta$ matrix of $\mathbf{Q}_t$. Output 10.4.2 on page 308 displays the log likelihood and the diffuse log-likelihood.

```
d = 0;
do i = 1 to t;
   dt = h*mt[(i-1)*nz+1:i*nz,]*h` + rt;
   d = d + log(det(dt));
end;
s = qt[(t-1)*(nd+1)+1:t*(nd+1)-1,1:nd];
log_l = -(t*log(s2) + d)/2;
dff_logl = log_l - log(det(s))/2;
print log_l dff_logl;
```

Output 10.4.2. Diffuse Likelihood Function

```
                   Diffuse Kalman Filtering

                                   LOG_L

          Log L          -11.42547
                                 DFF_LOGL

          Diffuse Log L -9.457596
```

Vector Time Series Analysis Subroutines

Vector time series analysis involves more than one dependent time series variable, with possible interrelations or feedback between the dependent variables.

The VARMASIM function gerernates various time series from the underlying VARMA models. Simulations of time series with known VARMA structure offer learning and developing vector time series analysis skills.

The VARMACOV subroutine provides the pattern of the autocovariance function of VARMA models and helps to identify and fit a proper model.

The VARMALIK subroutine provides the log-likelihood of a VARMA model and helps to obtainie estimates of the parameters of a regression model.

The following subroutines are supported:

VARMACOV   theoretical cross covariances for a multivariate ARMA model

VARMALIK   log-likelihood function for a multivariate ARMA model

VARMASIM   multivariate ARMA time series

VNORMAL   multivariate normal random series

VTSROOT   characteristic roots of a multivariate ARMA model

Getting Started

Stationary VAR Process

Generate the process following the first order stationary vector autoregressive model with zero mean

$$\mathbf{y}_t = \begin{pmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{pmatrix} \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t \ \text{ with } \ \Sigma = \begin{pmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{pmatrix}$$

The following statements compute the roots of characteristic function, compute the five lags of cross-covariance matrices, generate 100 observations simulated data, and evaluate the log-likelihood function of the VAR(1) model:

```
proc iml;
   /* Stationary VAR(1) model */
   sig = {1.0  0.5, 0.5 1.25};
   phi = {1.2 -0.5, 0.6 0.3};
   call varmasim(yt,phi) sigma = sig n = 100 seed=3243; print yt;
   call vtsroot(root,phi); print root;
   call varmacov(crosscov,phi) sigma = sig lag = 5;
   lag = {'0','','1','','2','','3','','4','','5',''};
   print lag crosscov;
   call varmalik(lnl,yt,phi) sigma = sig; print lnl;
```
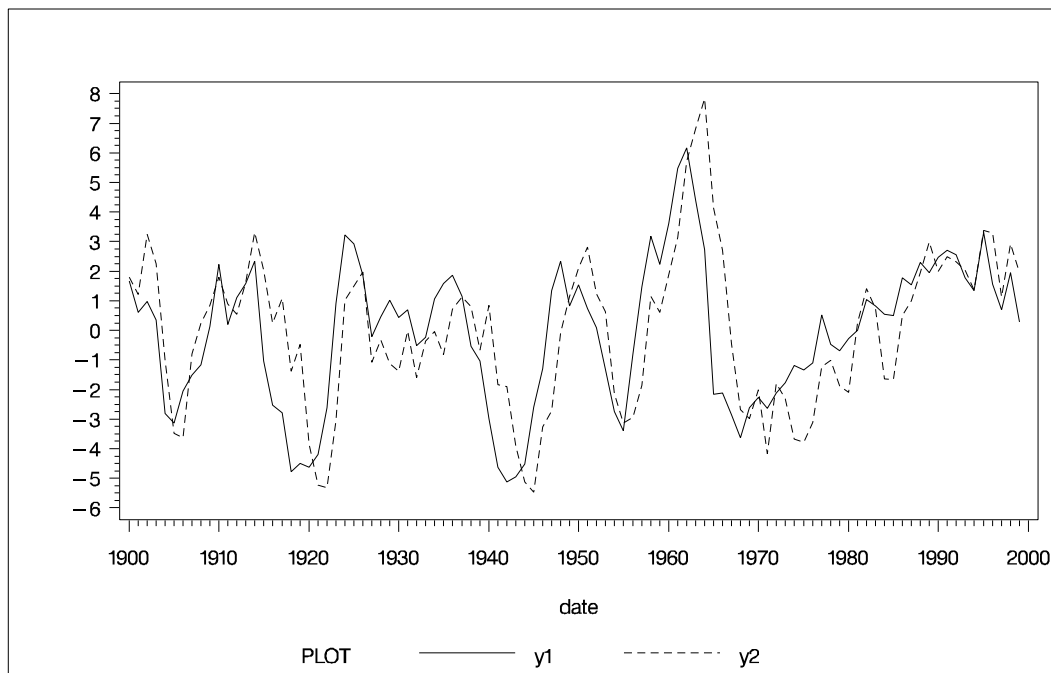

Figure 10.28. Plot of Generated VAR(1) Process (VARMASIM)

The stationary VAR(1) processes show in Figure 10.28.

```
                                  ROOT

                0.75 0.3122499 0.8124038 0.3945069 22.603583
                0.75  -0.31225 0.8124038 -0.394507 -22.60358
```

Figure 10.29. Roots of VAR(1) Model (VTSROOT)

In Figure 10.29, the first column is the real part (R) of the root of the characterstic function and the second one is the imaginary part (I). The third column is the moduls, that is the squared root of $R^2 + I^2$. The fourth column is $Tan^{-1}(I/R)$ and the last one is the degree. Since moduli are less than one from the third column, the series is obviously stationary.

```
                       LAG   CROSSCOV

                        0    5.3934173 3.8597124
                             3.8597124 5.0342051
                        1    4.5422445 4.3939641
                             2.1145523  3.826089
                        2    3.2537114 4.0435359
                             0.6244183 2.4165581
                        3    1.8826857 3.1652876
                             -0.458977 1.0996184
                        4     0.676579 2.0791977
                             -1.100582 0.0544993
                        5    -0.227704 1.0297067
                             -1.347948 -0.643999
```

Figure 10.30. Cross-covariance matrices of VAR(1) Model (VARMACOV)

In each matrix in Figure 10.30, the diagonal elements are corresponding to the autoco-variance functions of each time series. The off-diagonal elements are corresponding to the cross-covariance functions of between two series.

```
                                  LNL

                             -113.4708
                              2.5058678
                              224.43567
```

Figure 10.31. Log-likelihood function of VAR(1) Model (VARMALIK)

In Figure 10.31, the first row is the value of log-likelihood function; the second row is the sum of log determinant of the innovation variance; the last row is the weighted sum of squares of residulas.

Nonstationary VAR Process

Generate the process following the Error correction model with a cointegrated rank of 1:

$$(1 - B)\mathbf{y}_t = \begin{pmatrix} -0.4 \\ 0.1 \end{pmatrix} (1 - 2)\mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t$$

with

$$\Sigma = \left(\begin{array}{cc} 100 & 0 \\ 0 & 100 \end{array} \right) \text{ and } \mathbf{y}_0 = 0$$

The following statements compute the roots of characteristic function, generate simulated data.

```
proc iml;
   /* Nonstationary  model */
   sig = 100*i(2);
   phi = {0.6 0.8, 0.1 0.8};
   call varmasim(yt,phi) sigma = sig n = 100 seed=1324;
   call vtsroot(root,phi); print root;
   print yt;
```
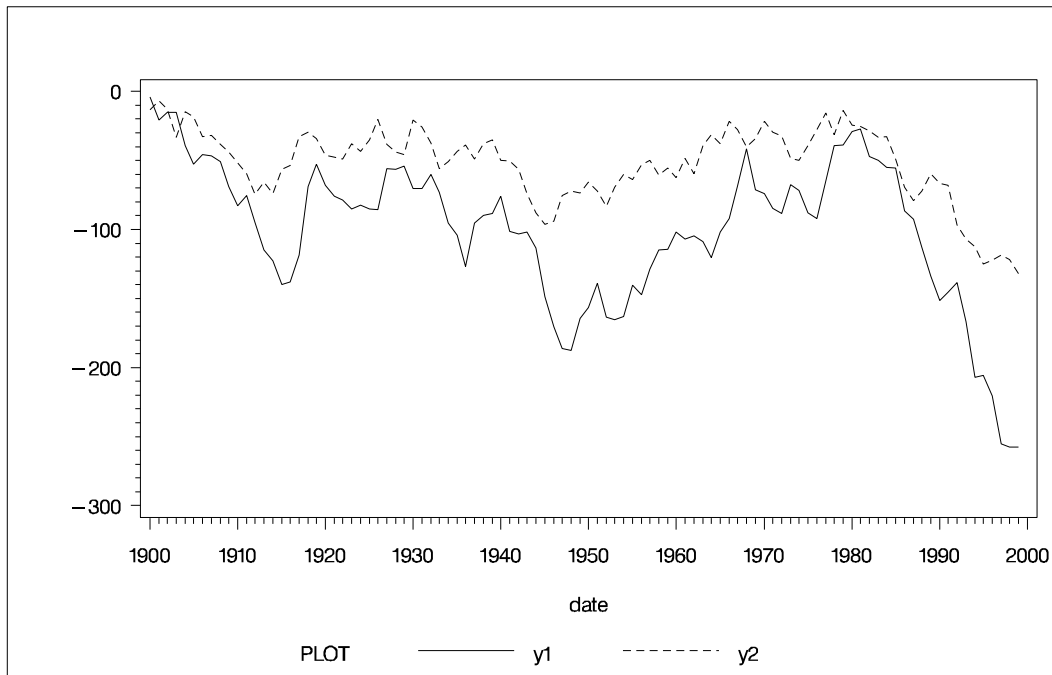

Figure 10.32.   Plot of Generated Nonstationary Vector Process (VARMASIM)

The nonstationary processes show in Figure 10.32 and have a covmovement.

ROOT				
1	0	1	0	0
0.4	0	0.4	0	0

Figure 10.33.   Roots of Nonstationary VAR(1) Model (VTSROOT)

In Figure 10.33, the first column is the real part (R) of the root of the characterstic function and the second one is the imaginary part (I). The third column is the moduls, that is the squared root of $R^2 + I^2$. The fourth column is $Tan^{-1}(I/R)$ and the last one

is the degree. Since the moduli are greater than equal to one from the third column, the series is obviously nonstationary.

Syntax

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

CALL VARMALIK(*lnl, series, phi, theta, sigma <, p, q, opt>* **);**

CALL VARMASIM(*series, phi, theta, mu, sigma, n <, p, q, initial, seed>***);**

CALL VNORMAL(*series, mu, sigma, n <, seed>***);**

CALL VTSROOT(*root, phi, theta<, p, q>***);**

Fractionally Integrated Time Series Analysis

This section describes subroutines related to fractionally integrated time series analysis. The phenomenon of long memory can be observed in hydrology, finance, economics, and so on. Unlike with a stationary process, the correlations between observations of a long memory series are slowly decaying to zero.

The following subroutines are supported:

FARMACOV   computes the autocovariance function for a fractionally integrated ARMA model

FARMAFIT   estimates the parameters for a fractionally integrated ARMA model

FARMALIK   computes the log-likelihood function for a fractionally integrated ARMA model

FARMASIM   generates a fractionally integrated ARMA process

FDIF   computes a fractionally differenced process

Getting Started

The fractional differencing enables the degree of differencing d to take any real value rather than being restricted to integer values. The fractionally differenced processes are capable of modeling long-term persistence. The process

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

is known as a fractional Gaussian noise process or an ARFIMA$(0, d, 0)$ process, where $d \in (-1, 1) \backslash \{0\}$, ϵ_t is a white noise process with mean zero and variance σ_ϵ^2, and B is the backshift operator such that $B^j \mathbf{y}_t = \mathbf{y}_{t-j}$. The extension of an ARFIMA$(0, d, 0)$ model combines fractional differencing with an ARMA(p, q) model, known as an ARFIMA(p, d, q) model.

Consider an ARFIMA$(0, 0.4, 0)$ represented as $(1 - B)^{0.4} y_t = \epsilon_t$ where $\epsilon_t \sim iid\ N(0, 2)$. With the following statements you can

- generate the simulated 300 observations data
- obtain the fractionally differenced data
- compute the autocovariance function
- compute the log-likelihood function
- fit a fractionally integrated time series model to the data

```
proc iml;
   /* ARFIMA(0,0.4,0) */
   lag = (0:12)`;
   call farmacov(autocov_D_IS_04, 0.4);
   call farmacov(D_IS_005, 0.05);
   print lag autocov_D_IS_04 D_IS_005;
   d = 0.4;
   call farmasim(yt, d) n = 300 sigma = 2 seed=5345; print yt;
   call fdif(zt, yt, d); print zt;
   call farmalik(lnl, yt, d); print lnl;
   call farmafit(d, ar, ma, sigma, yt); print d sigma;
```
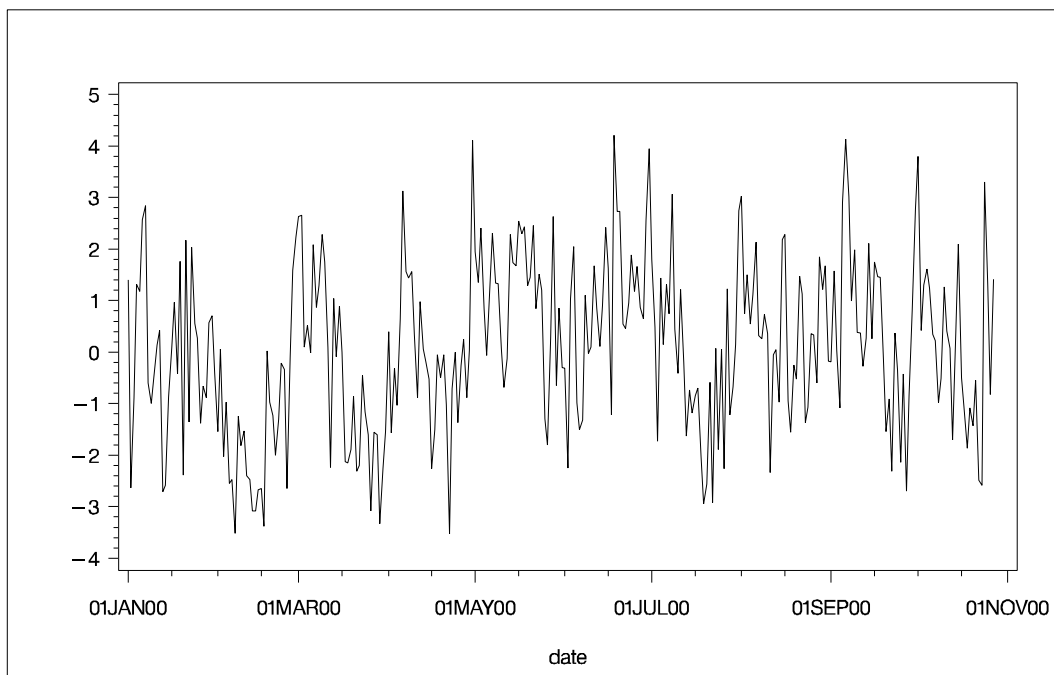

Figure 10.34.   Plot of Generated ARFIMA(0,0.4,0) Process (FARMASIM)

The FARMASIM function generates the data shown in Figure 10.34.
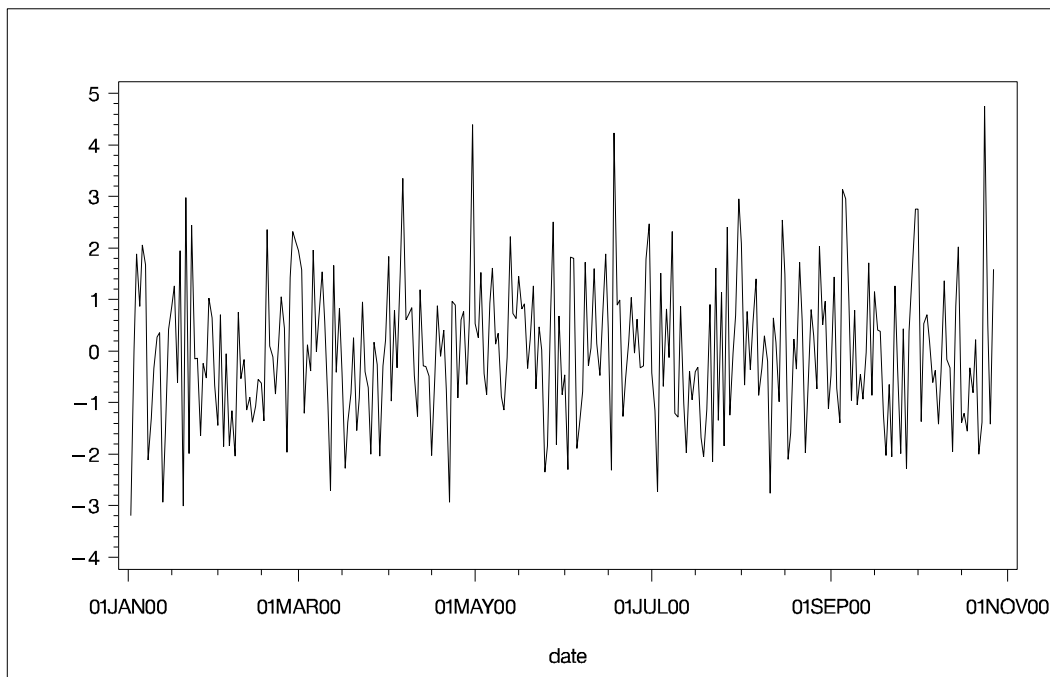
Figure 10.35. Plot of Fractionally Differenced Process (FDIF)

The FDIF function creates the fractionally differenced process. Figure 10.35 looks a white noise series.

```
LAG AUTOCOV_D_IS_04  D_IS_005

 0       2.0700983 1.0044485
 1       1.3800656 0.0528657
 2       1.2075574 0.0284662
 3       1.1146683 0.0197816
 4       1.0527423 0.0152744
 5       1.0069709 0.0124972
 6       0.9710077 0.0106069
 7       0.9415832 0.0092333
 8       0.9168047  0.008188
 9       0.8954836 0.0073647
10       0.8768277 0.0066985
11       0.8602838  0.006148
12       0.8454513 0.0056849
```

Figure 10.36. Autocovariance Functions of ARFIMA(0,0.4,0) and
ARFIMA(0,0.05,0) Models (FARMACOV)

The first column is the autocovariance function of the ARFIMA(0,0.4,0) model, and the second column is the autocovariance function of the ARFIMA(0,0.05,0) model. The first column decays to zero more slowly than the second column.

```
                        LNL

                     -101.2222
                         .
                         .
```

Figure 10.37. Log-likelihood Function of ARFIMA(0,0.4,0) Model (FARMALIK)

The first row value is the log-likelihood function of the ARFIMA(0,0.4,0) model. Since the default option of the estimates method is the conditional sum of squares, the last two rows of Figure 10.37 do not have the values since the default estimation method is used.

```
                   D       SIGMA

              0.386507 1.9631754
```

Figure 10.38. Parameter Estimation of ARFIMA(0,0.4,0) Model (FARMAFIT)

The final estimates of the parameters are $d = 0.387$ and $\sigma^2 = 1.96$, while the true values of the data generating process are $d = 0.4$ and $\sigma^2 = 2$.

Syntax

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

CALL FARMAFIT(*d, phi, theta, sigma, series <, p, q, opt>***);**

CALL FARMALIK(*lnl, series, d <, phi, theta, sigma, p, q, opt>***);**

CALL FARMASIM(*series, d <, phi, theta, mu, sigma, n, p, q, initial, seed>***);**

CALL FDIF(*out, series, d***);**

References

Afifi, A.A. and Elashoff, R.M. (1967), "Missing Observations in Multivariate Statistics II. Point Estimation in Simple Linear Regression," *Journal of the American Statistical Association*, 62, 10–29.

Akaike, H. (1974), "A New Look at Statistical Model Identification," *IEEE Transactions on Automatic Control*, 19, 716–723.

Akaike, H. (1977), "On Entropy Maximization Principle," in *Applications of Statistics*, ed. P.R. Krishnaiah, Amsterdam: North-Holland Publishing Co., 27–41.

Akaike, H. (1978a), "A Bayesian Analysis of the Minimum AIC Procedure," *Ann. Inst. Statist. Math.*, 30, 9–14.

Akaike, H. (1978b), "Time Series Analysis and Control through Parametric Models," in *Applied Time Series Analysis*, ed. D.F. Findley, New York: Academic Press, 1–23.

Akaike, H. (1979), "A Bayesian Extension of the Minimum AIC Procedure of Autoregressive Model Fitting," *Biometrika*, 66, 237–242.

Akaike, H. (1980a), "Likelihood and the Bayes Procedure," *Bay Statistics*, eds. J.M. Bernardo, M.H. DeGroot, D.V. Lindley, and M. Smith, Valencia, Spain: University Press.

Akaike, H. (1980b), "Seasonal Adjustment by a Bayesian Modeling," *Journal of Time Series Analysis*, 1, 1–13.

Akaike, H. (1981), "Likelihood of a Model and Information Criteria," *Journal of Econometrics*, 16, 3–14.

Akaike, H. (1986), "The Selection of Smoothness Priors for Distributed Lag Estimation," in *Bayesian Inference and Decision Techniques*, ed. P. Goel and A. Zellner, Elsevier Science Publishers, 109–118.

Akaike, H. and Ishiguro, M. (1980), "Trend Estimation with Missing Observations," *Ann. Inst. Statist. Math.*, 32, 481–488.

Akaike, H. and Nakagawa, T. (1988), *Statistical Analysis and Control of Dynamic Systems*, Tokyo: KTK Scientific Publishers.

Anderson, T.W. (1971), *The Statistical Analysis of Time Series*, New York: John Wiley & Sons, Inc.

Ansley, C.F. (1980), "Computation of the Theoretical Autocovariance Function for a Vector ARMA Process," *Journal 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.

Brockwell, P.J. and Davis, R.A. (1991), *Time Series: Theory and Methods*, Second Edition, New York: Springer-Verlag.

Chung, C.F. (1996), "A Generalized Fractionally Integrated ARMA Process," *Journal of Time Series Analysis*, 2, 111–140.

De Jong, P. (1991a), "The Diffuse Kalman Filter," *Annals of Statistics*, 19, 1073–1083.

Doan, T., Litterman, R., and Sims, C. (1984), "Forecasting and Conditional Projection using Realistic Prior Distributions," *Econometric Review*, 3, 1–100.

Gersch, W. and Kitagawa, G. (1983), "The Prediction of Time Series with Trends and Seasonalities," *Journal of Business and Economic Statistics*, 1, 253–264.

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.

Granger, C.W.J. and Joyeux, R. (1980), "An Introduction to Long Memory Time Series Models and Fractional Differencing," *Journal of Time Series Analysis*, 1, 15–39.

Harvey, A.C. (1989), *Forecasting, Structural Time Series Models and the Kalman Filter*, Cambridge: Cambridge University Press.

Hosking, J.R.M. (1981), "Fractional Differencing," *Biometrika*, 68, 165–176.

Ishiguro, M. (1984), "Computationally Efficient Implementation of a Bayesian Seasonal Adjustment Procedure," *Journal of Time Series Analysis*, 5, 245–253.

Ishiguro, M. (1987), "TIMSAC-84: A New Time Series Analysis and Control Package," *Proceedings of American Statistical Association: Business and Economic Section*, 33–42.

Jones, R.H. and Brelsford, W.M. (1967), "Time Series with Periodic Structure," *Biometrika*, 54, 403–408.

Kitagawa, G. (1981), "A Nonstationary Time Series Model and Its Fitting by a Recursive Filter," *Journal of Time Series Analysis*, 2, 103–116.

Kitagawa, G. (1983), "Changing Spectrum Estimation," *Journal of Sound and Vibration*, 89, 433–445.

Kitagawa, G. and Akaike, H. (1978), "A Procedure for the Modeling of Non-Stationary Time Series," *Ann. Inst. Statist. Math.*, 30, 351–363.

Kitagawa, G. and Akaike, H. (1981), "On TIMSAC-78," in *Applied Time Series Analysis II*, ed. D.F. Findley, New York: Academic Press, 499–547.

Kitagawa, G. and Akaike, H. (1982), "A Quasi Bayesian Approach to Outlier Detection," *Ann. Inst. Statist. Math.*, 34, 389–398.

Kitagawa, G. and Gersch, W. (1984), "A Smoothness Priors-State Space Modeling of Time Series with Trend and Seasonality," *Journal of the American Statistical Association*, 79, 378–389.

Kitagawa, G. and Gersch, W. (1985a), "A Smoothness Priors Time-Varying AR Coefficient Modeling of Nonstationary Covariance Time Series," *IEEE Transactions on Automatic Control*, 30, 48–56.

Kitagawa, G. and Gersch, W. (1985b), "A Smoothness Priors Long AR Model Method for Spectral Estimation," *IEEE Transactions on Automatic Control*, 30, 57–65.

Kohn, R. and Ansley, C.F. (1982), "A Note on Obtaining the Theoretical Autocovariances of an ARMA Process," *Journal Statistical Computation and Simulation*, 15, 273–283.

Li, W.K. and McLeod, A.I. (1986), "Fractional Time Series Modeling," *Biometrika*, 73, 217–221.

Lütkepohl, H. (1993), *Introduction to Multiple Time Series Analysis*, Berlin: Springer-Verlag.

Mittnik, S. (1990), "Computation of Theoretical Autocovariance Matrices of Multivariate Autoregressive Moving Average Time Series," *Journal of Royal Statistical Society*, B, 52, 151–155.

Nelson, C.P. and Plosser, C.I. (1982), "Trends and Random Walks in Macroeconomic Time Series: Some Evidence and Implications," *Journal of Monetary Economics*, 10, 139–162.

Pagano, M. (1978), "On Periodic and Multiple Autoregressions," *The Annals of Statistics*, 6, 1310–1317.

Reinsel, G.C. (1997), *Elements of Multivariate Time Series Analysis*, Second Edition, New York: Springer-Verlag.

Sakamoto, Y., Ishiguro, M., and Kitagawa, G. (1986), *Akaike Information Criterion Statistics*, Tokyo: KTK Scientific Publishers.

Shiller, R.J. (1973), "A Distributed Lag Estimator Derived from Smoothness Priors," *Econometrica*, 41, 775–788.

Sowell, F. (1992), "Maximum Likelihood Estimation of Stationary Univariate Fractionally Integrated Time Series Models," *Journal of Econometrics*, 53, 165–88.

Shumway, R.H. (1988), *Applied Statistical Time Series Analysis*, Englewood Cliffs, NJ: Prentice-Hall.

Tamura, Y. H. (1986), "An Approach to the Nonstationary Process Analysis," *Ann. Inst. Statist. Math.*, 39, 227–241.

Wei, W.W.S. (1990), *Time Series Analysis: Univariate and Multivariate Methods*, Redwood: Addison-Wesley.

Whittaker, E.T. (1923), "On a New Method of Graduation," *Proceedings of the Edinborough Mathematical Society*, 41, 63–75.

Whittaker, E.T. and Robinson, G. (1944), *Calculus of Observation*, Fourth Edition, London: Blackie & Son Limited.

Zellner, A. (1971), *An Introduction to Bayesian Inference in Econometrics*, New York: John Wiley & Sons, Inc.

Chapter 11
Nonlinear Optimization Examples

Chapter Contents

Chapter 11
Nonlinear Optimization Examples

Overview

The IML procedure offers a set of optimization subroutines for minimizing or maximizing a continuous nonlinear function $f = f(x)$ of n parameters, where $x = (x_1, \ldots, x_n)^T$. The parameters can be subject to boundary constraints and linear or nonlinear equality and inequality constraints. The following set of optimization subroutines is available:

NLPCG	Conjugate Gradient Method
NLPDD	Double Dogleg Method
NLPNMS	Nelder-Mead Simplex Method
NLPNRA	Newton-Raphson Method
NLPNRR	Newton-Raphson Ridge Method
NLPQN	(Dual) Quasi-Newton Method
NLPQUA	Quadratic Optimization Method
NLPTR	Trust-Region Method

The following subroutines are provided for solving nonlinear least-squares problems:

NLPLM	Levenberg-Marquardt Least-Squares Method
NLPHQN	Hybrid Quasi-Newton Least-Squares Methods

A least-squares problem is a special form of minimization problem where the objective function is defined as a sum of squares of other (nonlinear) functions.

$$f(x) = \frac{1}{2} \{ f_1^2(x) + \cdots + f_m^2(x) \}$$

Least-squares problems can usually be solved more efficiently by the least-squares subroutines than by the other optimization subroutines.

The following subroutines are provided for the related problems of computing finite difference approximations for first- and second-order derivatives and of determining a feasible point subject to boundary and linear constraints:

NLPFDD	Approximate Derivatives by Finite Differences
NLPFEA	Feasible Point Subject to Constraints

Each optimization subroutine works iteratively. If the parameters are subject only to linear constraints, all optimization and least-squares techniques are *feasible-point*

methods; that is, they move from feasible point $x^{(k)}$ to a better feasible point $x^{(k+1)}$ by a step in the search direction $s^{(k)}$, $k = 1, 2, 3, \ldots$. If you do not provide a feasible starting point $x^{(0)}$, the optimization methods call the algorithm used in the NLPFEA subroutine, which tries to compute a starting point that is feasible with respect to the boundary and linear constraints.

The NLPNMS and NLPQN subroutines permit nonlinear constraints on parameters. For problems with nonlinear constraints, these subroutines do not use a feasible-point method; instead, the algorithms begin with whatever starting point you specify, whether feasible or infeasible.

Each optimization technique requires a continuous objective function $f = f(x)$ and all optimization subroutines except the NLPNMS subroutine require continuous first-order derivatives of the objective function f. If you do not provide the derivatives of f, they are approximated by finite difference formulas. You can use the NLPFDD subroutine to check the correctness of analytical derivative specifications.

Most of the results obtained from the IML procedure optimization and least-squares subroutines can also be obtained by using the NLP procedure in the SAS/OR product.

The advantages of the IML procedure are as follows:

- You can use matrix algebra to specify the objective function, nonlinear constraints, and their derivatives in IML modules.

- The IML procedure offers several subroutines that can be used to specify the objective function or nonlinear constraints, many of which would be very difficult to write for the NLP procedure.

- You can formulate your own termination criteria by using the *"ptit"* module argument.

The advantages of the NLP procedure are as follows:

- Although identical optimization algorithms are used, the NLP procedure can be much faster because of the interactive and more general nature of the IML product.

- Analytic first- and second-order derivatives can be computed with a special compiler.

- Additional optimization methods are available in the NLP procedure that do not fit into the framework of this package.

- Data set processing is much easier than in the IML procedure. You can save results in output data sets and use them in following runs.

- The printed output contains more information.

Getting Started

Unconstrained Rosenbrock Function

The Rosenbrock function is defined as

$$
\begin{aligned}
f(x) &= \frac{1}{2}\{100(x_2 - x_1^2)^2 + (1 - x_1)^2\} \\
&= \frac{1}{2}\{f_1^2(x) + f_2^2(x)\}, \quad x = (x_1, x_2)
\end{aligned}
$$

The minimum function value $f^* = f(x^*) = 0$ is at the point $x^* = (1, 1)$.

The following code calls the NLPTR subroutine to solve the optimization problem:

```
proc iml;
   title 'Test of NLPTR subroutine: Gradient Specified';
   start F_ROSEN(x);
      y1 = 10. * (x[2] - x[1] * x[1]);
      y2 = 1. - x[1];
      f  = .5 * (y1 * y1 + y2 * y2);
      return(f);
   finish F_ROSEN;

   start G_ROSEN(x);
      g = j(1,2,0.);
      g[1] = -200.*x[1]*(x[2]-x[1]*x[1]) - (1.-x[1]);
      g[2] =  100.*(x[2]-x[1]*x[1]);
      return(g);
   finish G_ROSEN;

   x = {-1.2 1.};
   optn = {0 2};
   call nlptr(rc,xres,"F_ROSEN",x,optn) grd="G_ROSEN";
   quit;
```

The NLPTR is a trust-region optimization method. The F_ROSEN module represents the Rosenbrock function, and the G_ROSEN module represents its gradient. Specifying the gradient can reduce the number of function calls by the optimization subroutine. The optimization begins at the initial point $x = (-1.2, 1)$. For more information on the NLPTR subroutine and its arguments, see the section "NLPTR Call" on page 817. For details on the options vector, which is given by the OPTN vector in the preceding code, see the section "Options Vector" on page 343.

A portion of the output produced by the NLPTR subroutine is shown in Figure 11.1 on page 324.

```
                       Trust Region Optimization

                        Without Parameter Scaling
               CRP Jacobian Computed by Finite Differences

                     Parameter Estimates                  2

                         Optimization Start

Active Constraints                  0  Objective Function           12.1
Max Abs Gradient                107.8  Radius                          1
Element


                                           Max Abs          Trust
            Rest  Func Act   Objective  Obj Fun Gradient          Region
   Iter     arts Calls Con    Function   Change  Element  Lambda  Radius

      1       0    2   0     2.36594    9.7341   2.3189       0   1.000
      2       0    5   0     2.05926    0.3067   5.2875   0.385   1.526
      3       0    8   0     1.74390    0.3154   5.9934       0   1.086
      .
      .
      .

     22       0   31   0  1.3128E-16 6.96E-10 1.977E-7        0 0.00314

                        Optimization Results

Iterations                         22  Function Calls                  32
Hessian Calls                      23  Active Constraints               0
Objective Function    1.312814E-16  Max Abs Gradient      1.9773384E-7
                                        Element
Lambda                              0  Actual Over Pred                 0
                                        Change
Radius                    0.003140192

ABSGCONV convergence criterion satisfied.

                         Optimization Results
                          Parameter Estimates
                                                     Gradient
                                                    Objective
                 N Parameter          Estimate       Function

                 1 X1                  1.000000      0.000000198
                 2 X2                  1.000000     -0.000000105

                 Value of Objective Function = 1.312814E-16
```

Figure 11.1. NLPTR Solution to the Rosenbrock Problem

Since $f(x) = \frac{1}{2}\{f_1^2(x) + f_2^2(x)\}$, you can also use least-squares techniques in this situation. The following code calls the NLPLM subroutine to solve the problem. The output is shown in

```
proc iml;
   title 'Test of NLPLM subroutine: No Derivatives';
   start F_ROSEN(x);
      y = j(1,2,0.);
      y[1] = 10. * (x[2] - x[1] * x[1]);
      y[2] = 1. - x[1];
```

```
  return(y);
finish F_ROSEN;

x = {-1.2 1.};
optn = {2 2};
call nlplm(rc,xres,"F_ROSEN",x,optn);
quit;
```

Figure 11.2. NLPLM Solution Using the Least-Squares Technique

The Levenberg-Marquardt least-squares method, which is the method used by the NLPLM subroutine, is a modification of the trust-region method for nonlinear least-squares problems. The F_ROSEN module represents the Rosenbrock function. Note that for least-squares problems, the m functions $f_1(x), \ldots, f_m(x)$ are specified as elements of a vector; this is different from the manner in which $f(x)$ is specified for the other optimization techniques. No derivatives are specified in the preceding code, so the NLPLM subroutine computes finite difference approximations. For more information on the NLPLM subroutine, see the section "NLPLM Call" on page 795.

Constrained Betts Function

The linearly constrained Betts function (Hock & Schittkowski 1981) is defined as

$$f(x) = 0.01x_1^2 + x_2^2 - 100$$

with boundary constraints

$$2 \leq x_1 \leq 50$$
$$-50 \leq x_2 \leq 50$$

and linear constraint

$$10x_1 - x_2 \geq 10$$

The following code calls the NLPCG subroutine to solve the optimization problem. The infeasible initial point $x^0 = (-1, -1)$ is specified, and a portion of the output is shown in Figure 11.3.

```
proc iml;
   title 'Test of NLPCG subroutine: No Derivatives';
   start F_BETTS(x);
      f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
      return(f);
   finish F_BETTS;

   con = {  2. -50.  .   .,
           50.  50.  .   .,
           10.  -1. 1. 10.};
   x = {-1. -1.};
   optn = {0 2};
   call nlpcg(rc,xres,"F_BETTS",x,optn,con);
   quit;
```

The NLPCG subroutine performs conjugate gradient optimization. It requires only function and gradient calls. The F_BETTS module represents the Betts function, and since no module is defined to specify the gradient, first-order derivatives are computed by finite difference approximations. For more information on the NLPCG subroutine, see the section "NLPCG Call" on page 785. For details on the constraint matrix, which is represented by the CON matrix in the preceding code, see the section "Parameter Constraints" on page 341.

```
NOTE: Initial point was changed to be feasible for boundary and
                 linear constraints.


                     Optimization Start
                     Parameter Estimates
                                      Gradient           Lower
                                      Objective          Bound
     N Parameter          Estimate     Function        Constraint

     1 X1                 6.800000     0.136000         2.000000
     2 X2                -1.000000    -2.000000       -50.000000

                     Optimization Start
                     Parameter Estimates
                                Upper
                                Bound
                          Constraint

                           50.000000
                           50.000000

            Value of Objective Function = -98.5376



                     Linear Constraints

     1    59.00000 :      10.0000  <=   +   10.0000 * X1  -   1.0000 * X2


                 Conjugate-Gradient Optimization

        Automatic Restart Update (Powell, 1977; Beale, 1972)
              Gradient Computed by Finite Differences

               Parameter Estimates              2
               Lower Bounds                      2
               Upper Bounds                      2
               Linear Constraints               1
```

Figure 11.3. NLPCG Solution to Betts Problem

```
                        Optimization Start

Active Constraints                0  Objective Function      -98.5376
Max Abs Gradient                  2
Element


                                       Max Abs           Slope
         Rest  Func Act  Objective  Obj Fun Gradient   Step  Search
  Iter   arts Calls Con   Function   Change  Element   Size   Direc

    1      0    3   0   -99.54682    1.0092   0.1346   0.502  -4.018
    2      1    7   1   -99.96000    0.4132   0.00272 34.985 -0.0182
    3      2    9   1   -99.96000 1.851E-6        0    0.500  -74E-7

                      Optimization Results

Iterations                        3  Function Calls              10
Gradient Calls                    9  Active Constraints           1
Objective Function           -99.96  Max Abs Gradient             0
                                     Element
Slope of Search      -7.398365E-6
Direction

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            -1.24028E-10          0

             Value of Objective Function = -99.96



           Linear Constraints Evaluated at Solution

  [1]      10.00000  =  -10.0000  +   10.0000 * X1      -    1.0000 * X2
```

Figure 11.3. (continued)

Since the initial point $(-1, -1)$ is infeasible, the subroutine first computes a feasible starting point. Convergence is achieved after three iterations, and the optimal point is given to be $x^* = (2, 0)$ with an optimal function value of $f^* = f(x^*) = -99.96$. For more information on the printed output, see the section "Printing the Optimization History" on page 359.

Rosen-Suzuki Problem

The Rosen-Suzuki problem is a function of four variables with three nonlinear constraints on the variables. It is taken from Problem 43 of Hock and Schittkowski (1981). The objective function is

$$f(x) = x_1^2 + x_2^2 + 2x_3^2 + x_4^2 - 5x_1 - 5x_2 - 21x_3 + 7x_4$$

The nonlinear constraints are

$$
\begin{aligned}
0 &\leq 8 - x_1^2 - x_2^2 - x_3^2 - x_4^2 - x_1 + x_2 - x_3 + x_4 \\
0 &\leq 10 - x_1^2 - 2x_2^2 - x_3^2 - 2x_4^2 + x_1 + x_4 \\
0 &\leq 5 - 2x_1^2 - x_2^2 - x_3^2 - 2x_1 + x_2 + x_4
\end{aligned}
$$

Since this problem has nonlinear constraints, only the NLPQN and NLPNMS subroutines are available to perform the optimization. The following code solves the problem with the NLPQN subroutine:

```
proc iml;
   start F_HS43(x);
      f = x*x' + x[3]*x[3] - 5*(x[1] + x[2]) - 21*x[3] + 7*x[4];
      return(f);
   finish F_HS43;
   start C_HS43(x);
      c = j(3,1,0.);
      c[1] = 8 - x*x' - x[1] + x[2] - x[3] + x[4];
      c[2] = 10 - x*x' - x[2]*x[2] - x[4]*x[4] + x[1] + x[4];
      c[3] = 5 - 2.*x[1]*x[1] - x[2]*x[2] - x[3]*x[3]
                - 2.*x[1] + x[2] + x[4];
      return(c);
   finish C_HS43;
   x = j(1,4,1);
   optn= j(1,11,.); optn[2]= 3; optn[10]= 3; optn[11]=0;
   call nlpqn(rc,xres,"F_HS43",x,optn) nlc="C_HS43";
```

The F_HS43 module specifies the objective function, and the C_HS43 module specifies the nonlinear constraints. The OPTN vector is passed to the subroutine as the *opt* input argument. See the section "Options Vector" on page 343 for more information. The value of OPTN[10] represents the total number of nonlinear constraints, and the value of OPTN[11] represents the number of equality constraints. In the preceding code, OPTN[10]=3 and OPTN[11]=0, which indicate that there are three constraints, all of which are inequality constraints. In the subroutine calls, instead of separating missing input arguments with commas, you can specify optional arguments with keywords, as in the CALL NLPQN statement in the preceding code. For details on the CALL NLPQN statement, see the section "NLPQN Call" on page 808.

The initial point for the optimization procedure is $x = (1, 1, 1, 1)$, and the optimal point is $x^* = (0, 1, 2, -1)$, with an optimal function value of $f(x^*) = -44$. Part of the output produced is shown in Figure 11.4 on page 329.

```
                  Dual Quasi-Newton Optimization

          Modified VMCWD Algorithm of Powell (1978, 1982)

      Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
             Lagrange Multiplier Update of Powell(1982)
              Gradient Computed by Finite Differences
    Jacobian Nonlinear Constraints Computed by Finite Differences


              Parameter Estimates              4
              Nonlinear Constraints            3

                      Optimization Start

Objective Function           -19  Maximum Constraint            0
                                  Violation
Maximum Gradient of           17
the Lagran Func


                                                          Maximum
                                    Maximum                   Grad
                                      Con-                 Element
                                   straint Predicted        of the
         Rest   Func   Objective   Viola-  Function  Step  Lagran
  Iter   arts  Calls   Function      tion Reduction  Size    Func

     1      0      2  -41.88007    1.8988   13.6803  1.000   5.647
     2      0      3  -48.83264    3.0280    9.5464  1.000   5.041
     3      0      4  -45.33515    0.5452    2.6179  1.000   1.061
     4      0      5  -44.08667    0.0427    0.1732  1.000  0.0297
     5      0      6  -44.00011  0.000099  0.000218  1.000 0.00906
     6      0      7  -44.00001  2.573E-6  0.000014  1.000 0.00219
     7      0      8  -44.00000  9.118E-8  5.097E-7  1.000 0.00022
```

Figure 11.4. Solution to the Rosen-Suzuki Problem by the NLPQN Subroutine

```
                         Optimization Results

Iterations                       7  Function Calls                 9
Gradient Calls                   9  Active Constraints             2
Objective Function   -44.00000026  Maximum Constraint   9.1176306E-8
                                    Violation
Maximum Projected      0.0002265341 Value Lagrange               -44
Gradient                            Function
Maximum Gradient of     0.00022158  Slope of Search    -5.097332E-7
the Lagran Func                     Direction

FCONV2 convergence criterion satisfied.

WARNING: The point x is feasible only at the LCEPSILON= 1E-7 range.


                    Optimization Results
                     Parameter Estimates
                                       Gradient          Gradient
                                       Objective         Lagrange
     N Parameter          Estimate     Function          Function

     1 X1             -0.000001248     -5.000002      -0.000012804
     2 X2                 1.000027     -2.999945          0.000222
     3 X3                 1.999993    -13.000027      -0.000054166
     4 X4                -1.000003      4.999995      -0.000020681

           Value of Objective Function = -44.00000026


               Value of Lagrange Function = -44
```

Figure 11.4.   (continued)

In addition to the standard iteration history, the NLPQN subroutine includes the fol-
lowing information for problems with nonlinear constraints:

- *conmax* is the maximum value of all constraint violations.
- *pred* is the value of the predicted function reduction used with the GTOL and
  FTOL2 termination criteria.
- *alfa* is the step size α of the quasi-Newton step.
- *lfgmax* is the maximum element of the gradient of the Lagrange function.

Details

Global Versus Local Optima

All the IML optimization algorithms converge toward local rather than global optima. The smallest local minimum of an objective function is called the global minimum, and the largest local maximum of an objective function is called the global maximum. Hence, the subroutines may occasionally fail to find the global optimum. Suppose you have the function, $f(x) = \frac{1}{27}(3x_1^4 - 28x_1^3 + 84x_1^2 - 96x_1 + 64) + x_2^2$, which has a local minimum at $f(1,0) = 1$ and a global minimum at the point $f(4,0) = 0$.

The following statements use two calls of the NLPTR subroutine to minimize the preceding function. The first call specifies the initial point $xa = (0.5, 1.5)$, and the second call specifies the initial point $xb = (3, 1)$. The first call finds the local optimum $x^* = (1, 0)$, and the second call finds the global optimum $x^* = (4, 0)$.

```
proc iml;
   start F_GLOBAL(x);
      f=(3*x[1]**4-28*x[1]**3+84*x[1]**2-96*x[1]+64)/27 + x[2]**2;
      return(f);
   finish F_GLOBAL;
   xa = {.5 1.5};
   xb = {3 -1};
   optn = {0 2};
   call nlptr(rca,xra,"F_GLOBAL",xa,optn);
   call nlptr(rcb,xrb,"F_GLOBAL",xb,optn);
   print xra xrb;
```

One way to find out whether the objective function has more than one local optimum is to run various optimizations with a pattern of different starting points.

For a more mathematical definition of optimality, refer to the *Kuhn-Tucker theorem* in standard optimization literature. Using a rather nonmathematical language, a local minimizer x^* satisfies the following conditions:

- There exists a small, feasible neighborhood of x^* that does not contain any point x with a smaller function value $f(x) < f(x^*)$.

- The vector of first derivatives (gradient) $g(x^*) = \nabla f(x^*)$ of the objective function f (projected toward the feasible region) at the point x^* is zero.

- The matrix of second derivatives $G(x^*) = \nabla^2 f(x^*)$ (Hessian matrix) of the objective function f (projected toward the feasible region) at the point x^* is positive definite.

A local maximizer has the largest value in a feasible neighborhood and a negative definite Hessian.

The iterative optimization algorithm terminates at the point x^t, which should be in a small neighborhood (in terms of a user-specified termination criterion) of a local optimizer x^*. If the point x^t is located on one or more active boundary or general

linear constraints, the local optimization conditions are valid only for the feasible region. That is,

- the projected gradient, $Z^T g(x^t)$, must be sufficiently small
- the projected Hessian, $Z^T G(x^t) Z$, must be positive definite for minimization problems or negative definite for maximization problems

If there are n active constraints at the point x^t, the nullspace Z has zero columns and the projected Hessian has zero rows and columns. A matrix with zero rows and columns is considered positive as well as negative definite.

Kuhn-Tucker Conditions

The nonlinear programming (NLP) problem with one objective function f and m constraint functions c_i, which are continuously differentiable, is defined as follows:

$$\text{minimize} f(x), \quad x \in \mathcal{R}^n, \quad \text{subject to}$$
$$c_i(x) = 0, \quad i = 1, \ldots, m_e$$
$$c_i(x) \geq 0, \quad i = m_e + 1, \ldots, m$$

In the preceding notation, n is the dimension of the function $f(x)$, and m_e is the number of equality constraints. The linear combination of objective and constraint functions

$$L(x, \lambda) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x)$$

is the *Lagrange function,* and the coefficients λ_i are the *Lagrange multipliers.*

If the functions f and c_i are twice differentiable, the point x^* is an *isolated local minimizer* of the NLP problem, if there exists a vector $\lambda^* = (\lambda_1^*, \ldots, \lambda_m^*)$ that meets the following conditions:

- Kuhn-Tucker conditions

$$c_i(x^*) = 0, \qquad\qquad\qquad\qquad\qquad i = 1, \ldots, m_e$$
$$c_i(x^*) \geq 0, \ \lambda_i^* \geq 0, \ \lambda_i^* c_i(x^*) = 0, \quad i = m_e + 1, \ldots, m$$
$$\nabla_x L(x^*, \lambda^*) = 0$$

- Second-order condition

  Each nonzero vector $y \in \mathcal{R}^n$ with

$$y^T \nabla_x c_i(x^*) = 0 i = 1, \ldots, m_e, \ \text{ and } \forall i \in m_e + 1, \ldots, m; \lambda_i^* > 0$$

  satisfies

$$y^T \nabla_x^2 L(x^*, \lambda^*) y > 0$$

In practice, you cannot expect that the constraint functions $c_i(x^*)$ will vanish within machine precision, and determining the set of active constraints at the solution x^* may not be simple.

Definition of Return Codes

The return code, which is represented by the output parameter *rc* in the optimization subroutines, indicates the reason for optimization termination. A positive value indicates successful termination, while a negative value indicates unsuccessful termination. Table 11.1 gives the reason for termination associated with each return code.

Table 11.1.   Summary of Return Codes

rc	**Reason for Optimization Termination**
1	ABSTOL criterion satisfied (absolute F convergence)
2	ABSFTOL criterion satisfied (absolute F convergence)
3	ABSGTOL criterion satisfied (absolute G convergence)
4	ABSXTOL criterion satisfied (absolute X convergence)
5	FTOL criterion satisfied (relative F convergence)
6	GTOL criterion satisfied (relative G convergence)
7	XTOL criterion satisfied (relative X convergence)
8	FTOL2 criterion satisfied (relative F convergence)
9	GTOL2 criterion satisfied (relative G convergence)
10	n linear independent constraints are active at *xr* and none of them could be released to improve the function value
-1	objective function cannot be evaluated at starting point
-2	derivatives cannot be evaluated at starting point
-3	objective function cannot be evaluated during iteration
-4	derivatives cannot be evaluated during iteration
-5	optimization subroutine cannot improve the function value (this is a very general formulation and is used for various circumstances)
-6	there are problems in dealing with linearly dependent active constraints (changing the LCSING value in the *par* vector can be helpful)
-7	optimization process stepped outside the feasible region and the algorithm to return inside the feasible region was not successful (changing the LCEPS value in the *par* vector can be helpful)
-8	either the number of iterations or the number of function calls is larger than the prespecified values in the *tc* vector (MAXIT and MAXFU)
-9	this return code is temporarily not used (it is used in PROC NLP indicating that more CPU than a prespecified value was used)
-10	a feasible starting point cannot be computed

Objective Function and Derivatives

The input argument *fun* refers to an IML module that specifies a function that returns f, a vector of length m for least-squares subroutines or a scalar for other optimization subroutines. The returned f contains the values of the objective function (or the least-squares functions) at the point x. Note that for least-squares problems, you must specify the number of function values, m, with the first element of the *opt* argument to allocate memory for the return vector. All the modules that you can specify as input arguments (*"fun"*, *"grd"*, *"hes"*, *"jac"*, *"nlc"*, *"jacnlc"*, and *"ptit"*) allow only a single input argument, x, which is the parameter vector. Using the GLOBAL clause,

you can provide more input arguments for these modules. Refer to the middle of the section, Compartmental Analysis for an example.

All the optimization algorithms assume that f is continuous inside the feasible region. For nonlinearly constrained optimization, this is also required for points outside the feasible region. Sometimes the objective function cannot be computed for all points of the specified feasible region; for example, the function specification may contain the SQRT or LOG function, which cannot be evaluated for negative arguments. You must make sure that the function and derivatives of the starting point can be evaluated. There are two ways to prevent large steps into infeasible regions of the parameter space during the optimization process:

- The preferred way is to restrict the parameter space by introducing more boundary and linear constraints. For example, the boundary constraint $x_j >= 1\mathrm{E}-10$ prevents infeasible evaluations of $\log(x_j)$. If the function module takes the square root or the log of an intermediate result, you can use nonlinear constraints to try to avoid infeasible function evaluations. However, this may not ensure feasibility.

- Sometimes the preferred way is difficult to practice, in which case the function module may return a missing value. This may force the optimization algorithm to reduce the step length or the radius of the feasible region.

All the optimization techniques except the NLPNMS subroutine require continuous first-order derivatives of the objective function f. The NLPTR, NLPNRA, and NLPNRR techniques also require continuous second-order derivatives. If you do not provide the derivatives with the IML modules *"grd"*, *"hes"*, or *"jac"*, they are automatically approximated by finite difference formulas. Approximating first-order derivatives by finite differences usually requires n additional calls of the function module. Approximating second-order derivatives by finite differences using only function calls can be extremely computationally expensive. Hence, if you decide to use the NLPTR, NLPNRA, or NLPNRR subroutines, you should specify at least analytical first-order derivatives. Then, approximating second-order derivatives by finite differences requires only n or $2n$ additional calls of the function and gradient modules.

For all input and output arguments, the subroutines assume that

- the number of parameters n corresponds to the number of columns. For example, x, the input argument to the modules, and g, the output argument returned by the *"grd"* module, are row vectors with n entries, and G, the Hessian matrix returned by the *"hes"* module, must be a symmetric $n \times n$ matrix.

- the number of functions, m, corresponds to the number of rows. For example, the vector f returned by the *"fun"* module must be a column vector with m entries, and in least-squares problems, the Jacobian matrix **J** returned by the *"jac"* module must be an $m \times n$ matrix.

You can verify your analytical derivative specifications by computing finite difference approximations of the derivatives of f with the NLPFDD subroutine. For most

applications, the finite difference approximations of the derivatives will be very pre-cise. Occasionally, difficult objective functions and zero x coordinates will cause problems. You can use the *par* argument to specify the number of accurate dig-its in the evaluation of the objective function; this defines the step size h of the first- and second-order finite difference formulas. See the section "Finite Difference Approximations of Derivatives" on page 338.

Note: For some difficult applications, the finite difference approximations of deriva-tives that are generated by default may not be precise enough to solve the optimization or least-squares problem. In such cases, you may be able to specify better derivative approximations by using a better approximation formula. You can submit your own finite difference approximations using the IML modules *"grd"*, *"hes"*, *"jac"*, or *"jac-nlc"*. See Example 11.3 on page 368 for an illustration.

In many applications, calculations used in the computation of f can help compute derivatives at the same point efficiently. You can save and reuse such calculations with the GLOBAL clause. As with many other optimization packages, the subroutines perform calls of the *"grd"*, *"hes"*, or *"jac"* modules only after a call of the *"fun"* module.

The following statements specify modules for the function, gradient, and Hessian matrix of the Rosenbrock problem:

```
proc iml;
   start F_ROSEN(x);
      y1 = 10. * (x[2] - x[1] * x[1]);
      y2 = 1. - x[1];
      f  = .5 * (y1 * y1 + y2 * y2);
      return(f);
   finish F_ROSEN;

   start G_ROSEN(x);
      g = j(1,2,0.);
      g[1] = -200.*x[1]*(x[2]-x[1]*x[1]) - (1.-x[1]);
      g[2] =  100.*(x[2]-x[1]*x[1]);
      return(g);
   finish G_ROSEN;

   start H_ROSEN(x);
      h = j(2,2,0.);
      h[1,1] = -200.*(x[2] - 3.*x[1]*x[1]) + 1.;
      h[2,2] =  100.;
      h[1,2] = -200. * x[1];
      h[2,1] = h[1,2];
      return(h);
   finish H_ROSEN;
```

The following statements specify a module for the Rosenbrock function when con-sidered as a least-squares problem. They also specify the Jacobian matrix of the least-squares functions.

```
proc iml;
   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;

   start J_ROSEN(x);
      jac = j(2,2,0.);
      jac[1,1] = -20. * x[1]; jac[1,2] = 10.;
      jac[2,1] = -1.;         jac[2,2] =  0.;
      return(jac);
   finish J_ROSEN;
```

Diagonal or Sparse Hessian Matrices

In the unconstrained or only boundary constrained case, the NLPNRA algorithm can take advantage of diagonal or sparse Hessian matrices submitted by the *"hes"* module. If the Hessian matrix G of the objective function f has a large proportion of zeros, you may save computer time and memory by specifying a sparse Hessian of dimension $nn \times 3$ rather than a dense $n \times n$ Hessian. Each of the nn rows (i, j, g) of the matrix returned by the sparse Hessian module defines a nonzero element g_{ij} of the Hessian matrix. The row and column location is given by i and j, and g gives the nonzero value. During the optimization process, only the values g can be changed in each call of the Hessian module *"hes"*; the sparsity structure (i, j) must be kept the same. That means that some of the values g can be zero for particular values of x. To allocate sufficient memory before the first call of the Hessian module, you must specify the number of rows, nn, by setting the ninth element of the *opt* argument.

Example 22 of Moré, Garbow, and Hillstrom (1981) illustrates the sparse Hessian module input. The objective function, which is the Extended Powell's Singular Function, for $n = 40$ is a least-squares problem:

$$f(x) = \frac{1}{2}\{f_1^2(x) + \cdots + f_m^2(x)\}$$

with

$$
\begin{aligned}
f_{4i-3}(x) &= x_{4i-3} + 10x_{4i-2} \\
f_{4i-2}(x) &= \sqrt{5}(x_{4i-1} - x_{4i}) \\
f_{4i-1}(x) &= (x_{4i-2} - 2x_{4i-1})^2 \\
f_{4i}(x) &= \sqrt{10}(x_{4i-3} - x_{4i})^2
\end{aligned}
$$

The function and gradient modules are

```
start f_nlp22(x);
   n=ncol(x);
   f = 0.;
```

```
     do i=1 to n-3 by 4;
        f1 = x[i] + 10. * x[i+1];
        r2 = x[i+2] - x[i+3];
        f2 = 5. * r2;
        r3 = x[i+1] - 2. * x[i+2];
        f3 = r3 * r3;
        r4 = x[i] - x[i+3];
        f4 = 10. * r4 * r4;
        f = f + f1 * f1 + r2 * f2 + f3 * f3 + r4 * r4 * f4;
     end;
     f = .5 * f;
     return(f);
  finish f_nlp22;

  start g_nlp22(x);
     n=ncol(x);
     g = j(1,n,0.);
        do i=1 to n-3 by 4;
           f1 = x[i] + 10. * x[i+1];
           f2 = 5. * (x[i+2] - x[i+3]);
           r3 = x[i+1] - 2. * x[i+2];
           f3 = r3 * r3;
           r4 = x[i] - x[i+3];
           f4 = 10. * r4 * r4;
           g[i]   = f1 + 2. * r4 * f4;
           g[i+1] = 10. * f1 + 2. * r3 * f3;
           g[i+2] = f2 - 4. * r3 * f3;
           g[i+3] = -f2 - 2. * r4 * f4;
        end;
        return(g);
  finish g_nlp22;
```

You can specify the sparse Hessian with the following module:

```
  start hs_nlp22(x);
     n=ncol(x);
     nnz = 8 * (n / 4);
     h = j(nnz,3,0.);
     j = 0;
     do i=1 to n-3 by 4;
        f1 = x[i] + 10. * x[i+1];
        f2 = 5. * (x[i+2] - x[i+3]);
        r3 = x[i+1] - 2. * x[i+2];
        f3 = r3 * r3;
        r4 = x[i] - x[i+3];
        f4 = 10. * r4 * r4;
        j= j + 1; h[j,1] = i; h[j,2] = i;
        h[j,3] = 1. + 4. * f4;
        h[j,3] = h[j,3] + 2. * f4;
        j= j+1; h[j,1] = i; h[j,2] = i+1;
        h[j,3] = 10.;
        j= j+1; h[j,1] = i; h[j,2] = i+3;
        h[j,3] = -4. * f4;
```

```
         h[j,3] = h[j,3] - 2. * f4;
         j= j+1; h[j,1] = i+1; h[j,2] = i+1;
         h[j,3] = 100. + 4. * f3;
         h[j,3] = h[j,3] + 2. * f3;
         j= j+1; h[j,1] = i+1; h[j,2] = i+2;
         h[j,3] = -8. * f3;
         h[j,3] = h[j,3] - 4. * f3;
         j= j+1; h[j,1] = i+2; h[j,2] = i+2;
         h[j,3] = 5. + 16. * f3;
         h[j,3] = h[j,3] + 8. * f3;
         j= j+1; h[j,1] = i+2; h[j,2] = i+3;
         h[j,3] = -5.;
         j= j+1; h[j,1] = i+3; h[j,2] = i+3;
         h[j,3] = 5. + 4. * f4;
         h[j,3] = h[j,3] + 2. * f4;
      end;
      return(h);
   finish hs_nlp22;

   n = 40;
   x = j(1,n,0.);
   do i=1 to n-3 by 4;
      x[i] = 3.; x[i+1] = -1.; x[i+3] = 1.;
   end;
   opt = j(1,11,.); opt[2]= 3; opt[9]= 8 * (n / 4);
   call nlpnra(xr,rc,"f_nlp22",x,opt) grd="g_nlp22" hes="hs_nlp22";
```

Note: If the sparse form of Hessian defines a diagonal matrix (that is, $i = j$ in all nn rows), the NLPNRA algorithm stores and processes a diagonal matrix G. If you do not specify any general linear constraints, the NLPNRA subroutine uses only order n memory.

Finite Difference Approximations of Derivatives

If the optimization technique needs first- or second-order derivatives and you do not specify the corresponding IML modules *"grd"*, *"hes"*, *"jac"*, or *"jacnlc"*, the derivatives are approximated by finite difference formulas using only calls of the module *"fun"*. If the optimization technique needs second-order derivatives and you specify the *"grd"* module but not the *"hes"* module, the subroutine approximates the second-order derivatives by finite differences using n or $2n$ calls of the *"grd"* module.

The eighth element of the *opt* argument specifies the type of finite difference approximation used to compute first- or second-order derivatives and whether the finite difference intervals, h, should be computed by an algorithm of Gill, Murray, Saunders, and Wright (1983). The value of *opt*[8] is a two-digit integer, ij.

- If *opt*[8] is missing or $j = 0$, the fast but not very precise forward difference formulas are used; if $j \neq 0$, the numerically more expensive central difference formulas are used.

- If *opt*[8] is missing or $i \neq 1, 2,$ or 3, the finite difference intervals h are based only on the information of *par*[8] or *par*[9], which specifies the number of

accurate digits to use in evaluating the objective function and nonlinear constraints, respectively. If $i = 1, 2,$ or 3, the intervals are computed with an algorithm by Gill, Murray, Saunders, and Wright (1983). For $i = 1$, the interval is based on the behavior of the objective function; for $i = 2$, the interval is based on the behavior of the nonlinear constraint functions; and for $i = 3$, the interval is based on the behavior of both the objective function and the nonlinear constraint functions.

Forward Difference Approximations

- First-order derivatives: n additional function calls are needed.

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x)}{h_i}$$

- Second-order derivatives based on function calls only, when the *"grd"* module is not specified (Dennis and Schnabel 1983): for a dense Hessian matrix, $n + n^2/2$ additional function calls are needed.

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}$$

- Second-order derivatives based on gradient calls, when the *"grd"* module is specified (Dennis and Schnabel 1983): n additional gradient calls are needed.

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{g_i(x + h_j e_j) - g_i(x)}{2h_j} + \frac{g_j(x + h_i e_i) - g_j(x)}{2h_i}$$

Central Difference Approximations

- First-order derivatives: $2n$ additional function calls are needed.

$$g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + h_i e_i) - f(x - h_i e_i)}{2h_i}$$

- Second-order derivatives based on function calls only, when the *"grd"* module is not specified (Abramowitz and Stegun 1972): for a dense Hessian matrix, $2n + 2n^2$ additional function calls are needed.

$$\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}$$

- Second-order derivatives based on gradient calls, when the *"grd"* module is specified: $2n$ additional gradient calls are needed.

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{g_i(x + h_j e_j) - g_i(x - h_j e_j)}{4h_j} + \frac{g_j(x + h_i e_i) - g_j(x - h_i e_i)}{4h_i}$$

The step sizes h_j, $j = 1, \ldots, n$, are defined as follows:

- For the forward-difference approximation of first-order derivatives using only function calls and for second-order derivatives using only gradient calls, $h_j = \sqrt[2]{\eta_j}(1 + |x_j|)$.
- For the forward-difference approximation of second-order derivatives using only function calls and for central-difference formulas, $h_j = \sqrt[3]{\eta_j}(1 + |x_j|)$.

If the algorithm of Gill, Murray, Saunders, and Wright (1983) is not used to compute η_j, a constant value $\eta = \eta_j$ is used depending on the value of *par*[8].

- If the number of accurate digits is specified by *par*[8] $= k_1$, then η is set to 10^{-k_1}.
- If *par*[8] is not specified, η is set to the machine precision, ϵ.

If central difference formulas are not specified, the optimization algorithm will switch automatically from the forward-difference formula to a corresponding central-difference formula during the iteration process if one of the following two criteria is satisfied:

- The absolute maximum gradient element is less than or equal to 100 times the ABSGTOL threshold.
- The term on the left of the GTOL criterion is less than or equal to $\max(1\mathrm{E}{-6}, 100{\times}\mathrm{GTOL}\text{ threshold})$. The 1E−6 ensures that the switch is performed even if you set the GTOL threshold to zero.

The algorithm of Gill, Murray, Saunders, and Wright (1983) that computes the finite difference intervals h_j can be very expensive in the number of function calls it uses. If this algorithm is required, it is performed twice, once before the optimization process starts and once after the optimization terminates.

Many applications need considerably more time for computing second-order derivatives than for computing first-order derivatives. In such cases, you should use a quasi-Newton or conjugate gradient technique.

If you specify a vector, c, of nc nonlinear constraints with the "*nlc*" module but you do not specify the "*jacnlc*" module, the first-order formulas can be used to compute finite difference approximations of the $nc \times n$ Jacobian matrix of the nonlinear constraints.

$$(\nabla c_i) = \left(\frac{\partial c_i}{\partial x_j} \right), \quad i = 1, \ldots, nc, \ \ j = 1, \ldots, n$$

You can specify the number of accurate digits in the constraint evaluations with *par*[9]. This specification also defines the step sizes h_j, $j = 1, \ldots, n$.

Note: If you are not able to specify analytic derivatives and if the finite-difference approximations provided by the subroutines are not good enough to solve your optimization problem, you may be able to implement better finite-difference approximations with the "*grd*", "*hes*", "*jac*", and "*jacnlc*" module arguments.

Parameter Constraints

You can specify constraints in the following ways:

- The matrix input argument "*blc*" enables you to specify boundary and general linear constraints.

- The IML module input argument "*nlc*" enables you to specify general constraints, particularly nonlinear constraints.

Specifying the BLC Matrix

The input argument "*blc*" specifies an $n_1 \times n_2$ constraint matrix, where n_1 is two more than the number of linear constraints, and n_2 is given by

$$n2 = \begin{cases} n & \text{if } 1 \le n1 \le 2 \\ n+2 & \text{if } n1 > 2 \end{cases}$$

The first two rows define lower and upper bounds for the n parameters, and the remaining $c = n_1 - 2$ rows define general linear equality and inequality constraints. Missing values in the first row (lower bounds) substitute for the largest negative floating point value, and missing values in the second row (upper bounds) substitute for the largest positive floating point value. Columns $n + 1$ and $n + 2$ of the first two rows are not used.

The following c rows of the "*blc*" argument specify c linear equality or inequality constraints:

$$\sum_{j=1}^{n} a_{ij}x_j \quad (\le | = | \ge) \quad b_i, \quad i = 1, \dots, c$$

Each of these c rows contains the coefficients a_{ij} in the first n columns. Column $n+1$ specifies the kind of constraint, as follows:

- $blc[n + 1] = 0$ indicates an equality constraint.
- $blc[n + 1] = 1$ indicates a $\ge$ inequality constraint.
- $blc[n + 1] = -1$ indicates a $\le$ inequality constraint.

Column $n + 2$ specifies the right-hand side, b_i. A missing value in any of these rows corresponds to a value of zero.

For example, suppose you have a problem with the following constraints on $x_1, x_2,$ x_3, x_4:

$$\begin{array}{ccccc} 2 & \le & x_1 & \le & 100 \\ & & x_2 & \le & 40 \\ 0 & \le & x_4 & & \end{array}$$

$$
\begin{array}{rrrrrrrr}
4x_1 & + & 3x_2 & - & x_3 & & & \leq & 30 \\
& & x_2 & & & + & 6x_4 & \geq & 17 \\
x_1 & - & x_2 & & & & & = & 8
\end{array}
$$

The following statements specify the matrix CON, which can be used as the "*blc*" argument to specify the preceding constraints:

```
proc iml;
   con = {   2   .   .   0   .   . ,
           100  40   .   .   .   . ,
             4   3  -1   .  -1  30 ,
             .   1   .   6   1  17 ,
             1  -1   .   .   0   8   };
```

Specifying the NLC and JACNLC Modules

The input argument "*nlc*" specifies an IML module that returns a vector, c, of length nc, with the values, c_i, of the nc linear or nonlinear constraints

$$
\begin{aligned}
c_i(x) &= 0, & i &= 1, \ldots, nec, \\
c_i(x) &\geq 0, & i &= nec + 1, \ldots, nc,
\end{aligned}
$$

for a given input parameter point x.

Note: You must specify the number of equality constraints, nec, and the total number of constraints, nc, returned by the "*nlc*" module to allocate memory for the return vector. You can do this with the *opt*[11] and *opt*[10] arguments, respectively.

For example, consider the problem of minimizing the objective function $f(x_1, x_2) = x_1 x_2$ in the interior of the unit circle, $x_1^2 + x_2^2 \leq 1$. The constraint can also be written as $c_1(x) = 1 - x_1^2 - x_2^2 \geq 0$. The following statements specify modules for the objective and constraint functions and call the NLPNMS subroutine to solve the minimization problem:

```
proc iml;
   start F_UC2D(x);
      f = x[1] * x[2];
      return(f);
   finish F_UC2D;

   start C_UC2D(x);
      c = 1. - x * x`;
      return(c);
   finish C_UC2D;

   x = j(1,2,1.);
   optn= j(1,10,.); optn[2]= 3; optn[10]= 1;
   CALL NLPNMS(rc,xres,"F_UC2D",x,optn) nlc="C_UC2D";
```

To avoid typing multiple commas, you can specify the "*nlc*" input argument with a keyword, as in the preceding code. The number of elements of the return vector is specified by OPTN[10] = 1. There is a missing value in OPTN[11], so the subroutine assumes there are zero equality constraints.

The NLPQN algorithm uses the $nc \times n$ Jacobian matrix of first-order derivatives

$$(\nabla_x c_i(x)) = \left(\frac{\partial c_i}{\partial x_j} \right), \quad i = 1, \ldots, nc, \quad j = 1, \ldots, n$$

of the nc equality and inequality constraints, c_i, for each point passed during the iteration. You can use the "*jacnlc*" argument to specify an IML module that returns the Jacobian matrix **JC**. If you specify the "*nlc*" module without using the "*jacnlc*" argument, the subroutine uses finite difference approximations of the first-order derivatives of the constraints.

Note: The COBYLA algorithm in the NLPNMS subroutine and the NLPQN subroutine are the only optimization techniques that enable you to specify nonlinear constraints with the "*nlc*" input argument.

Options Vector

The options vector, represented by the *"opt"* argument, enables you to specify a variety of options, such as the amount of printed output or particular update or line-search techniques. Table 11.2 gives a summary of the available options.

Table 11.2. Summary of the Elements of the Options Vector

Index	Description
1	specifies minimization, maximization, or the number of least-squares functions
2	specifies the amount of printed output
3	NLPDD, NLPLM, NLPNRA, NLPNRR, NLPTR: specifies the scaling of the Hessian matrix (HESCAL)
4	NLPCG, NLPDD, NLPHQN, NLPQN: specifies the update technique (UPDATE)
5	NLPCG, NLPHQN, NLPNRA, NLPQN (with no nonlinear constraints): specifies the line-search technique (LIS)
6	NLPHQN: specifies version of hybrid algorithm (VERSION) NLPQN with nonlinear constraints: specifies version of μ update
7	NLPDD, NLPHQN, NLPQN: specifies initial Hessian matrix (INHESSIAN)
8	Finite Difference Derivatives: specifies type of differences and how to compute the difference interval
9	NLPNRA: specifies the number of rows returned by the sparse Hessian module
10	NLPNMS, NLPQN: specifies the total number of constraints returned by the *"nlc"* module
11	NLPNMS, NLPQN: specifies the number of equality constraints returned by the *"nlc"* module

The following list contains detailed explanations of the elements of the options vector:

- **opt[1]**
  indicates whether the problem is minimization or maximization. The default, $opt[1] = 0$, specifies a minimization problem, and $opt[1] = 1$ specifies a maximization problem. For least-squares problems, $opt[1] = m$ specifies the number of functions or observations, which is the number of values returned by the *"fun"* module. This information is necessary to allocate memory for the return vector of the *"fun"* module.

- **opt[2]**
  specifies the amount of output printed by the subroutine. The higher the value of $opt[2]$, the more printed output is produced. The following table indicates the specific items printed for each value.

Value of *opt*[2]	Printed Output
0	No printed output is produced. This is the default.
1	The summaries for optimization start and termination are produced, as well as the iteration history.
2	The initial and final parameter estimates are also printed.
3	The values of the termination criteria and other control parameters are also printed.
4	The parameter vector, x, is also printed after each iteration.
5	The gradient vector, g, is also printed after each iteration.

- **opt[3]**

  selects a scaling for the Hessian matrix, **G**. This option is relevant only for the NLPDD, NLPLM, NLPNRA, NLPNRR, and NLPTR subroutines. If $opt[3] \neq 0$, the first iteration and each restart iteration set the diagonal scaling matrix $\mathbf{D}^{(0)} = diag(d_i^{(0)})$, where

  $$d_i^{(0)} = \sqrt{\max(|G_{i,i}^{(0)}|, \epsilon)}$$

  and $G_{i,i}^{(0)}$ are the diagonal elements of the Hessian matrix, and ϵ is the machine precision. The diagonal scaling matrix $\mathbf{D}^{(0)} = diag(d_i^{(0)})$ is updated as indicated in the following table.

Value of *opt*[3]	Scaling Update
0	No scaling is done.
1	Moré (1978) scaling update: $$d_i^{(k+1)} = \max\left(d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$
2	Dennis, Gay, and Welsch (1981) scaling update: $$d_i^{(k+1)} = \max\left(0.6 * d_i^{(k)}, \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}\right)$$
3	d_i is reset in each iteration: $d_i^{(k+1)} = \sqrt{\max(|G_{i,i}^{(k)}|, \epsilon)}$

  For the NLPDD, NLPNRA, NLPNRR, and NLPTR subroutines, the default is $opt[3] = 0$; for the NLPLM subroutine, the default is $opt[3] = 1$.

- **opt[4]**

  defines the update technique for (dual) quasi-Newton and conjugate gradient techniques. This option applies to the NLPCG, NLPDD, NLPHQN, and NLPQN subroutines. For the NLPCG subroutine, the following update techniques are available.

Value of *opt*[4]	Update Method for NLPCG
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)

For the unconstrained or linearly constrained NLPQN subroutine, the following update techniques are available.

Value of *opt*[4]	Update Method for NLPQN
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

For the NLPQN subroutine used with the *"nlc"* module and for the NLPDD and NLPHQN subroutines, only the first two update techniques in the second table are available.

- **opt[5]**
  defines the line-search technique for the unconstrained or linearly constrained NLPQN subroutine, as well as the NLPCG, NLPHQN, and NLPNRA subroutines. Refer to Fletcher (1987) for an introduction to line-search techniques. The following table describes the available techniques.

Value of *opt*[5]	Line-Search Method
1	This method needs the same number of function and gradient calls for cubic interpolation and cubic extrapolation; it is similar to a method used by the Harwell subroutine library.
2	This method needs more function than gradient calls for quadratic and cubic interpolation and cubic extrapolation; it is implemented as shown in Fletcher (1987) and can be modified to exact line search with the *par*[6] argument (see the "Control Parameters Vector" section on page 356). This is the default for the NLPCG, NLPNRA, and NLPQN subroutines.
3	This method needs the same number of function and gradient calls for cubic interpolation and cubic extrapolation; it is implemented as shown in Fletcher (1987) and can be modified to exact line search with the *par*[6] argument.
4	This method needs the same number of function and gradient calls for stepwise extrapolation and cubic interpolation.
5	This method is a modified version of the *opt*[5]=4 method.
6	This method is the golden section line search of Polak (1971), which uses only function values for linear approximation.
7	This method is the bisection line search of Polak (1971), which uses only function values for linear approximation.
8	This method is the Armijo line-search technique of Polak (1971), which uses only function values for linear approximation.

For the NLPHQN least-squares subroutine, the default is a special line-search method that is based on an algorithm developed by Lindström and Wedin (1984). Although it needs more memory, this method sometimes works better with large least-squares problems.

- **opt[6]**
  is used only for the NLPHQN subroutine and the NLPQN subroutine with nonlinear constraints.

  In the NLPHQN subroutine, it defines the criterion for the decision of the hybrid algorithm to step in a Gauss-Newton or a quasi-Newton search direction. You can specify one of the three criteria that correspond to the methods of Fletcher and Xu (1987). The methods are HY1 (*opt*[6]=1), HY2 (*opt*[6]=2), and HY3 (*opt*[6]=2), and the default is HY2.

  In the NLPQN subroutine with nonlinear constraints, it defines the version of the algorithm used to update the vector μ of the Lagrange multipliers. The default is *opt*[6]=2, which specifies the approach of Powell (1982a,b). You can specify the approach of Powell (1978b) with *opt*[6]=1.

- **opt[7]**
  defines the type of start matrix, $G^{(0)}$, used for the Hessian approximation. This option applies only to the NLPDD, NLPHQN, and NLPQN subroutines. If

opt[7]=0, which is the default, the quasi-Newton algorithm starts with a multiple of the identity matrix where the scalar factor depends on *par*[10]; otherwise, it starts with the Hessian matrix computed at the starting point $x^{(0)}$.

- **opt[8]**
  defines the type of finite difference approximation used to compute first- or second-order derivatives and whether the finite difference intervals, h, should be computed using an algorithm of Gill, Murray, Saunders, and Wright (1983). The value of *opt*[8] is a two-digit integer, ij.

  > If *opt*[8] is missing or $j = 0$, the fast but not very precise forward difference formulas are used; if $j \neq 0$, the numerically more expensive central difference formulas are used.
  >
  > If *opt*[8] is missing or $i \neq 1, 2$, or 3, the finite difference intervals h are based only on the information of *par*[8] or *par*[9], which specifies the number of accurate digits to use in evaluating the objective function and nonlinear constraints, respectively. If $i = 1, 2$, or 3, the intervals are computed with an algorithm by Gill, Murray, Saunders, and Wright (1983). For $i = 1$, the interval is based on the behavior of the objective function; for $i = 2$, the interval is based on the behavior of the nonlinear constraint functions; and for $i = 3$, the interval is based on the behavior of both the objective function and the nonlinear constraint functions.

  The algorithm of Gill, Murray, Saunders, and Wright (1983) that computes the finite difference intervals h_j can be very expensive in the number of function calls it uses. If this algorithm is required, it is performed twice, once before the optimization process starts and once after the optimization terminates. See the "Finite Difference Approximations of Derivatives" section on page 338 for details.

- **opt[9]**
  indicates that the Hessian module *"hes"* returns a sparse definition of the Hessian, in the form of an $nn \times 3$ matrix instead of the default dense $n \times n$ matrix. If *opt[9]* is zero or missing, the Hessian module must return a dense $n \times n$ matrix. If you specify $opt[9] = nn$, the module must return a sparse $nn \times 3$ table. See the "Objective Function and Derivatives" section on page 333 for more details. This option applies only to the NLPNRA algorithm. If the dense specification contains a large proportion of analytical zero derivatives, the sparse specification may save memory and computer time.

- **opt[10]**
  specifies the total number of nonlinear constraints returned by the *"nlc"* module. If you specify nc nonlinear constraints with the *"nlc"* argument module, you must specify $opt[10] = nc$ to allocate memory for the return vector.

- **opt[11]**

  specifies the number of nonlinear equality constraints returned by the *"nlc"* module. If the first nec constraints are equality constraints, you must specify $opt[11] = nec$. The default value is $opt[11] = 0$.

Termination Criteria

The input argument *tc* specifies a vector of bounds corresponding to a set of termination criteria that are tested in each iteration. If you do not specify an IML module with the *"ptit"* argument, these bounds determine when the optimization process stops.

If you specify the *"ptit"* argument, the *"tc"* argument is ignored. The module specified by the *"ptit"* argument replaces the subroutine that is used by default to test the termination criteria. The module is called in each iteration with the current location, x, and the value, f, of the objective function at x. The module must give a return code, rc, that decides whether the optimization process is to be continued or terminated. As long as the module returns $rc = 0$, the optimization process continues. When the module returns $rc \neq 0$, the optimization process stops.

If you use the *tc* vector, the optimization techniques stop the iteration process when at least one of the corresponding set of termination criteria are satisfied. Table 11.3 and Table 11.4 indicate the criterion associated with each element of the *tc* vector. There is a default for each criterion, and if you specify a missing value for the corresponding element of the *tc* vector, the default value is used. You can avoid termination with respect to the ABSFTOL, ABSGTOL, ABSXTOL, FTOL, FTOL2, GTOL, GTOL2, and XTOL criteria by specifying a value of zero for the corresponding element of the *tc* vector.

Table 11.3.   Termination Criteria for the NLPNMS Subroutine

Index	Description
1	maximum number of iterations (MAXIT)
2	maximum number of function calls (MAXFU)
3	absolute function criterion (ABSTOL)
4	relative function criterion (FTOL)
5	relative function criterion (FTOL2)
6	absolute function criterion (ABSFTOL)
7	FSIZE value used in FTOL criterion
8	relative parameter criterion (XTOL)
9	absolute parameter criterion (ABSXTOL)
9	size of final trust-region radius ρ (COBYLA algorithm)
10	XSIZE value used in XTOL criterion

Table 11.4. Termination Criteria for Other Subroutines

Index	Description
1	maximum number of iterations (MAXIT)
2	maximum number of function calls (MAXFU)
3	absolute function criterion (ABSTOL)
4	relative gradient criterion (GTOL)
5	relative gradient criterion (GTOL2)
6	absolute gradient criterion (ABSGTOL)
7	relative function criterion (FTOL)
8	predicted function reduction criterion (FTOL2)
9	absolute function criterion (ABSFTOL)
10	FSIZE value used in GTOL and FTOL criterion
11	relative parameter criterion (XTOL)
12	absolute parameter criterion (ABSXTOL)
13	XSIZE value used in XTOL criterion

Criteria Used by All Techniques

The following list indicates the termination criteria that are used with all the optimization techniques:

- **tc[1]**

  specifies the maximum number of iterations in the optimization process (MAXIT). The default values are

NLPNMS:	MAXIT=1000
NLPCG:	MAXIT=400
Others:	MAXIT=200

- **tc[2]**

  specifies the maximum number of function calls in the optimization process (MAXFU). The default values are

NLPNMS:	MAXFU=3000
NLPCG:	MAXFU=1000
Others:	MAXFU=500

- **tc[3]**

  specifies the absolute function convergence criterion (ABSTOL). For minimization, termination requires $f^{(k)} = f(x^{(k)}) \leq ABSTOL$, while for maximization, termination requires $f^{(k)} = f(x^{(k)}) \geq ABSTOL$. The default values are the negative and positive square roots of the largest double precision value, for minimization and maximization, respectively.

These criteria are useful when you want to divide a time-consuming optimization problem into a series of smaller problems.

Termination Criteria for NLPNMS

Since the Nelder-Mead simplex algorithm does not use derivatives, no termination criteria are available that are based on the gradient of the objective function.

When the NLPNMS subroutine implements Powell's COBYLA algorithm, it uses only one criterion other than the three used by all the optimization techniques. The COBYLA algorithm is a trust-region method that sequentially reduces the radius, ρ, of a spheric trust region from the start radius, ρ_{beg}, which is controlled with the *par*[2] argument, to the final radius, ρ_{end}, which is controlled with the *tc*[9] argument. The default value for *tc*[9] is $\rho_{end} =$ 1E−4. Convergence to small values of ρ_{end} may take many calls of the function and constraint modules and may result in numerical problems.

In addition to the criteria used by all techniques, the original Nelder-Mead simplex algorithm uses several other termination criteria, which are described in the following list:

- **tc[4]**
  specifies the relative function convergence criterion (FTOL). Termination requires a small relative difference between the function values of the vertices in the simplex with the largest and smallest function values.

$$\frac{|f_{hi}^{(k)} - f_{lo}^{(k)}|}{\max(|f_{hi}^{(k)})|, \textit{FSIZE})} \leq \textit{FTOL}$$

  where *FSIZE* is defined by *tc*[7]. The default value is $tc[4] = 10^{-\text{FDIGITS}}$, where FDIGITS is controlled by the *par*[8] argument. The *par*[8] argument has a default value of $\log_{10}(\epsilon)$, where ϵ is the machine precision. Hence, the default value for *FTOL* is ϵ.

- **tc[5]**
  specifies another relative function convergence criterion (FTOL2). Termination requires a small standard deviation of the function values of the $n + 1$ simplex vertices $x_0^{(k)}, \ldots, x_n^{(k)}$.

$$\sqrt{\frac{1}{n+1} \sum_l (f(x_l^{(k)}) - \overline{f}(x^{(k)}))^2} \leq \textit{FTOL2}$$

  where $\overline{f}(x^{(k)}) = \frac{1}{n+1} \sum_l f(x_l^{(k)})$. If there are a active boundary constraints at $x^{(k)}$, the mean and standard deviation are computed only for the $n + 1 - a$ unconstrained vertices. The default is $tc[5] =$ 1E−6.

- **tc[6]**
  specifies the absolute function convergence criterion (ABSFTOL). Termination

requires a small absolute difference between the function values of the vertices in the simplex with the largest and smallest function values.

$$|f_{hi}^{(k)} - f_{lo}^{(k)}| \leq ABSFTOL$$

The default is $tc[6] = 0$.

- **tc[7]**
  specifies the FSIZE value used in the FTOL termination criterion. The default is $tc[7] = 0$.

- **tc[8]**
  specifies the relative parameter convergence criterion (XTOL). Termination requires a small relative parameter difference between the vertices with the largest and smallest function values.

$$\frac{\max_j |x_j^{lo} - x_j^{hi}|}{\max(|x_j^{lo}|, |x_j^{hi}|, XSIZE)} \leq XTOL$$

The default is $tc[8] = 1\mathrm{E}{-8}$.

- **tc[9]**
  specifies the absolute parameter convergence criterion (ABSXTOL). Termination requires either a small length, $\alpha^{(k)}$, of the vertices of a restart simplex or a small simplex size, $\delta^{(k)}$.

$$
\begin{aligned}
\alpha^{(k)} &\leq ABSXTOL \\
\delta^{(k)} &\leq ABSXTOL
\end{aligned}
$$

where $\delta^{(k)}$ is defined as the L1 distance of the simplex vertex with the smallest function value, $y^{(k)}$, to the other n simplex points, $x_l^{(k)} \neq y$.

$$\delta^{(k)} = \sum_{x_l \neq y} \| x_l^{(k)} - y^{(k)} \|_1$$

The default is $tc[9] = 1\mathrm{E}{-8}$.

- **tc[10]**
  specifies the XSIZE value used in the XTOL termination criterion. The default is $tc[10] = 0$.

Termination Criteria for Unconstrained and Linearly Constrained Techniques

- **tc[4]**

  specifies the relative gradient convergence criterion (GTOL). For all techniques except the NLPCG subroutine, termination requires that the normalized predicted function reduction is small.

  $$\frac{g(x^{(k)})^T [G^{(k)}]^{-1} g(x^{(k)})}{\max(|f(x^{(k)})|, \textit{FSIZE})} \leq \textit{GTOL}$$

  where *FSIZE* is defined by *tc*[10]. For the NLPCG technique (where a reliable Hessian estimate is not available),

  $$\frac{\| g(x^{(k)}) \|_2^2}{\| g(x^{(k)}) - g(x^{(k-1)}) \|_2} \frac{\| s(x^{(k)}) \|_2}{\max(|f(x^{(k)})|, \textit{FSIZE})} \leq \textit{GTOL}$$

  is used. The default is $tc[4] = $1E$-8$.

- **tc[5]**

  specifies another relative gradient convergence criterion (GTOL2). This criterion is used only by the NLPLM subroutine.

  $$\max_j \frac{|g_j(x^{(k)})|}{\sqrt{f(x^{(k)}) G_{j,j}^{(k)}}} \leq \textit{GTOL2}$$

  The default is $tc[5]=0$.

- **tc[6]**

  specifies the absolute gradient convergence criterion (ABSGTOL). Termination requires that the maximum absolute gradient element be small.

  $$\max_j |g_j(x^{(k)})| \leq \textit{ABSGTOL}$$

  The default is $tc[6] = $1E$-5$.

- **tc[7]**

  specifies the relative function convergence criterion (FTOL). Termination requires a small relative change of the function value in consecutive iterations.

  $$\frac{|f(x^{(k)}) - f(x^{(k-1)})|}{\max(|f(x^{(k-1)})|, FSIZE)} \leq \textit{FTOL}$$

  where $FSIZE$ is defined by *tc*[10]. The default is $tc[7] = 10^{-\text{FDIGITS}}$, where FDIGITS is controlled by the *par*[8] argument. The *par*[8] argument has a default value of $\log_{10}(\epsilon)$, where ϵ is the machine precision. Hence, the default for *FTOL* is ϵ.

- **tc[8]**

  specifies another function convergence criterion (FTOL2). For least-squares problems, termination requires a small predicted reduction of the objective function, $df^{(k)} \approx f(x^{(k)}) - f(x^{(k)} + s^{(k)})$. The predicted reduction is computed by approximating the objective function by the first two terms of the Taylor series and substituting the Newton step, $s^{(k)} = -G^{(k)-1}g^{(k)}$, as follows:

  $$
  \begin{aligned}
  df^{(k)} &= -g^{(k)T}s^{(k)} - \frac{1}{2}s^{(k)T}G^{(k)}s^{(k)} \\
  &= -\frac{1}{2}s^{(k)T}g^{(k)} \\
  &\leq FTOL2
  \end{aligned}
  $$

  The FTOL2 criterion is the unscaled version of the GTOL criterion. The default is $tc[8]=0$.

- **tc[9]**

  specifies the absolute function convergence criterion (ABSFTOL). Termination requires a small change of the function value in consecutive iterations.

  $$
  |f(x^{(k-1)}) - f(x^{(k)})| \leq ABSFTOL
  $$

  The default is $tc[9]=0$.

- **tc[10]**

  specifies the FSIZE value used in the GTOL and FTOL termination criteria. The default is $tc[10]=0$.

- **tc[11]**

  specifies the relative parameter convergence criterion (XTOL). Termination requires a small relative parameter change in consecutive iterations.

  $$
  \frac{\max_j |x_j^{(k)} - x_j^{(k-1)}|}{\max(|x_j^{(k)}|, |x_j^{(k-1)}|, XSIZE)} \leq XTOL
  $$

  The default is $tc[11]=0$.

- **tc[12]**

  specifies the absolute parameter convergence criterion (ABSXTOL). Termination requires a small Euclidean distance between parameter vectors in consecutive iterations.

  $$
  \| x^{(k)} - x^{(k-1)} \|_2 \leq ABSXTOL
  $$

  The default is $tc[12]=0$.

- **tc[13]**

  specifies the XSIZE value used in the XTOL termination criterion. The default is $tc[13]=0$.

Termination Criteria for Nonlinearly Constrained Techniques

The only algorithm available for nonlinearly constrained optimization other than the NLPNMS subroutine is the NLPQN subroutine, when you specify the *"nlc"* module argument. This method, unlike the other optimization methods, does not monotonically reduce the value of the objective function or some kind of merit function that combines objective and constraint functions. Instead, the algorithm uses the watchdog technique with backtracking of Chamberlain and others (1982). Therefore, no termination criteria are implemented that are based on the values x or f in consecutive iterations. In addition to the criteria used by all optimization techniques, there are three other termination criteria available; these are based on the Lagrange function

$$L(x, \lambda) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x)$$

and its gradient

$$\nabla_x L(x, \lambda) = g(x) - \sum_{i=1}^{m} \lambda_i \nabla_x c_i(x)$$

where m denotes the total number of constraints, $g = g(x)$ is the gradient of the objective function, and λ is the vector of Lagrange multipliers. The Kuhn-Tucker conditions require that the gradient of the Lagrange function is zero at the optimal point (x^*, λ^*), as follows:

$$\nabla_x L(x^*, \lambda^*) = 0$$

- **tc[4]**
  specifies the GTOL criterion, which requires that the normalized predicted function reduction be small.

  $$\frac{|g(x^{(k)})s(x^{(k)})| + \sum_{i=1}^{m} |\lambda_i c_i(x^{(k)})|}{\max(|f(x^{(k)})|, FSIZE)} \leq GTOL$$

  where *FSIZE* is defined by the *tc*[10] argument. The default is $tc[4] =$1E$-$8.

- **tc[6]**
  specifies the ABSGTOL criterion, which requires that the maximum absolute gradient element of the Lagrange function be small.

  $$\max_j |\{\nabla_x L(x^{(k)}, \lambda^{(k)})\}_j| \leq ABSGTOL$$

  The default is $tc[6] =$1E$-$5.

- **tc[8]**

   specifies the FTOL2 criterion, which requires that the predicted function re-
   duction be small.

$$|g(x^{(k)})s(x^{(k)})| + \sum_{i=1}^{m} |\lambda_i c_i| \leq \textit{FTOL2}.$$

   The default is $tc[8]$ $=$1E$-$6. This is the criterion used by the programs
   VMCWD and VF02AD of Powell (1982b).

Control Parameters Vector

For all optimization and least-squares subroutines, the input argument *par* specifies
a vector of parameters that control the optimization process. For the NLPFDD and
NLPFEA subroutines, the *par* argument is defined differently. For each element of the
par vector there exists a default value, and if you specify a missing value, the default
is used. Table 11.5 summarizes the uses of the *par* argument for the optimization and
least-squares subroutines.

Table 11.5. Summary of the Control Parameters Vector

Index	Description
1	specifies the singularity criterion (SINGULAR)
2	specifies the initial step length or trust-region radius
3	specifies the range for active (violated) constraints (LCEPS)
4	specifies the Lagrange multiplier threshold for constraints (LCDEACT)
5	specifies a criterion to determine linear dependence of constraints (LCSING)
6	specifies the required accuracy of the line-search algorithms (LSPRECISION)
7	reduces the line-search step size in successive iterations (DAMPSTEP)
8	specifies the number of accurate digits used in evaluating the objective function (FDIGITS)
9	specifies the number of accurate digits used in evaluating the nonlinear constraints (CDIGITS)
10	specifies a scalar factor for the diagonal of the initial Hessian (DIAHES)

- **par[1]**

   specifies the singularity criterion for the decomposition of the Hessian matrix
   (SINGULAR). The value must be between zero and one, and the default is
   $par[1]$ $=$1E$-$8.

- **par[2]**

   specifies different features depending on the subroutine in which it is used.
   In the NLPNMS subroutine, it defines the size of the start simplex. For the
   original Nelder-Mead simplex algorithm, the default value is $par[2] = 1$; for

the COBYLA algorithm, the default is $par[2] = 0.5$. In the NLPCG, NLPQN, and NLPHQN subroutines, the $par[2]$ argument specifies an upper bound for the initial step length for the line search during the first five iterations. The default initial step length is $par[2] = 1$. In the NLPTR, NLPDD, and NLPLM subroutines, the $par[2]$ argument specifies a factor for the initial trust-region radius, Δ. For highly nonlinear functions, the default step length or trust-region radius can result in arithmetic overflows. In that case, you can specify stepwise decreasing values of $par[2]$, such as $par[2]$=1E−1, $par[2]$=1E−2, $par[2]$=1E−4, until the subroutine starts to iterate successfully.

- **par[3]**
  specifies the range (LCEPS) for active and violated linear constraints. The ith constraint is considered an active constraint if the point $x^{(k)}$ satisfies the condition

$$\left| \sum_{j=1}^{n} a_{ij} x_j^{(k)} - b_i \right| \leq LCEPS(|b_i| + 1)$$

  where *LCEPS* is the value of $par[3]$ and a_{ij} and b_i are defined as in the section "Parameter Constraints" on page 341. Otherwise, the constraint i is either an inactive inequality or a violated inequality or equality constraint. The default is $par[3]$ =1E−8. During the optimization process, the introduction of rounding errors can force the subroutine to increase the value of $par[3]$ by a power of 10, but the value will never become larger than 1E−3.

- **par[4]**
  specifies a threshold (LCDEACT) for the Lagrange multiplier that decides whether an active inequality constraint must remain active or can be deactivated. For maximization, $par[4]$ must be positive, and for minimization, $par[4]$ must be negative. The default is

$$par[4] = \pm \min \left(0.01, \max \left(0.1 \times ABSGTOL, \ 0.001 \times gmax^{(k)} \right) \right)$$

  where the positive value is for maximization and the negative value is for minimization. *ABSGTOL* is the value of the absolute gradient criterion, and $gmax^{(k)}$ is the maximum absolute element of the gradient, $g^{(k)}$, or the projected gradient, $Z^T g^{(k)}$.

- **par[5]**
  specifies a criterion (LCSING) used in the update of the QR decomposition that decides whether an active constraint is linearly dependent on a set of other active constraints. The default is $par[5]$ =1E−8. As the value of $par[5]$ increases, more active constraints are recognized as being linearly dependent. If the value of $par[5]$ is larger than 0.1, it is reset to 0.1, and if it is negative, it is reset to zero.

- **par[6]**

  specifies the degree of accuracy (LSPRECISION) that should be obtained by the second or third line-search algorithm. This argument can be used with the NLPCG, NLPHQN, and NLPNRA algorithms and with the NLPQN algorithm if the *"nlc"* argument is specified. Usually, an imprecise line search is computationally inexpensive and successful, but for more difficult optimization problems, a more precise and time consuming line search may be necessary. Refer to Fletcher (1987) for details. If you have numerical problems, you should decrease the value of the *par*[6] argument to obtain a more precise line search. The default values are given in the following table.

Subroutine	Update Method	Default value
NLPCG	All	$par[6] = 0.1$
NLPHQN	DBFGS	$par[6] = 0.1$
NLPHQN	DDFP	$par[6] = 0.06$
NLPNRA	No update	$par[6] = 0.9$
NLPQN	BFGS, DBFGS	$par[6] = 0.4$
NLPQN	DFP, DDFP	$par[6] = 0.06$

- **par[7]**

  specifies a scalar factor (DAMPSTEP) that can be used to reduce the step size in each of the first five iterations. In each of these iterations, the starting step size, $\alpha^{(0)}$, can be no larger than the value of *par*[7] times the step size obtained by the line-search algorithm in the previous iteration. If *par*[7] is missing or if*par*[7]=0, which is the default, the starting step size in iteration t is computed as a function of the function change from the former iteration, $f^{(t-1)} - f^{(t)}$. If the computed value is outside the interval $[0.1, 10.0]$, it is moved to the next endpoint. You can further restrict the starting step size in the first five iterations with the *par*[2] argument.

- **par[8]**

  specifies the number of accurate digits (FDIGITS) used to evaluate the objective function. The default is $-\log_{10}(\epsilon)$, where ϵ is the machine precision, and fractional values are permitted. This value is used to compute the step size h for finite difference derivatives and the default value for the FTOL termination criterion.

- **par[9]**

  specifies the number of accurate digits (CDIGITS) used to evaluate the nonlinear constraint functions of the *"nlc"* module. The default is $-\log_{10}(\epsilon)$, where ϵ is the machine precision, and fractional values are permitted. The value is used to compute the step size h for finite difference derivatives. If first-order derivatives are specified by the *"jacnlc"* module, the *par*[9] argument is ignored.

- **par[10]**

  specifies a scalar factor (DIAHES) for the diagonal of the initial Hessian approximation. This argument is available in the NLPDD, NLPHQN, and NLPQN subroutines. If the *opt*[7] argument is not specified, the initial Hessian approximation is a multiple of the identity matrix determined by the magnitude of the initial gradient $g(x^{(0)})$. The value of the *par*[10] argument is used to specify $par[10] \times \mathbf{I}$ for the initial Hessian in the quasi-Newton algorithm.

Printing the Optimization History

Each optimization and least-squares subroutine prints the optimization history, as long as *opt*[2] $\geq$ 1 and you do not specify the *"ptit"* module argument. You can use this output to check for possible convergence problems. If you specify the *"ptit"* argument, you can enter a print command inside the module, which is called at each iteration.

The amount of information printed depends on the *opt*[2] argument. See the section "Options Vector" on page 343.

The output consists of three main parts:

- **Optimization Start Output**

  The following information about the initial state of the optimization can be printed:

  - the number of constraints that are active at the starting point, or, more precisely, the number of constraints that are currently members of the working set. If this number is followed by a plus sign (+), there are more active constraints, at least one of which is temporarily released from the working set due to negative Lagrange multipliers.
  - the value of the objective function at the starting point
  - the value of the largest absolute (projected) gradient element
  - the initial trust-region radius for the NLPTR and NLPLM subroutines

- **General Iteration History**

  In general, the iteration history consists of one line of printed output for each iteration, with the exception of the Nelder-Mead simplex method. The NLPNMS subroutine prints a line only after several internal iterations because some of the termination tests are time-consuming compared to the simplex operations and because the subroutine typically uses many iterations.

  The iteration history always includes the following columns:

  - *iter* is the iteration number.
  - *nrest* is the number of iteration restarts.
  - *nfun* is the number of function calls.
  - *act* is the number of active constraints.
  - *optcrit* is the value of the optimization criterion.
  - *difcrit* is the difference between adjacent function values.

– *maxgrad* is the maximum of the absolute (projected) gradient components.

An apostrophe trailing the number of active constraints indicates that at least one of the active constraints was released from the active set due to a significant Lagrange multiplier.

Some subroutines print additional information at each iteration; for details see the entry corresponding to each subroutine in the "Nonlinear Optimization and Related Subroutines" section on page 782.

- **Optimization Result Output**
  The output ends with the following information about the optimization result:

  – the number of constraints that are active at the final point, or more precisely, the number of constraints that are currently members of the working set. When this number is followed by a plus sign ($+$), there are more active constraints, at least one of which is temporarily released from the working set due to negative Lagrange multipliers.
  – the value of the objective function at the final point
  – the value of the largest absolute (projected) gradient element

Nonlinear Optimization Examples

Example 11.1. Chemical Equilibrium

The following example is used in many test libraries for nonlinear programming. It appeared originally in Bracken and McCormick (1968).

The problem is to determine the composition of a mixture of various chemicals that satisfy the mixture's chemical equilibrium state. The second law of thermodynamics implies that at a constant temperature and pressure, a mixture of chemicals satisfies its chemical equilibrium state when the free energy of the mixture is reduced to a minimum. Therefore, the composition of the chemicals satisfying its chemical equilibrium state can be found by minimizing the free energy of the mixture.

The following notation is used in this problem:

m	number of chemical elements in the mixture
n	number of compounds in the mixture
x_j	number of moles for compound j, $j = 1, \ldots, n$
s	total number of moles in the mixture, $s = \sum_{i=1}^{n} x_j$
a_{ij}	number of atoms of element i in a molecule of compound j
b_i	atomic weight of element i in the mixture $i = 1, \ldots, n$

The constraints for the mixture are as follows. Each of the compounds must have a nonnegative number of moles.

$$x_j \geq 0, \quad j = 1, \ldots, n$$

Example 11.1. Chemical Equilibrium ◆ 361

There is a mass balance relationship for each element. Each relation is given by a linear equality constraint.

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \quad i = 1, \ldots, m$$

The objective function is the total free energy of the mixture.

$$f(x) = \sum_{j=1}^{n} x_j \left[c_j + \ln \left(\frac{x_j}{s} \right) \right]$$

where

$$c_j = \left(\frac{F^0}{RT} \right)_j + \ln(P)$$

and $\left(F^0 / RT \right)_j$ is the model standard free energy function for the jth compound. The value of $\left(F^0 / RT \right)_j$ is found in existing tables. P is the total pressure in atmospheres.

The problem is to determine the parameters x_j that minimize the objective function $f(x)$ subject to the nonnegativity and linear balance constraints. To illustrate this, consider the following situation. Determine the equilibrium composition of compound $\frac{1}{2}N_2H_4 + \frac{1}{2}O_2$ at temperature $T = 3500°K$ and pressure $P = 750$ *psi*. The following table gives a summary of the information necessary to solve the problem.

| | | | | a_{ij} | | |
| | | | | $i=1$ | $i=2$ | $i=3$ |
j	Compound	$(F^0/RT)_j$	c_j	H	N	O
1	H	-10.021	-6.089	1		
2	H_2	-21.096	-17.164	2		
3	H_2O	-37.986	-34.054	2		1
4	N	-9.846	-5.914		1	
5	N_2	-28.653	-24.721		2	
6	NH	-18.918	-14.986	1	1	
7	NO	-28.032	-24.100		1	1
8	O	-14.640	-10.708			1
9	O_2	-30.594	-26.662			2
10	OH	-26.111	-22.179	1		1

The following statements solve the minimization problem:

```
proc iml;
   c = { -6.089 -17.164 -34.054  -5.914 -24.721
         -14.986 -24.100 -10.708 -26.662 -22.179 };
   start F_BRACK(x) global(c);
      s = x[+];
```

```
        f = sum(x # (c + log(x / s)));
        return(f);
     finish F_BRACK;

     con = { .   .   .   .   .   .   .   .   .   .     .     .   ,
             .   .   .   .   .   .   .   .   .   .     .     .   ,
            1.  2.  2.  .   .   1.  .   .   .   1.    0.    2.  ,
             .   .   .   1.  2.  1.  1.  .   .   .     0.    1.  ,
             .   .   1.  .   .   .   1.  1.  2.  1.    0.    1.  };
     con[1,1:10] = 1.e-6;

     x0 = j(1,10, .1);
     optn = {0 3};

     title 'NLPTR subroutine: No Derivatives';
     call nlptr(xres,rc,"F_BRACK",x0,optn,con);
```

The F_BRACK module specifies the objective function, $f(x)$. The matrix CON specifies the constraints. The first row gives the lower bound for each parameter, and to prevent the evaluation of the $\log(x)$ function for values of x that are too small, the lower bounds are set here to $1E-6$. The following three rows contain the three linear equality constraints.

The starting point, which must be given to specify the number of parameters, is represented by X0. The first element of the OPTN vector specifies a minimization problem, and the second element specifies the amount of printed output.

The CALL NLPTR statement runs trust-region minimization. In this case, since no analytic derivatives are specified, the F_BRACK module is used to generate finite difference approximations for the gradient vector and Hessian matrix.

The output is shown in the following figures. The iteration history does not show any problems.

Example 11.1. Chemical Equilibrium ◆ 363

```
                          Optimization Start

Active Constraints                3  Objective Function    -45.05516448
Max Abs Gradient       4.4710303342  Radius                          1
Element


                                         Max Abs            Trust
         Rest  Func Act   Objective  Obj Fun Gradient            Region
  Iter  arts Calls Con    Function    Change Element  Lambda   Radius

     1     0     2  3'   -47.33413    2.2790   4.3613   2.456   1.000
     2     0     3  3'   -47.70051    0.3664   7.0044   0.908   0.418
     3     0     4  3    -47.73117    0.0307   5.3051       0   0.359
     4     0     5  3    -47.73426   0.00310   3.7015       0   0.118
     5     0     6  3    -47.73982   0.00555   2.3054       0  0.0169
     6     0     7  3    -47.74846   0.00864   1.3029  90.184 0.00476
     7     0     9  3    -47.75796   0.00950   0.5073       0  0.0134
     8     0    10  3    -47.76094   0.00297   0.0988       0  0.0124
     9     0    11  3    -47.76109  0.000155  0.00447       0  0.0111
    10     0    12  3    -47.76109  3.385E-7 0.000011       0 0.00332


                         Optimization Results

Iterations                       10  Function Calls                 13
Hessian Calls                    11  Active Constraints              3
Objective Function     -47.76109086  Max Abs Gradient     7.3901293E-6
                                     Element
Lambda                            0  Actual Over Pred                0
                                     Change
Radius              0.0033214552
```

The output lists the optimal parameters with the gradient.

```
                    Optimization Results
                    Parameter Estimates
                                           Gradient
                                          Objective
          N Parameter          Estimate    Function

          1 X1                 0.040668   -9.785055
          2 X2                 0.147730  -19.570111
          3 X3                 0.783154  -34.792170
          4 X4                 0.001414  -12.968920
          5 X5                 0.485247  -25.937841
          6 X6                 0.000693  -22.753976
          7 X7                 0.027399  -28.190992
          8 X8                 0.017947  -15.222060
          9 X9                 0.037314  -30.444119
         10 X10                0.096871  -25.007115

          Value of Objective Function = -47.76109086
```

The three equality constraints are satisfied at the solution.

```
              Linear Constraints Evaluated at Solution


[1] ACT -3.053E-16  = -2.0000 + 1.0000 * X1 + 2.0000 * X2
          + 2.0000 * X3 + 1.0000 * X6 + 1.0000 * X10

[2] ACT -1.735E-17  = -1.0000 + 1.0000 * X4 + 2.0000 * X5
          + 1.0000 * X6 + 1.0000 * X7

[3] ACT -1.527E-16  = -1.0000 + 1.0000 * X3 + 1.0000 * X7
          + 1.0000 * X8 + 2.0000 * X9 +  1.0000 * X10
```

The Lagrange multipliers and the projected gradient are also printed. The elements
of the projected gradient must be small to satisfy a first-order optimality condition.

```
              First Order Lagrange Multipliers


                                        Lagrange
              Active Constraint         Multiplier

              Linear EC     [1]          -9.785055
              Linear EC     [2]         -12.968922
              Linear EC     [3]         -15.222061


                   Projected Gradient


                        Free     Projected
                   Dimension      Gradient

                          1  0.000000328
                          2 -9.703359E-8
                          3 3.2183113E-8
                          4 -0.000007390
                          5 -0.000005172
                          6 -0.000005669
                          7 -0.000000937
```

Example 11.2. Network Flow and Delay

The following example is taken from the user's guide of the GINO program
(Liebman, Lasdon, Schrage, and Waren 1986). A simple network of five roads (arcs)
can be illustrated by a path diagram.

The five roads connect four intersections illustrated by numbered nodes. Each minute,
F vehicles enter and leave the network. The parameter x_{ij} refers to the flow from
node i to node j. The requirement that traffic that flows into each intersection j must
also flow out is described by the linear equality constraint

$$\sum_i x_{ij} = \sum_i x_{ji} \quad , \quad j = 1, \ldots, n$$

Example 11.2. Network Flow and Delay ◆ 365

In general, roads also have an upper limit on the number of vehicles that can be handled per minute. These limits, denoted c_{ij}, can be enforced by boundary constraints:

$$0 \leq x_{ij} \leq c_{ij}$$

The goal in this problem is to maximize the flow, which is equivalent to maximizing the objective function $f(x)$, where $f(x)$ is

$$f(x) = x_{24} + x_{34}$$

The boundary constraints are

$$0 \leq \quad x_{12}, x_{32}, x_{34} \quad \leq 10$$
$$0 \leq \quad \quad x_{13}, x_{24} \quad \quad \leq 30$$

and the flow constraints are

$$x_{13} \quad = \quad x_{32} + x_{34}$$
$$x_{24} \quad = \quad x_{12} + x_{32}$$
$$x_{12} + x_{13} \quad = \quad x_{24} + x_{34}$$

The three linear equality constraints are linearly dependent. One of them is deleted automatically by the optimization subroutine. The following notation is used in this example:

$$X1 = x_{12}, \quad X2 = x_{13}, \quad X3 = x_{32}, \quad X4 = x_{24}, \quad X5 = x_{34}$$

Even though the NLPCG subroutine is used, any other optimization subroutine would also solve this small problem.

```
proc iml;
   title 'Maximum Flow Through a Network';
   start MAXFLOW(x);
      f = x[4] + x[5];
      return(f);
   finish MAXFLOW;

   con = {  0.   0.   0.   0.   0.    .    . ,
           10.  30.  10.  30.  10.    .    . ,
            0.   1.  -1.   0.  -1.   0.   0. ,
            1.   0.   1.  -1.   0.   0.   0. ,
            1.   1.   0.  -1.  -1.   0.   0. };
   x = j(1,5, 1.);
   optn = {1 3};
   call nlpcg(xres,rc,"MAXFLOW",x,optn,con);
```

The optimal solution is shown in the following output.

```
                   Optimization Results
                   Parameter Estimates
                                    Gradient      Active
                                    Objective     Bound
   N Parameter          Estimate    Function      Constraint

   1 X1              10.000000            0        Upper BC
   2 X2              10.000000            0
   3 X3              10.000000     1.000000        Upper BC
   4 X4              20.000000     1.000000
   5 X5            -1.11022E-16            0        Lower BC

              Value of Objective Function = 30
```

Finding the maximum flow through a network is equivalent to solving a simple linear optimization problem, and for large problems, the LP procedure or the NETFLOW procedure of the SAS/OR product can be used. On the other hand, finding a traffic pattern that minimizes the total delay to move F vehicles per minute from node 1 to node 4 includes nonlinearities that need nonlinear optimization techniques. As traffic volume increases, speed decreases. Let t_{ij} be the travel time on arc (i, j) and assume that the following formulas describe the travel time as decreasing functions of the amount of traffic:

$$
\begin{aligned}
t_{12} &= 5 + 0.1x_{12}/(1 - x_{12}/10) \\
t_{13} &= x_{13}/(1 - x_{13}/30) \\
t_{32} &= 1 + x_{32}/(1 - x_{32}/10) \\
t_{24} &= x_{24}/(1 - x_{24}/30) \\
t_{34} &= 5 + x_{34}/(1 - x_{34}/10)
\end{aligned}
$$

These formulas use the road capacities (upper bounds), and you can assume that $F = 5$ vehicles per minute have to be moved through the network. The objective is now to minimize

$$ f = f(x) = t_{12}x_{12} + t_{13}x_{13} + t_{32}x_{32} + t_{24}x_{24} + t_{34}x_{34} $$

The constraints are

$$
\begin{aligned}
0 &\le x_{12}, x_{32}, x_{34} \le 10 \\
0 &\le x_{13}, x_{24} \le 30
\end{aligned}
$$

$$
\begin{aligned}
x_{13} &= x_{32} + x_{34} \\
x_{24} &= x_{12} + x_{32} \\
x_{24} + x_{34} &= F = 5
\end{aligned}
$$

Example 11.2. Network Flow and Delay ◆ 367

In the following code, the NLPNRR subroutine is used to solve the minimization problem:

```
proc iml;
   title 'Minimize Total Delay in Network';
   start MINDEL(x);
      t12 = 5. + .1 * x[1] / (1. - x[1] / 10.);
      t13 = x[2] / (1. - x[2] / 30.);
      t32 = 1. + x[3] / (1. - x[3] / 10.);
      t24 = x[4] / (1. - x[4] / 30.);
      t34 = 5. + .1 * x[5] / (1. - x[5] / 10.);
      f = t12*x[1] + t13*x[2] + t32*x[3] + t24*x[4] + t34*x[5];
      return(f);
   finish MINDEL;

   con = {  0.  0.  0.  0.  0.   .   . ,
           10. 30. 10. 30. 10.   .   . ,
            0.  1. -1.  0. -1.  0.  0. ,
            1.  0.  1. -1.  0.  0.  0. ,
            0.  0.  0.  1.  1.  0.  5. };

   x = j(1,5, 1.);
   optn = {0 3};
   call nlpnrr(xres,rc,"MINDEL",x,optn,con);
```

The optimal solution is shown in the following output.

```
                   Optimization Results
                   Parameter Estimates
                                   Gradient      Active
                                   Objective     Bound
  N Parameter          Estimate    Function      Constraint

  1 X1                 2.500001    5.777778
  2 X2                 2.499999    5.702478
  3 X3            5.551115E-17     1.000000      Lower BC
  4 X4                 2.500001    5.702481
  5 X5                 2.499999    5.777778

        Value of Objective Function = 40.303030303
```

The active constraints and corresponding Lagrange multiplier estimates (costs) are shown in the following output.

```
                 Linear Constraints Evaluated at Solution

[1] ACT          0   =  0 + 1.0000 * X2 - 1.0000 * X3 - 1.0000 * X5

[2] ACT 4.4409E-16   =  0 + 1.0000 * X1 + 1.0000 * X3 - 1.0000 * X4

[3] ACT          0   = -5.0000 + 1.0000 * X4 + 1.0000 * X5




                    First Order Lagrange Multipliers


                                          Lagrange
                    Active Constraint     Multiplier

                  Lower BC      X3         0.924702
                  Linear EC     [1]        5.702479
                  Linear EC     [2]        5.777777
                  Linear EC     [3]       11.480257
```

Example 11.3. Compartmental Analysis

Numerical Considerations

An important class of nonlinear models involves a dynamic description of the response rather than an explicit description. These models arise often in chemical kinetics, pharmacokinetics, and ecological compartmental modeling. Two examples are presented in this section. Refer to Bates and Watts (1988) for a more general introduction to the topic.

In this class of problems, function evaluations, as well as gradient evaluations, are not done in full precision. Evaluating a function involves the numerical solution of a differential equation with some prescribed precision. Therefore, two choices exist for evaluating first- and second-order derivatives:

- differential equation approach
- finite difference approach

In the differential equation approach, the components of the Hessian and the gradient are written as a solution of a system of differential equations that can be solved simultaneously with the original system. However, the size of a system of differential equations, n, would suddenly increase to $n^2 + 2n$. This huge increase makes the finite difference approach an easier one.

With the finite difference approach, a very delicate balance of all the precision requirements of every routine must exist. In the examples that follow, notice the relative levels of precision that are imposed on different modules. Since finite differences are used to compute the first- and second-order derivatives, it is incorrect to set the precision of the ODE solver at a coarse level because that would render the numerical estimation finite difference worthless.

Example 11.3. Compartmental Analysis ◆ 369

A coarse computation of the solution of the differential equation cannot be accompanied by very fine computation of the finite difference estimates of the gradient and the Hessian. That is, you cannot set the precision of the differential equation solver to be 1E−4 and perform the finite difference estimation with a precision of 1E−10. In addition, this precision must be well-balanced with the termination criteria imposed on the optimization process.

In general, if the precision of the function evaluation is $O(\epsilon)$, the gradient should be computed by finite differences $O(\sqrt{\epsilon})$, and the Hessian should be computed with finite differences $O(\epsilon^{\frac{1}{3}})$. *

Diffusion of Tetracycline

Consider the concentration of tetracycline hydrochloride in blood serum. The tetracycline is administered to a subject orally, and the concentration of the tetracycline in the serum is measured. The biological system to be modeled will consist of two compartments: a gut compartment in which tetracycline is injected and a blood compartment that absorbs the tetracycline from the gut compartment for delivery to the body. Let $\gamma_1(t)$ and $\gamma_2(t)$ be the concentrations at time t in the gut and the serum, respectively. Let θ_1 and θ_2 be the transfer parameters. The model is depicted as follows.
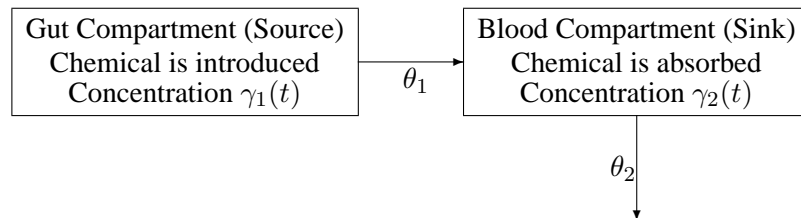
The rates of flow of the drug are described by the following pair of ordinary differential equations:

$$\frac{d\gamma_1(t)}{dt} = -\theta_1\gamma_1(t)$$
$$\frac{d\gamma_2(t)}{dt} = \theta_1\gamma_1(t) - \theta_2\gamma_2(t)$$

The initial concentration of the tetracycline in the gut is unknown, and while the concentration in the blood can be measured at all times, initially it is assumed to be zero. Therefore, for the differential equation, the initial conditions will be given by

$$\gamma_1(0) = \theta_3$$
$$\gamma_2(0) = 0$$

---

*In Release 6.09 and in later releases, you can specify the step size h in the finite difference formulas.

Also, a nonnegativity constraint is imposed on the parameters θ_1, θ_2, and θ_3, although for numerical purposes, you may need to use a small value instead of zero for these bounds (such as $1E{-}7$).

Suppose y_i is the observed serum concentration at time t_i. The parameters are estimated by minimizing the sum of squares of the differences between the observed and predicted serum concentrations:

$$\sum_i (y_i - \gamma_2(t_i))^2$$

The following IML program illustrates how to combine the NLPDD subroutine and the ODE subroutine to estimate the parameters $(\theta_1, \theta_2, \theta_3)$ of this model. The input data are the measurement time and the concentration of the tetracycline in the blood. For more information on the ODE call, see the "ODE Call" section on page 820.

```
data tetra;
   input t c @@;
   datalines;
 1 0.7    2 1.2    3 1.4    4 1.4    6 1.1
 8 0.8   10 0.6   12 0.5   16 0.3
;

proc iml;
   use tetra;
   read all into tetra;
   start f(theta) global(thmtrx,t,h,tetra,eps);
      thmtrx = ( -theta[1] || 0 )        //
               (  theta[1] || -theta[2] );
      c = theta[3]//0 ;
      t = 0 // tetra[,1];
      call ode( r1, "der",c , t, h) j="jac" eps=eps;
      f = ssq((r1[2,])'-tetra[,2]);
      return(f);
   finish;

   start der(t,x) global(thmtrx);
      y = thmtrx*x;
      return(y);
   finish;

   start jac(t,x) global(thmtrx);
      y = thmtrx;
      return(y);
   finish;

   h      = {1.e-14 1. 1.e-5};
   opt    = {0 2 0 1 };
   tc     = repeat(.,1,12);
   tc[1]  = 100;
   tc[7]  = 1.e-8;
   par    = { 1.e-13 . 1.e-10 . . . . };
```

Example 11.3. Compartmental Analysis ◆ 371

```
con     = j(1,3,0.);
itheta = { .1  .3  10};
eps     = 1.e-11;

call nlpdd(rc,rx,"f",itheta) blc=con opt=opt tc=tc par=par;
```

The output from the optimization process is shown in Output 11.3.1.

Output 11.3.1. Printed Output for Tetracycline Diffusion Problem

```
                          Optimization Start
                        Parameter Estimates
                                          Gradient        Lower       Upper
                                          Objective       Bound       Bound
    N Parameter            Estimate       Function     Constraint   Constraint

    1 X1                   0.100000       76.48208           0         .
    2 X2                   0.300000      -48.32095           0         .
    3 X3                  10.000000        1.66610           0         .


               Value of Objective Function = 4.1469872335




                        Double Dogleg Optimization

       Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

                        Without Parameter Scaling
                  Gradient Computed by Finite Differences

                  Parameter Estimates            3
                  Lower Bounds                   3
                  Upper Bounds                   0

                          Optimization Start

Active Constraints            0  Objective Function    4.1469872326
Max Abs Gradient         76.543381  Radius                        1
Element


                                       Max Abs            Slope
          Rest  Func Act  Objective  Obj Fun Gradient            Search
   Iter  arts Calls Con   Function   Change  Element   Lambda    Direc

     1     0    5   0      3.12117   1.0258     124.3  67.129   -8.023
     2     0    6   0      0.89524   2.2259   14.1471   1.885   -5.021
     3     0    7   0      0.32333   0.5719    3.7144   1.186   -0.786
     .
     .
     .
    31     0   38   0      0.03565 4.24E-11 3.196E-6       0  -18E-12
```

Output 11.3.1. (continued)

```
                    Optimization Results

Iterations                   31  Function Calls                39
Gradient Calls               33  Active Constraints             0
Objective Function    0.035648021  Max Abs Gradient     3.195746E-6
                                   Element




                    Optimization Results

Slope of Search      -1.76538E-11  Radius                     1
Direction

GCONV convergence criterion satisfied.




                    Optimization Results
                      Parameter Estimates
                                        Gradient
                                        Objective
          N Parameter         Estimate   Function

          1 X1               0.182440    -0.00251
          2 X2               0.436010     0.00122
          3 X3               6.020476    -0.0001875

          Value of Objective Function = 0.0356480211
```

The differential equation model is linear, and in fact, it can be solved using an eigenvalue decomposition (this is not always feasible without complex arithmetic). Alternately, the availability and the simplicity of the closed form representation of the solution enables you to replace the solution produced by the ODE routine with the simpler and faster analytical solution. Closed forms are not expected to be easily available for nonlinear systems of differential equations, which is why the preceding solution was introduced.

The closed form of the solution requires a change to the function $f(\cdot)$. The functions needed as arguments of the ODE routine, namely the *der* and *jac* modules, can be removed.

```
    start f(th) global(theta,tetra);
       theta = th;
       vv    = v(tetra[,1]);
       error = ssq(vv-tetra[,2]);
       return(error);
    finish;

    start v(t) global(theta);
       v = theta[3]*theta[1]/(theta[2]-theta[1])*
               (exp(-theta[1]*t)-exp(-theta[2]*t));
       return(v);
    finish;

    call nlpdd(rc,rx,"f",itheta) blc=con opt=opt tc=tc par=par;
```

Example 11.3. Compartmental Analysis ◆ 373

The parameter estimates, which are shown in Output 11.3.2, are close to those obtained by the first solution.

Output 11.3.2. Second Set of Parameter Estimates for Tetracycline Diffusion

```
                    Optimization Results
                    Parameter Estimates
                                            Gradient
                                           Objective
        N Parameter            Estimate     Function

        1 X1                   0.183025   -0.000003196
        2 X2                   0.434482    0.000002274
        3 X3                   5.995241   -0.000001035

        Value of Objective Function = 0.0356467763
```

Because of the nature of the closed form of the solution, you may want to add an additional constraint to guarantee that $\theta_2 \neq \theta_1$ at any time during the optimization. This will prevent a possible division by 0 or a value near 0 in the execution of the $v(\cdot)$ function. For example, you might add the constraint

$$\theta_2 - \theta_1 \geq 10^{-7}$$

Chemical Kinetics of Pyrolysis of Oil Shale

Pyrolysis is a chemical change effected by the action of heat, and this example considers the pyrolysis of oil shale described in Ziegel and Gorman (1980). Oil shale contains organic material that is bonded to the rock. To extract oil from the rock, heat is applied, and the organic material is decomposed into oil, bitumen, and other by-products. The model is given by

$$\frac{d\gamma_1(t)}{dt} = -(\theta_1 + \theta_4)\gamma_1(t)\iota(t, \theta_5)$$

$$\frac{d\gamma_2(t)}{dt} = [\theta_1\gamma_1(t) - (\theta_2 + \theta_3)\gamma_2(t)]\iota(t, \theta_5)$$

$$\frac{d\gamma_3(t)}{dt} = [\theta_4\gamma_1(t) + \theta_2\gamma_2(t)]\iota(t, \theta_5)$$

with the initial conditions

$$\gamma_1(t) = 100, \quad \gamma_2(t) = 0, \quad \gamma_3(t) = 0$$

A dead time is assumed to exist in the process. That is, no change occurs up to time θ_5. This is controlled by the indicator function $\iota(t, \theta_5)$, which is given by

$$\iota(t, \theta_5) = \begin{cases} 0 & \text{if } t < \theta_5 \\ 1 & \text{if } t \geq \theta_5 \end{cases}$$

where $\theta_5 \geq 0$. Only one of the cases in Ziegel and Gorman (1980) is analyzed in this report, but the others can be handled in a similar manner. The following IML program illustrates how to combine the NLPQN subroutine and the ODE subroutine to estimate the parameters θ_i in this model:

```
data oil ( drop=temp);
   input temp time bitumen oil;
   datalines;
673    5       0.      0.
673    7       2.2     0.
673   10      11.5     0.7
673   15      13.7     7.2
673   20      15.1    11.5
673   25      17.3    15.8
673   30      17.3    20.9
673   40      20.1    26.6
673   50      20.1    32.4
673   60      22.3    38.1
673   80      20.9    43.2
673  100      11.5    49.6
673  120       6.5    51.8
673  150       3.6    54.7
;

proc iml;
   use oil;
   read all into a;

/*****************************************************************/
/* The INS function inserts a value given by FROM into a vector */
/* given by INTO, sorts the result, and posts the global matrix */
/* that can be used to delete the effects of the point FROM.   */
/*****************************************************************/
   start ins(from,into) global(permm);
      in   =  into // from;
      x    =  i(nrow(in));
      permm = inv(x[rank(in),]);
      return(permm*in);
   finish;

   start der(t,x) global(thmtrx,thet);
      y     = thmtrx*x;
      if ( t <= thet[5] )  then y = 0*y;
      return(y);
   finish;

   start jac(t,x) global(thmtrx,thet);
      y     = thmtrx;
      if ( t <= thet[5] )  then y = 0*y;
      return(y);
   finish;

   start f(theta) global(thmtrx,thet,time,h,a,eps,permm);
      thet = theta;
      thmtrx = (-(theta[1]+theta[4]) ||          0           || 0 )//
               (theta[1]             || -(theta[2]+theta[3]) || 0 )//
               (theta[4]             || theta[2]             || 0 );
      t = ins( theta[5],time);
      c = { 100, 0, 0};
```

Example 11.3. Compartmental Analysis ◆ 375

```
          call ode( r1, "der",c , t , h) j="jac" eps=eps;

   /* send the intermediate value to the last column */
      r = (c ||r1) * permm;
      m = r[2:3,(2:nrow(time))];
      mm = m`- a[,2:3];
      call qr(q,r,piv,lindep,mm);
      v = det(r);
      return(abs(v));
   finish;

   opt = {0 2 0 1 };
   tc = repeat(.,1,12);
   tc[1] = 100;
   tc[7] = 1.e-7;
   par = { 1.e-13 . 1.e-10 . . . .};
   con = j(1,5,0.);
   h = {1.e-14 1. 1.e-5};
   time = (0 // a[,1]);
   eps = 1.e-5;
   itheta = { 1.e-3 1.e-3 1.e-3 1.e-3 1.};

   call nlpqn(rc,rx,"f",itheta)  blc=con opt=opt tc=tc par=par;
```

The parameter estimates are shown in Output 11.3.3.

Output 11.3.3. Parameter Estimates for Oil Shale Pyrolysis

```
                    Optimization Results
                    Parameter Estimates
                                           Gradient
                                           Objective
       N Parameter            Estimate      Function

       1 X1                   0.013692     150.14987
       2 X2                   0.012939     248.78071
       3 X3                   0.016303    -144.46645
       4 X4                   0.006638    -318.57862
       5 X5                   1.044177      -3.16737

       Value of Objective Function = 85.597262124
```

Again, compare the solution using the approximation produced by the ODE subroutine to the solution obtained through the closed form of the given differential equation. Impose the following additional constraint to avoid a possible division by 0 when evaluating the function:

$$\theta_2 + \theta_3 - \theta_1 - \theta_4 \geq 10^{-7}$$

The closed form of the solution requires a change in the function $f(\cdot)$. The functions needed as arguments of the ODE routine, namely the der and jac modules, can be removed.

```
   start f(thet) global(time,a);
      do i = 1 to nrow(time);
```

```
         t   = time[i];
         v1  = 100;
         if ( t >= thet[5] ) then
            v1 = 100*ev(t,thet[1],thet[4],thet[5]);
         v2 = 0;
         if ( t >= thet[5] ) then
            v2 = 100*thet[1]/(thet[2]+thet[3]-thet[1]-thet[4])*
                  (ev(t,thet[1],thet[4],thet[5])-
                   ev(t,thet[2],thet[3],thet[5]));
         v3 = 0;
         if ( t >= thet[5] ) then
            v3  = 100*thet[4]/(thet[1]+thet[4])*
              (1. - ev(t,thet[1],thet[4],thet[5])) +
              100*thet[1]*thet[2]/(thet[2]+thet[3]-thet[1]-thet[4])*(
              (1.-ev(t,thet[1],thet[4],thet[5]))/(thet[1]+thet[4]) -
              (1.-ev(t,thet[2],thet[3],thet[5]))/(thet[2]+thet[3])    );
         y = y // (v1 || v2 || v3);
      end;
      mm = y[,2:3]-a[,2:3];
      call qr(q,r,piv,lindep,mm);
      v = det(r);
      return(abs(v));
   finish;

   start ev(t,a,b,c);
      return(exp(-(a+b)*(t-c)));
   finish;

   con     = { 0.  0.  0.  0.   .   . . .,
                .   .   .   .   .    . .,
                -1   1   1  -1   .   1  1.e-7 };
   time    =  a[,1];
   par     = { 1.e-13 . 1.e-10 . . . .};
   itheta  = { 1.e-3 1.e-3 1.e-2 1.e-3 1.};

   call nlpqn(rc,rx,"f",itheta)  blc=con opt=opt tc=tc par=par;
```

The parameter estimates are shown in Output 11.3.4.

Output 11.3.4.   Second Set of Parameter Estimates for Oil Shale Pyrolysis

```
                 Optimization Results
                 Parameter Estimates
                                       Gradient
                                       Objective
      N Parameter          Estimate     Function

      1 X1                 0.017178    -0.005291
      2 X2                 0.008912     0.002413
      3 X3                 0.020007    -0.000520
      4 X4                 0.010494    -0.002890
      5 X5                 7.771534     0.000003217


      Value of Objective Function = 20.689350642
```

Example 11.4. MLEs for Two-Parameter Weibull Distribution ◆ 377

Example 11.4. MLEs for Two-Parameter Weibull Distribution

This example considers a data set given in Lawless (1982). The data are the number of days it took rats painted with a carcinogen to develop carcinoma. The last two observations are censored. Maximum likelihood estimates (MLEs) and confidence intervals for the parameters of the Weibull distribution are computed. In the following code, the data set is given in the vector CARCIN, and the variables P and M give the total number of observations and the number of uncensored observations. The set D represents the indices of the observations.

```
proc iml;
   carcin = { 143   164   188   188   190   192   206
                    209   213   216   220   227   230   234
                    246   265   304   216   244 };
   p = ncol(carcin); m = p - 2;
```

The three-parameter Weibull distribution uses three parameters: a scale parameter, a shape parameter, and a location parameter. This example computes MLEs and corresponding 95% confidence intervals for the scale parameter, σ, and the shape parameter, c, for a constant value of the location parameter, $\theta = 0$. The program can be generalized to estimate all three parameters. Note that Lawless (1982) denotes σ, c, and θ by α, β, and μ, respectively.

The observed likelihood function of the three-parameter Weibull distribution is

$$L(\theta, \sigma, c) = \frac{c^m}{\sigma^m} \prod_{i \in D} \left(\frac{t_i - \theta}{\sigma} \right)^{c-1} \prod_{i=1}^{p} \exp \left\{ - \left(\frac{t_i - \theta}{\sigma} \right)^c \right\}$$

and the log likelihood, $\ell(\theta, \sigma, c) = \log L(\theta, \sigma, c)$, is

$$\ell(\theta, \sigma, c) = m \log c - mc \log \sigma + (c - 1) \sum_{i \in D} \log(t_i - \theta) - \sum_{i=1}^{p} \left(\frac{t_i - \theta}{\sigma} \right)^c$$

The log-likelihood function, $\ell(\theta, \sigma, c)$, for $\theta = 0$ is the objective function to be maximized to obtain the MLEs $(\hat{\sigma}, \hat{c})$:

```
start f_weib2(x) global(carcin,thet);
   /* x[1]=sigma and x[2]=c */
   p = ncol(carcin); m = p - 2;
   sum1 = 0.; sum2 = 0.;
   do i = 1 to p;
      temp = carcin[i] - thet;
      if i <= m then sum1 = sum1 + log(temp);
      sum2 = sum2 + (temp / x[1])##x[2];
   end;
   f = m*log(x[2]) - m*x[2]*log(x[1]) + (x[2]-1)*sum1 - sum2;
   return(f);
finish f_weib2;
```

The derivatives of ℓ with respect to the parameters θ, σ, and c are given in Lawless (1982). The following code specifies a gradient module, which computes $\partial\ell/\partial\sigma$ and $\partial\ell/\partial c$:

```
start g_weib2(x) global(carcin,thet);
   /* x[1]=sigma and x[2]=c */
   p = ncol(carcin); m = p - 2;
   g = j(1,2,0.);
   sum1 = 0.; sum2 = 0.; sum3 = 0.;
   do i = 1 to p;
      temp = carcin[i] - thet;
      if i <= m then sum1 = sum1 + log(temp);
      sum2 = sum2 + (temp / x[1])##x[2];
      sum3 = sum3 + ((temp / x[1])##x[2]) * (log(temp / x[1]));
   end;
   g[1] = -m * x[2] / x[1] + sum2 * x[2] / x[1];
   g[2] = m / x[2] - m * log(x[1]) + sum1 - sum3;
   return(g);
finish g_weib2;
```

The MLEs are computed by maximizing the objective function with the trust-region algorithm in the NLPTR subroutine. The following code specifies starting values for the two parameters, $c = \sigma = 0.5$, and to avoid infeasible values during the optimization process, it imposes lower bounds of $c, \sigma >= 10^{-6}$. The optimal parameter values are saved in the variable XOPT, and the optimal objective function value is saved in the variable FOPT.

```
n = 2; thet = 0.;
x0 = j(1,n,.5);
optn = {1 2};
con = { 1.e-6 1.e-6 ,
          .      .    };
call nlptr(rc,xres,"f_weib2",x0,optn,con,,,,"g_weib2");
/*--- Save result in xopt, fopt ---*/
xopt = xres`; fopt = f_weib2(xopt);
```

The results shown in Output 11.4.1 are the same as those given in Lawless (1982).

Output 11.4.1. Parameter Estimates for Carcinogen Data

```
              Optimization Results
               Parameter Estimates
                                       Gradient
                                       Objective
   N Parameter          Estimate        Function

   1 X1               234.318611     1.3363283E-9
   2 X2                 6.083147    -7.850915E-9


   Value of Objective Function = -88.23273515
```

The following code computes confidence intervals based on the asymptotic normal distribution. These will be compared with the profile-likelihood-based confidence

Example 11.5. Profile-Likelihood-Based Confidence Intervals ◆ 379

intervals computed in the next example. The diagonal of the inverse Hessian (as calculated by the NLPFDD subroutine) is used to calculate the standard error.

```
call nlpfdd(f,g,hes2,"f_weib2",xopt,,"g_weib2");
hin2 = inv(hes2);
/* quantile of normal distribution */
prob = .05;
noqua = probit(1. - prob/2);
stderr = sqrt(abs(vecdiag(hin2)));
xlb = xopt - noqua * stderr;
xub = xopt + noqua * stderr;
print "Normal Distribution Confidence Interval";
print xlb xopt xub;
```

Output 11.4.2. Confidence Interval Based on Normal Distribution

```
       Normal Distribution Confidence Interval


              XLB       XOP2       XUB

        215.41298 234.31861 253.22425
        3.9894574 6.0831471 8.1768368
```

Example 11.5. Profile-Likelihood-Based Confidence Intervals

This example calculates confidence intervals based on the profile likelihood for the parameters estimated in the previous example. The following introduction on profile-likelihood methods is based on the paper of Venzon and Moolgavkar (1988).

Let $\hat{\theta}$ be the maximum likelihood estimate (MLE) of a parameter vector $\theta_0 \in \mathcal{R}^n$ and let $\ell(\theta)$ be the log-likelihood function defined for parameter values $\theta \in \Theta \subset \mathcal{R}^n$.

The profile-likelihood method reduces $\ell(\theta)$ to a function of a single parameter of interest, $\beta = \theta_j$, where $\theta = (\theta_1, \ldots, \theta_j, \ldots, \theta_n)'$, by treating the others as nuisance parameters and maximizing over them. The profile likelihood for β is defined as

$$\tilde{\ell}_j(\beta) = \max_{\theta \in \Theta_j(\beta)} \ell(\theta)$$

where $\Theta_j(\beta) = \{\theta \in \Theta : \theta_j = \beta\}$. Define the complementary parameter set $\omega = (\theta_1, \ldots, \theta_{j-1}, \theta_{j+1}, \ldots, \theta_n)'$ and $\hat{\omega}(\beta)$ as the optimizer of $\tilde{\ell}_j(\beta)$ for each value of β. Of course, the maximum of function $\tilde{\ell}_j(\beta)$ is located at $\beta = \hat{\theta}_j$. The profile-likelihood-based confidence interval for parameter θ_j is defined as

$$\{\beta : \ell(\hat{\theta}) - \tilde{\ell}_j(\beta) \leq \frac{1}{2}q_1(1 - \alpha)\}$$

where $q_1(1 - \alpha)$ is the $(1 - \alpha)^{th}$ quantile of the χ^2 distribution with one degree of freedom. The points (β_l, β_u) are the endpoints of the profile-likelihood-based confidence interval for parameter $\beta = \theta_j$. The points β_l and β_u can be computed

as the solutions of a system of n nonlinear equations $f_i(x)$ in n parameters, where $x = (\beta, \omega)$:

$$\left[\begin{array}{c} \ell(\theta) - \ell^* \\ \frac{\partial \ell}{\partial \omega}(\theta) \end{array} \right] = 0$$

where ℓ^* is the constant threshold $\ell^* = \ell(\hat{\theta}) - \frac{1}{2}q_1(1-\alpha)$. The first of these n equations defines the locations β_l and β_u where the function $\ell(\theta)$ cuts ℓ^*, and the remaining $n-1$ equations define the optimality of the $n-1$ parameters in ω. Jointly, the n equations define the locations β_l and β_u where the function $\tilde{\ell}_j(\beta)$ cuts the constant threshold ℓ^*, which is given by the roots of $\tilde{\ell}_j(\beta) - \ell^*$. Assuming that the two solutions $\{\beta_l, \beta_u\}$ exist (they do not if the quantile $q_1(1-\alpha)$ is too large), this system of n nonlinear equations can be solved by minimizing the sum of squares of the n functions $f_i(\beta, \omega)$:

$$F = \frac{1}{2} \sum_{i=1}^{n} f_i^2(\beta, \omega)$$

For a solution of the system of n nonlinear equations to exist, the minimum value of the convex function F must be zero.

The following code defines the module for the system of $n = 2$ nonlinear equations to be solved:

```
start f_plwei2(x) global(carcin,ipar,lstar);
   /* x[1]=sigma, x[2]=c */
   like = f_weib2(x);
   grad = g_weib2(x);
   grad[ipar] = like - lstar;
   return(grad`);
finish f_plwei2;
```

The following code implements the Levenberg-Marquardt algorithm with the NLPLM subroutine to solve the system of two equations for the left and right endpoints of the interval. The starting point is the optimizer $(\hat{\sigma}, \hat{c})$, as computed in the previous example, moved toward the left or right endpoint of the interval by an initial step (refer to Venzon and Moolgavkar 1988). This forces the algorithm to approach the specified endpoint.

```
/* quantile of chi**2 distribution */
chqua = cinv(1-prob,1); lstar = fopt - .5 * chqua;
optn = {2 0};
do ipar = 1 to 2;
/* Compute initial step: */
/* Choose (alfa,delt) to go in right direction */
/* Venzon & Moolgavkar (1988), p.89 */
   if ipar=1 then ind = 2; else ind = 1;
   delt = - inv(hes2[ind,ind]) * hes2[ind,ipar];
```

Example 11.6. Survival Curve for Interval Censored Data ◆ 381

```
      alfa = - (hes2[ipar,ipar] - delt` * hes2[ind,ipar]);
      if alfa > 0 then alfa = .5 * sqrt(chqua / alfa);
      else do;
         print "Bad alpha";
         alfa = .1 * xopt[ipar];
      end;
      if ipar=1 then delt = 1 || delt;
         else delt = delt || 1;

 /* Get upper end of interval */
      x0 = xopt + (alfa * delt)`;
 /* set lower bound to optimal value */
      con2 = con; con2[1,ipar] = xopt[ipar];
      call nlplm(rc,xres,"f_plwei2",x0,optn,con2);
      f = f_plwei2(xres); s = ssq(f);
      if (s < 1.e-6) then xub[ipar] = xres[ipar];
         else xub[ipar] = .;

 /* Get lower end of interval */
      x0 = xopt - (alfa * delt)`;
 /* reset lower bound and set upper bound to optimal value */
      con2[1,ipar] = con[1,ipar]; con2[2,ipar] = xopt[ipar];
      call nlplm(rc,xres,"f_plwei2",x0,optn,con2);
      f = f_plwei2(xres); s = ssq(f);
      if (s < 1.e-6) then xlb[ipar] = xres[ipar];
         else xlb[ipar] = .;
   end;
   print "Profile-Likelihood Confidence Interval";
   print xlb xopt xub;
```

The results, shown in Output 11.5.1, are close to the results shown in Output 11.4.2.

Output 11.5.1. Confidence Interval Based on Profile Likelihood

```
            Profile-Likelihood Confidence Interval


               XLB       XOP2       XUB

            215.1963 234.31861  255.2157
            4.1344126 6.0831471 8.3063797
```

Example 11.6. Survival Curve for Interval Censored Data

In some studies, subjects are assessed only periodically for outcomes or responses of interest. In such situations, the occurrence times of these events are not observed directly; instead they are known to have occurred within some time interval. The times of occurrence of these events are said to be *interval censored*. A first step in the analysis of these interval censored data is the estimation of the distribution of the event occurrence times.

In a study with n subjects, denote the raw interval censored observations by $\{(L_i, R_i] : 1 \leq i \leq n\}$. For the ith subject, the event occurrence time T_i lies in $(L_i, R_i]$, where L_i is the last assessment time at which there was no evidence of the

event, and R_i is the earliest time when a positive assessment was noted (if it was observed at all). If the event does not occur before the end of the study, R_i is given a value larger than any assessment time recorded in the data.

A set of nonoverlapping time intervals $I_j = (q_j, p_j]$, $1 \leq j \leq m$ is generated over which the survival curve $S(t) = \Pr[T > t]$ is estimated. Refer to Peto (1973) and Turnbull (1976) for details. Assuming the independence of T_i and $(L_i, R_i]$, and also independence across subjects, the likelihood of the data $\{T_i \in (L_i, R_i], 1 \leq i \leq n\}$ can be constructed in terms of the pseudo-parameters $\theta_j = \Pr[T_i \in I_j], 1 \leq i \leq m$. The conditional likelihood of $\theta = (\theta_1, \ldots, \theta_m)$ is

$$L(\theta) = \prod_{i=1}^{n} \left(\sum_{j=1}^{m} x_{ij} \theta_j \right)$$

where x_{ij} is 1 or 0 according to whether I_j is a subset of $(L_i, R_i]$. The maximum likelihood estimates, $\hat{\theta}_j, 1 \leq j \leq m$, yield an estimator $\hat{S}(t)$ of the survival function $S(t)$, which is given by

$$\hat{S}(t) = \begin{cases} 1 & t \leq q_1 \\ \sum_{i=j+1}^{m} \hat{\theta}_i & p_j \leq t \leq q_{j+1}, \quad 1 \leq j \leq m-1 \\ 0 & t \geq p_m \end{cases}$$

$\hat{S}(t)$ remains undefined in the intervals (q_j, p_j) where the function may decrease in an arbitrary way. The asymptotic covariance matrix of $\hat{\theta}$ is obtained by inverting the estimated matrix of second partial derivatives of the negative log likelihood (Peto 1973, Turnbull 1976). You can then compute the standard errors of the survival function estimators by the delta method and approximate the confidence intervals for survival function by using normal distribution theory.

The following code estimates the survival curve for interval censored data. As an illustration, consider an experiment to study the onset of a special kind of palpable tumor in mice. Forty mice exposed to a carcinogen were palpated for the tumor every two weeks. The times to the onset of the tumor are interval censored data. These data are contained in the data set CARCIN. The variable L represents the last time the tumor was not yet detected, and the variable R represents the first time the tumor was palpated. Three mice died tumor free, and one mouse was tumor free by the end of the 48-week experiment. The times to tumor for these four mice were considered right censored, and they were given an R value of 50 weeks.

```
data carcin;
   input id l r @@;
   datalines;
   1   20   22      11   30   32      21   22   24      31   34   36
   2   22   24      12   32   34      22   22   24      32   34   36
   3   26   28      13   32   34      23   28   30      33   36   38
   4   26   28      14   32   34      24   28   30      34   38   40
   5   26   28      15   34   36      25   32   34      35   38   40
```

Example 11.6. Survival Curve for Interval Censored Data ◆ 383

```
    6   26   28      16   36   38      26   32   34      36   42   44
    7   28   30      17   42   44      27   32   34      37   42   44
    8   28   30      18   30   50      28   32   34      38   46   48
    9   30   32      19   34   50      29   32   34      39   28   50
   10   30   32      20   20   22      30   32   34      40   48   50
    ;

proc iml;
   use carcin;
   read all var{l r};
   nobs= nrow(l);
  /***********************************************************
      construct the nonoverlapping intervals (Q,P) and
      determine the number of pseudo-parameters (NPARM)
      ***********************************************************/
   pp= unique(r); npp= ncol(pp);
   qq= unique(l); nqq= ncol(qq);
   q= j(1,npp, .);
   do;
      do i= 1 to npp;
         do j= 1 to nqq;
            if ( qq[j] < pp[i] ) then q[i]= qq[j];
         end;
         if q[i] = qq[nqq] then goto lab1;
      end;
   lab1:
   end;

   if i > npp then nq= npp;
   else            nq= i;
   q= unique(q[1:nq]);
   nparm= ncol(q);
   p= j(1,nparm, .);
   do i= 1 to nparm;
      do j= npp to 1 by -1;
         if ( pp[j] > q[i] ) then p[i]= pp[j];
      end;
   end;

  /***********************************************************
      generate the X-matrix for the likelihood
      ***********************************************************/
   _x= j(nobs, nparm, 0);
   do j= 1 to nparm;
      _x[,j]= choose(l <= q[j]  & p[j] <= r, 1, 0);
   end;

  /***********************************************************
     log-likelihood function (LL)
     ***********************************************************/
   start LL(theta) global(_x,nparm);
      xlt= log(_x * theta`);
      f= xlt[+];
      return(f);
```

```
      finish LL;

/************************************************************
    gradient vector (GRAD)
 ************************************************/
 start GRAD(theta) global(_x,nparm);
    g= j(1,nparm,0);
    tmp= _x # (1 /  (_x * theta`) );
    g= tmp[+,];
    return(g);
 finish GRAD;

/*************************************************************
    estimate the pseudo-parameters using quasi-newton technique
 *************************************************************/
 /* options */
 optn= {1 2};

 /* constraints */
 con= j(3, nparm + 2, .);
 con[1, 1:nparm]= 1.e-6;
 con[2:3, 1:nparm]= 1;
 con[3,nparm + 1]=0;
 con[3,nparm + 2]=1;

 /* initial estimates */
 x0= j(1, nparm, 1/nparm);

 /* call the optimization routine */
 call nlpqn(rc,rx,"LL",x0,optn,con,,,,"GRAD");

/*************************************************************
  survival function estimate (SDF)
 *************************************************/
 tmp1= cusum(rx[nparm:1]);
 sdf= tmp1[nparm-1:1];


/*************************************************************
  covariance matrix of the first nparm-1 pseudo-parameters (SIGMA2)
 *************************************************/
 mm= nparm - 1;
 _x= _x - _x[,nparm] * (j(1, mm, 1) || {0});
 h= j(mm, mm, 0);
 ixtheta= 1 / (_x * ((rx[,1:mm]) || {1})`);
 if _zfreq then
    do i= 1 to nobs;
       rowtmp= ixtheta[i] # _x[i,1:mm];
       h= h + (_freq[i] # (rowtmp` * rowtmp));
    end;
 else do i= 1 to nobs;
    rowtmp= ixtheta[i] # _x[i,1:mm];
    h= h + (rowtmp` * rowtmp);
 end;
```

Example 11.6. Survival Curve for Interval Censored Data ◆ 385

```
 sigma2= inv(h);

 /***********************************************************
   standard errors of the estimated survival curve (SIGMA3)
   ***********************************************************/
 sigma3= j(mm, 1, 0);
 tmp1= sigma3;
 do i= 1 to mm;
    tmp1[i]= 1;
    sigma3[i]= sqrt(tmp1` * sigma2 * tmp1);
 end;

 /***********************************************************
   95% confidence limits for the survival curve (LCL,UCL)
   ***********************************************************/
/* confidence limits */
 tmp1= probit(.975);
 *print tmp1;
 tmp1= tmp1 * sigma3;
 lcl= choose(sdf > tmp1, sdf - tmp1, 0);
 ucl= sdf + tmp1;
 ucl= choose( ucl > 1., 1., ucl);

 /***********************************************************
   print estimates of pseudo-parameters
   ***********************************************************/
 reset center noname;
 q= q`;
 p= p`;
 theta= rx`;
 print ,"Parameter Estimates", ,q[colname={q}] p[colname={p}]
       theta[colname={theta} format=12.7],;

 /***********************************************************
   print survival curve estimates and confidence limits
   ***********************************************************/
 left= {0} // p;
 right= q // p[nparm];
 sdf= {1} // sdf // {0};
 lcl= {.} // lcl //{.};
 ucl= {.} // ucl //{.};
 print , "Survival Curve Estimates and 95% Confidence Intervals", ,
       left[colname={left}] right[colname={right}]
       sdf[colname={estimate} format=12.4]
       lcl[colname={lower} format=12.4]
       ucl[colname={upper} format=12.4];
```

The iteration history produced by the NLPQN subroutine is shown in Output 11.6.1

Output 11.6.1. Iteration History for the NLPQN Subroutine

```
                        Dual Quasi-Newton Optimization


         Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)


                      Parameter Estimates            12
                      Lower Bounds                   12
                      Upper Bounds                   12
                      Linear Constraints              1


                           Optimization Start

Active Constraints         1  Objective Function          -93.3278404
Max Abs Gradient Element    65.361558529




                                      Objective Max Abs
                  Func.   Active   Objective   Function  Gradient  Step
  Iter    Rest    Calls   Constr.  Function    Change    Element   Size   Slope

    1       0       3       1      -88.51201   4.8158    16.6594   0.0256 -305.2
    2       0       4       1      -87.42665   1.0854    10.8769   1.000  -2.157
    3       0       5       1      -87.27408   0.1526     5.4965   1.000  -0.366
    4       0       7       1      -87.17314   0.1009     2.2856   2.000  -0.113
    5       0       8       1      -87.16611   0.00703    0.3444   1.000  -0.0149
    6       0      10       1      -87.16582   0.000287   0.0522   1.001  -0.0006
    7       0      12       1      -87.16581   9.128E-6   0.00691  1.133  -161E-7
    8       0      14       1      -87.16581   1.712E-7   0.00101  1.128  -303E-9

                           Optimization Results

Iterations                          8  Function Calls                    15
Gradient Calls                     11  Active Constraints                 1
Objective Function         -87.16581343  Max Abs Gradient Element   0.0010060788
Slope of Search Direction  -3.033154E-7


NOTE:GCONV convergence criterion satisfied.
NOTE: At least one element of the (projected) gradient is greater than 1e-3.
```

The estimates of the pseudo-parameter for the nonoverlapping intervals are shown in Output 11.6.2.

Output 11.6.2. Estimates for the Probability of Event Occurrence

```
                     Parameter Estimates


              Q         P        THETA

             20        22      0.0499997
             22        24      0.0749988
             26        28      0.0999978
             28        30      0.1033349
             30        32      0.0806014
             32        34      0.2418023
             34        36      0.0873152
             36        38      0.0582119
             38        40      0.0582119
             42        44      0.0873152
             46        48      0.0291055
             48        50      0.0291055
```

The survival curve estimates and confidence intervals are displayed in Output 11.6.3.

Example 11.7. A Two-Equation Maximum Likelihood Problem ◆ 387

Output 11.6.3. Survival Estimates and Confidence Intervals

```
          Survival Curve Estimates and 95% Confidence Intervals


        LEFT       RIGHT      ESTIMATE      LOWER        UPPER

          0          20        1.0000         .            .
         22          22        0.9500       0.8825       1.0000
         24          26        0.8750       0.7725       0.9775
         28          28        0.7750       0.6456       0.9044
         30          30        0.6717       0.5252       0.8182
         32          32        0.5911       0.4363       0.7458
         34          34        0.3493       0.1973       0.5013
         36          36        0.2619       0.1194       0.4045
         38          38        0.2037       0.0720       0.3355
         40          42        0.1455       0.0293       0.2617
         44          46        0.0582       0.0000       0.1361
         48          48        0.0291       0.0000       0.0852
         50          50        0.0000         .            .
```

In this program, the quasi-Newton technique is used to maximize the likelihood function. You can replace the quasi-Newton routine by other optimization routines, such as the NLPNRR subroutine, which performs Newton-Raphson ridge optimization, or the NLPCG subroutine, which performs conjugate gradient optimization. Depending on the number of parameters and the number of observations, these optimization routines do not perform equally well. For survival curve estimation, the quasi-Newton technique seems to work fairly well since the number of parameters to be estimated is usually not too large.

Example 11.7. A Two-Equation Maximum Likelihood Problem

This example and notation are taken from Bard (1974). A two-equation model is used to fit U.S. production data for the years 1909-1949, where z_1 is capital input, z_2 is labor input, z_3 is real output, z_4 is time in years (with 1929 as the origin), and z_5 is the ratio of price of capital services to wage scale.

```
    proc iml;
      z={ 1.33135 0.64629 0.4026 -20 0.24447,
          1.39235 0.66302 0.4084 -19 0.23454,
          1.41640 0.65272 0.4223 -18 0.23206,
          1.48773 0.67318 0.4389 -17 0.22291,
          1.51015 0.67720 0.4605 -16 0.22487,
          1.43385 0.65175 0.4445 -15 0.21879,
          1.48188 0.65570 0.4387 -14 0.23203,
          1.67115 0.71417 0.4999 -13 0.23828,
          1.71327 0.77524 0.5264 -12 0.26571,
          1.76412 0.79465 0.5793 -11 0.23410,
          1.76869 0.71607 0.5492 -10 0.22181,
          1.80776 0.70068 0.5052  -9 0.18157,
          1.54947 0.60764 0.4679  -8 0.22931,
          1.66933 0.67041 0.5283  -7 0.20595,
          1.93377 0.74091 0.5994  -6 0.19472,
```

```
1.95460 0.71336 0.5964   -5 0.17981,
2.11198 0.75159 0.6554   -4 0.18010,
2.26266 0.78838 0.6851   -3 0.16933,
2.33228 0.79600 0.6933   -2 0.16279,
2.43980 0.80788 0.7061   -1 0.16906,
2.58714 0.84547 0.7567    0 0.16239,
2.54865 0.77232 0.6796    1 0.16103,
2.26042 0.67880 0.6136    2 0.14456,
1.91974 0.58529 0.5145    3 0.20079,
1.80000 0.58065 0.5046    4 0.18307,
1.86020 0.62007 0.5711    5 0.18352,
1.88201 0.65575 0.6184    6 0.18847,
1.97018 0.72433 0.7113    7 0.20415,
2.08232 0.76838 0.7461    8 0.18847,
1.94062 0.69806 0.6981    9 0.17800,
1.98646 0.74679 0.7722   10 0.19979,
2.07987 0.79083 0.8557   11 0.21115,
2.28232 0.88462 0.9925   12 0.23453,
2.52779 0.95750 1.0877   13 0.20937,
2.62747 1.00285 1.1834   14 0.19843,
2.61235 0.99329 1.2565   15 0.18898,
2.52320 0.94857 1.2293   16 0.17203,
2.44632 0.97853 1.1889   17 0.18140,
2.56478 1.02591 1.2249   18 0.19431,
2.64588 1.03760 1.2669   19 0.19492,
2.69105 0.99669 1.2708   20 0.17912 };
```

The two-equation model in five parameters $c_1, \ldots, c_5$ is

$$
\begin{aligned}
g_1 &= c_1 10^{c_2 z_4} [c_5 z_1^{-c_4} + (1 - c_5) z_2^{-c_4}]^{-c_3/c_4} - z_3 = 0 \\
g_2 &= [\frac{c_5}{1 - c_5}] \left(\frac{z_1}{z_2}\right)^{-1-c_4} - z_5 = 0
\end{aligned}
$$

where the variables z_1 and z_2 are considered dependent (endogenous) and the variables z_3, z_4, and z_5 are considered independent (exogenous).

Differentiation of the two equations g_1 and g_2 with respect to the endogenous variables z_1 and z_2 yields the Jacobian matrix $\partial g_i / \partial z_j$ for $i = 1, 2$ and $j = 1, 2$, where i corresponds to rows (equations) and j corresponds to endogenous variables (refer to Bard 1974). You must consider parameter sets for which the elements of the Jacobian and the logarithm of the determinant cannot be computed. In such cases, the function module must return a missing value.

```
start fiml(pr) global(z);
   c1 = pr[1]; c2 = pr[2]; c3 = pr[3]; c4 = pr[4]; c5 = pr[5];
   /* 1. Compute Jacobian */
   lndet = 0 ;
   do t= 1 to 41;
```

Example 11.7. A Two-Equation Maximum Likelihood Problem ◆ 389

```
j11 = (-c3/c4) * c1 * 10 ##(c2 * z[t,4]) * (-c4) * c5 *
        z[t,1]##(-c4-1) * (c5 * z[t,1]##(-c4) + (1-c5) *
        z[t,2]##(-c4))##(-c3/c4 -1);
j12 = (-c3/c4) * (-c4) * c1 * 10 ##(c2 * z[t,4]) * (1-c5) *
        z[t,2]##(-c4-1) * (c5 * z[t,1]##(-c4) + (1-c5) *
        z[t,2]##(-c4))##(-c3/c4 -1);
j21 = (-1-c4)*(c5/(1-c5))*z[t,1]##( -2-c4)/ (z[t,2]##(-1-c4));
j22 = (1+c4)*(c5/(1-c5))*z[t,1]##( -1-c4)/ (z[t,2]##(-c4));

j = (j11 || j12 ) // (j21 || j22) ;
if any(j = .) then detj = 0.;
   else detj = det(j);
if abs(detj) < 1.e-30 then do;
   print t detj j;
   return(.);
end;
lndet = lndet + log(abs(detj));
   end;
```

Assuming that the residuals of the two equations are normally distributed, the likelihood is then computed as in Bard (1974). The following code computes the logarithm of the likelihood function:

```
/* 2. Compute Sigma */
 sb = j(2,2,0.);
 do t= 1 to 41;
    eq_g1 = c1 * 10##(c2 * z[t,4]) * (c5*z[t,1]##(-c4)
            + (1-c5)*z[t,2]##(-c4))##(-c3/c4) - z[t,3];
    eq_g2 = (c5/(1-c5)) * (z[t,1] / z[t,2])##(-1-c4) - z[t,5];
    resid = eq_g1 // eq_g2;
    sb = sb + resid * resid`;
 end;
 sb = sb / 41;
/* 3. Compute log L */
 const = 41. * (log(2 * 3.1415) + 1.);
 lnds = 0.5 * 41 * log(det(sb));
 logl = const - lndet + lnds;
 return(logl);
finish fiml;
```

There are potential problems in computing the power and log functions for an unrestricted parameter set. As a result, optimization algorithms that use line search will fail more often than algorithms that restrict the search area. For that reason, the NLPDD subroutine is used in the following code to maximize the log-likelihood function:

```
pr = j(1,5,0.001);
optn = {0 2};
tc = {. . . 0};
call nlpdd(rc, xr,"fiml", pr, optn,,tc);
print "Start" pr, "RC=" rc, "Opt Par" xr;
```

Part of the iteration history is shown in Output 11.7.1.

Output 11.7.1. Iteration History for Two-Equation ML Problem

```
                        Double Dogleg Optimization

        Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

                        Without Parameter Scaling
                Gradient Computed by Finite Differences

                    Parameter Estimates              5

                                            Optimization Start

Active Constraints                       0  Objective Function       909.72691311
Max Abs Gradient Element      41115.729089  Radius                             1




                                    Objective  Max Abs
                     Func.   Active    Obj.    Function   Gradient
     Iter   Rest    Calls   Constr.   Func.     Change    Element    Lambda    Slope

       1     0        2        0     85.24836   824.5     3676.4     711.8    -71032
       2     0        7        0     45.14682   40.1015   3382.0    2881.2    -29.683
       3     0       10        0     43.46797   1.6788    208.4      95.020   -3.348
      35     0       64        0    -110.77858  5.68E-14  0.000111   41.795   -34E-17
      36     1      101        0    -110.77858  5.68E-14  0.000119   4E12     -32E-20
      36     2      145        0    -110.77858         0  0.000119   3.2E16   -46E-24


                        Optimization Results

Iterations                            36   Function Calls                      146
Gradient Calls                        41   Active Constraints                    0
Objective Function          -110.7785811   Max Abs Gradient Element    0.0001186267
Slope of Search Direction   -4.55096E-23   Radius                      3.771173E-19
```

The results are very close to those reported by Bard (1974). Bard also reports different approaches to the same problem that can lead to very different MLEs.

Output 11.7.2. Parameter Estimates

```
                        Parameter Estimates
                                            Gradient
                                            Objective
                N Parameter      Estimate    Function

                1 X1             0.583884   -0.000004817
                2 X2             0.005882    0.000011377
                3 X3             1.362817   -0.000003229
                4 X4             0.475091   -0.000018103
                5 X5             0.447072    0.000119

                Value of Objective Function = -110.7785811
```

Example 11.8. Time-Optimal Heat Conduction ♦ 391

Example 11.8. Time-Optimal Heat Conduction

This example illustrates a nontrivial application of the NLPQN algorithm that requires nonlinear constraints, which are specified by the *nlc* module. The example is listed as problem 91 in Hock & Schittkowski (1981). The problem describes a time-optimal heating process minimizing the simple objective function

$$f(x) = \sum_{j=1}^{n} x_j^2$$

subjected to a rather difficult inequality constraint:

$$c(x) = 10^{-4} - h(x) \geq 0$$

where $h(x)$ is defined as

$$
\begin{aligned}
h(x) &= \int_0^1 \left(\sum_{i=1}^{30} \alpha_i(s)\rho_i(x) - k_0(s) \right)^2 ds \\
\alpha_i(s) &= \mu_i^2 A_i \cos(\mu_i s) \\
\rho_i(x) &= -\mu_i^2 \left[\exp\left(-\mu_i^2 \sum_{j=1}^{n} x_j^2 \right) - 2\exp\left(-\mu_i^2 \sum_{j=2}^{n} x_j^2 \right) + \cdots \right. \\
&\qquad \left. + (-1)^{n-1} 2\exp\left(-\mu_i^2 x_n^2 \right) + (-1)^n \right] \\
k_0(s) &= 0.5(1 - s^2) \\
A_i &= \frac{2\sin\mu_i}{\mu_i + \sin\mu_i \cos\mu_i}, \\
\mu &= (\mu_1, \ldots, \mu_{30})', \quad \text{where } \mu_i \tan(\mu_i) = 1
\end{aligned}
$$

The gradient of the objective function f, $g(x) = 2x$, is easily supplied to the NLPQN subroutine. However, the analytical derivatives of the constraint are not used; instead, finite difference derivatives are computed.

In the following code, the vector MU represents the first 30 positive values μ_i that satisfy $\mu_i \tan(\mu_i) = 1$:

```
proc iml;
   mu = { 8.6033358901938E-01 ,    3.4256184594817E+00 ,
          6.4372981791719E+00 ,    9.5293344053619E+00 ,
          1.2645287223856E+01 ,    1.5771284874815E+01 ,
          1.8902409956860E+01 ,    2.2036496727938E+01 ,
          2.5172446326646E+01 ,    2.8309642854452E+01 ,
          3.1447714637546E+01 ,    3.4586424215288E+01 ,
          3.7725612827776E+01 ,    4.0865170330488E+01 ,
          4.4005017920830E+01 ,    4.7145097736761E+01 ,
```

```
             5.0285366337773E+01 ,     5.3425790477394E+01 ,
             5.6566344279821E+01 ,     5.9707007305335E+01 ,
             6.2847763194454E+01 ,     6.5988598698490E+01 ,
             6.9129502973895E+01 ,     7.2270467060309E+01 ,
             7.5411483488848E+01 ,     7.8552545984243E+01 ,
             8.1693649235601E+01 ,     8.4834788718042E+01 ,
             8.7975960552493E+01 ,     9.1117161394464E+01  };
```

The vector $A = (A_1, \ldots, A_{30})'$ depends only on μ and is computed only once, before the optimization starts:

```
nmu  = nrow(mu);
a = j(1,nmu,0.);
do i = 1 to nmu;
   a[i] = 2*sin(mu[i]) / (mu[i] + sin(mu[i])*cos(mu[i]));
end;
```

The constraint is implemented with the QUAD subroutine, which performs numerical integration of scalar functions in one dimension. The subroutine calls the module fquad that supplies the integrand for $h(x)$. For details on the QUAD call, see the "QUAD Call" section on page 851.

```
/* This is the integrand used in h(x) */
 start fquad(s) global(mu,rho);
    z = (rho * cos(s*mu) - 0.5*(1. - s##2))##2;
    return(z);
 finish;

/* Obtain nonlinear constraint h(x) */
 start h(x) global(n,nmu,mu,a,rho);
    xx = x##2;
    do i= n-1 to 1 by -1;
       xx[i] = xx[i+1] + xx[i];
    end;
    rho = j(1,nmu,0.);
    do i=1 to nmu;
       mu2 = mu[i]##2;
       sum = 0; t1n = -1.;
       do j=2 to n;
          t1n = -t1n;
          sum = sum + t1n * exp(-mu2*xx[j]);
       end;
       sum = -2*sum + exp(-mu2*xx[1]) + t1n;
       rho[i] = -a[i] * sum;
    end;
    aint = do(0,1,.5);
    call quad(z,"fquad",aint) eps=1.e-10;
    v = sum(z);
    return(v);
 finish;
```

Example 11.8. Time-Optimal Heat Conduction ◆ 393

The modules for the objective function, its gradient, and the constraint $c(x) \geq 0$ are given in the following code:

```
/* Define modules for NLPQN call: f, g, and c */
 start F_HS88(x);
    f = x * x`;
    return(f);
 finish F_HS88;

 start G_HS88(x);
    g = 2 * x;
    return(g);
 finish G_HS88;

 start C_HS88(x);
    c = 1.e-4 - h(x);
    return(c);
 finish C_HS88;
```

The number of constraints returned by the *"nlc"* module is defined by $opt[10] = 1$. The ABSGTOL termination criterion (maximum absolute value of the gradient of the Lagrange function) is set by $tc[6] = 1\mathrm{E}{-4}$.

```
print 'Hock & Schittkowski Problem #91 (1981) n=5, INSTEP=1';
opt  = j(1,10,.);
opt[2]=3;
opt[10]=1;
tc   = j(1,12,.);
tc[6]=1.e-4;
x0 = {.5 .5 .5 .5 .5};
n = ncol(x0);
call nlpqn(rc,rx,"F_HS88",x0,opt,,tc) grd="G_HS88" nlc="C_HS88";
```

Part of the iteration history and the parameter estimates are shown in

Output 11.8.1. Iteration History and Parameter Estimates

```
                      Dual Quasi-Newton Optimization
               Modified VMCWD Algorithm of Powell (1978, 1982)

          Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
                  Lagrange Multiplier Update of Powell(1982)
         Jacobian Nonlinear Constraints Computed by Finite Differences


                   Parameter Estimates                 5
                   Nonlinear Constraints               1


                                    Optimization Start

Objective Function                     1.25  Max Constr. Violation  0.0952775105
Max Grad of the Lagran Func    1.1433393372



                                                    Maximum
                                                    Gradient
                                                    Element
                                 Max.      Pred.      of the
                  Func.  Obj.   Constr.    Func.     Step   Lagrange
Iter   Rest  Calls  Func.   Viol.     Red.      Size   Function

 1      0    3    0.81165  0.0869    1.7562    0.100   1.325
 2      0    4    0.18232  0.1175    0.6220    1.000   1.207
 3*     0    5    0.34567  0.0690    0.9321    1.000   0.639
 4      0    6    0.77700  0.0132    0.3498    1.000   1.329
 .
 .
 .
21      0   30    1.36266  8.02E-12  1.079E-6  1.000   0.00009

                            Optimization Results

Iterations                 21  Function Calls                    31
Grad. Calls                23  Active Constraints                 1
Obj. Func.      1.3626568064  Max. Constr. Viol.       8.017286E-12
Max. Proj. Grad. 0.000096451  Value Lagrange Function    1.3626568149
Max. Grad. of the Lagran Func    0.0000887635  Slope  -1.079452E-6

NOTE:  ABSGCONV convergence criterion satisfied.

                         Optimization Results
                         Parameter Estimates

                                    Gradient        Gradient
                                    Objective       Lagrange
  N Parameter         Estimate      Function        Function

  1 X1                0.860296      1.720593      0.000030988
  2 X2               -0.000002262  -0.000004524  -0.000015387
  3 X3                0.643468      1.286936      0.000021570
  4 X4               -0.456614     -0.913227      0.000088763
  5 X5                0.000000904   0.000001807   0.000077409

          Value of Objective Function = 1.3626568064
          Value of Lagrange Function  = 1.3626568149
```

Problems 88 to 92 of Hock and Schittkowski (1981) specify the same optimization problem for $n = 2$ to $n = 6$. You can solve any of these problems with the preceding code by submitting a vector of length n as the initial estimate, x_0.

References

Abramowitz, M. and Stegun, I.A. (1972), *Handbook of Mathematical Functions*, New York: Dover Publications, Inc.

Al-Baali, M. and Fletcher, R. (1985), "Variational Methods for Nonlinear Least Squares," *J. Oper. Res. Soc.*, 36, 405–421.

Al-Baali, M. and Fletcher, R. (1986), "An Efficient Line Search for Nonlinear Least Squares," *Journal of Optimization Theory and Applications*, 48, 359–377.

Anderson, B.D.O. and Moore, J.B. (1979), *Optimal Filtering*, New Jersey: Prentice-Hall.

Bard, Y. (1974), *Nonlinear Parameter Estimation*, New York: Academic Press.

Bates, D.M. and Watts, D.G. (1988), *Nonlinear Regression Analysis and Its Applications*, New York: John Wiley & Sons, Inc.

Beale, E.M.L. (1972), "A Derivation of Conjugate Gradients," *Numerical Methods for Nonlinear Optimization*, ed. F. A. Lootsma, London: Academic Press.

Betts, J. T. (1977), "An Accelerated Multiplier Method for Nonlinear Programming," *Journal of Optimization Theory and Applications*, 21, 137–174.

Bracken, J. and McCormick, G.P. (1968), *Selected Applications of Nonlinear Programming*, New York: John Wiley & Sons, Inc.

Chamberlain, R.M.; Powell, M.J.D.; Lemarechal, C.; and Pedersen, H.C. (1982), "The Watchdog Technique for Forcing Convergence in Algorithms for Constrained Optimization," *Mathematical Programming*, 16, 1–17.

De Jong, P. (1988), "The Likelihood for a State Space Model," *Biometrika*, 75, 165–169.

Dennis, J.E.; Gay, D.M.; and Welsch, R.E. (1981), "An Adaptive Nonlinear Least-Squares Algorithm," *ACM Trans. Math. Software*, 7, 348–368.

Dennis, J.E. and Mei, H.H.W. (1979), "Two New Unconstrained Optimization Algorithms which Use Function and Gradient Values," *J. Optim. Theory Appl.*, 28, 453–482.

Dennis, J.E. and Schnabel, R.B. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, New Jersey: Prentice-Hall.

Eskow, E. and Schnabel, R.B. (1991), "Algorithm 695: Software for a New Modified Cholesky Factorization," *ACM Trans. Math. Software*, 17, 306–312.

Fletcher, R. (1987), *Practical Methods of Optimization*, Second Ed., Chichester: John Wiley & Sons, Inc.

Fletcher, R. and Powell, M.J.D. (1963), "A Rapidly Convergent Descent Method for Minimization," *Computer Journal*, 6, 163–168.

Fletcher, R. and Xu, C. (1987), "Hybrid Methods for Nonlinear Least Squares," *Journal of Numerical Analysis*, 7, 371–389.

Gay, D.M. (1983), "Subroutines for Unconstrained Minimization," *ACM Trans. Math. Software*, 9, 503–524.

George, J.A. and Liu, J.W. (1981), *Computer Solution of Large Sparse Positive Definite Systems*, New Jersey: Prentice-Hall.

Gill, E.P.; Murray, W.; and Wright, M.H. (1981), *Practical Optimization*, London: Academic Press.

Gill, E.P.; Murray, W.; Saunders, M.A.; and Wright, M.H. (1983), "Computing Forward-Difference Intervals for Numerical Optimization," *SIAM J. Sci. Stat. Comput.*, 4, 310–321.

Gill, E.P.; Murray, W.; Saunders, M.A.; and Wright, M.H. (1984), "Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints," *ACM Trans. Math. Software*, 10, 282–298.

Goldfeld, S.M.; Quandt, R.E.; and Trotter, H.F. (1966), "Maximisation by Quadratic Hill-Climbing," *Econometrica*, 34, 541–551.

Hartmann, W. (1991), *The NLP Procedure: Extended User's Guide*, Releases 6.08 and 6.10, Cary, NC: SAS Institute Inc.

Hock, W. and Schittkowski, K. (1981), "Test Examples for Nonlinear Programming Codes," *Lecture Notes in Economics and Mathematical Systems 187*, New York: Springer-Verlag.

Jennrich, R.I. and Sampson, P.F. (1968), "Application of Stepwise Regression to Nonlinear Estimation," *Technometrics*, 10, 63–72.

Lawless, J.F. (1982), *Statistical Models and Methods for Lifetime Data*, New York: John Wiley & Sons, Inc.

Liebman, J.; Lasdon, L.; Schrage, L.; and Waren, A. (1986), *Modeling and Optimization with GINO*, California: The Scientific Press.

Lindström, P. and Wedin, P.A. (1984), "A New Linesearch Algorithm for Nonlinear Least-Squares Problems," *Mathematical Programming*, 29, 268–296.

Lütkepohl, H. (1991), *Introduction to Multiple Time Series Analysis*, Berlin: Springer-Verlag.

Moré, J.J. (1978), "The Levenberg-Marquardt Algorithm: Implementation and Theory," *Lecture Notes in Mathematics 630*, ed. G.A. Watson, New York: Springer-Verlag, 105–116.

Moré, J.J.; Garbow, B.S.; and Hillstrom, K.E. (1981), "Testing Unconstrained Optimization Software," *ACM Trans. Math. Software*, 7, 17–41.

Moré, J.J. and Sorensen, D.C. (1983), "Computing a Trust-Region Step," *SIAM J. Sci. Stat. Comput.*, 4, 553–572.

Moré, J.J. and Wright, S.J. (1993), *Optimization Software Guide*, Philadelphia: SIAM.

Murtagh, B.A. and Saunders, M.A. (1983), *MINOS 5.0 User's Guide*; Technical Report SOL 83-20, Stanford University.

Nelder, J.A. and Mead, R. (1965), "A Simplex Method for Function Minimization," *Computer Journal*, 7, 308–313.

Peto, R. (1973), "Experimental Survival Curves for Interval-Censored Data," *Appl. Statist*, 22, 86–91.

Polak, E. (1971), *Computational Methods in Optimization*, New York, San Francisco, London: Academic Press, Inc.

Powell, J.M.D. (1977), "Restart Procedures for the Conjugate Gradient Method," *Mathematical Programming*, 12, 241–254.

Powell, J.M.D. (1978a), "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations," *Numerical Analysis, Dundee 1977, Lecture Notes in Mathematics 630*, ed. G.A. Watson, Berlin: Springer-Verlag, 144–175.

Powell, J.M.D. (1978b), "Algorithms for Nonlinear Constraints that use Lagrangian Functions," *Mathematical Programming*, 14, 224–248.

Powell, J.M.D. (1982a), "Extensions to Subroutine VF02AD," *Systems Modeling and Optimization, Lecture Notes In Control and Information Sciences 38*, eds. R.F. Drenick and F. Kozin, Berlin: Springer-Verlag, 529–538.

Powell, J.M.D. (1982b), "VMCWD: A Fortran Subroutine for Constrained Optimization," *DAMTP 1982/NA4*, Cambridge, England.

Powell, J.M.D. (1992), "A Direct Search Optimization Method that Models the Objective and Constraint Functions by Linear Interpolation," *DAMTP/NA5*, Cambridge, England.

Rosenbrock, H.H. (1960), "An Automatic Method for Finding the Greatest or Least Value of a Function," *Computer Journal*, 3, 175–184.

Schittkowski, K. (1978), "An Adaptive Precision Method for the Numerical Solution of Constrained Optimization Problems Applied to a Time-Optimal Heating Process," *Proceedings of the 8th IFIP Conference on Optimization Techniques*, Heidelberg, New York: Springer-Verlag.

Schittkowski, K. (1987), *More Test Examples for Nonlinear Programming Codes*, *Lecture Notes in Economics and Mathematical Systems 282*, Berlin, Heidelberg: Springer-Verlag.

Schittkowski, K. and Stoer, J. (1979), "A Factorization Method for the Solution of Constrained Linear Least Squares Problems Allowing Subsequent Data Changes," *Numer. Math.*, 31, 431–463.

Turnbull, B.W. (1976), "The Empirical Distribution Function from Arbitrarily Grouped, Censored and Truncated Data," *J. Royal Statist. Soc. Ser. B*, 38, 290–295.

Venzon, D.J. and Moolgavkar, S.H. (1988), "A Method for Computing Profile-Likelihood-Based Confidence Intervals," *Applied Statistics*, 37, 87–94.

Wedin, P.A. and Lindström, P. (1987), *Methods and Software for Nonlinear Least Squares Problems*, University of Umea, Report No. UMINF 133.87.

Ziegel, E. R. and J. W. Gorman (1980), "Kinetic Modelling with Multipurpose Data," *Technometrics*, 27, 352–357.

Chapter 12
Graphics Examples

Chapter Contents

Chapter 12
Graphics Examples

Overview

SAS/IML software provides you with a powerful set of graphics commands (called graphics primitives) from which to create customized displays. Several basic commands are GDRAW (for drawing a line), GPOINT (for plotting points), and GPOLY (for drawing a polygon). With each primitive, you can associate a set of attributes such as color or line style.

In this chapter you learn about

- plotting simple two-dimensional plots
- naming and saving a graph
- changing attributes such as color and line style
- specifying the location and scale of your graph
- adding axes and text

SAS/IML graphics commands depend on the libraries and device drivers distributed with SAS/GRAPH software, and they do not work unless you have SAS/GRAPH software.

An Introductory Graph

Suppose that you have data for ACME Corporation's stock price and you want a simple PRICE $\times$ DAY graph to see the overall trend of the stock's price. The data are as follows.

Day	Price
0	43.75
5	48.00
10	59.75
15	75.5
20	59.75
25	71.50
30	70.575
35	61.125
40	79.50
45	72.375
50	67.00
55	54.125
60	58.750
65	43.625
70	47.125
75	45.50

To graph a scatter plot of these points, enter the following statements. These statements generate Figure 12.1.

```
proc iml;                           /* invoke IML       */
   call gstart;                     /* start graphics   */
   xbox={0 100 100 0};
   ybox={0 0 100 100};
   day=do(0,75,5);                  /* initialize day   */
   price={43.75,48,59.75,75.5,      /* initialize price */
        59.75,71.5,70.575,
        61.125,79.5,72.375,67,
        54.125,58.75,43.625,
        47.125,45.50};
   call gopen;                      /* start new graph      */
   call gpoly(xbox,ybox);       /* draw a box around plot */
   call gpoint(day,price);          /* plot the points    */
   call gshow;                      /* display the graph */
```
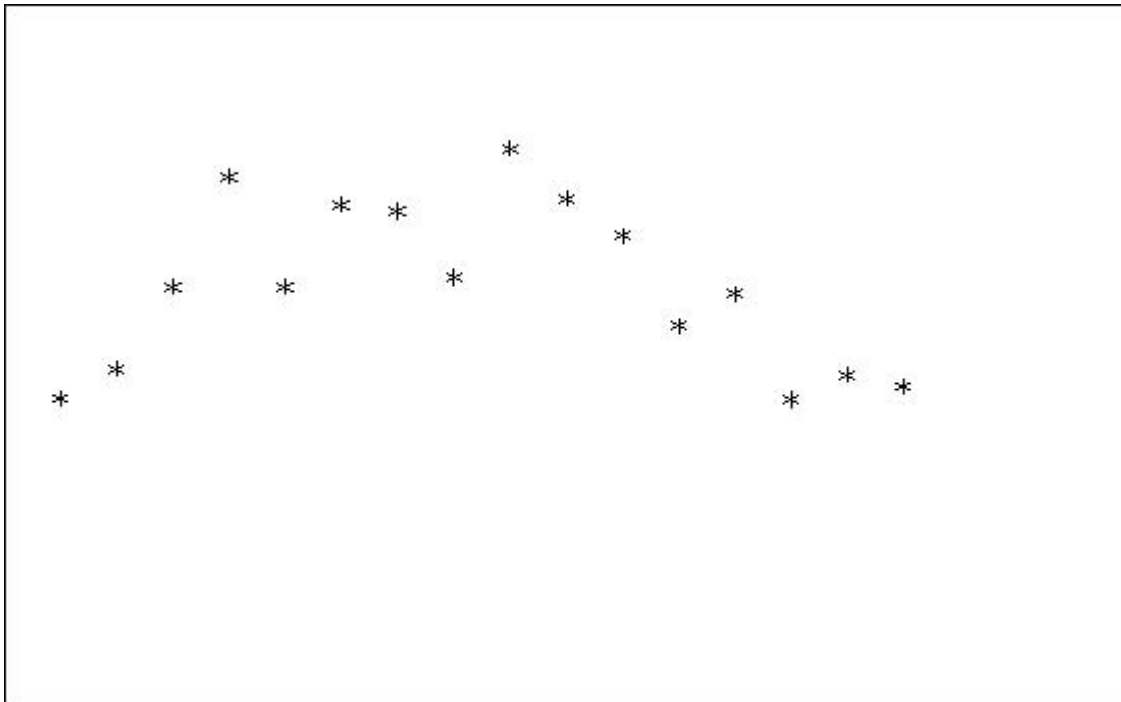
Figure 12.1. Scatter plot

Note that the GSTART statement initializes the graphics session. It usually needs to be called only once. Next, you enter the data matrices. Then you open a graphics segment (that is, begin a new graph) with the GOPEN command. The GPOINT command draws the scatter plot of points of DAY versus PRICE. The GSHOW command displays the graph.

Notice also that, for this example, the x coordinate of the data is DAY and that $0 \leq$ DAY ≤ 100. The y coordinate is PRICE, which ranges from $0 \leq$ PRICE ≤ 100. For this example, the ranges are this way because the IML default ranges are from 0 to 100 on both the x and y axes. Later on you learn how to change the default ranges for the axes with the GWINDOW statement so that you can handle data with any range of values.

Of course, this graph is quite simple. By the end of this chapter, you will know how to add axes and titles, to scale axes, and to connect the points with lines.

Details

Graphics Segments

A graph is saved in what is called a graphics segment. A *graphics segment* is simply a collection of primitives and their associated attributes that creates a graph.

Each time you create a new segment, it is named and stored in a SAS graphics catalog called WORK.GSEG. If you want to store your graphics segments in a permanent SAS catalog, do this with options to the GSTART call. You can name the segments yourself in the GOPEN statement, or you can let the IML procedure automatically

generate a segment name. In this way, graphics segments that are used several times can be included in subsequent graphs by using the GINCLUDE command with the segment name. You can also manage and replay a segment using the GREPLAY procedure as well as replay it in another IML session with the GSHOW command.

To name a segment, include the name as an argument to the GOPEN statement. For example, to begin a new segment and name it STOCK1, use the statement

```
call gopen("stock1");
```

For more information about SAS catalogs and graphics, refer to the chapter on graphics in *SAS/GRAPH Software: Reference*.

Segment Attributes

A set of attributes is initialized for each graphics segment. These attributes are color, line style, line thickness, fill pattern, font, character height, and aspect ratio. You can change any of these attributes for a graphics segment by using the GSET command. Some IML graphics commands take optional attribute arguments. The values of these arguments affect only the graphics output associated with the call.

The IML graphics subsystem uses the same conventions that SAS/GRAPH software uses in setting the default attributes. It also uses the options set in the GOPTIONS statement when applicable. The SAS/IML default values for the GSET command are given by their corresponding GOPTIONS default values. To change the default, you need to issue a GOPTIONS statement. The GOPTIONS statement can also be used to set graphics options not available through the GSET command (for example, the ROTATE option).

For more information about GOPTIONS, refer to the chapter on the GOPTIONS statement in *SAS/GRAPH Software: Reference*.

Coordinate Systems

Each IML graph is associated with two independent cartesian coordinate systems, a *world coordinate system* and a *normalized coordinate system*.

Understanding World Coordinates

The *world coordinate system* is the coordinate system defined by your data. Because these coordinates help define objects in the data's two-dimensional world, these are referred to as *world coordinates*. For example, suppose that you have a data set containing heights and weights and that you are interested in plotting height versus weight. Your data induces a world coordinate system in which each point (x, y) represents a pair of data values (*height,weight*). The world could be defined by the observed ranges of heights and weights, or it could be enlarged to include a range of all reasonable values for heights and weights.

Now consider a more realistic example of the stock price data for ACME Corporation. Suppose that the stock price data were actually the year end prices of ACME stock for the years 1971 through 1986, as follows.

```
YEAR    PRICE
  71    123.75
  72    128.00
  73    139.75
  74    155.50
  75    139.75
  76    151.50
  77    150.375
  78    149.125
  79    159.50
  80    152.375
  81    147.00
  82    134.125
  83    138.75
  84    123.625
  85    127.125
  86    125.500
```

The actual range of YEAR is from 71 to 86, and the range of PRICE is from $123.625 to $159.50. These are the ranges in world coordinate space for the stock data. Of course, you could say that the range for PRICE could start at $0 and range upwards to, for example, $200. Or, if you were interested only in prices during the 80s, you could say the range for PRICE is from $123.625 to $152.375. As you see, it all depends on how you want to define your world.

Figure 12.2 shows a graph of the stock data with the world defined as the actual data given. The corners of the rectangle give the actual boundaries for this data.
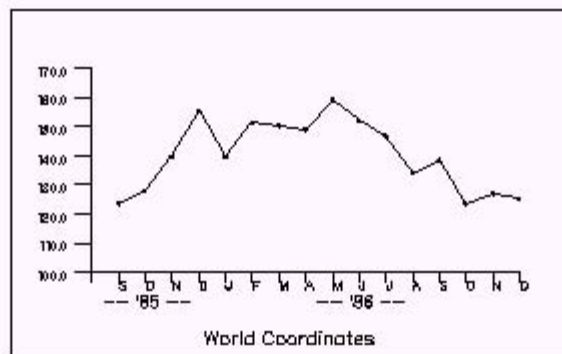
Figure 12.2. World Coordinates

Understanding Normalized Coordinates

The *normalized coordinate system* is defined relative to your display device, usually a monitor or plotter. It is always defined with points varying between (0,0) and (100,100), where (0,0) refers to the lower left corner and (100,100) refers to the upper right corner.

In summary,

- the world coordinate system is defined relative to your data
- the normalized coordinate system is defined relative to the display device

Figure 12.3 shows the ACME stock data in terms of normalized coordinates. There is a natural mathematical relationship between each point in world and normalized coordinates. The normalized device coordinate system is mapped to the device display area so that (0,0), the lower left corner, corresponds to (71, 123.625) in world coordinates, and (100,100), the upper right corner, corresponds to (86,159.5) in world coordinates.
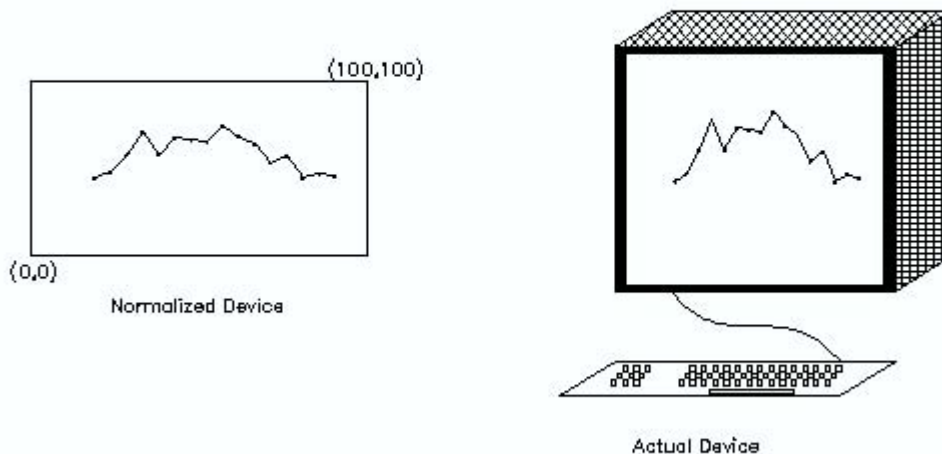
Figure 12.3. Normalized Coordinates

Windows and Viewports

A *window* defines a rectangular area in world coordinates. You define a window with a GWINDOW statement. You can define the window to be larger than, the same size as, or smaller than the actual range of data values, depending on whether you want to show all of the data or only part of the data.

A *viewport* defines in normalized coordinates a rectangular area on the display device where the image of the data appears. You define a viewport with the GPORT command. You can have your graph take up the entire display device or show it in only a portion, say the upper right part.

Mapping Windows to Viewports

A *window* and a *viewport* are related by the linear transformation that maps the window onto the viewport. A line segment in the window is mapped to a line segment in the viewport such that the relative positions are preserved.

You don't have to display all of your data in a graph. In Figure 12.4, the graph on the left displays all of the ACME stock data, and the graph on the right displays only a part of the data. Suppose that you wanted to graph only the last ten years of the stock data, say from 1977 to 1986. You would want to define a window where the YEAR axis ranges from 77 to 86, while the PRICE axis could range from 120 to 160. Figure 12.4 shows stock prices in a window defined for data from 1977 to 1986 along the horizontal direction and from 120 to 160 along the vertical direction. The window is mapped to a viewport defined by the points (20,20) and (70,60). The appropriate GPORT and GWINDOW specifications are as follows.

```
call gwindow({77 120, 86 160});
call gport({20 20, 70 60});
```

The window, in effect, defines the portion of the graph that is to be displayed in world coordinates, and the viewport specifies the area on the device on which the image is to appear.
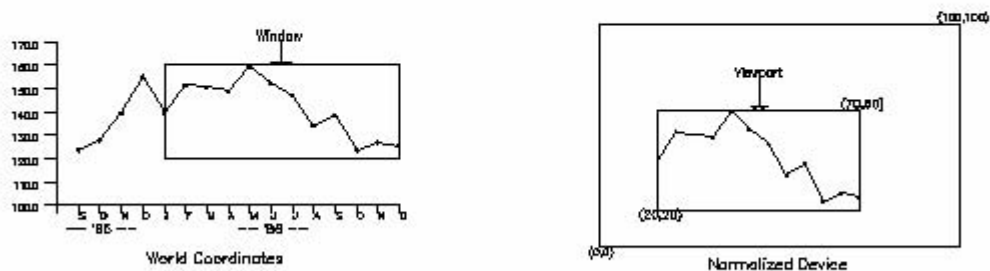
Figure 12.4.   Window to Viewport Mapping

Understanding Windows

Because the default world coordinate system ranges from (0,0) to (100,100), you usually need to define a *window* in order to set the world coordinates corresponding to your data. A window specifies which part of the data in world coordinate space is to be shown. Sometimes you want all of the data shown; other times, you want to show only part of the data.

A window is defined by an array of four numbers, which define a rectangular area. You define this area by specifying the *world coordinates* of the lower left and upper right corners in the GWINDOW statement, which has the general form

CALL GWINDOW(*minimum-x minimum-y maximum-x maximum-y* **);**

The argument can be either a matrix or a literal. The order of the elements is important. The array of coordinates can be a 2×2, 1×4, or $4 \times 1 matrix$. These coordinates can be specified as matrix literals or as the name of a numeric matrix containing the coordinates. If you do not define a window, the default is to assume both x and y range between 0 and 100.

In summary, a window

- defines the portion of the graph that appears in the viewport
- is a rectangular area
- is defined by an array of four numbers
- is defined in world coordinates
- scales the data relative to world coordinates

In the previous example, the variable YEAR ranges from 1971 to 1986, while PRICE ranges from 123.625 to 159.50. Because the data do not fit nicely into the default, you want to define a window that reflects the ranges of the variables YEAR and PRICE. To draw the graph of this data to scale, you can let the YEAR axis range from 70 to 87 and the PRICE axis range from 100 to 200. Use the following statements to draw the graph, shown in Figure 12.5.

```
call gstart;
xbox={0 100 100 0};
ybox={0 0 100 100};
call gopen("stocks1");        /* begin new graph STOCKS1 */
call gset("height", 2.0);
year=do(71,86,1);                      /* initialize YEAR */
price={123.75 128.00 139.75         /* initialize PRICE */
       155.50 139.750 151.500
       150.375 149.125 159.500
       152.375 147.000 134.125
       138.750 123.625 127.125
       125.50};
call gwindow({70 100 87 200});              /* define window */
call gpoint(year,price,"diamond","green"); /* graph the points */
call gdraw(year,price,1,"green");          /* connect points */
call gshow;                                /* show the graph */
```
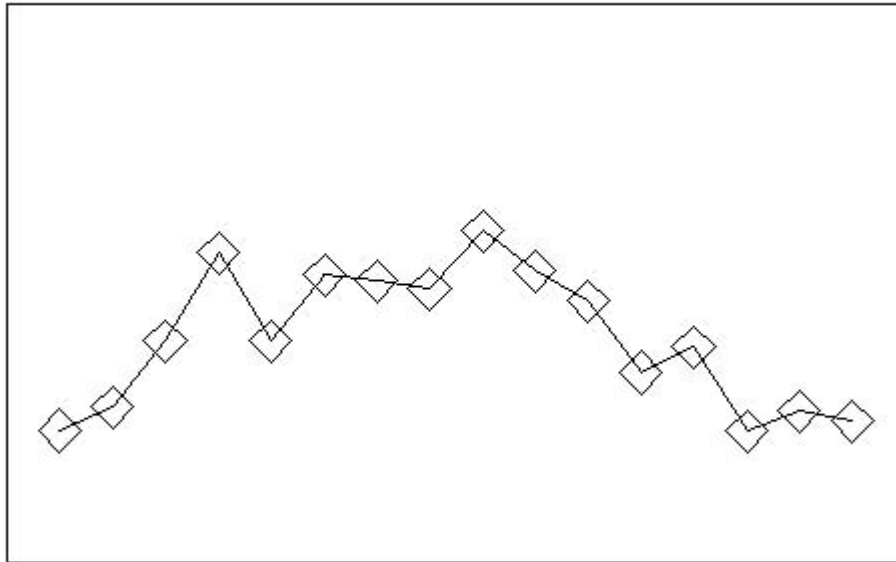
Figure 12.5. Stock Data

In this example, you perform several steps that you did not do with the previous graph:

- You associate the name STOCKS1 with this graphics segment in the GOPEN command.

- You define a window that reflects the actual ranges of the data with a GWINDOW command.

- You associate a plotting symbol, the diamond, and the color green with the GPOINT command.

- You connect the points with line segments with the GDRAW command. The GDRAW command requests that the line segments be drawn in style 1 and be green.

Understanding Viewports

A *viewport* specifies a rectangular area on the display device where the graph appears. You define this area by specifying the *normalized* coordinates, the lower left corner and the upper right corner, in the GPORT statement, which has the general form

CALL GPORT(*minimum-x minimum-y maximum-x maximum-y* **);**

The argument can be either a matrix or a literal. Note that both x and y must range between 0 and 100. As with the GWINDOW specification, you can give the coordinates either as a matrix literal enclosed in braces or as the name of a numeric matrix containing the coordinates. The array can be a 2×2, 1×4, or 4×1 matrix. If you do not define a viewport, the default is to span the entire display device.

In summary, a viewport

- specifies where the image appears on the display
- is a rectangular area
- is specified by an array of four numbers
- is defined in normalized coordinates
- scales the data relative to the shape of the viewport

To display the stock price data in a smaller area on the display device, you must define a viewport. While you are at it, add some text to the graph. You can use the graph that you created and named STOCKS1 in this new graph. The following statements create the graph shown in Figure 12.6.

```
     /* module centers text strings */
start gscenter(x,y,str);
   call gstrlen(len,str);                  /* find string length */
   call gscript(x-len/2,y,str);                /* print text */
finish gscenter;

call gopen("stocks2");                   /* open a new segment */
call gset("font","swiss");               /* set character font */
call gpoly(xbox,ybox);                   /* draw a border       */
call gwindow({70 100,87 200});           /* define a window     */
call gport({15 15,85 85});               /* define a viewport   */
call ginclude("stocks1");            /* include segment STOCKS1 */
call gxaxis({70 100},17,18, ,            /* draw x-axis          */
            ,"2.",1.5);
call gyaxis({70 100},100,11, ,                 /* draw y-axis */
            ,"dollar5.",1.5);
call gset("height",2.0);                 /* set character height */
call gtext(77,89,"Year");                /* print horizontal text */
call gvtext(68,200,"Price");             /* print vertical text   */
call gscenter(79,210,"ACME Stock Data");     /* print title  */
call gshow;
```
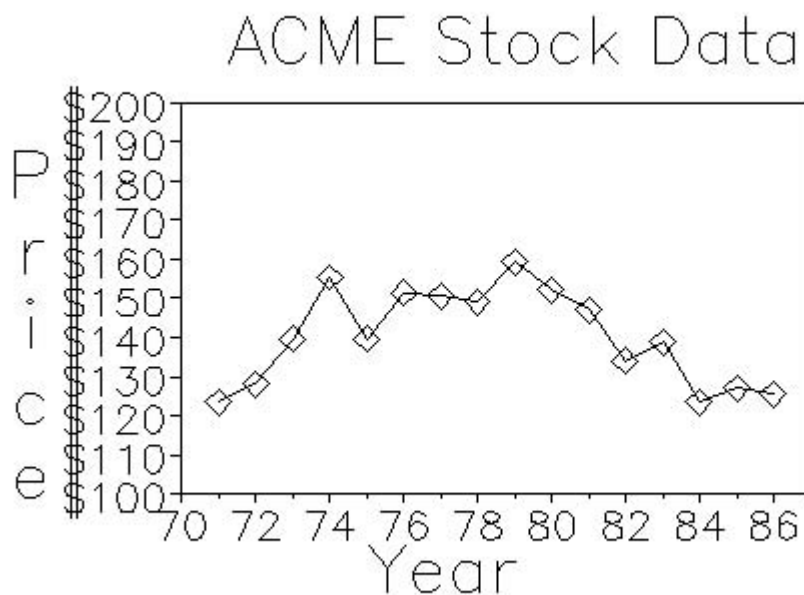
Figure 12.6. Stock Data with Axes and Labels

The statements that generated this graph are described below:

- GOPEN begins a new graph and names it STOCKS2.

- GPOLY draws a box around the display area.

- GWINDOW defines the world coordinate space to be larger than the actual range of stock data values.

- GPORT defines a viewport. It causes the graph to appear in the center of the display, with a border around it for text. The lower left corner has coordinates (15,15) and the upper right corner has coordinates (85,85).

- GINCLUDE includes the graphics segment STOCKS1. This saves you from having to plot points you have already created.

- GXAXIS draws the x axis. It begins at the point (70,100) and is 17 units (years) long, divided with 18 tick marks. The axis tick marks are printed with the numeric 2.0 format, and they have a height of 1.5 units.

- GYAXIS draws the y axis. It also begins at (70,100) but is 100 units (dollars) long, divided with 11 tick marks. The axis tick marks are printed with the DOLLAR5.0 format and have a height of 1.5 units.

- GSET sets the text font to be Swiss and the height of the letters to be 2.0 units. The height of the characters has been increased because the viewport definition scales character sizes relative to the viewport.

- GTEXT prints horizontal text. It prints the text string **Year** beginning at the world coordinate point (77,89).

- GVTEXT prints vertical text. It prints the text string **Price** beginning at the world coordinate point (68,200).

- GSCENTER runs the module to print centered text strings.

- GSHOW displays the graph.

Changing Windows and Viewports

Windows and viewports can be changed for the graphics segment any time that the segment is active. Using the stock price example, you can first define a window for the data during the years 1971 to 1974 and map this to the viewport defined on the upper half of the normalized device; then you can redefine the window to enclose the data for 1983 to 1986 and map this to an area in the lower half of the normalized device. Notice how the shape of the viewport affects the shape of the curve. Changing the viewport can affect the height of any printed characters as well. In this case, you can modify the HEIGHT parameter.

The following statements generate the graph in Figure 12.7:

```
        /* figure 12.7  */
reset clip;                             /* clip outside viewport */
call gopen;                                 /* open a new segment */
call gset("color","blue");
call gset("height",2.0);
call gwindow({71 120,74 175});              /* define a window */
call gport({20 55,80 90});                  /* define a viewport */
call gpoly({71 74 74 71},{120 120 170 170}); /* draw a border */
call gscript(71.5,162,"Viewport #1 1971-74",,   /* print text */
             ,3.0,"complex","red");
call gpoint(year,price,"diamond","green");     /* draw points */
call gdraw(year,price,1,"green");              /* connect points */
call gblkvpd;
call gwindow({83 120,86 170});              /* define new window */
call gport({20 10,80 45});                  /* define new viewport */
call gpoly({83 86 86 83},{120 120 170 170});   /* draw border */
call gpoint(year,price,"diamond","green");     /* draw points */
call gdraw(year,price,1,"green");              /* connect points */
call gscript(83.5,162,"Viewport #2 1983-86",,   /* print text */
             ,3.0,"complex","red");
call gshow;
```
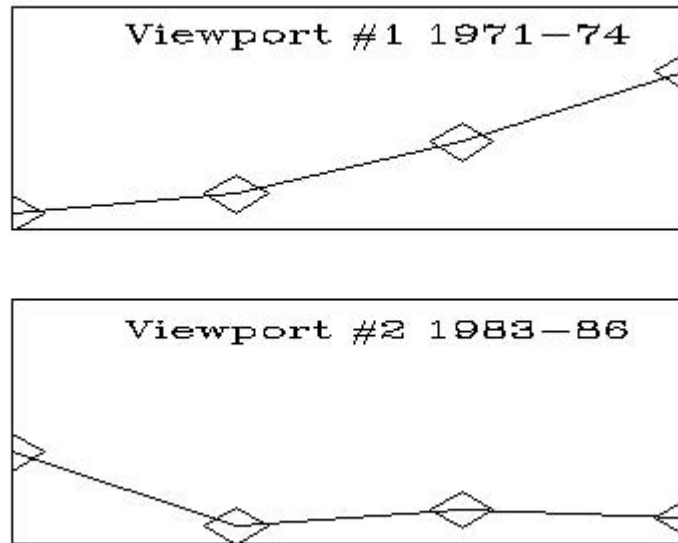
Figure 12.7.   Multiple Viewports

The RESET CLIP command is necessary because you are graphing only a part of the data in the window. You want to clip the data that falls outside of the window. See "Clipping Your Graphs" later in this chapter for more on clipping. In this graph, you

- open a new segment (GOPEN)
- define the first window for the first four years' data (GWINDOW)
- define a viewport in the upper part of the display device (GPORT)
- draw a box around the viewport (GPOLY)
- add text (GSCRIPT)
- graph the points and connect them (GPOINT and GDRAW)
- define the second window for the last four years (GWINDOW)
- define a viewport in the lower part of the display device (GPORT)
- draw a box around the viewport (GPOLY)
- graph the points and connect them (GPOINT and GDRAW)
- add text (GSCRIPT)
- display the graph (GSHOW)

Stacking Viewports

Viewports can be stacked; that is, a viewport can be defined relative to another viewport so that you have a viewport within a viewport.

A window or a viewport is changed globally through the IML graphics commands: the GWINDOW command for windows, and the GPORT, GPORTSTK, and GPORTPOP commands for viewports. When a window or viewport is defined, it persists across IML graphics commands until another window- or viewport-altering command is encountered. Stacking helps you define a viewport without losing the effect of a previously defined viewport. When a stacked viewport is *popped*, you are placed into the environment of the previous viewport.

Windows and viewports are associated with a particular segment; thus, they automatically become undefined when the segment is closed. A segment is closed whenever IML encounters a GCLOSE command or a GOPEN command. A window or a viewport can also be changed for a single graphics command. Either one can be passed as an argument to a graphics primitive, in which case any graphics output associated with the call is defined in the specified window or viewport. When a viewport is passed as an argument, it is stacked, or defined relative to the current viewport, and *popped* when the graphics command is complete.

For example, suppose you want to create a legend that shows the low and peak points of the data for the ACME stock graph. Create a graphics segment showing this information:

```
call gopen("legend");
call gset('height',5);    /* enlarged to accommodate viewport later */
call gset('font','swiss');
call gscript(5,75,"Stock Peak:   159.5 in 1979");
call gscript(5,65,"Stock Low:    123.6 in 1984");
call gclose;
```

Now create a segment that highlights and labels the low and peak points of the data:

```
   /* Highlight and label the low and peak points of the stock */
call gopen("labels");
call gwindow({70 100 87 200}); /* define window */
call gpoint(84,123.625,"circle","red",4) ;
call gtext(84,120,"LOW","red");
call gpoint(79,159.5,"circle","red",4);
call gtext(79,162,"PEAK","red");
call gclose;
```

Open a new graphics segment and include the STOCK1 segment created earlier in the chapter, placing the segment in the viewport {10 10 90 90}.

```
call gopen;
call gportstk ({10 10 90 90}); /* viewport for the plot itself */
call ginclude('stocks2');
```

To place the legend in the upper right corner of this viewport, use the GPORTSTK command instead of the GPORT command to define the legend's viewport relative to the one used for the plot of the stock data:

```
call gportstk ({70 70 100 100});   /* viewport for the legend */
call ginclude("legend");
```

Now pop the legend's viewport to get back to the viewport of the plot itself and
include the segment that labels and highlights the low and peak stock points:

```
call gportpop;   /* viewport for the legend */
call ginclude ("labels");
```

Finally, display the graph.

```
call gshow;
```
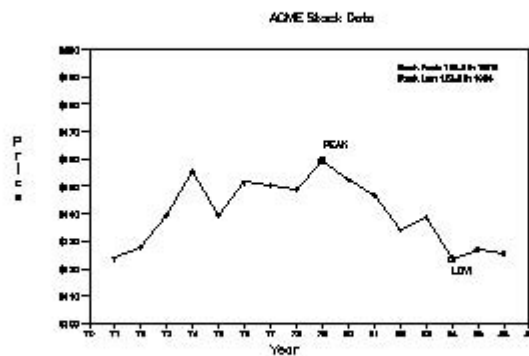

Figure 12.8.   Stacking Viewports

Clipping Your Graphs

The IML graphics subsystem does not automatically clip the output to the viewport.
Thus, it is possible that data are graphed outside of the defined viewport. This hap-
pens when there are data points lying outside of the defined window. For instance,
if you specify a window to be a subset of the world, then there will be data lying
outside of the window and these points will be graphed outside of the viewport. This
is usually not what you want. To clean up such graphs, you either delete the points
you do not want to graph or clip the graph.

There are two ways to clip a graph. You can use the RESET CLIP command, which
clips outside of a viewport. The CLIP option remains in effect until you submit
a RESET NOCLIP command. You can also use the GBLKVP command, which

clips either inside or outside of a viewport. Use the GBLKVP command to define a blanking area in which nothing can be drawn until the blanking area is released. Use the GBLKVPD command to release the blanking area.

Common Arguments

IML graphics commands are available in the form of call subroutines. They generally take a set of required arguments followed by a set of optional arguments. All graphics primitives take *window* and *viewport* as optional arguments. Some IML graphics commands, like GPOINT or GPIE, allow implicit repetition factors in the argument lists. The GPOINT command places as many markers as there are well-defined (x, y) pairs. The GPIE command draws as many slices as there are well-defined pies. In those cases, some of the attribute matrices can have more than one element, which are used in order. If an attribute list is exhausted before the repetition factor is completed, the last element of the list is used as the attribute for the remaining primitives.

The arguments to the IML graphics commands are positional. Thus, to skip over an optional argument from the middle of a list, you must specify a comma to hold its place. For example, the command

```
call gpoint(x,y, ,"red");
```

omits the third argument from the argument list.

The following list details the arguments commonly used in IML graphics commands:

color is a character matrix or literal that names a valid color as specified in the GOPTIONS statement. The default color is the first color specified in the COLORS= list in the GOPTIONS statement. If no such list is given, IML uses the first default color for the graphics device. Note that *color* can be specified either as a quoted literal, such as "RED", a color number, such as 1, or the name of a matrix containing a reference to a valid color. A color number n refers to the nth color in the color list.

You can change the default color with the GSET command.

font is a character matrix or quoted literal that specifies a valid font name. The default font is the hardware font, which can be changed by the GSET command unless a viewport is in effect.

height is a numeric matrix or literal that specifies the character height. The unit of height is the *gunit* of the GOPTIONS statement, when specified; otherwise, the unit is a character cell. The default height is 1 *gunit*, which you can change using the GSET command.

pattern is a character matrix or quoted literal that specifies the pattern to fill the interior of a closed curve. You specify a pattern by a coded character string as documented in the V= option in the PATTERN statement (refer to the chapter on the PATTERN ttatement in *SAS/GRAPH Software: Reference*.

The default pattern set by the IML graphics subsystem is "E", that is, empty. The default pattern can be changed using the GSET command.

segment-name   is a character matrix or quoted literal that specifies a valid SAS name used to identify a graphics segment. The *segment-name* is associated with the graphics segment opened with a GOPEN command. If you do not specify *segment-name*, IML generates default names. For example, to create a graphics segment called PLOTA, enter

```
call gopen("plota");
```

Graphics segments are not allowed to have the same name as an existing segment. If you try to create a second segment named PLOTA, (that is, when the *replace flag* is turned off), then the second segment is named PLOTA1. The *replace* flag is set by the GOPEN command for the segment that is being created. To open a new segment named PLOTA and replace an existing segment with the same name, enter

```
call gopen("plota",1);
```

If you do not specify a *replace* argument to the GOPEN command, the default is set by the GSTART command for all subsequent segments that are created. By default, the GSTART command sets the *replace* flag to 0, so that new segments do not replace like-named segments.

style   is a numeric matrix or literal that specifies an index corresponding to the line style documented for the SYMBOL statement in the chapter on the Symbol statement in *SAS/GRAPH Software: Reference*. The IML graphics subsystem sets the default line style to be 1, a solid line. The default line style can be changed using the GSET command.

symbol   is a character matrix or quoted literal that specifies either a character string corresponding to a symbol as defined for the V= option of the SYMBOL statement or specifies the corresponding identifying symbol number. STAR is the default symbol used by the IML graphics subsystem.

SAS/IML graphics commands are described in detail in Chapter 20, "Language Reference," .

Refer also to *SAS/GRAPH Software: Reference* for additional information.

Graphics Examples

This section provides the details and code for three examples involving SAS/IML graphics. The first example shows a 2×2 matrix of scatter plots and a 3×3 matrix of scatter plots. A matrix of scatter plots is useful when you have several variables that you want to investigate simultaneously rather than in pairs. The second example draws a grid for representing a train schedule, with arrival and departure dates on the horizontal axis and destinations along the vertical axis. The final example plots Fisher's Iris data. This example shows how to plot several graphs on one page.

Example 12.1. Scatter Plot Matrix

With the viewport capability of the IML graphics subroutine, you can arrange several graphs on a page. In this example, multiple graphs are generated from three variables and are displayed in a scatterplot matrix. For each variable, one contour plot is generated with each of the other variables as the dependent variable. For the graphs on the main diagonal, a box and whisker plot is generated for each variable.

This example takes advantage of user-defined IML modules:

BOXWHSKR   computes median and quartiles.

GBXWHSKR   draws box and whisker plots.

CONTOUR   generates confidence ellipses assuming bivariate normal data.

GCONTOUR   draws the confidence ellipses for each pair of variables.

GSCATMAT   produces the $n \times n$ scatter plot matrix, where n is the number of variables.

The code for the five modules and a sample data set follow. The modules produce Figure 12.9 on page 424 and Figure 12.10 on page 424.

```
      /* This program generates a data set and uses iml graphics    */
      /* subsystem to draw a scatterplot matrix.                     */
      /*                                                             */
   data factory;
     input recno prod temp a defect mon;
     datalines;
       1    1.82675     71.124    1.12404    1.79845        2
       2    1.67179    70.9245   0.924523    1.05246        3
       3    2.22397     71.507    1.50696    2.36035        4
       4    2.39049    74.8912    4.89122    1.93917        5
       5    2.45503    73.5338    3.53382     2.0664        6
       6    1.68758    71.6764    1.67642    1.90495        7
       7    1.98233    72.4222    2.42221    1.65469        8
       8    1.17144    74.0884    4.08839    1.91366        9
       9    1.32697    71.7609    1.76087    1.21824       10
      10    1.86376    70.3978   0.397753    1.21775       11
      11    1.25541     74.888    4.88795    1.87875       12
      12    1.17617    73.3528    3.35277    1.15393        1
      13    2.38103    77.1762    7.17619    2.26703        2
      14    1.13669    73.0157    3.01566          1        3
```

Example 12.1. Scatter Plot Matrix ◆ 419

15	1.01569	70.4645	0.464485	1	4
16	2.36641	74.1699	4.16991	1.73009	5
17	2.27131	73.1005	3.10048	1.79657	6
18	1.80597	72.6299	2.62986	1.8497	7
19	2.41142	81.1973	11.1973	2.137	8
20	1.69218	71.4521	1.45212	1.47894	9
21	1.95271	74.8427	4.8427	1.93493	10
22	1.28452	76.7901	6.79008	2.09208	11
23	1.51663	83.4782	13.4782	1.81162	12
24	1.34177	73.4237	3.42369	1.57054	1
25	1.4309	70.7504	0.750369	1.22444	2
26	1.84851	72.9226	2.92256	2.04468	3
27	2.08114	78.4248	8.42476	1.78175	4
28	1.99175	71.0635	1.06346	1.25951	5
29	2.01235	72.2634	2.2634	1.36943	6
30	2.38742	74.2037	4.20372	1.82846	7
31	1.28055	71.2495	1.24953	1.8286	8
32	2.05698	76.0557	6.05571	2.03548	9
33	1.05429	77.721	7.72096	1.57831	10
34	2.15398	70.8861	0.886068	2.1353	11
35	2.46624	70.9682	0.968163	2.26856	12
36	1.4406	73.5243	3.52429	1.72608	1
37	1.71475	71.527	1.52703	1.72932	2
38	1.51423	78.5824	8.5824	1.97685	3
39	2.41538	73.7909	3.79093	2.07129	4
40	2.28402	71.131	1.13101	2.25293	5
41	1.70251	72.3616	2.36156	2.04926	6
42	1.19747	72.3894	2.3894	1	7
43	1.08089	71.1729	1.17288	1	8
44	2.21695	72.5905	2.59049	1.50915	9
45	1.52717	71.1402	1.14023	1.88717	10
46	1.5463	74.6696	4.66958	1.25725	11
47	2.34151	90	20	3.57864	12
48	1.10737	71.1989	1.19893	1.62447	1
49	2.2491	76.6415	6.64147	2.50868	2
50	1.76659	71.7038	1.70377	1.231	3
51	1.25174	76.9657	6.96572	1.99521	4
52	1.81153	73.0722	3.07225	2.15915	5
53	1.72942	71.9639	1.96392	1.86142	6
54	2.17748	78.1207	8.12068	2.54388	7
55	1.29186	77.0589	7.05886	1.82777	8
56	1.92399	72.6126	2.61256	1.32816	9
57	1.38008	70.8872	0.887228	1.37826	10
58	1.96143	73.8529	3.85289	1.87809	11
59	1.61795	74.6957	4.69565	1.65806	12
60	2.02756	75.7877	5.78773	1.72684	1
61	2.41378	75.9826	5.98255	2.76309	2
62	1.41413	71.3419	1.34194	1.75285	3
63	2.31185	72.5469	2.54685	2.27947	4
64	1.94336	71.5592	1.55922	1.96157	5
65	2.094	74.7338	4.73385	2.07885	6
66	1.19458	72.233	2.23301	1	7
67	2.13118	79.1225	9.1225	1.84193	8
68	1.48076	87.0511	17.0511	2.94927	9
69	1.98502	79.0913	9.09131	2.47104	10
70	2.25937	73.8232	3.82322	2.49798	12
71	1.18744	70.6821	0.682067	1.2848	1
72	1.20189	70.7053	0.705311	1.33293	2
73	1.69115	73.9781	3.9781	1.87517	3

```
            74    1.0556    73.2146    3.21459          1          4
            75    1.59936   71.4165    1.41653    1.29695          5
            76    1.66044   70.7151    0.715145   1.22362          6
            77    1.79167   74.8072    4.80722    1.86081          7
            78    2.30484   71.5028    1.50285    1.60626          8
            79    2.49073   71.5908    1.59084    1.80815          9
            80    1.32729   70.9077    0.907698   1.12889         10
            81    2.48874   83.0079    13.0079    2.59237         11
            82    2.46786   84.1806    14.1806    3.35518         12
            83    2.12407   73.5826    3.58261    1.98482          1
            84    2.46982   76.6556    6.65559    2.48936          2
            85    1.00777   70.2504    0.250364         1          3
            86    1.93118   73.9276    3.92763    1.84407          4
            87    1.00017   72.6359    2.63594     1.3882          5
            88    1.90622    71.047      1.047     1.7595          6
            89    2.43744    72.321    2.32097    1.67244          7
            90    1.25712        90         20    2.63949          8
            91    1.10811   71.8299    1.82987          1          9
            92    2.25545   71.8849     1.8849    1.94247         10
            93    2.47971   73.4697     3.4697    1.87842         11
            94    1.93378   74.2952     4.2952    1.52478         12
            95    2.17525   73.0547    3.05466    2.23563          1
            96    2.18723   70.8299    0.829929   1.75177          2
            97    1.69984   72.0026    2.00263    1.45564          3
            98    1.12504   70.4229    0.422904   1.06042          4
            99    2.41723   73.7324    3.73238    2.18307          5
        ;

        proc iml;

          /*-- Load graphics --*/
          call gstart;


          /*--------------------*/
          /*-- Define modules --*/
          /*--------------------*/

          /*   Module : compute contours   */
          start contour(c,x,y,npoints,pvalues);

          /*   This routine computes contours for a scatter plot        */
          /*   c returns the contours as consecutive pairs of columns   */
          /*   x and y are the x and y coordinates of the points        */
          /*   npoints is the number of points in a contour             */
          /*   pvalues is a column vector of contour probabilities       */
          /*   the number of contours is controlled by the ncol(pvalue) */


            xx=x||y;
            n=nrow(x);
        /* Correct for the mean */
            mean=xx[+,]/n;
            xx=xx-mean@j(n,1,1);

        /* Find principle axes of ellipses */
            xx=xx` *xx/n;
            call eigen(v,e,xx);
```

Example 12.1. Scatter Plot Matrix ◆ 421

```
   /* Set contour levels */
      c=-2*log(1-pvalues);
      a=sqrt(c*v[1]); b=sqrt(c*v[2]);


   /* Parameterize the ellipse by angle */
      t=((1:npoints)-{1})#atan(1)#8/(npoints-1);
      s=sin(t);
      t=cos(t);
      s=s` *a;
      t=t` *b;


   /* Form contour points */
      s=((e*(shape(s,1)//shape(t,1)))+mean`@j(1,npoints*ncol(c),1))`;
      c=shape(s,npoints);


   /* Returned as ncol pairs of columns for contours */
   finish contour;
   /*-- Module : draw contour curves --*/
   start gcontour(t1, t2);
      run contour(t12, t1, t2, 30, {.5 .8 .9});
      window=(min(t12[,{1 3}],t1)||min(t12[,{2 4}],t2))//
              (max(t12[,{1 3}],t1)||max(t12[,{2 4}],t2));
      call gwindow(window);
      call gdraw(t12[,1],t12[,2],,'blue');
      call gdraw(t12[,3],t12[,4],,'blue');
      call gdraw(t12[,5],t12[,6],,'blue');
      call gpoint(t1,t2,,'red');
   finish gcontour;


   /*-- Module : find median, quartiles for box and whisker plot --*/
   start boxwhskr(x, u, q2, m, q1, l);
      rx=rank(x);
      s=x;
      s[rx,]=x;
      n=nrow(x);

   /*-- Median --*/
      m=floor(((n+1)/2)||((n+2)/2));
      m=(s[m,])[+,]/2;

   /*-- Compute quartiles --*/
      q1=floor(((n+3)/4)||((n+6)/4));
      q1=(s[q1,])[+,]/2;
      q2=ceil(((3*n+1)/4)||((3*n-2)/4));
      q2=(s[q2,])[+,]/2;
      h=1.5*(q2-q1);    /*-- step=1.5*(interquartile range) --*/
      u=q2+h;
      l=q1-h;
      u=(u>s)[+,];       /*-- adjacent values ----------------*/
      u=s[u,];
      l=(l>s)[+,];
      l=s[l+1,];

   finish boxwhskr;


   /*-- Box and Whisker plot --*/
   start gbxwhskr(t, ht);
```

```
      run boxwhskr(t, up, q2,med, q1, lo);

   /*---Adjust screen viewport and data window  */
      y=min(t)//max(t);
      call gwindow({0, 100} || y);
      mid  = 50;
      wlen = 20;

   /*-- Add whiskers */
      wstart=mid-(wlen/2);
      from=(wstart||up)//(wstart||lo);
      to=((wstart//wstart)+wlen)||from[,2];

   /*-- Add box  */
      len=50;
      wstart=mid-(len/2);
      wstop=wstart+len;
      from=from//(wstart||q2)//(wstart||q1)//
           (wstart||q2)//(wstop||q2);
      to=to//(wstop||q2)//(wstop||q1)//
           (wstart||q1)//(wstop||q1);

   /*---Add median line  */
      from=from//(wstart||med);
      to=to//(wstop||med);

   /*---Attach whiskers to box  */
      from=from//(mid||up)//(mid||lo);
      to=to//(mid||q2)//(mid||q1);

   /*-- Draw box and whiskers  */
      call gdrawl(from, to,,'red');

   /*---Add minimum and maximum data points */
      call gpoint(mid, y ,3,'red');

   /*---Label min, max, and mean  */
      y=med//y;
      s={'med' 'min' 'max'};
      call gset("font","swiss");
      call gset('height',13);
      call gscript(wstop+ht, y, char(y,5,2),,,,,'blue');
      call gstrlen(len, s);
      call gscript(wstart-len-ht,y,s,,,,,'blue');
      call gset('height');
   finish gbxwhskr;


   /*-- Module : do scatter plot matrix --*/
   start gscatmat(data, vname);
      call gopen('scatter');
      nv=ncol(vname);
      if (nv=1) then nv=nrow(vname);
      cellwid=int(90/nv);
      dist=0.1*cellwid;
      width=cellwid-2*dist;
      xstart=int((90 -cellwid * nv)/2) + 5;
      xgrid=((0:nv)#cellwid + xstart)`;
```

Example 12.1. Scatter Plot Matrix ◆ 423

```
   /*-- Delineate cells --*/
      cell1=xgrid;
      cell1=cell1||(cell1[nv+1]//cell1[nv+1-(0:nv-1)]);
      cell2=j(nv+1, 1, xstart);
      cell2=cell1[,1]||cell2;
      call gdrawl(cell1, cell2);
      call gdrawl(cell1[,{2 1}], cell2[,{2 1}]);
      xstart = xstart + dist;  ystart = xgrid[nv] + dist;

   /*-- Label variables ---*/
      call gset("height", 5);
      call gset("font","swiss");
      call gstrlen(len, vname);
      where=xgrid[1:nv] + (cellwid-len)/2;
      call gscript(where, 0, vname) ;
      len=len[nv-(0:nv-1)];
      where=xgrid[1:nv] + (cellwid-len)/2;
      call gscript(4,where, vname[nv - (0:nv-1)],90);

   /*-- First viewport --*/
      vp=(xstart || ystart)//((xstart || ystart) + width) ;

   /*  Since the characters are scaled to the viewport       */
   /*    (which is inversely porportional to the             */
   /*    number of variables),                               */
   /*    enlarge it proportional to the number of variables  */

      ht=2*nv;
      call gset("height", ht);
      do i=1 to nv;
         do j=1 to i;
            call gportstk(vp);
            if (i=j) then run gbxwhskr(data[,i], ht);
            else run gcontour(data[,j], data[,i]);
      /*-- onto the next viewport --*/
            vp[,1] = vp[,1] + cellwid;
            call gportpop;
         end;
         vp=(xstart // xstart + width) || (vp[,2] - cellwid);
      end;
      call gshow;
finish gscatmat;

   /*-- Placement of text is based on the character height.     */
   /* The IML modules defined here assume percent as the unit of */
   /* character height for device independent control.          */
goptions gunit=pct;

use factory;
vname={prod, temp, defect};
read all var vname into xyz;
run gscatmat(xyz, vname[1:2]);   /*-- 2 x 2 scatter plot matrix --*/
run gscatmat(xyz, vname);        /*-- 3 x 3 scatter plot matrix --*/
quit;

goptions gunit=cell;             /*-- reset back to default --*/
```
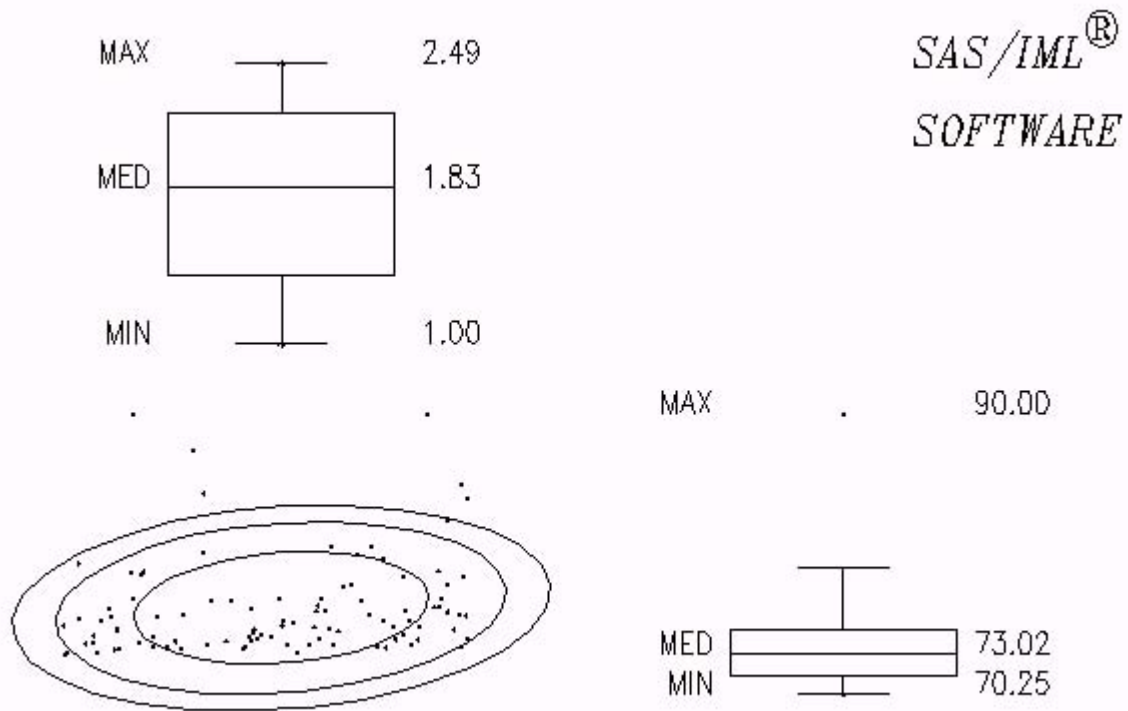
Figure 12.9. 2×2 Scatter Plot Matrix
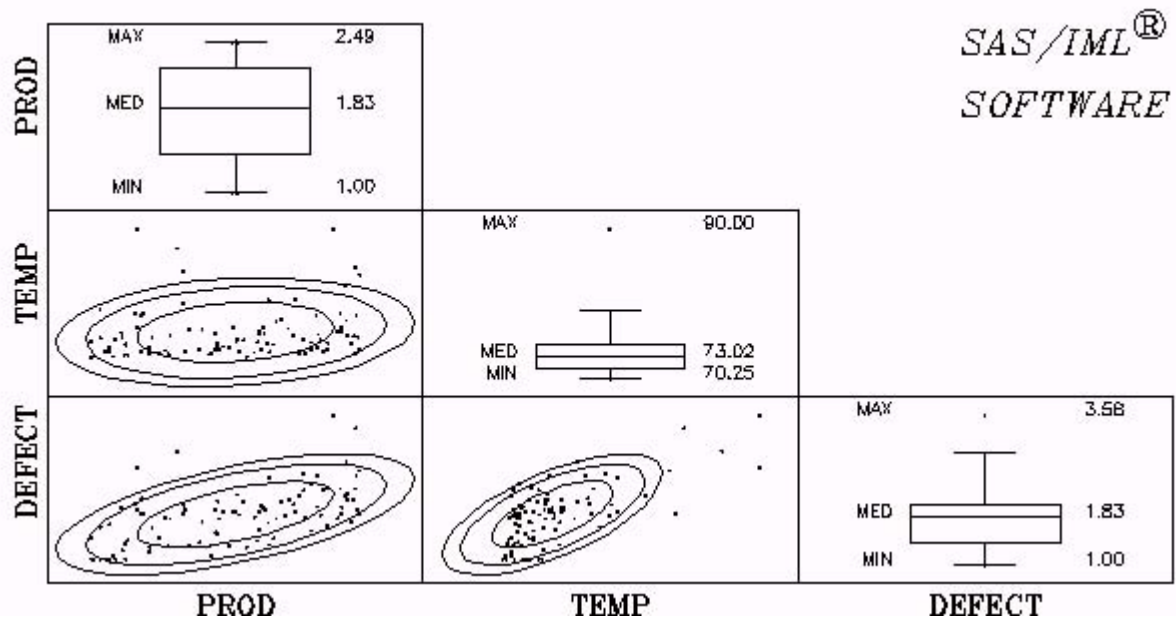
Figure 12.10. 3×3 Scatter Plot Matrix

Example 12.2. Train Schedule ⬥ 425

Example 12.2. Train Schedule

This example draws a grid on which the horizontal dimension gives the arrival/departure data and the vertical dimension gives the destination. The first section of the code defines the matrices used. The following section generates the graph. The following example code shows some applications of the GGRID, GDRAWL, GSTRLEN, and GSCRIPT subroutines. This code produces Figure 12.11 on page 426.

```
proc iml;
/*  Placement of text is based on the character height.    */
/*  The graphics segment defined here assumes percent as the */
/*  unit of character height for device independent control. */
   goptions gunit=pct;

   call gstart;
   /* Define several necessary matrices  */
   cityloc={0 27 66 110 153 180}`;
   cityname={"Paris" "Montereau" "Tonnerre" "Dijon" "Macon" "Lyons"};
   timeloc=0:30;
   timename=char(timeloc,2,0);
   /* Define a data matrix  */
   schedule=
      /* origin dest start  end       comment */
         { 1     2    11.0  12.5,   /* train 1 */
           2     3    12.6  14.9,
           3     4    15.5  18.1,
           4     5    18.2  20.6,
           5     6    20.7  22.3,
           6     5    22.6  24.0,
           5     4     0.1   2.3,
           4     3     2.5   4.5,
           3     2     4.6   6.8,
           2     1     6.9   8.5,
           1     2    19.2  20.5,   /* train 2 */
           2     3    20.6  22.7,
           3     4    22.8  25.0,
           4     5     1.0   3.3,
           5     6     3.4   4.5,
           6     5     6.9   8.5,
           5     4     8.6  11.2,
           4     3    11.6  13.9,
           3     2    14.1  16.2,
           2     1    16.3  18.0
         };

   xy1=schedule[,3]||cityloc[schedule[,1]];
   xy2=schedule[,4]||cityloc[schedule[,2]];

   call gopen;
   call gwindow({-8 -35, 36 240});
   call ggrid(timeloc,cityloc,1,"red");
   call gdrawl(xy1,xy2,,"blue");

   /*-- center title -- */
   s = "Train Schedule: Paris to Lyons";
   call gstrlen(m, s,5,"titalic");
```

```
        call gscript(15-m/2,185,s,,,5,"titalic");

        /*-- find max graphics text length of cityname --*/
        call gset("height",3);
        call gset("font","italic");
        call gstrlen(len, cityname);
        m = max(len) +1.0
        call gscript(-m, cityloc,cityname);
        call gscript(timeloc - .5,-12,timename,-90,90);
        call gshow;

    quit;
    goptions gunit=cell;                /*-- reset back to default --*/
```
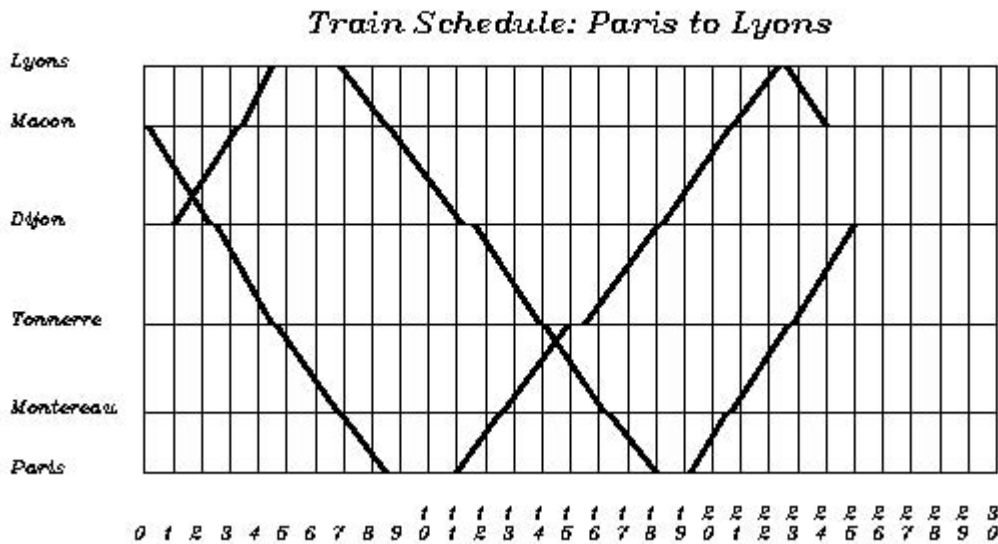

Figure 12.11. Train Schedule

Example 12.3. Fisher's Iris Data

This example generates four scatter plots and prints them on a single page. Scatter
plots of sepal length versus petal length, sepal width versus petal width, sepal length
versus sepal width, and petal length versus petal width are generated. The following
code produces Figure 12.12 on page 429.

```
data iris;
    title 'Fisher (1936) Iris Data';
    input sepallen sepalwid petallen petalwid spec_no @@;
    if spec_no=1 then species='setosa    ';
    if spec_no=2 then species='versicolor';
    if spec_no=3 then species='virginica ';
    label sepallen='sepal length in mm.'
          sepalwid='sepal width  in mm.'
          petallen='petal length in mm.'
```

Example 12.3. Fisher's Iris Data   ◆   427

```
          petalwid='petal width   in mm.';
   datalines;

 50 33 14 02 1 64 28 56 22 3 65 28 46 15 2
 67 31 56 24 3 63 28 51 15 3 46 34 14 03 1
 69 31 51 23 3 62 22 45 15 2 59 32 48 18 2
 46 36 10 02 1 61 30 46 14 2 60 27 51 16 2
 65 30 52 20 3 56 25 39 11 2 65 30 55 18 3
 58 27 51 19 3 68 32 59 23 3 51 33 17 05 1
 57 28 45 13 2 62 34 54 23 3 77 38 67 22 3
 63 33 47 16 2 67 33 57 25 3 76 30 66 21 3
 49 25 45 17 3 55 35 13 02 1 67 30 52 23 3
 70 32 47 14 2 64 32 45 15 2 61 28 40 13 2
 48 31 16 02 1 59 30 51 18 3 55 24 38 11 2
 63 25 50 19 3 64 32 53 23 3 52 34 14 02 1
 49 36 14 01 1 54 30 45 15 2 79 38 64 20 3
 44 32 13 02 1 67 33 57 21 3 50 35 16 06 1
 58 26 40 12 2 44 30 13 02 1 77 28 67 20 3
 63 27 49 18 3 47 32 16 02 1 55 26 44 12 2
 50 23 33 10 2 72 32 60 18 3 48 30 14 03 1
 51 38 16 02 1 61 30 49 18 3 48 34 19 02 1
 50 30 16 02 1 50 32 12 02 1 61 26 56 14 3
 64 28 56 21 3 43 30 11 01 1 58 40 12 02 1
 51 38 19 04 1 67 31 44 14 2 62 28 48 18 3
 49 30 14 02 1 51 35 14 02 1 56 30 45 15 2
 58 27 41 10 2 50 34 16 04 1 46 32 14 02 1
 60 29 45 15 2 57 26 35 10 2 57 44 15 04 1
 50 36 14 02 1 77 30 61 23 3 63 34 56 24 3
 58 27 51 19 3 57 29 42 13 2 72 30 58 16 3
 54 34 15 04 1 52 41 15 01 1 71 30 59 21 3
 64 31 55 18 3 60 30 48 18 3 63 29 56 18 3
 49 24 33 10 2 56 27 42 13 2 57 30 42 12 2
 55 42 14 02 1 49 31 15 02 1 77 26 69 23 3
 60 22 50 15 3 54 39 17 04 1 66 29 46 13 2
 52 27 39 14 2 60 34 45 16 2 50 34 15 02 1
 44 29 14 02 1 50 20 35 10 2 55 24 37 10 2
 58 27 39 12 2 47 32 13 02 1 46 31 15 02 1
 69 32 57 23 3 62 29 43 13 2 74 28 61 19 3
 59 30 42 15 2 51 34 15 02 1 50 35 13 03 1
 56 28 49 20 3 60 22 40 10 2 73 29 63 18 3
 67 25 58 18 3 49 31 15 01 1 67 31 47 15 2
 63 23 44 13 2 54 37 15 02 1 56 30 41 13 2
 63 25 49 15 2 61 28 47 12 2 64 29 43 13 2
 51 25 30 11 2 57 28 41 13 2 65 30 58 22 3
 69 31 54 21 3 54 39 13 04 1 51 35 14 03 1
 72 36 61 25 3 65 32 51 20 3 61 29 47 14 2
 56 29 36 13 2 69 31 49 15 2 64 27 53 19 3
 68 30 55 21 3 55 25 40 13 2 48 34 16 02 1
 48 30 14 01 1 45 23 13 03 1 57 25 50 20 3
 57 38 17 03 1 51 38 15 03 1 55 23 40 13 2
 66 30 44 14 2 68 28 48 14 2 54 34 17 02 1
 51 37 15 04 1 52 35 15 02 1 58 28 51 24 3
 67 30 50 17 2 63 33 60 25 3 53 37 15 02 1
 ;

 proc iml;

 use iris; read all;
```

```
   /*------------------------------------------------------  */
   /*  Create 5 graphs, PETAL, SEPAL, SPWID, SPLEN, and ALL4 */
   /*   After the graphs are created, to see any one, type    */
   /*             CALL GSHOW("name");                          */
   /*   where name is the name of any one of the 5 graphs     */
   /* ---------------------------------------------------     */


call gstart;                     /*-- always start with GSTART --*/

   /*-- Spec_no will be used as marker index, change 3 to 4 */
   /*-- 1 is + , 2 is x, 3 is *, 4 is a square -------------*/
do i=1 to 150;
   if (spec_no[i] = 3) then spec_no[i] = 4;
end;

   /*-- Creates 4 x-y plots stored in 4 different segments */

   /*-- Creates a segment called petal, petallen by petalwid --*/
call gopen("petal");
   wp = { -10 -5, 90 30};
   call gwindow(wp);
   call gxaxis({0 0}, 75, 6,,,'5.1');
   call gyaxis({0 0}, 25, 5,,,'5.1');
   call gpoint(petallen, petalwid, spec_no, 'blue');
   labs = "Petallen vs Petalwid";
   call gstrlen(len, labs,2, 'swiss');
   call gscript(40-len/2,-4,labs,,,2,'swiss');

   /*-- Creates a segment called sepal, sepallen by sepalwid --*/
call gopen("sepal");
   ws = {35 15 85 55};
   call gwindow(ws);
   call gxaxis({40 20}, 40, 9, , ,'5.1');
   call gyaxis({40 20}, 28, 7, , ,'5.1');
   call gpoint(sepallen, sepalwid, spec_no, 'blue');
   labs = "Sepallen vs Sepalwid";
   call gstrlen(len, labs,2, 'swiss');
   call gscript(60-len/2,16,labs,,,2,'swiss');

   /*-- Creates a segment called spwid, petalwid by sepalwid --*/
call gopen("spwid");
   wspwid = { 15 -5 55 30};
   call gwindow(wspwid);
   call gxaxis({20 0}, 28, 7,,,'5.1');
   call gyaxis({20 0}, 25, 5,,,'5.1');
   call gpoint(sepalwid, petalwid, spec_no, 'green');
   labs = "Sepalwid vs Petalwid";
   call gstrlen(len, labs,2,'swiss');
   call gscript(35-len/2,-4,labs,,,2,'swiss');


   /*-- Creates a segment called splen, petallen by sepallen --*/
call gopen("splen");
   wsplen = {35 -15 85 90};
   call gwindow(wsplen);
   call gxaxis({40 0}, 40, 9,,,'5.1');
```

Example 12.3. Fisher's Iris Data ⬧ 429

```
        call gyaxis({40 0}, 75, 6,,,'5.1');
        call gpoint(sepallen, petallen, spec_no, 'red');
        labs = "Sepallen vs Petallen";
        call gstrlen(len, labs,2,'swiss');
        call gscript(60-len/2,-14,labs,,,2,'swiss');


        /*-- Create a new segment */
    call gopen("all4");
        call gport({50 0, 100 50});  /* change viewport, lower right ----*/
        call ginclude("sepal");      /* include sepal in this graph -----*/
        call gport({0 50, 50 100});  /* change the viewport, upper left  */
        call ginclude("petal");      /* include petal ------------------*/
        call gport({0 0, 50 50});    /* change the viewport, lower left  */
        call ginclude("spwid");      /* include spwid ------------------*/
        call gport({50 50, 100 100});/* change the viewport, upper right */
        call ginclude("splen");      /* include splen ------------------*/

    call gshow("Petal");
```
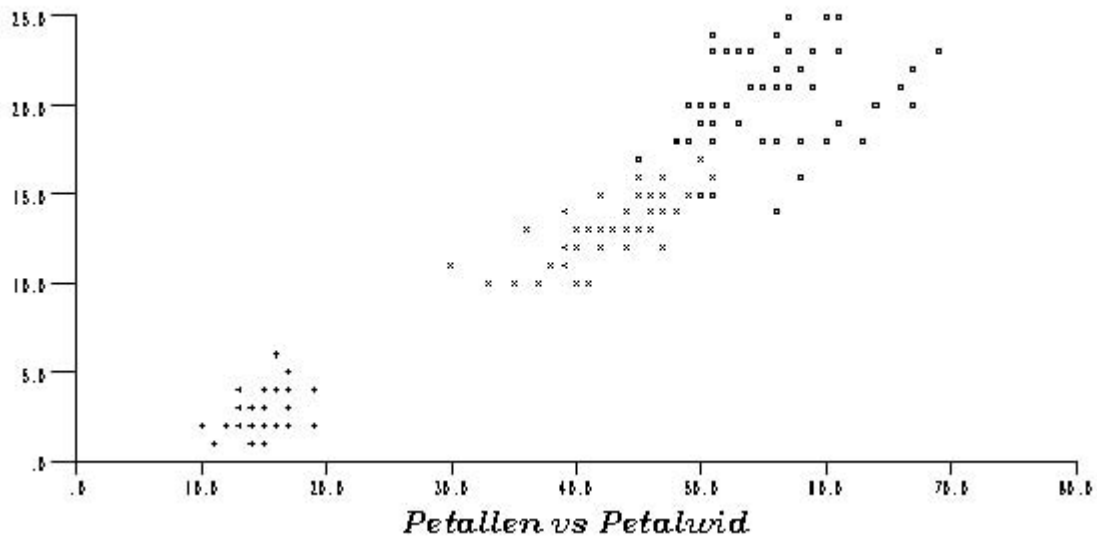

Figure 12.12. Petal Length versus Petal Width

Chapter 13
Window and Display Features

Chapter Contents

Chapter 13
Window and Display Features

Overview

The dynamic nature of IML gives you the ability to create windows on your display for full-screen data entry or menuing. Using the WINDOW statement, you can define a window, its fields, and its attributes. Using the DISPLAY statement, you can display a window and await data entry.

These statements are similar in form and function to the corresponding statements in the SAS DATA step. The specification of fields in the WINDOW or DISPLAY statements is similar to the specifications used in the INPUT and PUT statements. Using these statements you can write applications that behave similarly to other full-screen facilities in the SAS System, such as the BUILD procedure in SAS/AF software and the FSEDIT procedure in SAS/FSP software.

Creating a Display Window for Data Entry

Suppose that your application is a data entry system for a mailing list. You want to create a data set called MAILLIST by prompting the user with a window that displays all the entry fields. You want the data entry window to look as follows.

```
+--MAILLIST--------------------------------------------+
| Command==>                                           |
|                                                      |
|                                                      |
| NAME:                                                |
| ADDRESS:                                             |
| CITY: STATE: ZIP:                                    |
| PHONE:                                               |
|                                                      |
+------------------------------------------------------+
```

The process for creating a display window for this application consists of

- initializing the variables
- creating a SAS data set
- defining a module for collecting data that

    1. defines a window
    2. defines the data fields
    3. defines a loop for collecting data
    4. provides an exit from the loop

● executing the data-collecting routine

The whole system can be implemented with the following code to define modules
INITIAL and MAILGET:

```
/*   module to initialize the variables                        */
/*                                                             */
start initial;
   name='         ';
   addr='                    ';
   city='                    ';
   state='  ';
   zip='    ';
   phone='        ';
finish initial;
```

This defines a module named INITIAL that initializes the variables you want to col-
lect. The initialization sets the string length for the character fields. You need to do
this prior to creating your data set.

Now define a module for collecting the data:

```
/* module to collect data                                    */
/*                                                            */
start mailget;
/* define the window                                         */
   window maillist cmndline=cmnd msgline=msg
      group=addr
      #2 " NAME: " name
      #3 " ADDRESS:" addr
      #4 " CITY: " city +2 "STATE: " state +2 "ZIP: " zip
      #5 " PHONE: " phone;
/*                                                           */
/* collect addresses until the user enters exit              */
/*                                                           */
   do until(cmnd="EXIT");
      run initial;
      msg="ENTER SUBMIT TO APPEND OBSERVATION, EXIT TO END";
/*                                                           */
/* loop until user types submit or exit                      */
/*                                                           */
      do until(cmnd="SUBMIT"|cmnd="EXIT");
         display maillist.addr;
      end;
      if cmnd="SUBMIT" then append;
   end;
   window close=maillist;
finish mailget;
/* initialize variables                                      */
run initial;
/* create the new data set                                   */
create maillist var{name addr city state zip phone};
/* collect data                                              */
```

```
     run mailget;
     /* close the new data set                                    */
     close maillist;
```

In the module MAILGET, the WINDOW statement creates a window named MAILLIST with a group of fields (the group is named ADDR) presenting data fields for data entry. The program sends messages to the window through the MSGLINE= variable MSG. The program receives commands you enter through the CMNDLINE= variable CMND.

You can enter data into the fields after each prompt field. After you are finished with the entry, press a key defined as SUBMIT, or type SUBMIT in the command field. The data are appended to the data set MAILLIST. When data entry is complete, enter EXIT in the command field. If you enter a command other than SUBMIT, EXIT, or a valid SAS windowing environment command in the command field, you get this message on the message line:

```
     ENTER SUBMIT TO APPEND OBSERVATION, EXIT TO END.
```

Using the WINDOW Statement

You use the WINDOW statement to define a window, its fields, and its attributes. The general form of the WINDOW statement is

> **WINDOW** <**CLOSE=**> *window-name* < *window-options* >
>
> > <**GROUP=***group-name-1 field-specs*
> >
> > < . . .**GROUP=***group-name-n field-specs* >>**;**

The following options can be used with the WINDOW statement:

CLOSE=
is used only when you want to close the window.

window-name
is a valid SAS name for the window. This name is displayed in the upper left border of the window.

window-options
control the size, position, and other attributes of the window. You can change the attributes interactively with window commands such as WGROW, WDEF, WSHRINK, and COLOR. These options are described in the next section.

GROUP=*group-name*
starts a repeating sequence of groups of fields defined for the window. The *group-name* is a valid SAS variable name used to identify a group of fields in a DISPLAY statement that occurs later in the program.

field-specs
is a sequence of field specifications made up of positionals, field operands, formats, and options. These are described in "Field Specifications" later in this chapter.

Window Options

Window-options control the attributes of the window. The following options are valid in the WINDOW statement:

CMNDLINE=*name*
names a character variable in which the command line entered by the user is stored.

COLOR=*operand*
specifies the background color for the window. The *operand* can be either a quoted character literal or the name of a character variable containing the color. The valid values are BLACK, GREEN, MAGENTA, RED, CYAN, GRAY, and BLUE. This default is BLACK.

COLUMNS=*operand*
specifies the starting number of columns of the window. The *operand* can be either a literal number, a variable name, or an expression in parentheses. The default is 78.

ICOLUMN=*operand*
specifies the initial column position of the window on the display screen. The *operand* can be either a literal number or a variable name. The default is column 1.

IROW=*operand*
specifies the initial row position of the window on the display screen. The *operand* can be either a literal number or a variable name. The default 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 a quoted character literal or the name of a character variable containing the message.

ROWS=*operand*
determines the starting number of rows of the window. The *operand* is either a literal number, the name of a variable containing the number, or an expression in parentheses yielding the number. The default is 23 rows.

Field Specifications

Both the WINDOW and DISPLAY statements allow field specifications. Field specifications have the general form

   <*positionals*> *field-operand* <*format*> <*field-options*>

Positionals

The *positionals* are directives specifying the position on the screen in which to begin the field. There are four kinds of positionals, any number of which are allowed for each field operand. Positionals are the following:

   # *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 variable name, or an expression in parentheses. The expression
must evaluate to a positive number.

/   instructs IML to go to column 1 of the next row.

@ *operand*   specifies the column position. The *operand* is either a number, a
variable name, or an expression in parentheses. The @ directive
should come after the pound sign (#) positional, if it is specified.

+ *operand*   instructs IML to skip columns. The *operand* is either a number, a
variable name, or an expression in parentheses.

Field 0perands

The *field-operand* specifies what goes in the field. It is either a character literal in
quotes or the name of a character variable.

Formats

The *format* is the format used for display, for the value, and also as the informat
applied to entered values. If no format is specified, the standard numeric or character
format is used.

Field Options

The *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* can be either 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. The default is 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.

Using the DISPLAY Statement

After you have opened a window with the WINDOW statement, you can use the
DISPLAY statement to display the fields in the window.

The DISPLAY statement specifies a list of groups to be displayed. Each group is
separated from the next by a comma. The general form of the DISPLAY statement is
as follows.

DISPLAY ⟨*group-spec-1 group-options,*⟨ *. . ., group-spec-n group-options*⟩⟩;

Group Specifications

The group specification names a group, either a compound name of the form *windowname.groupname* or a *windowname* followed by a group defined by fields and enclosed in parentheses. For example, you can specify *windowname.groupname* or *windowname*(*field-specs*), where *field-specs* are as defined earlier for the WINDOW statement.

In the example, you used the statement

```
display maillist.addr;
```

to display the window MAILLIST and the group ADDR.

Group Options

The *group-options* can be any of the following:

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

NOINPUT
requests that the group be displayed with all the fields protected so that no data entry can be done.

REPEAT
specifies that the group be repeated for each element of the matrices specified as *field-operands*. See "Repeating Fields" later in this chapter.

Details about Windows

The following sections discuss some of the ideas behind windows.

The Number and Position of Windows

You can have any number of windows. They can overlap each other or be disjoint. Each window behaves independently from the others. You can specify the starting size, position, and color of the window when you create it. Each window responds to SAS windowing environment commands so that it can be moved, sized, popped, or changed in color dynamically by the user.

You can list all active windows in a session by using the SHOW WINDOWS command. This makes it easy to keep track of multiple windows.

Windows and the Display Surface

A window is really a viewport into a display. The display can be larger or smaller than the window. If the display is larger than the window, you can use scrolling commands to move the surface under the window (or equivalently, move the window over the display surface). The scrolling commands are as follows:

RIGHT $< n >$	scrolls right.
LEFT $< n >$	scrolls left.
FORWARD $< n >$	scrolls forward (down).
BACKWARD $< n >$	scrolls backward (up).
TOP	scrolls to the top of the display surface.
BOTTOM	scrolls to the bottom of the display surface.

The argument n is an optional numeric argument that indicates the number of positions to scroll. The default is 5.

Only one window is active at a time. You can move, zoom, enlarge, shrink, or recolor inactive windows, but you cannot scroll or enter data.

Each display starts with the same standard lines: first a command line for entering commands, then a message line for displaying messages (such as error messages).

The remainder of the display is up to you to design. You can put fields in any positive row and column position of the display surface, even if it is off the displayed viewport.

Deciding Where to Define Fields

You have a choice of whether to define your fields in the WINDOW statement, the DISPLAY statement, or both. Defining field groups in the WINDOW statement saves work if you access the window from many different DISPLAY statements. Specifying field groups in the DISPLAY statement provides more flexibility.

Groups of Fields

All fields must be part of field groups. The group is just a mechanism to treat multiple fields together as a unit in the DISPLAY statement. There is only one rule about the field positions of different groups: active fields must not overlap. Overlapping is acceptable among fields as long as they are not simultaneously active. Active fields are the ones that are specified together in the current DISPLAY statement.

You name groups specified in the WINDOW statement. You specify groups in the DISPLAY statement just by putting them in parentheses; they are not named.

Field Attributes

There are two types of fields you can define:

- Protected fields are for constants on the screen.
- Unprotected fields accept data entry.

If the field consists of a character string in quotes, it is protected. If the field is a variable name, it is not protected unless you specify PROTECT=YES as a field option. If you want all fields protected, specify the NOINPUT group option in the DISPLAY statement.

Display Execution

When you execute a DISPLAY statement, the SAS System displays the window with all current values of the variables. You can then enter data into the unprotected fields. All the basic editing keys (cursor controls, delete, end, insert, and so forth) work, as well as SAS windowing environment commands to scroll or otherwise manage the window. Control does not return to the IML code until you enter a command on the command line that is not recognized as a SAS windowing environment command. Typically, a SUBMIT command is used since most users define a function key for this command. Before control is returned to you, IML moves all modified field values from the screen back into IML variables using standard or specified informat routines. If you have specified the CMNDLINE= option in the WINDOW statement, the current command line is passed back to the specified variable.

The window remains visible with the last values entered until the next DISPLAY statement or until the window is closed by a WINDOW statement with the CLOSE= option.

Only one window is active at a time. Every window may be subject to SAS windowing environment commands, but only the window specified in the current DISPLAY statement transfers data to IML.

Each window is composed dynamically every time it is displayed. If you position fields by variables, you can make them move to different parts of the screen simply by programming the values of the variables.

The DISPLAY statement even allows general expressions in parentheses as positional or field operands. The WINDOW statement only allows literal constants or variable names as operands. If a field operand is an expression in parentheses, then it is always a protected field. You cannot use the statement

```
display w(log(X));
```

and expect it to return the log function of the data entered. You would need the following code to do that:

```
lx=log(x);
display w(lx);
```

Field Formatting and Inputting

The length of a field on the screen is specified in the format after the field operand, if you give one. If a format is not given, IML uses standard character or numeric formats and informats. Numeric informats allow scientific notation and missing values (represented with periods). The default length for character variables is the size of the variable element. The default size for numeric fields is as given with the FW= option (see the discussion of the RESET statement in Chapter 20, "Language Reference.")

If you specify a named format (such as DATE7.), IML attempts to use it for both the output format and the input informat. If IML cannot find an input informat of that name, it uses the standard informats.

Display-only Windows

If a window consists only of protected fields, it is merely displayed; that is, it does not wait for user input. These display-only windows can be displayed rapidly.

Opening Windows

The WINDOW statement is executable. When a WINDOW statement is executed, it checks to see if the specific window has already been opened. If it has not been opened, then the WINDOW statement opens it; otherwise, the WINDOW statement does nothing.

Closing Windows

To close a window, use the CLOSE= option in the WINDOW statement. In the example given earlier, you closed MAILLIST with the statement

```
window close=maillist;
```

Repeating Fields

If you specify an operand for a field that is a multi-element matrix, the routines deal with the first value of the matrix. However, there is a special group option, REPEAT, that enables you to display and retrieve values from all the elements of a matrix. If the REPEAT option is specified, IML determines the maximum number of elements of any field-operand matrix, and then it repeats the group that number of times. If any field operand has fewer elements, the last element is repeated the required number of times (the last one becomes the data entered). Be sure to write your specifications so that the fields do not overlap. If the fields overlap, an error message results. Although the fields must be matrices, the positional operands are never treated as matrices.

The repeat feature can come in very handy in situations where you want to menu a list of items. For example, suppose you want to build a restaurant billing system and you have stored the menu items and prices in the matrices ITEM and PRICE. You want to obtain the quantity ordered in a matrix called AMOUNT. Enter the following code:

```
item={ "Hamburger", "Hot Dog", "Salad Bar", "Milk" };
price={1.10 .90 1.95 .45};
amount= repeat(0,nrow(item),1);
window menu
group=top
#1 @2 "Item" @44 "Price" @54 "Amount"
group=list
/ @2 item $10. @44 price 6.2 @54 amount 4.
;
display menu.top, menu.list repeat;
```

This defines the window

```
+-----Menu------------------------------------+
+   Command --->                              +
+                                             +
+  Item                Price    Amount        +
+                                             +
+  Hamburger           1.10       0           +
+  Hot Dog             0.90       0           +
+  Salad Bar           1.95       0           +
+  Milk                0.45       0           +
+                                             +
+--------------------------------------------- +
```

Example

This example illustrates the following features:

- multiple windows
- the repeat feature
- command- and message-line usage
- a large display surface needing scrolling
- windows linked to data set transactions

This example uses two windows, FIND and ED. The FIND window instructs you to enter a name. Then a data set is searched for all the names starting with the entered value. If no observations are found, you receive the following message:

```
Not found, enter request
```

If any observations are found, they are displayed in the ED window. You can then edit all the fields. If several observations are found, you need to use the scrolling commands to view the entire display surface. If you enter the SUBMIT command, the data are updated in place in the data set. Otherwise, you receive the following message:

```
Not replaced, enter request
```

If you enter a blank field for the request, you are advised that EXIT is the keyword needed to exit the system.

```
start findedit;
    window ed rows=10 columns=40 icolumn=40 cmndline=c;
    window find rows=5 columns=35 icolumn=1 msgline=msg;
    edit user.class;
    display ed ( "Enter a name in the FIND window, and this"
    / "window will display the observations "
    / "starting with that name. Then you can"
    / "edit them and enter the submit command"
    / "to replace them in the data set. Enter cancel"
    / "to not replace the values in the data set."
    /
    / "Enter exit as a name to exit the program." );
    do while(1);
       msg=' ';
       again:
       name=" ";
       display find ("Search for name: " name);
       if name=" " then
          do;
              msg='Enter exit to end';
              goto again;
          end;
       if name="exit" then goto x;
       if name="PAUSE" then
          do;
              pause;
              msg='Enter again';
              goto again;
          end;
       find all where(name=:name) into p;
       if nrow(p)=0 then
          do;
              msg='Not found, enter request';
              goto again;
          end;
       read point p;
       display ed (//" name: " name
          " sex: " sex
          " age: " age
          /" height: " height
          " weight: " weight ) repeat;
       if c='submit' then
          do;
              msg="replaced, enter request";
              replace point p;
          end;
       else
          do;
             msg='Not replaced, enter request';
          end;
    end;
```

```
     x:
     display find ("Closing Data Set and Exiting");
     close user.class;
     window close=ed;
     window close=find;
  finish findedit;
  run findedit;
```

Chapter 14
Storage Features

Chapter Contents

Chapter 14
Storage Features

Overview

SAS/IML software can store user-defined modules and the values of matrices in special library storage on disk for later retrieval. The library storage feature enables you to perform the following tasks:

- store and reload IML modules and matrices
- save work for a later session
- keep records of work
- conserve space by saving large, intermediate results for later use
- communicate data to other applications through the library
- store and retrieve data in general

Storage Catalogs

SAS/IML storage catalogs are specially structured SAS files that are located in a SAS data library. A SAS/IML catalog contains *entries* that are either matrices or modules. Like other SAS files, SAS/IML catalogs have two-level names in the form *libref.catalog*. The first-level name, *libref*, is a name assigned to the SAS data library to which the catalog belongs. The second-level name, *catalog*, is the name of the catalog file.

The default libref is initially SASUSER, and the default catalog is IMLSTOR. Thus, the default storage catalog is called SASUSER.IMLSTOR. You can change the storage catalog with the RESET STORAGE command (see the discussion of the RESET statement in Chapter 20, "Language Reference.")

Using this command, you can change either the catalog or the libref.

When you store a matrix, IML automatically stores the matrix name, its type, its dimension, and its current values. Modules are stored in the form of their compiled code. Once modules are loaded, they do not need to be parsed again, making their use very efficient.

Catalog Management

IML provides you with all the commands necessary to reference a particular storage catalog, to list the modules and matrices in that catalog, to store and remove modules and matrices, and to load modules and matrices back to IML. The following commands enable you to perform all necessary catalog management functions:

LOAD	recalls entries from storage.
REMOVE	removes entries from storage.
RESET STORAGE	specifies the library name.
SHOW STORAGE	lists all entries currently in storage.
STORE	saves modules or matrices to storage.

Restoring Matrices and Modules

You can restore matrices and modules from storage back into the IML active workspace using the LOAD command. The LOAD command has the general form

LOAD ;

LOAD *matrices***;**

LOAD MODULE= *module***;**

LOAD MODULE=(*modules***);**

LOAD MODULE=(*modules***)** *matrices***;**

Some examples of valid LOAD commands are shown below:

```
load a b c;                      /* load matrices A,B, and C   */
load module=mymod1;              /* load module MYMOD1         */
load module=(mymod1 mymod2) a b; /* load modules and matrices  */
```

The special operand _ALL_ can be used to load all matrices or modules, or both. For example, if you want to load all modules, use the statement

```
load module=_all_;
```

If you want to load all matrices and modules in storage, use the LOAD command by itself:

```
load;                 /* loads all matrices and modules  */
```

The LOAD command can be used with the STORE statement to save and restore an IML environment between sessions.

Removing Matrices and Modules

You can remove modules or matrices from the catalog using the REMOVE command. The REMOVE command has the same form as the LOAD command. Some examples of valid REMOVE statements are shown below:

```
remove a b c;                      /* remove matrices A,B, and C */
remove module=mymod1;                    /* remove module MYMOD1 */
remove module=(mymod1 mymod2) a; /* remove modules and matrices */
```

The special operand _ALL_ can be used to remove all matrices or modules, or both. For example, if you want to remove all matrices, use the statement

```
remove _all_;
```

If you want to remove everything from storage, use the REMOVE command by itself:

```
remove;
```

Specifying the Storage Catalog

To specify the name of the storage catalog, use one of the following general forms of the STORAGE= option in the RESET statement:

RESET STORAGE= *catalog*;

RESET STORAGE= *libref.catalog*;

Each time you specify the STORAGE= option, the previously opened catalog is closed before the new one is opened.

You can have any number of catalogs, but you can have only one open at a time. A SAS data library can contain many IML storage catalogs, and an IML storage catalog can contain many entries (that is, many matrices and modules).

For example, you can change the name of the storage catalog without changing the libref with the statement

```
reset storage=mystor;
```

To change the libref as well, use the statement

```
reset storage=mylib.mystor;
```

Listing Storage Entries

You can list all modules and matrices in the current storage catalog using the SHOW STORAGE command, which has the general form

SHOW STORAGE ;

Storing Matrices and Modules

You can save modules or matrices in the storage catalog using the STORE command. The STORE command has the same general form as the LOAD command. Several examples of valid STORE statements are shown below:

```
store a b c;                      /* store matrices A,B, and C    */
store module=mymod1;              /* store module MYMOD1          */
store module=(mymod1 mymod2) a; /* storing modules and matrices */
```

The special operand _ALL_ can be used to store all matrices or modules. For example, if you want to store everything, use the statement

```
store _all_ module=_all_;
```

Alternatively, to store everything, you can also enter the STORE command by itself:

```
store;
```

This can help you to save your complete IML environment before exiting an IML session. Then you can use the LOAD statement in a subsequent session to restore the environment and resume your work.

Chapter 15
Using SAS/IML Software to Generate IML Statements

Chapter Contents

Chapter 15
Using SAS/IML Software to Generate IML Statements

Overview

This chapter describes ways of using SAS/IML software to generate and execute statements from within the Interactive Matrix Language. You can execute statements generated at run time, execute global SAS commands under program control, or create statements dynamically to get more flexibility.

Generating and Executing Statements

You can push generated statements into the input command stream (queue) with the PUSH, QUEUE, and EXECUTE subroutines. This can be very useful in situations that require added flexibility, such as menu-driven applications or interrupt handling.

The PUSH command inserts program statements at the front of the input command stream, whereas the QUEUE command inserts program statements at the back. In either case, if they are not input to an interactive application, the statements remain in the queue until IML enters a pause state, at which point they are executed. The pause state is usually induced by a program error or an interrupt control sequence. Any subsequent RESUME statement resumes execution of the module from the point where the PAUSE command was issued. For this reason, the last statement put into the command stream for PUSH or QUEUE is usually a RESUME command.

The EXECUTE statement also pushes program statements like PUSH and QUEUE, but it executes them immediately and returns. It is not necessary to push a RESUME statement when you use the CALL EXECUTE command.

Executing a String Immediately

The PUSH, QUEUE, and EXECUTE commands are especially useful when used in conjunction with the pause and resume features because they enable you to generate a pause-interrupt command to execute the code you push and return from it via a pushed RESUME statement. In fact, this is precisely how the EXECUTE subroutine is implemented generally.

CAUTION: Note that the push and resume features work this way only in the context of being inside modules. You cannot resume an interrupted sequence of statements in immediate mode, that is, not inside a module.

For example, suppose that you collect program statements in a matrix called CODE. You push the code to the command input stream along with a RESUME statement and then execute a PAUSE statement. The PAUSE statement interrupts the execution,

parses and executes the pushed code, and returns to the original execution via the RESUME statement.

```
proc iml;
start testpush;
   print '*** ENTERING MODULE TESTPUSH ***';
   print '*** I should be 1,2,3: ';
   /* constructed code * /
   code = ' do i = 1 to 3; print i; end;   ';
   /* push code+resume */
   call push (code, 'resume;');
   /* pause interrupt  */
   pause;
   print '*** EXITING MODULE TESTPUSH ***';
finish;
```

When the PAUSE statement interrupts the program, the IML procedure then parses and executes the line:

```
do i=1 to 3; print i; end; resume;
```

The RESUME command then causes the IML procedure to resume the module that issued the PAUSE.

Note: The EXECUTE routine is equivalent to a PUSH command, but it also adds the push of a RESUME command, then issues a pause automatically.

A CALL EXECUTE command should be used only from inside a module because pause and resume features do not support returning to a sequence of statements in immediate mode.

Feeding an Interactive Program

Suppose that an interactive program gets responses from the statement INFILE CARDS. If you want to feed it under program control, you can push lines to the command stream that is read.

For example, suppose that a subroutine prompts a user to respond **YES** before performing some action. If you want to run the subroutine and feed the **YES** response without the user being bothered, you push the response as follows:

```
   /* the function that prompts the user */
start delall;
   file log;
   put 'Do you really want to delete all records? (yes/no)';
   infile cards;
   input answer $;
   if upcase(answer)='YES' then
      do;
         delete all;
         purge;
         print "*** FROM DELALL:
```

```
              should see End of File (no records to list)";
              list all;
          end;
    finish;
```

The latter DO group is necessary so that the pushed **YES** is not read before the RUN statement. The following example illustrates the use of the module DELALL given above:

```
    /* Create a dummy data set for delall to delete records  */
xnum = {1 2 3, 4 5 6, 7 8 0};
create dsnum1 from xnum;
append from xnum;
    do;
        call push ('yes');
        run delall;
    end;
```

Calling the Operating System

Suppose that you want to construct and execute an operating system command. Just push it to the token stream in the form of an X statement and have it executed under a pause interrupt.

The following module executes any system command given as an argument:

```
start system(command);
    call push(" x '",command,"'; resume;");
    pause;
finish;
run system('listc');
```

The call generates and executes a LISTC command under MVS:

```
x 'listc'; resume;
```

Calling the SAS Windowing Environment

The same strategy used for calling the operating system works for SAS global statements as well, including calling the SAS windowing environment by generating DM statements.

The following subroutine executes a SAS windowing environment command:

```
start dm(command);
    call push(" dm '",command,"'; resume;");
    pause;
finish;
```

```
run dm('log; color source red');
```

The call generates and executes the statements

```
dm 'log; color source red'; resume;
```

which take you to the LOG window, where all source code is written in red.

Executing Any Command in an EXECUTE Call

The EXECUTE command executes the statements contained in the arguments using the same facilities as a sequence of CALL PUSH, PAUSE, and RESUME statements. The statements use the same symbol environment as that of the subroutine that calls them. For example, consider the following subroutine:

```
proc iml;
start exectest;
/*    IML STATEMENTS  */
   call execute ("xnum = {1 2 3, 4 5 6, 7 8 0};");
   call execute ("create dsnum1 from xnum;");
   call execute ("append from xnum;");
   call execute ("print 'DSNUM should have 3 obs and 3 var:';");
   call execute ("list all;");
/*    global (options) statement  */
   call execute ("options linesize=68;");
   call execute ("print 'Linesize should be 68';");
finish;
run exectest;
```

The output generated from EXECTEST is exactly the same as if you had entered the statements one at a time:

```
            DSNUM should have 3 obs and 3 var:

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

                    Linesize should be 68
```

CALL EXECUTE could almost be programmed in IML as shown here; the difference between this and the built-in command is that the following subroutine would not necessarily have access to the same symbols as the calling environment:

```
start execute(command1,...);
   call push(command1,...," resume;");
   pause;
finish;
```

Making Operands More Flexible

Suppose that you want to write a program that prompts a user for the name of a data set. Unfortunately the USE, EDIT, and CREATE commands expect the data set name as a hardcoded operand rather than an indirect one. However, you can construct and execute a function that prompts the user for the data set name for a USE statement.

```
/* prompt the user to give dsname for use statement */
start flexible;
   file log;
   put 'What data set shall I use?';
   infile cards;
   input dsname $;
  call execute('use', dsname, ';');
finish;
run flexible;
```

If you enter USER.A, the program generates and executes the line

```
use user.a;
```

Interrupt Control

Whenever a program error or interrupt occurs, IML automatically issues a pause, which places the module in a paused state. At this time, any statements pushed to the input command queue get executed. Any subsequent RESUME statement (including pushed RESUME statements) resume executing the module from the point where the error or interrupt occurred.

If you have a long application such as reading a large data set and you want to be able to find out where the data processing is just by entering a break-interrupt (sometimes called an attention signal), you push the interrupt text. The pushed text can, in turn, push its own text on each interrupt, followed by a RESUME statement to continue execution.

For example, suppose you have a data set called TESTDATA that has 4096 observations. You want to print the current observation number if an attention signal is given. The following code does this:

```
start obsnum;
   use testdata;
   brkcode={"print 'now on observation number',i;"
            "if (i<4096) then do;"
            "call push(brkcode);"
            "resume;"
            "end;"
            };
   call push(brkcode);
   do i=1 to 4096;
      read point i;
```

```
      end;
   finish;
   run obsnum;
```

After the module has been run, enter the interrupt control sequence for your operating system. Type S to suspend execution. The IML procedure prints a message telling which observation is being processed. Because the pushed code is executed at the completion of the module, the message is also printed when OBSNUM ends.

Each time the attention signal is given, OBSNUM executes the code contained in the variable BRKCODE. This code prints the current itteration number and pushes commands for the next interrupt. Note that the PUSH and RESUME commands are inside a DO group, making them conditional and ensuring that they are parsed before the effect of the PUSH command is realized.

Specific Error Control

A PAUSE command is automatically issued whenever an execution error occurs, putting the module in a holding state. If you have some way of checking for specific errors, you can write an interrupt routine to correct them during the pause state.

In this example, if a singular matrix is passed to the INV function, the IML procedure pauses and executes the pushed code to make the result for the inverse be set to missing values. The code uses the variable SINGULAR to detect if the interrupt occurred during the INV operation.

This is particularly necessary because the pushed code is executed on completion of the routine, as well as on interrupts.

```
   proc iml;
   a = {3 3, 3 3};                              /* singular matrix */
   /* If a singular matrix is sent to the INV function,        */
   /* IML normally sets the resulting matrix to be empty       */
   /* and prints an error message.                             */
   b = inv(a);
   print "*** A should be non-singular", a;
   start singtest;
      msg="    Matrix is singular - result set to missing ";
      onerror=
         "if singular then do; b=a#.; print msg; print b;
          resume; end;";
      call push(onerror);
      singular = 1;
      b = inv(a);
      singular = 0;
   finish ;
   call singtest;
```

The resulting output is shown below:

```
   ERROR: (execution) Matrix should be non-singular.
```

```
        Error occurred in module SINGTEST at line      67 column    9
        operation : INV                        at line     67 column   16
        operands  : A

 A                 2 rows      2 cols     (numeric)

             3          3
             3          3

   stmt: ASSIGN                         at line     67 column    9

   Paused in module SINGTEST.


              MSG
                  Matrix is singular - result set to missing


                                    B
                                    .        .
                                    .        .


   Resuming execution in module SINGTEST.
```

General Error Control

Sometimes, you may want to process or step over errors. To do this, put all the code into modules and push a code to abort if the error count goes above some maximum. Often, you may submit a batch job and get a trivial mistake that causes an error, but you do not want to cause the whole run to fail because of it. On the other hand, if you have many errors, you do not want to let the routine run.

In the following example, up to three errors are tolerated. A singular matrix **A** is passed to the INV function, which would, by itself, generate an error message and issue a pause in the module. This module pushes three RESUME statements, so that the first three errors are tolerated. Messages are printed and execution is resumed. The DO loop in the module OOPS is executed four times, and on the fourth iteration, an ABORT statement is issued and you exit IML.

```
proc iml;
a={3 3, 3 3};                               /* singular matrix */
   /*                                                          */
   /* GENERAL ERROR CONTROL -- exit iml for 3 or more errors */
   /*                                                          */
 start;                             /* module will be named MAIN */
    errcode = {" if errors >= 0 then do;",
               "    errors = errors + 1;",
               "    if errors > 2 then abort;",
               "    else do; call push(errcode); resume; end;",
               " end;" } ;
    call push (errcode);
    errors = 0;
    start oops;                             /* start module OOPS */
```

```
      do i = 1 to 4;
          b = inv(a);
      end;
   finish;                                /* finish OOPS */
   run oops;
finish;                                   /* finish MAIN */
errors=-1;                                /* disable     */
run;
```

The output generated from this example is shown below:

```
   ERROR: (execution) Matrix should be non-singular.

   Error occurred in module OOPS     at line    41 column  17
   called   from    module MAIN      at line    44 column  10
   operation : INV                   at line    41 column  24
   operands  : A

 A               2 rows      2 cols     (numeric)

         3          3
         3          3

  stmt: ASSIGN                       at line    41 column  17

  Paused in module OOPS.

  Resuming execution in module OOPS.
  ERROR: (execution) Matrix should be non-singular.

   Error occurred in module OOPS     at line    41 column  17
   called   from    module MAIN      at line    44 column  10
   operation : INV                   at line    41 column  24
   operands  : A

 A               2 rows      2 cols     (numeric)

         3          3
         3          3

  stmt: ASSIGN                       at line    41 column  17

  Paused in module OOPS.

  Resuming execution in module OOPS.
  ERROR: (execution) Matrix should be non-singular.
   Error occurred in module OOPS     at line    41 column  17
   called   from    module MAIN      at line    44 column  10
   operation : INV                   at line    41 column  24
   operands  : A

  A               2 rows      2 cols     (numeric)
```

```
          3          3
          3          3

   stmt: ASSIGN                              at line     41 column   17

   Paused in module OOPS.
   Exiting IML.
```

Actually, in this particular case it would probably be simpler to put three RESUME statements after the RUN statement to resume execution after each of the first three errors.

Macro Interface

The pushed text is scanned by the macro processor; therefore, the text can contain macro instructions. For example, here is an all-purpose routine that shows what the expansion of any macro is, assuming that it does not have embedded double quotes:

```
     /* function: y = macxpand(x);                        */
     /* will macro-process the text in x,                 */
     /* and return the expanded text in the result.       */
     /* Do not use double quotes in the argument          */
     /*                                                    */
start macxpand(x);
   call execute('Y="',x,'";');
   return(y);
finish;
```

Consider the following statements:

```
%macro verify(index);
   data _null_;
      infile junk&index;
      file print;
      input;
      put _infile_;
   run;
%mend;
y = macxpand('%verify(1)');
print y;
```

The output produced is shown below:

```
     Y

     DATA _NULL_;        INFILE JUNK1;        FILE PRINT;        INPUT;
     PUT _INFILE_;        RUN;
```

IML Line Pushing Contrasted with Using the Macro Facility

The SAS macro language is a language embedded in and running on top of another language; it generates text to feed the other language. Sometimes it is more convenient to generate the text using the primary language directly rather than embedding the text generation in macros. The preceding examples show that this can even be done at execution time, whereas pure macro processing is done only at parse time. The advantage of the macro language is its embedded, yet independent, nature: it needs little quoting, and it works for all parts of the SAS language, not just IML. The disadvantage is that it is a separate language that has its own learning burden, and it uses extra reserved characters to mark its programming constructs and variables. Consider the quoting of IML versus the embedding characters of the macro facility: IML makes you quote every text constant, whereas the macro facility makes you use the special characters percent sign (%) and ampersand (&) on every macro item. There are some languages, such as REXX, that give you the benefits of both (no macro characters and no required quotes), but the cost is that the language forces you to discipline your naming so that names are not expanded inadvertently.

Example 15.1. Full-Screen Editing

The ability to form and submit statements dynamically provides a very powerful mechanism for making systems flexible. For example, consider the building of a data entry system for a file. It is straightforward to write a system using WINDOW and DISPLAY statements for the data entry and data processing statements for the I/O, but once you get the system built, it is good only for that one file. With the ability to push statements dynamically, however, it is possible to make a system that dynamically generates the components that are customized for each file. For example, you can change your systems from static systems to dynamic systems.

To illustrate this point, consider an IML system to edit an arbitrary file, a system like the FSEDIT procedure in SAS/FSP software but programmed in IML. You cannot just write it with open code because the I/O statements hardcode the filenames and the WINDOW and DISPLAY statements must hardcode the fields. However, if you generate just these components dynamically, the problem is solved for any file, not just one.

```
proc iml;
/*                  FSEDIT                                */
/* This program defines and stores the modules FSEINIT,   */
/* FSEDT, FSEDIT, and FSETERM in a storage catalog called */
/* FSED.  To use it, load the modules and issue the command */
/* RUN FSEDIT;  The system prompts or menus the files and  */
/* variables to edit, then runs a full screen editing      */
/* routine that behaves similar to PROC FSEDIT             */
/*                                                        */
/* These commands are currently supported:                */
/*                                                        */
/* END        gets out of the system.  The user is prompted */
/*            as to whether or not to close the files and  */
/*            window.                                      */
/* SUBMIT     forces current values to be written out,     */
```

Example 15.1. Full-Screen Editing ⬩ 463

```
/*              either to append a new record or replace     */
/*              existing ones                                */
/* ADD          displays a screen variable with blank values */
/*              for appending to the end of a file           */
/* DUP          takes the current values and appends them to */
/*              the end of the file                          */
/* number       goes to that line number                     */
/* DELETE       deletes the current record after confirmation */
/*              by a Y response                              */
/* FORWARD1     moves to the next record, unless at eof       */
/* BACKWARD1    moves to the previous record, unless at eof   */
/* EXEC         executes any IML statement                    */
/* FIND         finds records and displays them              */
/*                                                           */
/* Use: proc iml;                                            */
/*         reset storage='fsed';                             */
/*         load module=_all_;                                */
/*         run fsedit;                                       */
/*                                                           */
/*---routine to set up display values for new problem---     */
start fseinit;
   window fsed0 rows=15 columns=60 icolumn=18 color='GRAY'
   cmndline=cmnd group=title +30 'Editing a data set' color='BLUE';
   /*---get file name---                                     */
   _file="                  ";
   msg =
      'Please Enter Data Set Name or Nothing For Selection List';
   display  fsed0.title,
      fsed0 ( / @5  'Enter Data Set:'
               +1  _file
               +4  '(or nothing to get selection list)' );
   if _file=' ' then
      do;
         loop:
         _f=datasets(); _nf=nrow(_f); _sel=repeat("_",_nf,1);
         display fsed0.title,
            fsed0 (/ "Select? File Name"/) ,
            fsed0 (/ @5 _sel +1 _f protect=yes ) repeat ;
         _l = loc(_sel^='_');
         if nrow(_l)^=1 then
            do;
               msg='Enter one S somewhere';
               goto loop;
            end;
         _file = _f[_l];
      end;
   /*---open file, get number of records---                  */
   call queue(" edit ",_file,";
               setin ",_file," NOBS _nobs; resume;"); pause *;
   /*---get variables---                                     */
   _var = contents();
   _nv = nrow(_var);
   _sel = repeat("_",_nv,1);
   display fsed0.title,
           fsed0 (/ "File:" _file) noinput,
           fsed0 (/ @10 'Enter S to select each var, or select none
                    to get all.'
                    // @3  'select?  Variable  '  ),
                    fsed0 ( / @5  _sel  +5 _var protect=yes ) repeat;
```

```
      /*---reopen if subset of variables---                        */
      if any(_sel^='_') then
         do;
            _var = _var[loc(_sel^='_')];
            _nv = nrow(_var);
            call push('close ',_file,'; edit ',_file,' var
            _var;resume;');pause *;
         end;
      /*---close old window---                                      */
      window close=fsed0;
      /*---make the window---*/
      call queue('window fsed columns=55 icolumn=25 cmndline=cmnd
                  msgline=msg ', 'group=var/@20 "Record " _obs
                  protect=yes');
      call queue( concat('/"',_var,': " color="YELLOW" ',
                  _var,' color="WHITE"'));
      call queue(';');
      /*---make a missing routine---*/
      call queue('start vmiss; ');
      do i=1 to _nv;
         val = value(_var[i]);
         if type(val)='N' then call queue(_var[i],'=.;');
         else call queue(_var[i],'="',
                         cshape(' ',1,1,nleng(val)),'";');
      end;
      call queue('finish; resume;');
      pause *;
      /*---initialize current observation---*/
      _obs = 1;
      msg = Concat('Now Editing File ',_file);
   finish;
    /*                                                              */
     /*---The Editor Runtime Controller---                          */
   start fsedt;
      _old = 0; go=1;
      do while(go);
      /*--get any needed data--*/
         if any(_obs^=_old) then do; read point _obs; _old = _obs;
      end;
      /*---display the record---*/
      display fsed.var repeat;
      cmnd = upcase(left(cmnd));
      msg=' ';
      if cmnd='END' then go=0;
      else if cmnd='SUBMIT' then
         do;
            if _obs<=_nobs then
               do;
                  replace point _obs; msg='replaced';
               end;
            else do;
                    append;
                    _nobs=_nobs+nrow(_obs);
                    msg='appended';
                 end;
         end;
      else if cmnd="ADD" then
          do;
             run vmiss;
```

Example 15.1. Full-Screen Editing ◆ 465

```
              _obs = _nobs+1;
               msg='New Record';
           end;
      else if cmnd='DUP' then
         do;
             append;
             _nobs=_nobs+1;
             _obs=_nobs;
             msg='As Duplicated';
         end;
      else if cmnd>'0' & cmnd<'999999' then
         do;
             _obs = num(cmnd);
             msg=concat('record number ',cmnd);
         end;
      else if cmnd='FORWARD1' then _obs=min(_obs+1,_nobs);
      else if cmnd='BACKWARD1' then _obs=max(_obs-1,1);
      else if cmnd='DELETE' then
         do;
             records=cshape(char(_obs,5),1,1);
             msg=concat('Enter command Y to Confirm delete of'
                         ,records);
             display fsed.var repeat;
             if (upcase(cmnd)='Y') then
                do;
                    delete point _obs;
                    _obs=1;
                    msg=concat('Deleted Records',records);
                end;
             else msg='Not Confirmed, Not Deleted';
         end;
      else if substr(cmnd,1,4)='FIND' then
         do;
             call execute("find all where(",
                            substr(cmnd,5),
                            ") into _obs;" );
              _nfound=nrow(_obs);
              if _nfound=0 then
                 do;
                     _obs=1;
                     msg='Not Found';
                 end;
               else
                  do;
                      msg=concat("Found ",char(_nfound,5)," records");
                  end;
         end;
      else if substr(cmnd,1,4)='EXEC' then
         do;
             msg=substr(cmnd,5);
             call execute(msg);
         end;
      else msg='Unrecognized Command; Use END to exit.';
      end;
finish;
  /*---routine to close files and windows, clean up---*/
start fseterm;
     window close=fsed;
     call execute('close ',_file,';');
```

```
      free _q;
finish;
    /*---main routine for FSEDIT---*/
start fsedit;
  if (nrow(_q)=0) then
     do;
        run fseinit;
     end;
  else msg = concat('Returning to Edit File ',_file);
  run fsedt;
  _q='_';
  display fsed ( "Enter 'q' if you want to close files and windows"
                 _q " (anything else if you want to return later"
                 pause 'paused before termination';
  run fseterm;
finish;
reset storage='fsed';
store module=_all_;
```

Summary

In this chapter you learned how to use SAS/IML software to generate IML statements. You learned how to use the PUSH, QUEUE, EXECUTE, and RESUME commands to interact with the operating system or with the SAS windowing environment. You also saw how to add flexibility to programs by adding interrupt control features and by modifying error control. Finally, you learned how IML compares to the SAS macro language.

Chapter 16
Wavelet Analysis

Chapter Contents

Chapter 16
Wavelet Analysis

Overview

Wavelets are a versatile tool for understanding and analyzing data, with important applications in nonparametric modeling, pattern recognition, feature identification, data compression, and image analysis. Wavelets provide a description of your data that localizes information at a range of scales and positions. Moreover, they can be computed very efficiently, and there is an intuitive and elegant mathematical theory to guide you in applying them.

Some Brief Mathematical Preliminaries

The discrete wavelet transform decomposes a function as a sum of basis functions called wavelets. These basis functions have the property that they can be obtained by dilating and translating two basic types of wavelets known as the *scaling function* or *father wavelet* ϕ, and the *mother wavelet* ψ. These translates and dilations are defined as follows:

$$\phi_{j,k}(x) = 2^{j/2}\phi(2^j x - k)$$
$$\psi_{j,k}(x) = 2^{j/2}\psi(2^j x - k)$$

The index j defines the dilation or *level* while the index k defines the translate. Loosely speaking, sums of the $\phi_{j,k}(x)$ capture low frequencies and sums of the $\psi_{j,k}(x)$ represent high frequencies in the data. More precisely, for any suitable function $f(x)$ and for any j_0,

$$f(x) = \sum_k c_k^{j_0}\phi_{j_0,k}(x) + \sum_{j \geq j_0}\sum_k d_k^j\psi_{j,k}(x)$$

where the c_k^j and d_k^j are known as the scaling coefficients and the detail coefficients respectively. For orthonormal wavelet families these coefficients can be computed by

$$c_k^j = \int f(x)\phi_{j,k}(x)\,dx$$
$$d_k^j = \int f(x)\psi_{j,k}(x)\,dx$$

The key to obtaining fast numerical algorithms for computing the detail and scaling coefficients for a given function $f(x)$ is that there are simple recurrence relationships

that enable you to compute the coefficients at level $j-1$ from the values of the scaling coefficients at level j. These formulae are

$$c_k^{j-1} = \sum_i h_{i-2k} c_i^j$$

$$d_k^{j-1} = \sum_i g_{i-2k} c_i^j$$

The coefficients h_k and g_k that appear in these formulae are called *filter coefficients*. The h_k are determined by the father wavelet and they form a low-pass filter; $g_k = (-1)^k h_{1-k}$ and form a high-pass filter. The preceding sums are formally over the entire (infinite) range of integers. However, for wavelets that are zero except on a finite interval, only finitely many of the filter coefficients are non-zero and so in this case the sums in the recurrence relationships for the detail and scaling coefficients are finite.

Conversely, if you know the detail and scaling coefficients at level $j-1$ then you can obtain the scaling coefficients at level j using the relationship

$$c_k^j = \sum_i h_{k-2i} c_i^{j-1} + \sum_i g_{k-2i} d_i^{j-1}$$

Suppose that you have data values

$$y_k = f(x_k), \qquad k = 0, 1, 2, \cdots, N-1$$

at $N = 2^J$ equally spaced points x_k. It turns out that the values $2^{-J/2} y_k$ are good approximations of the scaling coefficients c_k^J. Then using the recurrence formula you can find c_k^{J-1} and d_k^{J-1}, $k = 0, 1, 2, \cdots, N/2 - 1$. The discrete wavelet transform of the y_k at level $J-1$ consists of the $N/2$ scaling and $N/2$ detail coefficients at level $J-1$. A technical point that arises is that in applying the recurrence relationships to finite data, a few values of the c_k^J for $k < 0$ or $k \geq N$ may be needed. One way to cope with this difficulty is to extend the sequence c_k^J to the left and right using some specified boundary treatment.

Continuing by replacing the scaling coefficients at any level j by the scaling and detail coefficients at level $j-1$ yields a sequence of N coefficients

$$\{c_0^0, d_0^0, d_0^1, d_1^1, d_0^2, d_1^2, d_2^2, d_3^2, d_1^3, \ldots, d_7^3, \ldots, d_0^{J-1}, \ldots, d_{N/2-1}^{J-1}\}$$

This sequence is the finite discrete wavelet transform of the input data $\{y_k\}$. At any level j_0 the finite dimensional approximation of the function $f(x)$ is

$$f(x) \approx \sum_k c_k^{j_0} \phi_{j_0,k}(x) + \sum_{j=j_0}^{J-1} \sum_k d_k^j \psi_{j,k}(x)$$

Getting Started

Fourier Transform Infrared (FT-IR) spectroscopy is an important tool in analytic chemistry. This example demonstrates wavelet analysis applied to an FT-IR spectrum of quartz (Sullivan 2000). The following DATA step creates a data set containing the spectrum, expressed as an absorbance value for each of 850 wave numbers.

```
data quartzInfraredSpectrum;
   WaveNumber=4000.6167786 - _N_ *4.00084378;
   input Absorbance @@;
datalines;
 4783  4426  4419  4652  4764  4764  4621  4475  4430  4618
 4735  4735  4655  4538  4431  4714  4738  4707  4627  4523
 4512  4708  4802  4811  4769  4506  4642  4799  4811  4732
 4583  4676  4856  4868  4796  4849  4829  4677  4962  4994
 4924  4673  4737  5078  5094  4987  4632  4636  5010  5166
 5166  4864  4547  4682  5161  5291  5143  4684  4662  5221
 5640  5640  5244  4791  4832  5629  5766  5723  5121  4690
 5513  6023  6023  5503  4675  5031  6071  6426  6426  5723
 5198  5943  6961  7135  6729  5828  6511  7500  7960  7960
 7299  6484  7257  8180  8542  8537  7154  7255  8262  8898
 8898  8263  7319  7638  8645  8991  8991  8292  7309  8005
 9024  9024  8565  7520  7858  8652  8966  8966  8323  7513
 8130  8744  8879  8516  7722  8099  8602  8729  8726  8238
 7885  8350  8600  8603  8487  7995  8194  8613  8613  8408
 7953  8236  8696  8696  8552  8102  7852  8570  8818  8818
 8339  7682  8535  9038  9038  8503  7669  7794  8864  9163
 9115  8221  7275  8012  9317  9317  8512  7295  7623  9021
 9409  9338  8116  6860  7873  9282  9490  9191  7012  7392
 9001  9483  9457  8107  6642  7695  9269  9532  9246  7641
 6547  8886  9457  9457  8089  6535  7537  9092  9406  9178
 7591  6470  7838  9156  9222  7974  6506  7360  8746  9057
 8877  7455  6504  7605  8698  8794  8439  7057  7202  8240
 8505  8392  7287  6634  7418  8186  8229  7944  6920  6829
 7499  7949  7831  7057  6866  7262  7626  7626  7403  6791
 7062  7289  7397  7397  7063  6985  7221  7221  7199  6977
 7088  7380  7380  7195  6957  6847  7426  7570  7508  6952
 6833  7489  7721  7718  7254  6855  7132  7914  8040  7880
 7198  6864  7575  8270  8229  7545  7036  7637  8470  8570
 8364  7591  7413  8195  8878  8878  8115  7681  8313  9102
 9185  8981  8283  8197  8932  9511  9511  9101  8510  8670
 9686  9709  9504  8944  8926  9504  9964  9964  9627  9212
 9366  9889 10100  9939  9540  9512  9860 10121 10121  9828
 9567  9513  9782  9890  9851  9510  9385  9339  9451  9451
 9181  9076  9015  8960  9014  8957  8760  8760  8602  8584
 8584  8459  8469  8373  8279  8327  8282  8341  8341  8155
 8260  8260  8250  8350  8245  8358  8403  8355  8490  8490
 8439  8689  8689  8621  8680  8661  8897  9028  8900  8873
 8873  9187  9377  9377  9078  9002  9147  9635  9687  9535
 9127  9242  9824  9928  9775  9200  9047  9572 10102 10102
 9631  9024  9209 10020 10271  9830  9062  9234 10154 10483
10453  9582  9011  9713 10643 10701 10372  9368  9857 10865
10936 10572  9574  9691 10820 11452 11452 10623  9903 10787
```

```
      11931 12094 11302 10604 11458 12608 12808 12589 11629 11795
      12863 13575 13575 12968 12498 13268 14469 14469 13971 13727
      14441 15334 15515 15410 14986 15458 16208 16722 16722 16618
      17061 17661 18089 18089 18184 18617 19015 19467 19633 19830
      20334 20655 20947 21347 21756 22350 22584 22736 22986 23412
      24126 24498 24501 24598 24986 25729 26356 26356 26271 26754
      27624 28162 28162 28028 28305 29223 30073 30219 30185 30308
      31831 32699 32819 32793 33320 34466 35600 36038 36086 36518
      37517 38765 39462 39681 40209 41243 42274 42772 42876 43172
      43929 44842 45351 45395 45551 46035 46774 47353 47353 47362
      47908 48539 48936 48978 49057 49497 50101 50670 50914 51134
      51603 52276 53007 53399 53769 54281 54815 54914 55365 55874
      56180 56272 56669 57076 57422 57458 57525 57681 57679 57318
      57318 57181 57417 57409 57144 57047 56377 56551 56483 56098
      56034 55598 55364 55364 55146 54904 54990 55501 55533 55362
      54387 55340 55240 54748 53710 55346 55795 55795 55060 55945
      55945 55753 56759 56859 57509 56741 56273 56961 58566 58566
      58104 59275 59275 59051 59090 59461 60362 60560 61103 61272
      61380 61878 62067 62237 62214 61182 61532 62173 62253 60473
      61346 63143 63378 61519 61753 63078 63841 63841 62115 61227
      63237 63237 61338 63951 63951 63604 63633 64625 65135 64976
      63630 63494 63834 63338 63218 62324 64131 64234 65122 64551
      64127 64415 64621 64621 63142 65344 65585 65476 65074 64714
      63803 65085 65085 65646 65646 64851 65390 65390 64997 65541
      65587 65682 65952 65952 65390 65702 65846 65734 65734 65628
      65509 65571 65636 65636 65620 65487 65544 65547 65738 65758
      65711 65360 65362 65362 65231 65333 65453 65473 65435 65302
      65412 65412 65351 65242 65242 65170 65221 65297 65297 65202
      65177 65183 65184 65179 65209 65209 65144 65134 65113 65009
      64919 64945 64988 64988 64856 64686 64529 64370 64282 64233
      64169 63869 63685 63480 63373 63349 63307 63131 63017 62885
      62736 62736 62706 62666 62622 62671 62781 62853 62950 63106
      63135 63141 63220 63263 63489 63807 63966 64132 64294 64612
      64841 64985 65159 65204 65259 65540 65707 65749 65732 65719
      65820 65895 65925 65925 65888 65937 66059 66109 66109 66078
      66007 65897 65897 65747 65490 64947 64598 64363 64140 63801
      63571 63395 63333 63442 63442 63339 63196 62911 62118 61795
      61454 61456 61607 62025 62190 62190 62023 61780 61502 61482
      61458 61320 61015 60852 60708 60684 60522 60488 60506 60640
      60797 60995 61141 61141 61036 60664 60522 60017 59681 59129
      58605 58035 57192 56137 54995 53586 52037 50283 48565 45419
      43341 41111 36131 35377 34431 31679 29237 26898 24655 22417
      19876 17244 15176 12575 10532  8180  6040  4059  2210   575
   ;
```

The following statements produce the line plot of these data displayed in Figure 16.1.

```
   symbol1 c=black i=join v=none;
 proc gplot data=quartzInfraredSpectrum;
    plot Absorbance*WaveNumber/
        hminor = 0    vminor = 0
        vaxis  = axis1
        hreverse frame;
```

```
        axis1 label = ( r=0 a=90 );
    run;
```
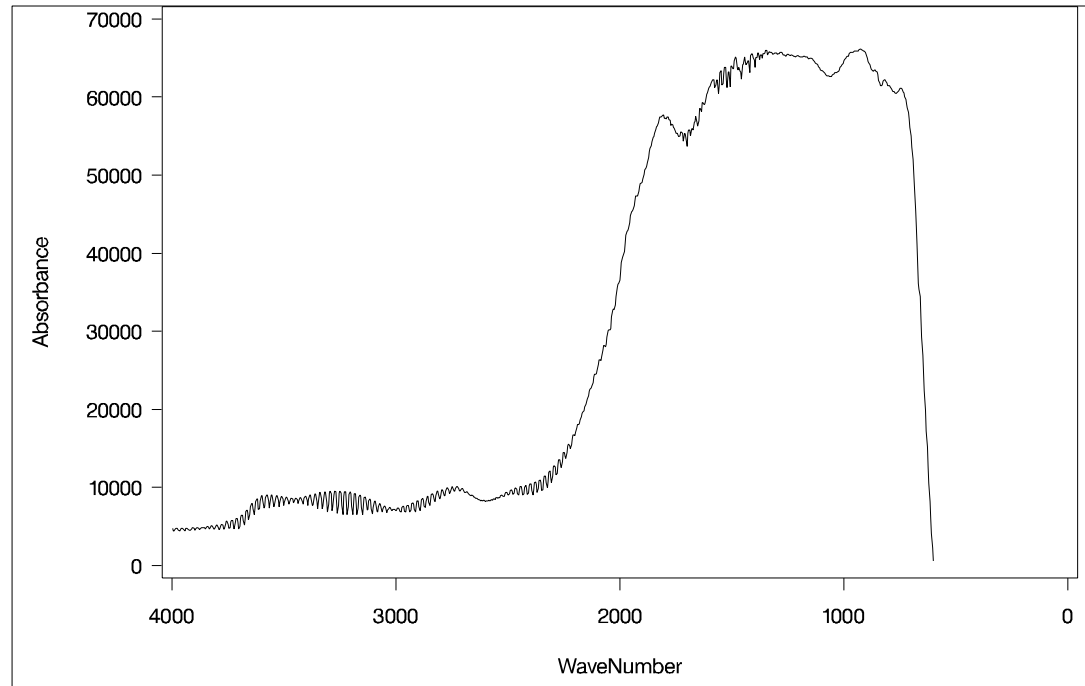

Figure 16.1. FT-IR Spectrum of Quartz

This data contains information at two distinct scales, namely a low frequency underlying curve superimposed with a high frequency oscillation. Notice that the oscillation is not uniform but that it occurs in several distinct bands. Wavelet analysis is an appropriate tool for providing insight into this type of data as it enables you to identify the frequencies present in the absorbance data as the wave number changes. This property of wavelets is known as "time frequency localization"; in this case the role of time is played by WaveNumber. Also note that the dependent variable Absorbance is measured at equally spaced values of the independent variable WaveNumber. This condition is necessary for the direct use of the discrete wavelet transform that is implemented in the SAS/IML wavelet functions.

Creating the Wavelet Decomposition

The following SAS code starts the wavelet analysis:

```
    %wavginit;
    proc iml;
        %wavinit;
```

Notice that the previous code segment includes two SAS macro calls. You can use the IML wavelet functions without using the WAVGINIT and WAVINIT macros. The

macros are called to initialize and load IML modules that you can use to produce several standard wavelet diagnostic plots. These macros have been provided as autocall macros that you can invoke directly in your SAS code.

The WAVGINIT macro must be called prior to invoking PROC IML. This macro defines several macro variables that are used to adjust the size, aspect ratio, and font size for the plots produced by the wavelet plot modules. This macro can also take several optional arguments that control the positioning and and size of the wavelet diagnostic plots. See the "Obtaining Help for the Wavelet Macros and Modules" section on page 491 for details on getting help about this macro call.

The WAVINIT macro must be called from within PROC IML. It loads the IML modules that you can use to produce wavelet diagnostic plots. This macro also defines symbolic macro variables that you can use to improve the readability of your code.

The following statements read the absorbance variable into an IML vector:

```
use quartzInfraredSpectrum;
read all var{Absorbance} into absorbance;
```

You are now in a position to begin the wavelet analysis. The first step is to set up the options vector that specifies which wavelet and what boundary handling you want to use. You do this as follows:

```
optn              =   &waveSpec;    /* optn=j(1,4,.); */
optn[&family]     =   &daubechies;  /* optn[3] = 1;   */
optn[&member]     =   3;            /* optn[4] = 3;   */
optn[&boundary]   =   &polynomial;  /* optn[1] = 3;   */
optn[&degree]     =   &linear;      /* optn[2] = 1;   */
```

These statements use macro variables that are defined in the WAVINIT macro. The equivalent code without using these macro variables is given in the adjacent comments. As indicated by the suggestive macro variable names, this options vector specifies that the wavelet to be used is the third member of the Daubechies wavelet family and that boundaries are to be handled by extending the signal as a linear polynomial at each endpoint.

The next step is to create the wavelet decomposition with the following call:

```
call wavft(decomp,absorbance,optn);
```

This call computes the wavelet transform specified by the vector optn of the input vector absorbance. The specified transform is encapsulated in the vector decomp. This vector is not intended to be used directly. Rather you use this vector as an argument to other IML wavelet subroutines and plot modules. For example, you use the WAVPRINT subroutine to print the information encapsulated in a wavelet decomposition. The following code produces output in Figure 16.2.

```
call wavprint(decomp,&summary);
call wavprint(decomp,&detailCoeffs,1,4);
```

```
                        Decomposition Summary

        Decomposition Name                          DECOMP
        Wavelet Family            Daubechies Extremal Phase
        Family Member                                    3
        Boundary Treatment     Recursive Linear Extension
        Number of Data Points                          850
        Start Level                                      0


                Wavelet Detail Coefficients for DECOMP

  Translate         Level 1          Level 2          Level 3          Level 4

          0      -1.71343E-9     1.36819E-10      -6.6097E-12      5.23868E-11
          1       1340085.30      -128245.70       191.084707          4501.36
          2                         62636.70          6160.27         -1358.23
          3                       -238445.36        -54836.56      -797.724143
          4                                          39866.95       676.034389
          5                                         -28836.85         -5166.59
          6                                         223421.00         -6088.99
          7                                                          -5794.67
          8                                                          30144.74
          9                                                          -3903.53
         10                                                        638.063264
         11                                                         -10803.45
         12                                                          33616.35
         13                                                         -50790.30
```

Figure 16.2.   Output of WAVPRINT CALLS

Usually such displayed output is of limited use. More frequently you will want to
represent the transformed data graphically or use the results in further computational
routines. As an example, you can estimate the noise level of the data using a robust
measure of the standard deviation of the highest level detail coefficients, as demon-
strated in the following statements:

```
call wavget(tLevel,decomp,&topLevel);
call wavget(noiseCoeffs,decomp,&detailCoeffs,tLevel-1);

noiseScale=mad(noiseCoeffs,"nmad");
print "Noise scale = " noiseScale;
```

The result is shown in Figure 16.3;

```
                              NOISESCALE

                    Noise scale =   169.18717
```

Figure 16.3.   Scale of Noise in the Absorbance Data

The first WAVGET call is used to obtain the top level number in the wavelet decom-
position decomp. The highest level of detail coefficients are defined at one level

below the top level in the decomposition. The second WAVGET call returns these coefficients in the vector noiseCoeffs. Finally, the MAD function computes a robust estimate of the standard deviation of these coefficients.

Wavelet Coefficient Plots

Diagnostic plots greatly facilitate the interpretation of a wavelet decomposition. One standard plot is the detail coefficients arranged by level. Using a module included by the WAVINIT macro call, you can produce the plot shown in Figure 16.5 as follows:

```
call coefficientPlot(decomp, , , , ,"Quartz Spectrum");
```

The first argument specifies the wavelet decomposition and is required. All other arguments are optional and need not be specified. You can use the WAVHELP macro to obtain a description of the arguments of this and other wavelet plot modules. The WAVHELP macro is defined in autocall the WAVINIT macro. For example, invoking the WAVHELP macro as follows writes the calling information shown in Figure 16.4 to the SAS log.

```
%wavhelp(coefficientPlot);
```

```
           coefficientPlot Module

  Function: Plots wavelet detail coefficients

  Usage: call coefficientPlot(decomposition,
                              threshopt,
                              startLevel,
                              endLevel,
                              howScaled,
                              header);

  Arguments:
     decomposition - (required) valid wavelet decompostion produced
                                by the IML subroutine WAVFT
     threshopt     - (optional) numeric vector of 4 elements
                                specifying thresholding to be used
                                Default: no thresholding
     startLevel    - (optional) numeric scalar specifying the lowest
                                level to be displayed in the plot
                                Default: start level of decomposition
     endLevel      - (optional) numeric scalar specifying the highest
                                level to be displayed in the plot
                                Default: end level of decomposition
     howScaled     - (optional) character: 'absolute' or 'uniform'
                                specifies coefficients are scaled uniformly
                                Default: independent level scaling
     header        - (optional) character string specifying a header
                                Default: no header
```

Figure 16.4. Log Output Produced by %wavhelp(coefficientPlot) Call
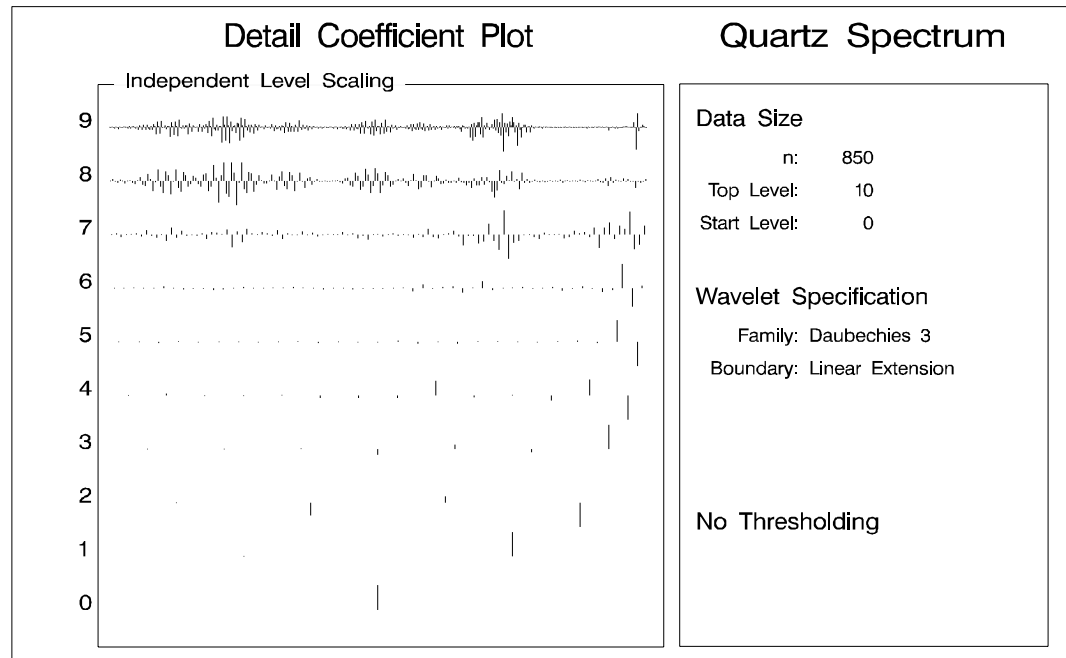
Figure 16.5. Detail Coefficients Scaled by Level

In this plot the detail coefficients at each level are scaled independently. The oscillations present in the absorbance data are captured in the detail coefficients at levels 7, 8, and 9. The following statement produces a coefficient plot of just these higher level detail coefficients and shows them scaled uniformly.

```
call coefficientPlot(decomp, ,7, ,
                     'uniform',"Quartz Spectrum");
```

The plot is shown in Figure 16.6.
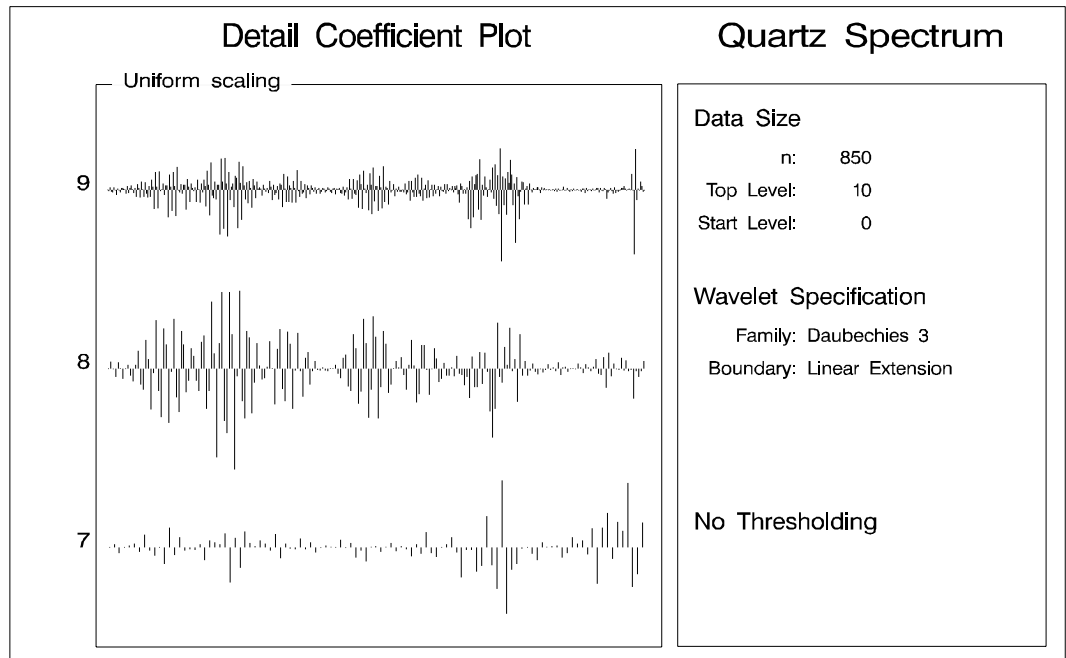
**Figure 16.6.** Uniformly Scaled Detail Coefficients

As noted earlier, noise in the data is captured in the detail coefficients, particularly in the small coefficients at higher levels in the decomposition. By zeroing or shrinking these coefficients, you can get smoother reconstructions of the input data. This is done by specifying a threshold value for each level of detail coefficients and then zeroing or shrinking all the detail coefficients below this threshold value. The IML wavelet functions and modules support several policies for how this thresholding is performed as well as for selecting the thresholding value at each level. See the the "WAVIFT Call" section on page 983 for details.

An options vector is used to specify the desired thresholding; several standard choices are predefined as macro variables in the WAVINIT module. The following statements produce the detail coefficient plot with the "SureShrink" thresholding algorithm of Donoho and Johnstone (1995).

```
call coefficientPlot(decomp,&SureShrink,6,, ,
                     "Quartz Spectrum");
```

The plot is shown in Figure 16.7.
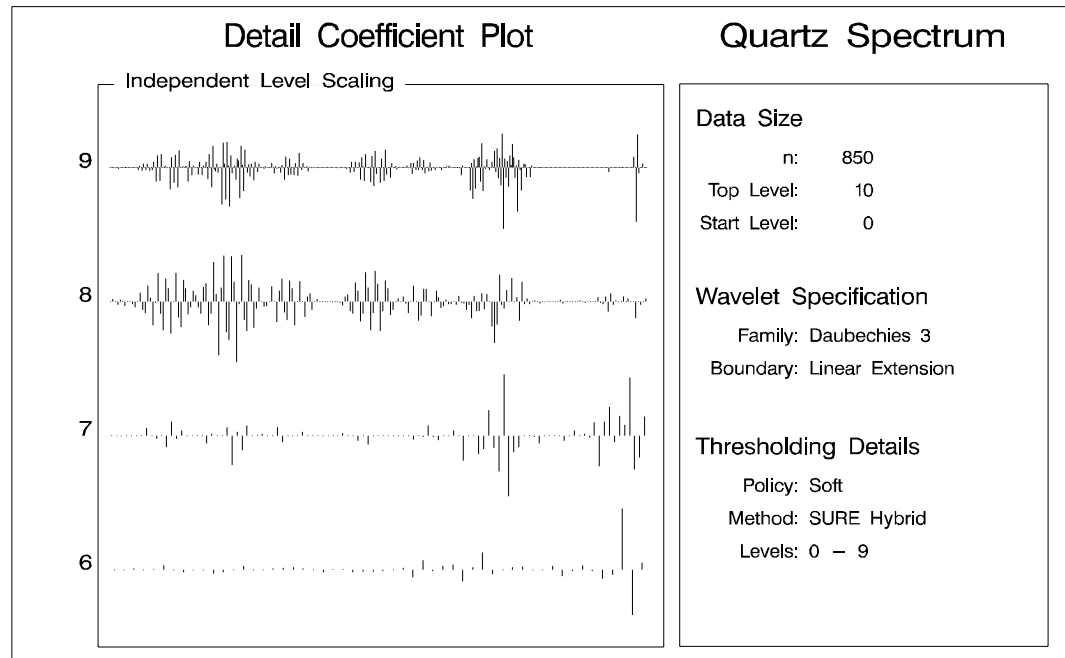
Figure 16.7. Thresholded Detail Coefficients

You can see that "SureShrink" thresholding has zeroed some of the detail coefficients at the higher levels but the larger coefficients that capture the oscillation in the data are still present. Consequently, reconstructions of the the input signal using the thresholded detail coefficients will still capture the essential features of the data, but will be smoother as much of the very fine scale detail has been eliminated.

Multiresolution Approximation Plots

One way of presenting reconstructions is in a multiresolution approximation plot. In this plot reconstructions of the input data are shown by level. At any level the reconstruction at that level uses only the detail and scaling coefficients defined below that level.

The following statement produces such a plot, starting at level 3:

```
call mraApprox(decomp, ,3, ,"Quartz Spectrum");
```

The results are shown in Figure 16.8.

Multiresolution Approximation | Quartz Spectrum
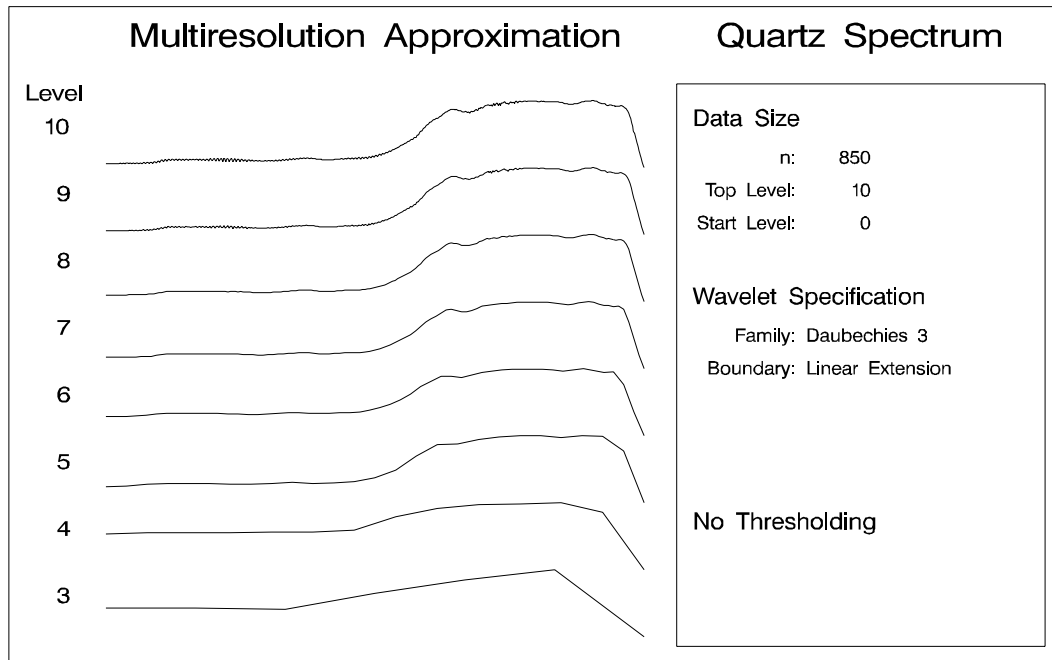
Figure 16.8. Multiresolution Approximation

You can see that even at level 3, the basic form of the input signal has been captured. As noted earlier, the oscillation present in the absorbance data is captured in the detail coefficients above level 7. Thus, the reconstructions at level 7 and below are largely free of these oscillation since they do not use any of the higher detail coefficients. You can confirm this observation by plotting just this level in the multiresolution analysis as follows:

```
call mraApprox(decomp, ,7,7,"Quartz Spectrum");
```

The results are shown in Figure 16.9.
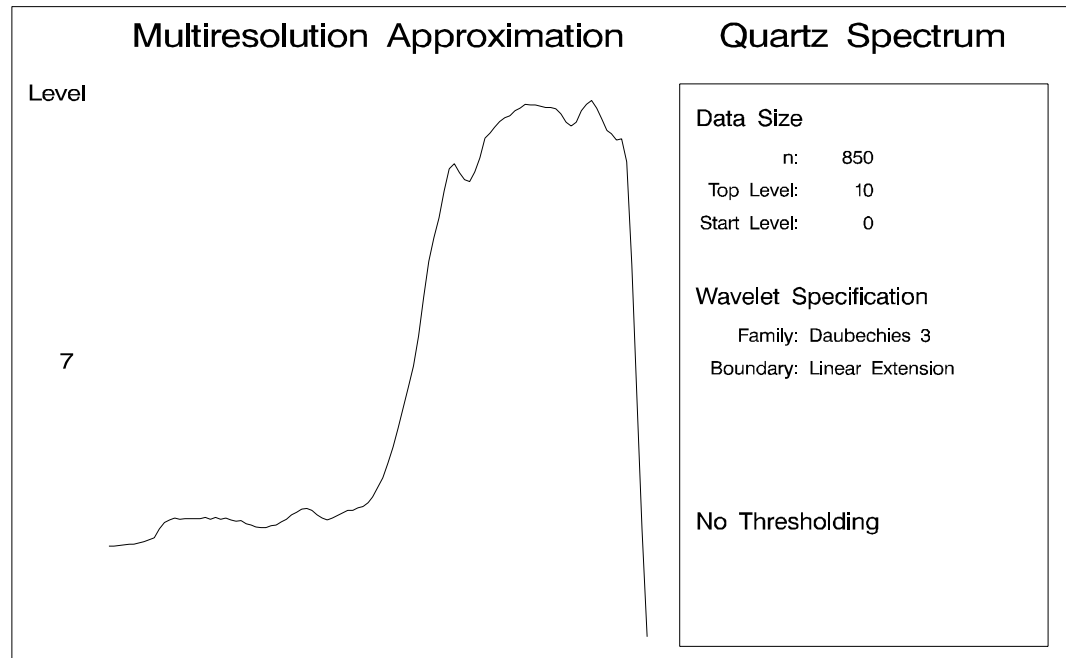
**Figure 16.9.**   Level 7 of the Multiresolution Approximation

You can also plot the multiresolution approximations obtained with thresholded detail coefficients. For example, the following statement plots the top level reconstruction obtained using the "SureShrink" threshold:

```
call mraApprox(decomp,&SureShrink,10,10,
                "Quartz Spectrum");
```

The results are shown in Figure 16.10.
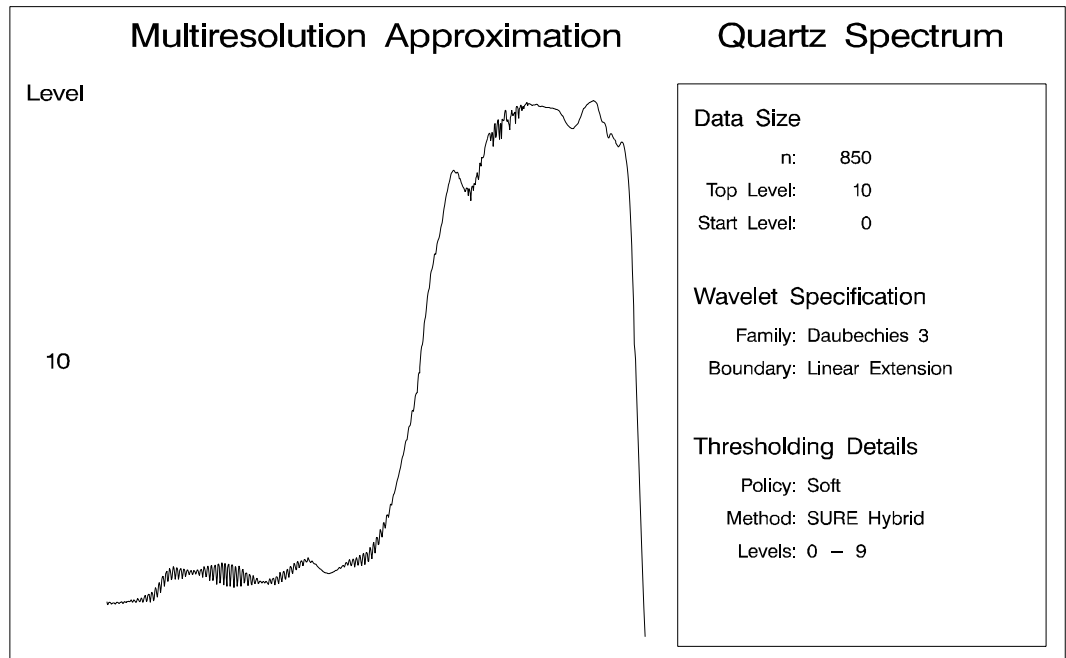
Figure 16.10.　Top Level of Multiresolution Approximation with SureShrink
Thresholding Applied

Note that the high frequency oscillation is still present in the reconstruction even with
"SureShrink" thresholding applied.

Multiresolution Decomposition Plots

A related plot is the multiresolution decomposition plot, which shows the detail coefficients at each level. For convenience, the starting level reconstruction at the lowest level of the plot and the reconstruction at the highest level the plot are also included. Adding suitably scaled versions of all the detail levels to the starting level reconstruction recovers the final reconstruction. The following statement produces such a plot, yielding the results shown in Figure 16.11.

```
call mraDecomp(decomp, ,5, , ,"Quartz Spectrum");
```
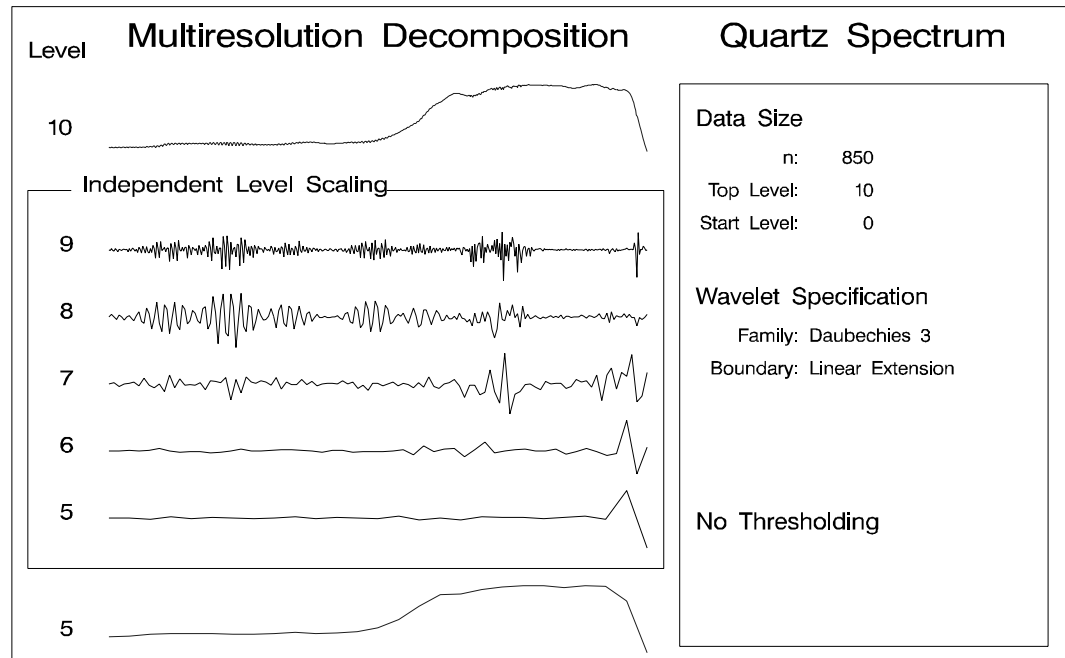
Figure 16.11. Multiresolution Decomposition

Wavelet Scalograms

Wavelet scalograms communicate the time frequency localization property of the discrete wavelet transform. In this plot each detail coefficient is plotted as a filled rectangle whose color corresponds to the magnitude of the coefficient. The location and size of the rectangle are related to the time interval and the frequency range for this coefficient. Ccoefficients at low levels are plotted as wide and short rectangles to indicate that they localize a wide time interval but a narrow range of frequencies in the data. In contrast, rectangles for coefficients at high levels are plotted thin and tall to indicate that they localize small time ranges but large frequency ranges in the data. The heights of the rectangles grow as a power of 2 as the level increases. If you include all levels of coefficients in such a plot, the heights of the rectangles at the lowest levels are so small that they will not be visible. You can use an option to plot the heights of the rectangles on a logarithmic scale. This results in rectangles of uniform height but requires that you interpret the frequency localization of the coefficients with care.

The following statement produces a scalogram plot of all levels with "SureShrink" thresholding applied:

```
call scalogram(decomp,&SureShrink, , ,0.25,
               'log',"Quartz Spectrum");
```

The sixth argument specifies that the rectangle heights are to be plotted on a logarithmic scale. The role of the fifth argument (0.25) is to amplify the magnitude of the small detail coefficients. This is necessary since the detail coefficients at the lower

levels are orders of magnitude larger than those at the higher levels. The amplification is done by first scaling the magnitudes of all detail coefficients to lie in the interval $[0, 1]$ and then raising these scaled magnitudes to the power 0.25. Note that smaller powers yield larger amplification of the small detail coefficient magnitudes. The default amplification is $1/3$.

The results are shown in Figure 16.12.
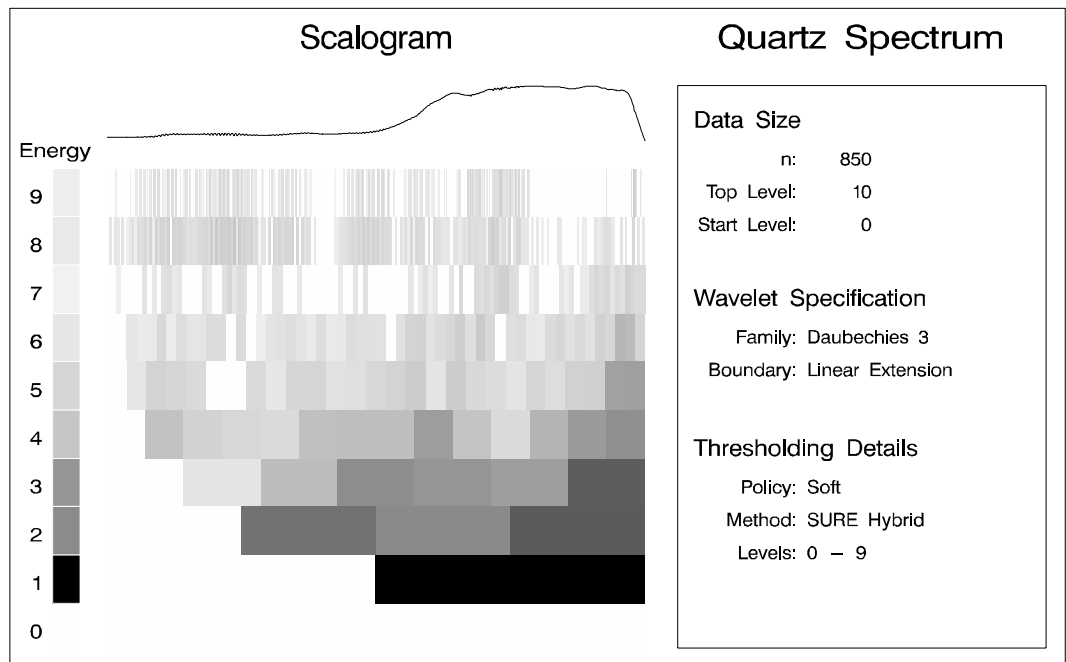
Figure 16.12.   Scalogram Showing All Levels

The bar on the left-hand side of the scalogram plot indicates the overall energy of each level. This energy is defined as the sum of the squares of the detail coefficients for each level. These energies are amplified using the same algorithm for amplifying the detail coefficient magnitudes. The energy bar in Figure 16.12 shows that higher energies occur at the lower levels whose coefficients capture the gross features of the data. In order to interpret the finer-scale details of the data it is helpful to focus on just the higher levels. The following statement produces a scalogram for levels 6 and above without using a logarithmic scale for the rectangle heights, and using the default coefficient amplification.

```
call scalogram(decomp,&SureShrink,6, , , ,
               "Quartz Spectrum");
```

The result is shown in Figure 16.13.
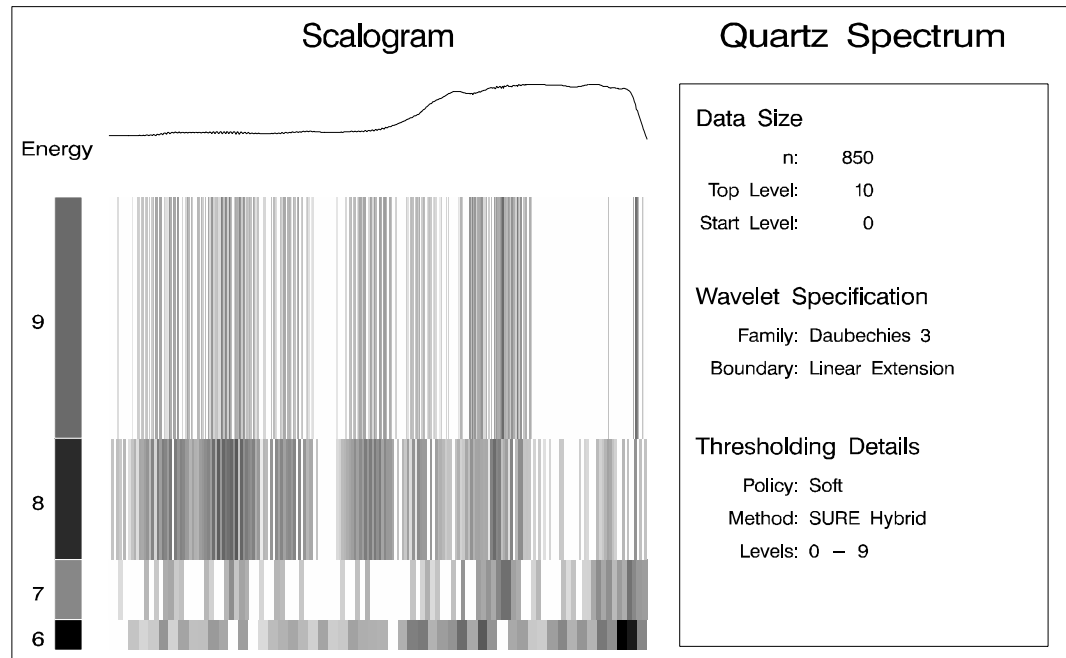
Figure 16.13. Scalogram of Levels 6 and Above Using SureShrink Thresholding

The scalogram in Figure 16.13 reveals that most of the energy of the oscillation in the data is captured in the detail coefficients at level 8. Also note that many of the coefficients at the higher levels are set to zero by "SureShrink" thresholding. You can verify this by comparing Figure 16.13 with Figure 16.14, which shows the corresponding scalogram except that no thresholding is done. The following statement produces Figure 16.14:

```
call scalogram(decomp, ,6, , , ,"Quartz Spectrum");
```
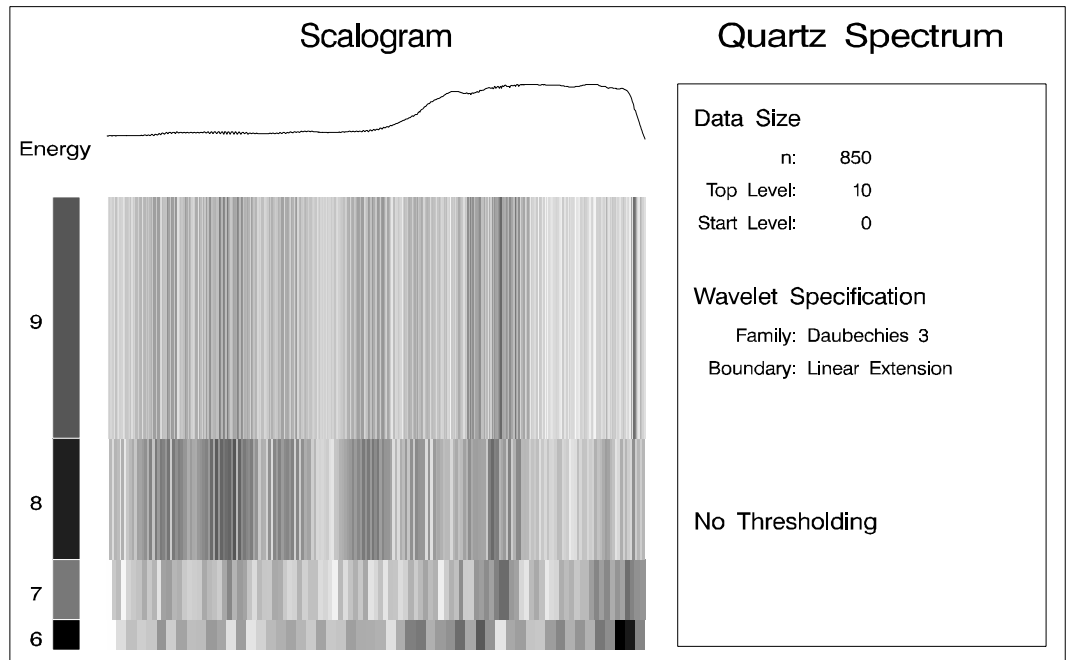
Figure 16.14. Scalogram of Levels 6 and Above Using No Thresholding

Reconstructing the Signal from the Wavelet Decomposition

You can use the WAVIFT subroutine to invert a wavelet transformation computed using the WAVFT subroutine. If no thresholding is specified, then up to numerical rounding error this inversion is exact. The following statements provide an illustration of this:

```
call wavift(reconstructedAbsorbance,decomp);
errorSS=ssq(absorbance-reconstructedAbsorbance);
print "The reconstruction error sum of squares = " errorSS;
```

The output is shown in Figure 16.15.

```
                                          ERRORSS

         The reconstruction error sum of squares =  1.321E-16
```

Figure 16.15.   Exact Reconstruction Property of WAVIFT

Usually you use the WAVIFT subroutine with thresholding specified. This yields a smoothed reconstruction of the input data. You can use the following statements to create a smoothed reconstruction of absorbance and add this variable to the QuartzInfraredSpectrum data set.

```
    call wavift(smoothedAbsorbance,decomp,&SureShrink);
    create temp from smoothedAbsorbance[colname='smoothedAbsorbance'];
        append from smoothedAbsorbance;
    close temp;
quit;

data quartzInfraredSpectrum;
  set quartzInfraredSpectrum;
  set temp;
run;
```

The following statements produce the line plot of the smoothed absorbance data shown in Figure 16.16:

```
symbol1 c=black i=join v=none;
proc gplot data=quartzInfraredSpectrum;
    plot smoothedAbsorbance*WaveNumber/
        hminor = 0    vminor = 0
        vaxis  = axis1
        hreverse frame;
        axis1 label = ( r=0 a=90 );
run;
```
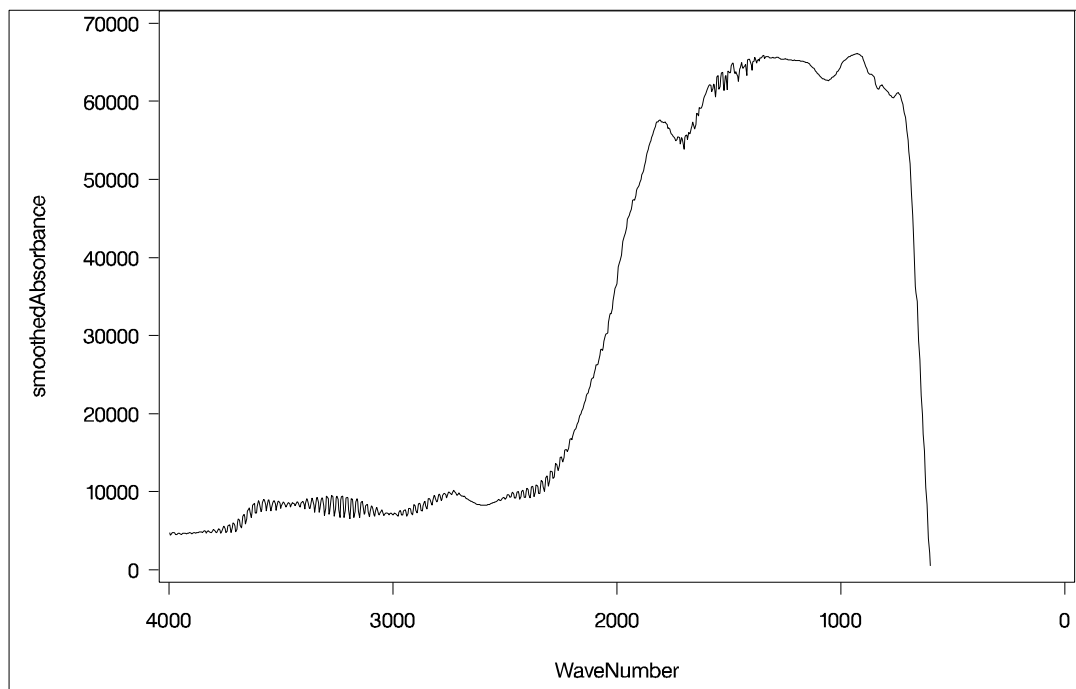
Figure 16.16. Smoothed FT-IR Spectrum of Quartz

You can see by comparing Figure 16.1 with Figure 16.16 that the wavelet smooth of the absorbance data has preserved all the essential features of this data.

Details

Using Symbolic Names

Several of the wavelet subroutines take arguments that are options vectors that specify user input. For example, the third argument in a WAVFT subroutine call is an options vector that specifies which wavelet and which boundary treatment are used in computing the wavelet transform. Typical code that defines this options vector is

```
optn    = j(1, 4, .);
optn[1] = 0;
optn[3] = 1;
optn[4] = 3;
```

A problem with such code is that it is not easily readable. By using symbolic names readability is greatly enhanced. SAS macro variables provide a convenient mechanism for creating such symbolic names. For example, the previous code could be replaced by

```
optn            = &waveSpec;
optn[&family]   = &daubechies;
optn[&member]   = 3;
optn[&boundary] = &zeroExtension;
```

where the symbolic macro variables (names with a preceding ampersand) resolve to the relevant quantities. Another example where symbolic names improve code readability is to use symbolic names for an integer argument that controls what action a multipurpose subroutine performs. An illustration is replacing code such as

```
call wavget(n,decomposition,1);
call wavget(fWavelet,decompostion,8);
```

by

```
call wavget(n,decomposition,&numPoints);
call wavget(fWavelet,decompostion,&fatherWavelet);
```

A set of symbolic names is defined in the autocall WAVINIT macro. The following tables list the symbolic names that are defined in this macro:

Table 16.1. Macro Variables for Wavelet Specification

Position		Admissible Values	
Name	**Value**	**Name**	**Value**
&boundary	1	&zeroExtension	0
		&periodic	1
		&polynomial	2
		&reflection	3
		&antisymmetricReflection	4
&degree	2	&constant	0
		&linear	1
		&quadratic	2
&family	3	&daubechies	1
		&symmlet	2
&member	4	1 - 10	

Table 16.2. Macro Variables for Threshold Specification

Position		Admissible Values	
Name	**Value**	**Name**	**Value**
&policy	1	&none	0
		&hard	1
		&soft	2
		&garrote	3
&method	2	&absolute	0
		&minimax	1
		&universal	2
		&sure	3
		&sureHybrid	4
		&nhoodCoeffs	5
&value	3	*positive real*	
&levels	4	&all	-1
		positive integer	

Table 16.3. Symbolic Names for the Third Argument of WAVGET

Name	**Value**
&numPoints	1
&detailCoeffs	2
&scalingCoeffs	3
&thresholdingStatus	4
&specification	5
&topLevel	6
&startLevel	7
&fatherWavelet	8

Table 16.4. Macro Variables for the Second Argument of WAVPRINT

Name	Value
&summary	1
&detailCoeffs	2
&scalingCoeffs	3
&thresholdedDetailCoeffs	4

Table 16.5. Macro Variables for Predefined Wavelet Specifications

Name		&boundary	&degree	&family	&member	
&waveSpec	{	.	.	.	.	}
&haar	{	&periodic	.	&daubechies	1	}
&daubechies3	{	&periodic	.	&daubechies	3	}
&daubechies5	{	&periodic	.	&daubechies	5	}
&symmlet5	{	&periodic	.	&symmlet	5	}
&symmlet8	{	&periodic	.	&symmlet	8	}

Table 16.6. Macro Variables for Predefined Threshold Specifications

Name		&policy	&method	&value	&levels	
&threshSpec	{	.	.	.	.	}
&RiskShrink	{	&hard	&minimax	.	&all	}
&VisuShrink	{	&soft	&universal	.	&all	}
&SureShrink	{	&soft	&sureHybrid	.	&all	}

Obtaining Help for the Wavelet Macros and Modules

The WAVINIT macro that you call to define symbolic macro variables and wavelet plot modules also defines a macro WAVHELP that you can call to obtain help for the wavelet macros and plot modules. The syntax for calling the WAVHELP macro is

> **%WAVHELP** $<$ *(name)*$>$; ;

where *name* is one of wavginit, wavinit, coefficientPlot, mraApprox, mraDecomp, or scalogram. This macro displays usage and argument information for the specified macro or module. If you call the WAVHELP macro with no arguments, it lists the names of the macros and modules for which help is available. Note that you can obtain help for the built-in IML wavelet subroutines using the SAS Online Help.

References

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.

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.

Mallat, S. (1989), "Multiresolution Approximation and Wavelets," *Transactions of the American Mathematical Society*, 315, 69–88.

Ogden, R.T. (1997), *Essential Wavelets for Statistical Applications and Data Analysis,* Boston: Birkhäuser.

Sullivan, D. (2000), "FT-IR Library," [http://www.che.utexas.edu/~dls/ir/ir_dir.html], accessed 16 October 2000.

Chapter 17
Genetic Algorithms (Experimental)

Chapter Contents

Chapter 17
Genetic Algorithms (Experimental)

Overview

Genetic algorithms (referred to hereafter as GAs) are a family of search algorithms that seek optimal solutions to problems using the principles of natural selection and evolution. GAs can be applied to almost any optimization problem and are especially useful for problems where other calculus-based techniques do not work, such as when the objective function has many local optimums, is not differentiable or continuous, or solution elements are constrained to be integers or sequences. In most cases GAs require more computation than specialized techniques that take advantage of specific problem structure or characteristics. However, for optimization problems with no such techniques available, GAs provide a robust general method of solution. The current GA implementation in IML is experimental, and will be furthur developed and tested in later SAS releases.

In general, GAs use the following procedure to search for an optimum solution:

initialization: An initial population of solutions is randomly generated, and an objective function value is evaluated for each member of the solution population.

regeneration: A new solution population is generated from the current solution population. First, individual members are chosen stochastically to parent the next generation such that those who are the "fittest" (have the best objective function values) are more likely to be picked. This process is called *selection*. Those chosen solutions will either copied directly to the next generation or will be passed to a crossover operator, with a user-specified crossover probabilty. The crossover operator combines two or more parents to produce new offspring solutions for the next generation. A fraction of the next generation solutions, selected according to a user-specified mutation probability, will be passed to a mutation operator which introduces random variations in the solutions.

The crossover and mutation operators are commonly called *genetic operators*. The crossover operator passes characteristics from each parent to the offspring, especially those characteristics shared in common. It is selection and crossover that distinguish GAs from a purely random search, and direct the algorithm toward finding an optimum. Mutation is designed to ensure diversity in the search to prevent premature convergence to a local optimum.

As the final step in regeneration, the current population is replaced by the new solutions generated by selection, crossover, and mutation.

The objective function values are evaluated for the new generation. A common variation on this approach that is supported in IML is to pass one or more of the best solutions from the current population on to the next population unchanged. This often leads to faster convergence, and assures that the best solution generated at any time during the optimization is never lost.

repeat: After *regeneration*, the process checks some stopping criteria, such as the number of iterations or some other convergence criteria. If the stopping criteria is not met, then the algorithm loops back to the *regeneration* step.

Although GAs have been demonstrated to work well for a variety of problems, there is no guarantee of convergence to a global optimum. Also, the convergence of GAs can be sensitive to the choice of genetic operators, mutation probability, and selection criteria, so that some initial experimentation and fine-tuning of these parameters is often required.

In the traditional formulation of GAs, the parameter set to be searched is mapped into finite-length bit strings, and the genetic operators applied to these strings, or chromosomes, are based on biological processes. While there is a theoretical basis for the effectiveness of GAs formulated in this way (Goldberg 1989), in practice most problems don't fit naturally into this paradigm. Modern research has shown that optimizations can be set up using the natural solution domain (for example, a real vector or integer sequence) and applying crossover and mutation operators analogous to the traditional genetic operators, but more appropriate to the natural formulation of the problem (Michalewicz 1996). This latter approach is sometimes called *evolutionary computing*. IML implements the evolutionary computing approach because it makes it much easier to formulate practical problems with realistic constraints. Throughout this documentation, the term "genetic algorithm" is to be interpreted as evolutionary computing.

IML provides a flexible framework for implementing GAs, enabling you to write your own modules for the genetic operators and objective function, as well as providing some standard genetic operators that you can specify. This framework will also enable you to introduce some variations to the usual GA, such as adapting the optimization parameters during the optimization, or incorporating some problem-specific local optimizations into the process to enhance convergence.

An IML program to do GA optimization is structured differently from a program doing non-linear optimization with the nlp routines. With the nlp routines, generally a single call is made in which the user specifies the objective and optimization parameters, and that call runs the optimization process to completion. In contrast, to perform a GA optimization you use separate calls to the GA routines to specify the problem encoding (GASETUP), genetic operators (GASETMUT and GASETCRO), objective function (GASETOBJ), and selection criteria (GASETSEL). You then call the GAINIT routine to initialize the problem population. After that, you advance the optimization process by calling GAREGEN (for the regeneration step) within an IML loop. Within the loop you can use GAGETMEM and GAGETVAL calls to retrieve population members and objective function values for examination. This strategy al-

lows you to monitor the convergence of the GA, adjust optimization parameters with GA routine calls within the loop, and exit the loop when the GA is not producing furthur improvement in the objective function. The next section explains the optimization parameters in more detail and gives guidance on how they should be set.

Formulating a Genetic Algorithm Optimization

To formulate a GA in IML you must decide on five basic optimization parameters:

1. *Encoding:* The general structure and form of the solution.
2. *Objective:* The function to be optimized. IML also enables you to specify whether the function is to be minimized or maximized.
3. *Selection:* How members of the current solution population will be chosen to be parents to propagate the next generation.
4. *Crossover:* How the attributes of parent solutions will be combined to produce new offspring solutions.
5. *Mutation:* How random variation will be introduced into the new offspring solutions to maintain genetic diversity.

The following section discusses each of these items in more detail.

Choosing the Problem Encoding

Problem encoding refers to the structure or type of solution space that is to be optimized, such as real-valued fixed-length vectors or integer sequences. IML offers encoding options appropriate to several types of optimization problems.

General Numeric Matrix: With this encoding, solutions can take the form of a numeric matrix of any shape. Also, different solutions can have different dimensions. This is the most flexible option. If you use this encoding, IML makes no assumptions about the form of the solution, so you are required to specify user modules for crossover and mutation operators, and a user module for creating the initial solution population.

Fixed-length Real-valued Row Vector: If you use this encoding, you must also specify the number of components in the solution vector. Using this option, you can use some IML-supplied crossover and mutation operators later, or you can supply custom modules. You can also specify upper and lower bounds for each component in the vector, and IML will generate an initial population for the GA randomly distributed between the bounds. If you don't explicitly set crossover and mutation operators, IML will provide default operators to be used in the optimization. This type of encoding is often used for general non-linear optimization problems.

Fixed-length Integer-valued Row Vector: This option is similar to the fixed-length real-valued encoding already described, except that the IML-supplied genetic operators and initialization process will preserve and generate integer solutions. This type

of encoding might be applicable, for example, in an assignment problem where the positions within the vector represent different tasks, and the integer values represent different machines or other resources that might be applied to each task.

Fixed-length Integer Sequence: In this encoding, each solution is composed of a sequence of integers ranging from 1 to the length of the sequence, with different solutions distinguished by different ordering of the elements. For example, $s1$ and $s2$ are two integer sequences of length 6:

```
s1 = {1 2 3 4 5 6};
s2 = {2 6 5 3 4 1};
```

This type of encoding is often used for routing problems like the Traveling Salesman Problem, where each element represents a city in a circular route, or scheduling problems.

Setting the Objective Function

Before executing a GA, the you must specify the objective function to be optimized. There are currently two options available: a user function module, or an IML-supplied Traveling Salesman Problem (TSP) objective function.

User Function Module: The module must take exactly one parameter, which will be one solution, and return a numeric scalar objective function value. The module can also have a global clause, which may be used to pass in any other information required to determine the objective function value. If global parameters are used, you must be careful about changing them after the optimization has been initialized. If a change in a global parameter affects the objective function values, you must reevaluate the entire solution population (see GAREEVAL call) to ensure that the values are consistent with the changed global parameter.

The solution parameter passed into the routine is also written back out to the solution population when the module exits, so you should take care not to modify the parameter and therefore the solution population unintentionally. However, it is permissible and may prove very effective to add logic to the module to improve the solution through some heuristic technique or local optimization, and deliberately pass that improved solution back to the solution population by updating the parameter before returning. Using this hybrid approach may significantly improve the convergence of the GA, especially in later stages when solutions may be near an optimum.

TSP Objective Function: An objective function for the Traveling Salesman Problem can be specified with integer sequence encoding. For the TSP, a solution sequence represents a circular route. For example, a solution s with the value

```
s = {2 4 3 1 5};
```

represents a route going from location 2 to location 4 to 3 to 1 to 5 and back to 2. You must also specify a cost matrix c, where $c[i,j]$ is the cost of going from location i to location j. The objective function is just the cost of traversing the route determined by s, and is equivalent to the IML code:

```
start TSPObjectiveFunction(s) global(c);
nc = ncol(s);
cost = c[s[nc],s[1]];
do i = 1 to nc-1;
 cost = cost + c[s[i],s[i+1]];
end;
return (cost);
finish;
```

The IML-supplied order crossover operator and invert mutation operator are especially appropriate for the TSP and other routing problems.

Controlling the Selection Process

There are two competing factors that need to be balanced in the selection process, the *selective pressure* and *genetic diversity*. Selective pressure, the tendency to select only the best members of the current generation to propagate to the next, is required to direct the GA to an optimum. Genetic diversity, the maintenance of a diverse solution population, is also required to ensure that the solution space is adequately searched, especially in the earlier stages of the optimization process. Too much selective pressure can lower the genetic diversity so that the global optimum is overlooked and the GA converges to a local optimum. Yet, with too little selective pressure the GA may not converge to an optimum in a reasonable time. A proper balance between the selective pressure and genetic diversity must be maintained for the GA to converge in a reasonable time to a global optimum.

IML offers two variants of a standard technique for the selection process commonly known as *tournament selection* (Miller and Goldberg 1995). In general, the tournament selection process randomly chooses a group of members from the current population, compares their objective values, and picks the one with the best objective value to be a parent for the next generation. Tournament selection was chosen for IML because it is one of the fastest selection methods, and offers you good control over the selection pressure. Other selection methods such as roulette and rank selection may be offered as options in the future.

In the first variant of tournament selection, you can control the selective pressure by specifying the tournament size, the number of members chosen to compete for parenthood in each tournament. This number should be two or greater, with two implying the weakest selection pressure. Tournament sizes from two to ten have been successfully applied to various GA optimizations, with sizes over four to five considered to represent strong selective pressure.

The second variant of tournament selection provides weaker selective pressure than the first variant just described. The tournament size is set at two, and the member with the best objective value is chosen with a probability that you specify. This best-player-wins probability can range from 0.5 to 1.0, with 1.0 implying that the best member is always chosen (equivalent to a conventional tournament of size two) and 0.5 implying an equal chance of either member being chosen (equivalent to pure random selection). Using this option, you could set the best-player-wins probabilty close to 0.5 in the initial stages of the optimization, and gradually increase it to strengthen

the selective pressure as the optimization progresses, in a similar manner to the simulated annealing optimization technique.

Another important selection option supported in IML is the *elite* parameter. If an elite value of n is specified, then the best n solutions will be carried over to the next generation unchanged, with the rest of the new population filled in by tournament selection, crossover, and mutation. Setting the elite parameter to one or greater will therefore guarantee that the best solution is never lost through selection and propagation, which often improves the convergence of the algorithm.

Using Crossover and Mutation Operators

IML enables you to use user modules for crossover and mutation operators, or you may choose from the operators provided by IML. The IML operators are tied to the problem encoding options, and IML will check to make sure a specified operator is appropriate to the problem encoding. You can also turn off crossover, in which case the current population will pass on to the next generation subject only to mutation. Mutation can be turned off by setting the mutation probability to 0.

The IML-supplied genetic operators are described below, beginning with the crossover operators:

simple: This operator is defined for fixed-length integer and real vector encoding. To apply this operator, a position k within a vector of length n is chosen at random, such that $1 \leq k < n$. Then for parents *p1* and *p2* the offspring are

```
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, 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 a is always 1.

two-point: This operator is defined for fixed-length integer and real vector encoding with length $n \geq 3$. To apply this operator, two positions *k1* and *k2* within the vector are chosen at random, such that $1 \leq k1 < k2 < n$. Element values between those positions are swapped between parents. For parents *p1* and *p2* the offspring are

```
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, 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 a is always 1.

arithmetic: This operator is defined for real and integer fixed-length vector encoding. This operator computes offspring of parents *p1* and *p2* as

```
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. It has the advantage that it will always produce feasible offspring for a convex solution space. A disadvantage of this operator is that it will tend to produce offspring toward the interior of the search region, so that it may be less effective if the optimum lies on or near the search region boundary.

heuristic: This operator is defined for real fixed-length vector encoding. It computes the first offspring from the two parents *p1* and *p2* as

```
        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:

```
        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 the solution space has upper and lower bound constraints the offspring will be checked against the bounds, and any component outside its bound will be set equal to that bound. The heuristic operator will perform best when the objective function is smooth, and may not work well if the objective function or its first derivative is discontinuous.

pmatch: The partial match operator is defined for sequence encoding. It 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 cutpoints as indicated:

```
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 (. indicates positions yet to be determined):

```
c1 = {. . 9 3 4 1 . . .};
c2 = {. . 3 4 5 6 . . .};
```

Next, define a mapping according to the two selected segments:

```
9-3, 3-4, 4-5, 1-6
```

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 $5 \rightarrow 4$, $4 \rightarrow 3$, $3 \rightarrow 9$ 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 will tend 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.

order: This operator is defined for sequence encoding. It 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:

```
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 cutpoint, the elements of *p2* in order are (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*, we have:

```
2 8 7 9 1
```

Placing these back in order starting at the second cutpoint yields

```
c1 = {9 1 3 4 5 6 2 8 7};
```

Applying this logic to *c2* yields

```
c2 = {5 6 9 3 4 1 7 8 2};
```

This operator maintains the similarity of the relative order, or adjacency, of the sequence elements of the parents. It is especially effective for circular path-oriented optimizations, such as the Traveling Salesman Problem.

cycle: This operator is defined for sequence encoding. It produces offspring such that the position of each element value in the offspring comes from one of the parents. For example, for 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:

```
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

```
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.

The mutation operators supported by IML are as follows:

uniform: This operator is defined for fixed-length real or integer encoding with specified upper and lower bounds. 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 may prove especially useful in early stages of the optimization, since it will tend 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 hone in on an optimum, the uniform operator may hinder the optimization.

delta: This operator is defined for integer and real fixed-length vector encoding. It first chooses an element of the solution at random, and then perturbs that element by a fixed amount, set by a *delta* input parameter. *delta* has the same dimension as the solution vectors. To apply the mutation, a randomly chosen element *k* of the solution vector *v* is modified such that

```
v[k] = v[k] + delta[k]; /* with probability 0.5 */
        or
v[k] = v[k] - delta[k];
```

If upper and lower bounds are specified for the problem, then $v[k]$ is adjusted as necessary to fit within the bounds. This operator gives you the ability to control the scope of the search with the *delta* vector. One possible strategy is to start with a larger *delta* value, and then reduce it 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.

swap: This operator is defined for sequence problem encoding. It picks two random locations in the solution vector, and swaps their value. You can also specify that multiple swaps be made for each mutation.

invert: This operator is defined for sequence encoding. It 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 IML-supplied crossover and mutation operators that are allowed for each problem encoding are summarized in the following table.

Table 17.1. Valid Genetic Operators for Each Encoding

Encoding	Crossover	Mutation
general	user module	user module
fixed-length real vector	user module simple two-point arithmetic heuristic	user module uniform delta
fixed-length integer vector	user module simple two-point arithmetic	user module uniform delta
fixed-length integer sequence	user module pmatch order cycle	user module swap invert

A user module specified as a crossover operator must be a subroutine with four parameters. The module should compute and return two new offspring solutions in the

first two parameters, based on the two parent solutions, which will be passed into the module in the last two parameters. The module should not modify the parent solutions passed into it. A global clause can be used to pass in any additional information that the module might use.

A user module specified as a mutation operator must be a subroutine with exactly one parameter. When the module is called, the parameter will contain the solution that is to be mutated. The module will be expected to update the parameter with the new mutated value for the solution. As with crossover, a global clause can be used to pass in any additional information that the module might use.

Executing a Genetic Algorithm

Setting Up the IML Program

After you formulate the GA optimization problem as described in the previous section, executing the genetic algorithm in IML is simple and straightforward. Remember that the current GA implementation in IML is experimental, and will be furthur developed and tested in later SAS releases. The following table summarizes the IML GA modules used to set each of the optimization parameters. IML will use reasonable default values for some of the parameters if they are not specified by the GA calls, and these default values are also listed. Parameters shown in italics are not required in all cases.

Table 17.2. Establishing Optimization Parameters

Type	Set By	Parameter	Value
encoding	GASETUP	encoding	0 →general
			1 →fixed-length real
			2 →fixed-length integer
			3 →fixed-length sequence
		size	fixed-length size
		seed	initial random seed
objective	GASETOBJ	id	returned from GASETUP
		objtype	0 →minimize user module
			1 →maximize user module
			2 →Traveling Salesman Problem
		parm	if objtype=0 or 1, user module
			if objtype=2, cost coefficients
selection	GASETSEL	id	returned from GASETUP
		elite	integer in [0, population size]
		type	0 →conventional tournament
			1 →dual tournament with BPW prob
		parm	if type = 0, tournament size
			if type = 1, real number in [0.5,1]
	default if	elite	1
	not set	type	conventional tournament
		parm	2

crossover	GASETCRO	id	returned from GASETUP
		crossprob	crossover probability
		type	0 →user module
			1 →simple
			2 →two-point
			3 →arithmetic
			4 →heuristic
			5 →pmatch
			6 →cycle
			7 →order
		parm	module name for type = 0
			0 <val≤ 1 if encoding=1, 0<type<3
	default if	crossprob	1.0
	not set	type	heuristic if encoding=1
			simple if encoding=2
			pmatch if encoding=3, objtype 0
			order if objtype=2 (TSP)
mutation	GASETMUT	id	returned from GASETUP
		mutprob	mutation probability
		type	0 →user module
			1 →uniform
			2 →delta
			3 →swap
			4 →invert
		parm	delta value if type=2
			number of swaps if type=3
	default if	mutprob	0.05
	not set	type	uniform if encoding=1 or 2, bounded
			delta if encoding=1 or 2, no bounds
			swap if encoding=3, not TSP
			invert if objtype=1 (TSP)

After setting the optimization parameters, you are ready to execute the GA. First, an initial solution population must be generated with a GAINIT call. GAINIT implements the *initialization* phase of the GA, generating an initial population of solutions and evaluating the objective value of each member solution. In the GAINIT call you specify the population size and any constant bounds on the solution domain. Next comes an IML loop containing a GAREGEN call. GAREGEN implements the *regeneration* phase of the GA, which generates a new solution population based on selection, crossover, and mutation of the current solution population, then replaces the current population with the new population and computes the new objective function values.

After the GAREGEN call, you can monitor the convergence of the GA by retrieving the objective function values for the current population with the GAGETVAL call. You might check the average value of the objective population, or check only the best

value. If the elite parameter is 1 or more, then it is easy to check the best member of the population, since it will always be the first member retrieved.

After your stopping criteria has been reached, you can retrieve the members of the solution population with the GAGETMEM call. To end the optimization, you should always use the GAEND call to free up memory resources allocated to the GA.

Below are some example programs to illustrate setting up and executing a genetic algorithm. The first example illustrates a simple program, a ten-city TSP using all IML defaults. The cost coefficients correspond to the cities being laid out on a two-by-five grid. The optimal route has a total distance of ten.

```
proc iml;

/* cost coefficients for TSP problem */
coeffs = { 0 1 2 3 4 5 4 3 2 1,
           1 0 1 2 3 4 5 4 3 2,
           2 1 0 1 2 3 4 5 4 3,
           3 2 1 0 1 2 3 4 5 4,
           4 3 2 1 0 1 2 3 4 5,
           5 4 3 2 1 0 1 2 3 4,
           4 5 4 3 2 1 0 1 2 3,
           3 4 5 4 3 2 1 0 1 2,
           2 3 4 5 4 3 2 1 0 1,
           1 2 3 4 5 4 3 2 1 0 };

/* problem setup */
  id = gasetup(3,   /* 3 -> integer sequence encoding */
               10,  /* number of locations */
               1234 /* initial seed */
              );
  /* set objective function */
  call gasetobj(id,
                2, /* 2 -> Traveling Salesman Problem */
                coeffs /* cost coefficient matrix */
               );

/* initialization phase */
  call gainit(id,
              100  /* initial population size */
             );

/* execute regeneration loop */

  niter = 20;            /* number of iterations */
  bestValue = j(niter,1); /* to store results */

  call gagetval(value, id, 1); /* gets first value */
  bestValue[1] = value;

  do i = 2 to niter;
    call garegen(id);
    call gagetval(value, id, 1);
```

```
   bestValue[i] = value;
end;

/* print solution history */
print (t(1:niter))[l = "iteration"] bestValue;

/* print final solution */
call gagetmem(bestMember, value, id, 1);
print  "best member " bestMember [f = 3.0 l = ""],,
      "final best value " value [l = ""];

call gaend(id);
```

For this test case, there is no call to GASETSEL. Therefore IML will use default se-
lection parameters: an elite value of 1 and a conventional tournament of size 2. Also,
since there is no GASETCRO or GASETMUT call, IML will use default genetic op-
erators: the order operator for crossover and the invert operator for mutation, and a
default mutation probability of 0.05. The output results are

```
                         iteration BESTVALUE

                              1        18
                              2        18
                              3        16
                              4        14
                              5        14
                              6        14
                              7        12
                              8        12
                              9        12
                             10        12
                             11        12
                             12        12
                             13        12
                             14        12
                             15        12
                             16        12
                             17        12
                             18        12
                             19        10
                             20        10


     best member    10   1   2   3   4   5   6   7   8   9

               final best value        10
```

The optimal value was reached after 19 iterations. Because the elite value was 1, the
best solution was retained and passed on to each successive generation, and therefore
never lost. Note that out of 3,628,800 possible solutions (representing 362,800 unique

paths), the GA found the optimum after only 1,900 function evaluations, without using any problem-specific information to assist the optimization. You could also do some experimentation, and specify different genetic operators with a GASETCRO and GASETMUT call, and different selection parameters with a GASETSEL call:

```
/* alternate problem setup */
  id = gasetup(3,  /* 3 -> integer sequence encoding */
               10,  /* number of locations */
               1234 /* initial seed */
              );
  /* set objective function */
  call gasetobj(id,
                2, /* 2 -> Traveling Salesman Problem */
                coeffs /* cost coefficient matrix */
               );
  call gasetcro(id,
                1.0, /* crossover probabilty 1 */
                5  /* 5 -> pmatch operator */
               );
  call gasetmut(id,
                0.05, /* mutation probability */
                3     /* 3 -> swap operator */
               );
  call gasetsel(id,
                3,  /* set elite to 3 */
                1,  /* dual tournament */
                0.95 /* best-player-wins probability 0.95 */
               );
/* initialization phase */
  call gainit(id,
              100  /* initial population size */
             );

/* execute regeneration loop */

  niter = 15;          /* number of iterations */
  bestValue = j(niter,1); /* to store results */

  call gagetval(value, id, 1); /* gets first value */
  bestValue[1] = value;

  do i = 2 to niter;
    call garegen(id);
    call gagetval(value, id, 1);
    bestValue[i] = value;
  end;

  /* print solution history */
  print (t(1:niter))[l = "iteration"] bestValue;

  /* print final solution */
  call gagetmem(bestMember, value, id, 1);
  print  "best member " bestMember [f = 3.0 l = ""],,
```

```
          "final best value " value [l = ""];

   call gaend(id);
```

The output of this test case is

```
               iteration BESTVALUE

                        1          24
                        2          18
                        3          18
                        4          16
                        5          16
                        6          14
                        7          14
                        8          14
                        9          14
                       10          12
                       11          12
                       12          12
                       13          10
                       14          10
                       15          10


   best member    3   4   5   6   7   8   9  10   1   2

               final best value         10
```

Note that convergence was faster than for the previous case, reaching an optimum after 13 iterations. This illustrates that the convergence of a GA may be very sensitive to the choice of genetic operators and selection parameters, and for practical problems some experimental fine-tuning of optimization parameters may be required to obtain acceptable convergence.

Incorporating Local Optimization

One commonly used technique is to combine the GA with a local optimization technique specific to the problem being solved. This can be done within the IML GA framework by incorporating a local optimization into the objective function evaluation: return a locally optimized objective value, and optionally replace the original solution passed into the module with the optimized solution.

Always replacing the original solution with the locally optimized one will cause faster convergence, but it is also more likely to converge prematurely to a local optimum. One way to reduce this possibility is to not replace the original solution in every case, but replace it with some probability p. For some problems, values of p from 5 to 15 percent have been shown to significantly improve convergence, while avoiding premature convergence to a local optimum (Michalewicz 1996) .

Example 17.1. Genetic Algorithm with Local Optimization ◆ 511

Handling Constraints

Practical optimization problems often come with constraints, which may make the problem difficult to solve. Constraints are handled in GAs in a variety of ways.

If it is possible, the most straightforward approach is to set the problem encoding, genetic operators, and initialization such that the constraints are automatically met. For example, a non-linear optimization problem over n real variables with constant upper and lower bounds is easily formulated in IML using real fixed-length encoding, arithmetic crossover, and uniform mutation. The arithmetic crossover operator can be used without modification in any optimization over a convex solution space, when the optimum is expected to be an interior point of the domain.

Another approach to satisfying constraints is to repair solutions after genetic operators have been applied. This is what IML does when using the heuristic crossover operator or delta mutation operator with fixed bounds: it adjusts any individual component that violates an upper or lower bound. You can repair a solution inside a user crossover or mutation module, or repairs can be made by modifying the solution in a user objective function module, as was described in the previous section.

Another technique is to allow solutions to violate constraints, but to impose a penalty in the objective function for unsatisfied constraints. If the penalty is severe enough, the algorithm should converge to an optimum point within the constraints. This approach should be used carefully. If most of the points in the solution space violate the constraints, then this technique may converge prematurely to the first feasible solution found. Also, convergence may be poor to a solution that lies on or near a constraint boundary.

Example 17.1. Genetic Algorithm with Local Optimization

For the symmetric Traveling Salesman Problem, there is a simple local optimization that can be incorporated into a user objective function module, which is to check each pair of adjacent locations in the solution and swap their positions if that would improve the objective function value. Here is the previous TSP example, modified to use an objective function module that implements this strategy. In this initial example, the optimized solution is not written back out to the solution population (except to get the final solution at the end).

```
proc iml;

/* cost coefficients for TSP problem */
coeffs = { 0 1 2 3 4 5 4 3 2 1,
           1 0 1 2 3 4 5 4 3 2,
           2 1 0 1 2 3 4 5 4 3,
           3 2 1 0 1 2 3 4 5 4,
           4 3 2 1 0 1 2 3 4 5,
           5 4 3 2 1 0 1 2 3 4,
           4 5 4 3 2 1 0 1 2 3,
           3 4 5 4 3 2 1 0 1 2,
           2 3 4 5 4 3 2 1 0 1,
           1 2 3 4 5 4 3 2 1 0 };
```

```
start TSPObjectiveFunction(r) global(coeffs, p);
  s = r;
  nc = ncol(s);
  /* local optimization, assumes symmetric cost *
   * coefficients                              */
  do i = 1 to nc;
    city1 = s[i];
    inext = 1 + mod(i,nc);
    city2 = s[inext];
    if i=1 then
      before = s[nc];
    else
      before = s[i-1];
    after = s[1 + mod(inext,nc)];
    if (coeffs[before,city1] + coeffs[city2, after]) >
       (coeffs[before,city2] + coeffs[city1, after])
    then do;
      s[i] = city2;
      s[inext] = city1;
    end;
  end;
  /* compute objective function */
  cost = coeffs[s[nc],s[1]];
  do i = 1 to nc-1;
    cost = cost + coeffs[s[i],s[i+1]];
  end;
  if uniform(1234)<=p then
     r = s;
  return (cost);
finish;

/* problem setup */
  id = gasetup(3,  /* 3 -> integer sequence encoding */
               10,  /* number of locations */
               123  /* initial random seed */
              );
  /* set objective function */
  call gasetobj(id,
                0, /* 0 -> minimize a user-defined module */
                "TSPObjectiveFunction"
               );
  call gasetcro(id, 1.0, 6);
  call gasetmut(id, 0.05, 4);
  call gasetsel(id, 1, 1, 0.95);
  p = 0; /* probability of writing locally optimized
          * solution back out to population  */
/* initialization phase */
  call gainit(id,
              100  /* initial population size */
             );
/* execute regeneration loop */

  niter = 10;              /* number of iterations */
```

Example 17.1. Genetic Algorithm with Local Optimization ◆ 513

```
bestValue = j(niter,1); /* to store results */

call gagetval(value, id, 1); /* gets first (and best) value */
bestValue[1] = value;

do i = 2 to niter;
  call garegen(id);
  call gagetval(value, id, 1);
  bestValue[i] = value;
end;

/* print solution history */
print (t(1:niter))[l = "iteration"] bestValue;

/* make sure local optimization is
 * written back to all solutions */
p = 1.; /* set global probability to 1 */
call gareeval(id);

/* print final solution */
call gagetmem(bestMember, value, id, 1);
print  "best member " bestMember [f = 3.0 l = ""],,
     "final best value " value [l = ""];
call gaend(id);
```

The results of running this program are

```
                 iteration   BESTVALUE

                     1           12
                     2           12
                     3           12
                     4           12
                     5           10
                     6           10
                     7           10
                     8           10
                     9           10
                    10           10


 best member     7   6   5   4   3   2   1  10   9   8

              final best value        10
```

Convergence is much improved by the local optimization, reaching the optimum in just 5 iterations compared to 13 with no local optimization. Writing some of the optimized solutions back to the solution population, by setting the global probability *p* to 0.05 or 0.15, will improve convergence even more.

Example 17.2. Real-Valued Objective Optimization with Constant Bounds

The next example illustrates some of the strengths and weaknesses of the arithmetic and heuristic crossover operators. The objective function to be minimized is

```
start sin_obj(x) global(xopt);
  r = abs(sin(sum(abs(x-xopt))));
  return(r);
finish;
```

This function obviously has a minimum at *x=xopt*, and is not differentiable at all points. The following program sets *xopt*= 0 and specifies constant boundary constraints such that the optimum is in the interior of the search space, and specifies the heuristic crossover operator:

```
proc iml;

/* objective function, has minimum of 0 at x = xopt */
start sin_obj(x) global(xopt);
  r = abs(sin(sum(abs(x-xopt))));
  return(r);
finish;

xopt = { 0 0 0 };
optimum = xopt;
optval = sin_obj(optimum);

id = gasetup(1,   /* 1-> fixed-length floating point vector encoding */
             3,   /* 3-> length of solution vectors */
             1234 /* 0-> initial random seed */
            );
call gasetobj(id,0,"sin_obj"); /* 0->minimize a user module,
                                * "sin_obj" is name of module */
call gasetcro(id, 0.9, 4); /* crossover probabilty 0.9,
                            * 4-> heuristic crossover operator   */

call gasetmut(id,0.05,2,0.01); /* mutation probability 0.05,
                                * 2-> delta mutation operator
                                * 0.01 is delta value  */

call gasetsel(id, 5, 1, 0.95); /* carry best 5 solutions over
                                * to the next generation, dual
                                * tournment with 0.95 best-player
                                * wins probability   */

bounds = {-1 -1 -1, 1 1 1};
call gainit(id,200,bounds);    /* initialize population with
                                * 200 members, "bounds" gives
                                * upper and lower bounds for
                                * components of randomly
                                * randomly generated vectors */
summary = j(20,2);
```

```
mattrib summary [c = {"bestValue", "avgValue"}];
call gagetval(value, id);
summary[1,1] = value[1];
summary[1,2] = value[:];


do i = 2 to 20;

   call garegen(id);

   call gagetval(value, id); /* get all objective values of
                               * the population */
   summary[i,1] = value[1];
   summary[i,2] = value[:];

end;

iteration = t(1:20);
print iteration summary;
call gaend(id);
```

The output results are

```
                     SUMMARY
            ITERATION bestValue avgValue

                   1  0.894517 0.8926763
                   2  0.894517  0.752227
                   3 0.1840732 0.6087493
                   4   0.14112 0.4848342
                   5   0.14112 0.3991614
                   6   0.14112 0.3539561
                   7 0.0481937 0.3680798
                   8 0.0481937 0.3243406
                   9 0.0481937 0.3027395
                  10 0.0481937 0.2679123
                  11 0.0481937 0.2550643
                  12 0.0481937 0.2582514
                  13 0.0481937 0.2652337
                  14 0.0481937 0.2799655
                  15 0.0383933  0.237546
                  16 0.0383933 0.3008743
                  17 0.0383933 0.2341022
                  18 0.0383933 0.1966969
                  19 0.0383933 0.2778152
                  20 0.0383933 0.2690036
```

To show the convergence of the overall population, the average value of the objective function for the whole population is printed out as well as the best value. The optimum value for this formulation is 0, and the optimum solution is (0 0 0). The output shows the convergence of the GA to be slow, especially as the solutions get near the optimum. This is the result of applying the heuristic crossover operator to an

ill-behaved objective function. If you change the crossover to the arithmetic operator
by changing the GASETCRO call to

```
call gasetcro(id, 0.9, 3); /* 3-> arithmetic crossover operator */
```

you get the following output:

```
                 SUMMARY
     ITERATION bestValue avgValue

          1  0.894517 0.8926763
          2  0.894517 0.8014329
          3 0.1840732 0.6496871
          4 0.1705931 0.4703868
          5 0.0984926 0.2892114
          6  0.076859 0.1832358
          7 0.0287965 0.1123732
          8 0.0273074 0.0720792
          9  0.018713 0.0456323
         10 0.0129708 0.0309648
         11 0.0087931 0.0240822
         12 0.0087931 0.0172102
         13 0.0050753 0.0128258
         14 0.0019603 0.0092872
         15 0.0016225 0.0070575
         16 0.0016225 0.0051149
         17 0.0012465 0.0036445
         18 0.0011895  0.002712
         19 0.0007646 0.0023329
         20 0.0007646 0.0020842
```

For this case, the arithmetic operator shows improved convergence. If we change the
problem characteristics again by changing the constraints so that the optimum lies on
a boundary

```
bounds = {0 0 0, 1 1 1};
```

The output using the arithmetic operator is

```
                 SUMMARY
     ITERATION bestValue avgValue

          1 0.8813497 0.8749132
          2 0.8813497  0.860011
          3 0.3721446 0.8339357
          4 0.3721446   0.79106
          5 0.3721446  0.743336
          6 0.3721446 0.7061592
          7 0.3721446 0.6797346
          8 0.3721446 0.6302206
          9 0.3721446 0.5818008
```

```
10 0.3721446 0.5327339
11 0.3721446 0.5149562
12 0.3721446   0.48525
13 0.3721446 0.4708617
14 0.3721446 0.4582203
15 0.3721446  0.433538
16 0.3721446 0.4256162
17 0.3721446 0.4236062
18 0.3721446 0.4149336
19 0.3721446 0.4135214
20 0.3721446 0.4078068
```

In this case, the algorithm fails to converge to the true optimum, given the characteristic of the arithmetic operator to converge on interior points. However, if you switch back to the heuristic crossover operator the results are

```
              SUMMARY
ITERATION bestValue avgValue

        1 0.8813497 0.8749132
        2 0.8813497 0.7360591
        3 0.3721446 0.5465098
        4         0 0.3427185
        5         0 0.2006271
        6         0 0.0826017
        7         0 0.0158228
        8         0 0.0002602
        9         0   0.00005
       10         0   0.00065
       11         0    0.0003
       12         0    0.0002
       13         0    0.0002
       14         0  0.000285
       15         0    0.0005
       16         0 0.0002952
       17         0    0.0002
       18         0 0.0001761
       19         0   0.00035
       20         0   0.00035
```

These results show a rapid convergence to the optimum. This example illustrates how the results of a GA are very operator-dependent. For complicated problems with unknown solution, you might need to try a number of different combinations of parameters in order to have confidence that you have converged to a true global optimum.

Example 17.3. Integer Programming Knapsack Problem

The next example uses the integer encoding, along with user modules for crossover and mutation. It formulates the Knapsack Problem using fixed-length integer encoding. The integer vector solution *s* is a vector of ones and zeros, where *s*[*i*]=1 implies that item *i* is packed in the knapsack. The weight constraints of the problem are not handled explicitly, but are accounted for by including a penalty for overweight in the objective function. The crossover operator randomly chooses a value for each element of the solution vector from each parent. The mutation operator randomly changes the values of a user-set number of elements in the solution vector. For this problem the value of the global optimum is 18.

```
proc iml;
weight = {2 3 4 4 1 1 1 1 1 1 1 1 1 1 1};
limit = 9; /* weight limit */
reward = {6 6 6 5 1.3 1.2 1.1 1.0 1.1 1.3 1.0 1.0 0.9 0.8 0.6};

start knapsack( x ) global( weight, reward, limit);
        wsum = sum(weight # x);
        rew = sum(reward # x);
        /* subtract penalty for exceeding weight */
        if wsum>limit then
                rew = rew - 5 * (wsum - limit);
        return(rew);
finish;

start switch_mut(s) global(nswitches);
   n = ncol(s);
   do i = 1 to nswitches;
      k = int(uniform(1234) * n) + 1;
      if s[k]=0 then
            s[k] = 1;
         else
            s[k] = 0;
   end;
finish;

start uniform_cross(child1, child2, parent1, parent2);
    child1 = parent1;
    child2 = parent2;
    do i = 1 to ncol(parent1);
       r = uniform(1234);
       if r<=0.5 then do;
         child1[i] = parent2[i];
         child2[i] = parent1[i];
       end;
    end;
finish;

id = gasetup(2,15, 123);
call gasetobj(id, 1, "knapsack"); /* maximize objective module */
call gasetcro(id, 1.0, 0,"uniform_cross"); /* user crossover module */
```

Example 17.3. Integer Programming Knapsack Problem ◆ 519

```
call gasetmut(id,
              0.20,                  /* mutation probabilty */
              0, "switch_mut"   /* user mutation module */
            );
nswitches = 3;
call gasetsel(id, 3,      /* carry 3 over to next generation */
                  1,      /* dual tournament */
                  0.95    /* best-player-wins probabilty */
            );
call gainit(id,100,{0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0,
                    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1});
niter = 20;
summary = j(niter,2);
mattrib summary [c = {"bestValue", "avgValue"}];
call gagetval(value, id);
summary[1,1] = value[1];
summary[1,2] = value[:];

do i = 1 to niter;
   call garegen(id);
   call gagetval(value, id);
   summary[i,1] = value[1];
   summary[i,2] = value[:];
end;
call gagetmem(mem, value, id, 1);
print  "best member " mem[f = 1.0 l = ""],
       "best value " value[l = ""];
iteration = t(1:niter);
print iteration summary;
call gaend(id);
```

The output of the program is

```
          best member   1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
              best value          18


                        SUMMARY
              ITERATION bestValue avgValue

                    1        16       2.44
                    2        16      6.257
                    3        16      6.501
                    4      16.7      7.964
                    5      16.7      8.812
                    6      16.7      9.254
                    7      16.7     10.021
                    8      16.8     11.216
                    9      16.9     12.279
                   10      16.9     12.094
                   11      16.9     11.633
                   12      16.9     11.431
                   13        18     11.502
```

14	18	13.2
15	18	13.128
16	18	13.282
17	18	12.876
18	18	13.715
19	18	12.889
20	18	13.15

Note that for this problem, the mutation parameters are set higher than is often seen for GAs. For this example, this is necessary to prevent premature convergence.

Example 17.4. Optimization with Linear Constraints Using Repair Strategy

This problem seeks a minimum within a convex domain specified by a convex hull, a set of points such that all points in the search space are normalized linear combinations of those points. Each solution is represented by a set of weights w such that there is one w_i for each point in the convex hull, $0 \le w_i \le 1$, and $\Sigma w_i = 1$. In this example the feasible region is the convex hull defined by the set of points (-3 -2), (3 -2), (-3 2) and (3 2). The objective function is a six-hump camel-back function (see Michalewicz, Appendix B), with a known global minimum value of -1.0316 at two different points, (-0.0898,0.7126) and (0.0898,-0.7126). A user mutation module is specified, and the simple crossover operator is used. Both the mutation operator and the crossover operator will produce solutions that violate the constraints, so in the objective function each solution will be checked and re-normalized to bring it back within the convex hull.

```
proc iml;

/* Test case using user modules for the mutation operator and
 * for initialization
 */

start sixhump(w) global(cvxhull);
/* Function has global minimum value of -1.0316
 * at x = {-0.0898  0.7126} and
 *    x = { 0.0898 -0.7126}
 */
  sum = w[1,+];

  /* guard against the remote possibility of all-0 weights */
  if sum=0 then do;
    nc = ncol(w);
    w = j(1, nc, 1/nc );
        sum = 1;
  end;

  /* re-normalize weights */
  w = w/sum;

  /* convert to x-coordinate form */
```

```
  x = (w * cvxhull)[+,];
  x1 = x[1];
  x2 = x[2];

  /* compute objective value */
  r = (4 - 2.1*x1##2 + x1##4/3)*x1##2 + x1*x2 +
        (-4 + 4*x2*x2)*x2##2;
  return(r);
finish;

/* each row is one point on the boundary of
 * the convex hull */
cvxhull = {-3 -2,
            3 -2,
           -3 2,
            3 2};

/* initialization module */
start cvxinit( w ) global(cvxhull);
sum = 0;
a = j(1, nrow(cvxhull), 1234);
do while(sum = 0);
  r = uniform(a);
  sum = r[1,+];
end;
w = r / sum;
finish;

/* mutation module */
start cvxmut(w)global(cvxhull);
  row = int(uniform(1234) * nrow(cvxhull)) + 1;
  r = uniform(1234);
  w[1,row] = r;
finish;

id = gasetup(1, /* real fixed-length vector encoding */
             nrow(cvxhull), /* vector size = number of points
                              * specifying convex hull
                              */
             1234);
call gasetobj(id,
               0, /* minimize a user-specified objective function */
              "sixhump"
              );
call gasetsel( id,
               5, /* carry over the best 5 from each generation */
               1, /* dual tournament */
              0.95  /* best-player-wins probability */
              );
call gasetcro(id,
              0.8, /* crossover probability */
              1    /* simple crossover operator */
              );
call gasetmut(id,0.05,0,"cvxmut");
```

```
call gainit( id,
            100,            /* population size */
                 ,          /* not using constant bounds */
         "cvxinit"          /* initialization module */
             );

niter = 35; /* number of iterations */
summary = j(niter,2);
mattrib summary [c = {"bestValue", "avgValue"}];
call gagetval(value, id);
summary[1,1] = value[1];
summary[1,2] = value[:];

do i = 1 to niter;
  call garegen(id);
  call gagetval(value, id);
  summary[i,1] = value[1];
  summary[i,2] = value[:];
end;
call gagetmem(mem, value, id, 1);
bestX = (mem * cvxhull)[+,];
print  "best X " bestX[l = ""],
       "best value " value[l = ""];
iteration = t(1:niter);
print iteration summary;
call gaend(id);
```

The output results are

```
            best X   0.089842 -0.712658
               best value  -1.031628

                    SUMMARY
            ITERATION bestValue avgValue

                    1 -0.082301 0.9235856
                    2 -0.948434 0.1262678
                    3 -0.956136 0.2745601
                    4 -1.017636 0.1367912
                    5 -1.028457 -0.241069
                    6 -1.028457 -0.353218
                    7 -1.028457  -0.56789
                    8 -1.028457  -0.73044
                    9 -1.028457 -0.854496
                   10 -1.028509 -0.941693
                   11 -1.031334 -0.936541
                   12 -1.031334  -0.90363
                   13 -1.031373 -0.774917
                   14 -1.031614 -0.873418
                   15 -1.031614 -0.886818
                   16 -1.031618  -0.95678
                   17 -1.031619 -0.933061
```

```
18 -1.031626 -0.885132
19 -1.031628 -0.936944
20 -1.031628 -0.906637
21 -1.031628 -0.925809
22 -1.031628 -0.860156
23 -1.031628 -0.946146
24 -1.031628 -0.817196
25 -1.031628 -0.883284
26 -1.031628 -0.904361
27 -1.031628 -0.974893
28 -1.031628 -0.975647
29 -1.031628 -0.872004
30 -1.031628 -1.031628
31 -1.031628 -0.897558
32 -1.031628 -0.922121
33 -1.031628 -0.855045
34 -1.031628 -0.922061
35 -1.031628 -0.958257
```

Any problem with linear constraints could be formulated in this way, by determining the convex hull corresponding to the constraints. The genetic operators and the repair strategy are straightforward to apply, and as this case shows, can give reasonable convergence to a global optimum.

References

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning,* Reading, MA: Addison-Wesley Longman.

Miller, B. L. and Goldberg, D. E. (1995), *Genetic Algorithms, Tournament Selecton, and the Effects of Noise*, Technical Report 95006, Illinois Genetic Algorithm Laboratory, University of Urbana-Champaign.

Michalewicz, Zbigniew (1996), *Genetic Algorithms + Data Structures = Evolution Programs*, New York: Springer-Verlag.

Chapter 18
Sparse Matrix Algorithms (Experimental)

Chapter Contents

Chapter 18
Sparse Matrix Algorithms
(Experimental)

Overview

This chapter documents direct and iterative algorithms for large sparse systems of linear equations:

$$Ax = b, \quad A \in R^{n \times n}, \ x, b \in R^n$$

where A is a nonsingular square matrix.

The following classes of iterative solvers are supported:

- Conjugate Gradient for symmetric positive-definite systems (ITCGRAD)
- Minimum Residual for symmetric indefinite systems (ITMINRES)
- Biconjugate Gradient for general nonsingular systems (ITBICG)

Iterative algorithms incur zero or controlled amounts of fill-in, have relatively small working memory requirements, and can converge as fast as $O(n)$ or $O(n^2)$ versus direct dense methods that are typically $O(n^3)$. Each iteration of an iterative algorithm is very inexpensive and typically involves a single matrix-vector multiplication and a pair of forward/backward substitutions.

Convergence of an iterative method depends upon the distribution of eigenvalues for the matrix A, and can be rather slow for badly conditioned matrices. For such cases SAS/IML offers hybrid algorithms, which combine an incomplete factorization (a modified direct method) used in the preconditioning phase with an iterative refinement procedure. The following preconditioners are supported:

- Incomplete Cholesky factorization ("IC")
- Diagonal Jacobi preconditioner ("DIAG")
- Modified incomplete LU factorization ("MILU")

For more information, see the description of the *precond* parameter in the section "Input Data Description" on page 528.

The following direct sparse solvers are supported for symmetric positive-definite systems:

- Symbolic LDL (SOLVELIN)
- Cholesky (SOLVELIN)

Classical factorization-based algorithms share one common complication: the matrix A usually suffers *fill-in*, which means additional operations and computer memory are required to complete the algorithm. A symmetric permutation of matrix rows and columns can lead to a dramatic reduction of fill-in. To compute such a permutation, SAS/IML implements a minimum degree ordering algorithm, which is an automatic step in the SOLVELIN function.

Iterative Methods

The conjugate gradient algorithm can be interpreted as the following optimization problem: minimize $\phi(x)$ defined by

$$\phi(x) = 1/2 x^T A x - x^T b$$

where $b \in R^n$ and $A \in R^{n \times n}$ are symmetric and positive definite.

At each iteration $\phi(x)$ is minimized along an A-conjugate direction, constructing orthogonal residuals:

$$r_i \perp \mathcal{K}_i(A; r_0), \quad r_i = A x_i - b$$

where $\mathcal{K}_i$ is a Krylov subspace:

$$\mathcal{K}_i(A; r) = \text{span}\{r, Ar, A^2 r, \ldots, A^{i-1} r\}$$

Minimum residual algorithms work by minimizing the Euclidean norm $\|Ax - b\|_2$ over $\mathcal{K}_i$. At each iteration, x_i is the vector in $\mathcal{K}_i$ that gives the smallest residual.

The biconjugate gradient algorithm belongs to a more general class of Petrov-Galerkin methods, where orthogonality is enforced in a different i-dimensional subspace (x_i remains in $\mathcal{K}_i$):

$$r_i \perp \{w, A^T w, (A^T)^2 w, \ldots, (A^T)^{i-1} w\}$$

Input Data Description

All subroutines have the same calling syntax and arguments:

```
ITCGRAD (x, error, iter, A, b, precond, tol, maxiter, start, history);
ITMINRES(x, error, iter, A, b, precond, tol, maxiter, start, history);
ITBICG  (x, error, iter, A, b, precond, tol, maxiter, start, history);
```

The subroutines ITCGRAD and ITMINRES require A to be symmetric, hence you must specify only the lower triangular part of A, while ITBICG requires *all* nonzero coefficients to be listed. The following table lists valid values for the *precond* parameter for each subroutine.

Table 18.1. Subroutine Definitions and Valid Preconditioners

IML Subroutine	Algorithm	Preconditioners
ITCGRAD	conjugate gradient	"NONE" "IC" "DIAG"
ITMINRES	minimum residual	"NONE" "IC" "DIAG"
ITBICG	biconjugate gradient	"NONE" "MILU"

x	solution vector
error	final solution error (optional)
iter	resultant number of iterations (optional)
A	three-column matrix of triplets, where the first column contains the value, the next column contains the row indices, and the third column contains the column indices of the nonzero matrix coefficients. The order in which triplets are listed is insignificant. For symmetric matrices specify only the lower triangular part, including the main diagonal (row indices must be greater than or equal to the corresponding column indices.) Zero coefficients should not be included. No missing values or duplicate entries are allowed.
b	the right-hand-side vector
precond	preconditioner, default value "NONE"
tol	desired tolerance, default value 10^{-7}
maxiter	maximum number of iterations, default value 10^5
start	initial guess
history	the history of errors for each iteration

Example: Conjugate Gradient Algorithm

Consider the following small example: $Ax = b$, where

$$A = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 3 \\ 0 & 1 & 10 & 0 \\ 0 & 3 & 0 & 3 \end{pmatrix}$$

and the vector of right-hand-side $b = (1\,1\,1\,1)^T$. Since the matrix is positive-definite and symmetric, you can apply the conjugate gradient algorithm to solve the system. Remember that you must specify only the lower-triangular part of the matrix (so row indices must be greater than or equal to the corresponding column indices.) The code for this example is as follows:

```
/* value    row    col */
A = { 3      1      1,
      1      2      1,
      4      2      2,
```

```
       1     3     2,
       3     4     2,
      10     3     3,
       3     4     4 };

/* right-hand-side */
b = {1, 1, 1, 1};

/* desired solution tolerance (optional) */
tol = 1e-7;

/* maximum number of iterations (optional) */
maxit = 200;

/* allocate iteration progress (optional) */
hist = j(50, 1);

/* provide an initial guess (optional) */
start = {2, 3, 4, 5};

/* invoke conjugate gradient method */
call itcgrad (
   x, st, it,     /* output parameters */
   A, b, "ic",    /* input parameters */
   tol,           /* optional control parameters */
   maxit,
   start,
   hist
 );

print  x;  /* print solution */
print st;  /* print solution tolerance */
print it;  /* print resultant number of iterations */
```

Notice that the example used an incomplete Cholesky preconditioner (which is recommended.) Here is the program output:

```
             X
       0.5882353
       -0.764706
       0.1764706
       1.0980392

            ST
       1.961E-16

            IT
               3
```

The conjugate gradient method converged successfully within three iterations. You can also print out the hist (iteration progress) array. Different starting points will result in different iterative histories.

Example: Minimum Residual Algorithm

For symmetric indefinite matrices it is best to use the minimum residual algorithm. The following example is slightly modified from the previous example by negating the first matrix element:

```
/* minimum residual algorithm */

/* value    row    col */
A = { -3     1      1,
       1     2      1,
       4     2      2,
       1     3      2,
       3     4      2,
      10     3      3,
       3     4      4 };

/* right-hand-sides b = (1 1 1 1) */
b = {1, 1, 1, 1};

/* desired solution tolerance (optional) */
tol = 1e-7;

/* maximum number of iterations (optional) */
maxit = 200;

/* allocate iteration progress (optional) */
hist = j(50, 1);

/* initial guess (optional) */
start = {2, 3, 4, 5};

/* invoke minimum residual method */
call itminres (
   x, st, it,    /* output parameters */
   a, b, "ic",   /* input parameters */
   tol,          /* optional control parameters */
   maxit,
   start,
   hist
 );

print  x;  /* print solution */
print st;  /* print solution tolerance */
print it;  /* print resultant number of iterations */

              X
          -0.27027
          0.1891892
          0.0810811
          0.1441441
```

```
                         ST
                     1.283E-15


                         IT
                          4
```

Example: Biconjugate Gradient Algorithm

The biconjugate gradient algorithm is meant for general sparse linear systems. Matrix symmetry is no longer assumed, and a complete list of nonzero coefficients must be provided. Consider the following matrix:

$$A = \begin{pmatrix} 10 & 0 & 0.2 \\ 0.1 & 3 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

with $b = (1\,1\,1)^T$.

The code for this example is as follows:

```
/* biconjugate gradient algorithm */

/*  value   row column */
A = { 10     1      1,
       3     2      2,
       4     3      3,
       0.1   2      1,
       0.2   1      3 };

/* vector of right-hand-sides */
b = {1, 1, 1};

/* desired solution tolerance */
tol = 1e-9;

/* maximum number of iterations */
maxit = 10000;

/* allocate history/progress */
hist = j(50, 1);

/* initial guess (optional) */
start = {2, 3, 4};

/* call biconjugate gradient subroutine */
call itbicg (
   x, st, it,      /* output parameters */
   a, b, "milu",   /* input parameters */
   tol,            /* optional control parameters */
   maxit,
   start,
   hist);
```

```
/* Print results */
print  x;
print st;
print it;
```

Here is the output:

```
        X
    0.095
0.3301667
     0.25


       ST
1.993E-16


       IT
        3
```

It is important to observe the resultant tolerance, in order to know how effective the solution is.

Symbolic LDL and Cholesky Factorizations

Symbolic LDL and Cholesky factorization algorithms are meant for symmetric positive-definite systems; hence, again, only the lower-triangular part of the matrix must be provided. The PROC IML function SOLVELIN provides an interface to both algorithms; the minimum degree ordering heuristic is invoked automatically:

```
SOLVELIN (x, status, A, b, method)
```

x　　　　　solution vector

status　　　status indicator 0 success, 1 matrix is not positive-definite, 2 out of memory

A　　　　　sparse matrix (lower-triangular part)

b　　　　　vector of right-hand-sides

method　　a character string, which specifies factorization type, possible values: "LDL" for LDL factorization, and "CHOL" for Cholesky.

The code for this example is as follows:

```
/* value    row    col */
A = { 3      1      1,
      1      2      1,
      4      2      2,
      1      3      2,
```

```
        3       4       2,
       10       3       3,
        3       4       4 };

/* right-hand-side */
b = {1, 1, 1, 1};

/* invoke LDL factorization */
call solvelin (x, status, a, b, "LDL");

print x; /* print solution */
```

Here is the program output:

```
           x
      0.5882353
      -0.764706
      0.1764706
      1.0980392
```

References

Golub, G. H. and Van Loan, C. F. (1996), "*Matrix Computations,*" Third Edition, Baltimore: Johns Hopkins University Press.

Greenbaum, A. (1997), "*Iterative Methods for Solving Linear Systems,*" *Society for Industrial and Applied Mathematics (SIAM)*, Philadelphia, PA.

Hestenes, M. R. and Stiefel, E. (1952), "*Methods of Conjugate Gradients for Solving Linear Systems,*" *J. Res. Natl. Bur. Standards*, B49, 409–436.

Paige, C. C. and Saunders, M. A. (1975), "*Solution of Sparse Indefinite Systems of Linear Equations,*" *SIAM Journal on Numerical Analysis*, 12:4, 617–629.

Chapter 19
Further Notes

Chapter Contents

Chapter 19
Further Notes

Memory and Workspace

You do not need to be concerned with the details of memory usage because memory allocation is done automatically. However, if you are interested, the following sections explain how it works.

There are two logical areas of memory, *symbolspace* and *workspace*. Symbolspace contains symbol table information and compiled statements. Workspace contains matrix data values. Workspace itself is divided into one or more extents.

At the start of a session, the symbolspace and the first extent of workspace are allocated automatically. More workspace is allocated as the need to store data values grows. The SYMSIZE= and WORKSIZE= options in the PROC IML statement give you control over the size of symbolspace and the size of each extent of workspace. If you do not specify these options, PROC IML uses host dependent defaults. For example, you can begin an IML session and set the SYMSIZE= and WORKSIZE= options with the statement

```
proc iml symsize=n1 worksize=n2;
```

where $n1$ and $n2$ are specified in kilobytes.

If the symbolspace memory becomes exhausted, more memory is automatically acquired. The symbolspace is stable memory and is not compressible like workspace. Symbolspace is recycled whenever possible for reuse as the same type of object. For example, temporary symbols may be deleted after they are used in evaluating an expression. The symbolspace formerly used by these temporaries is added to a list of free symbol-table nodes. When allocating temporary variables to evaluate another expression, IML looks for symbol-table nodes in this list first before consuming unused symbolspace.

Workspace is compressible memory. Workspace extents fill up as more matrices are defined by operations. Holes in extents appear as you free matrices or as IML frees temporary intermediate results. When an extent fills up, compression reclaims the holes that have appeared in the extent. If compression does not reclaim enough memory for the current allocation, IML allocates a new extent. This procedure results in the existence of a list of extents, each of which contains a mixture of active memory and holes of unused memory. There is always a current extent, the one in which the last allocation was made.

For a new allocation, the search for free space begins in the current extent and proceeds around the extent list until finding enough memory or returning to the current extent. If the search returns to the current extent, IML begins a second transversal

of the extent list, compressing each extent until either finding sufficient memory or returning to the current extent. If the second search returns to the current extent, IML opens a new extent and makes it the current one.

If the SAS System cannot provide enough memory to open a new extent with the full extent size, IML repeatedly reduces its request by 2K. In this case, the successfully opened extent will be smaller than the standard size.

If a single allocation is larger than the standard extent size, IML requests an allocation large enough to hold the matrix.

The WORKSIZE= and SYMSIZE= options offer tools for tuning memory usage. For data intensive applications involving a few large matrices, use a high WORKSIZE= value and a low SYMSIZE= value. For symbol intensive applications involving many matrices, perhaps through the use of many IML modules, use a high SYMSIZE= value.

You can use the SHOW SPACE command to display the current status of IML memory usage. This command also lists the total number of compressions done on all extents.

Setting the DETAILS option in the RESET command prints messages in the output file when IML compresses an extent, opens a new extent, allocates a large object, or acquires more symbolspace. These messages can be useful because these actions normally occur without the user's knowledge. The information can be used to tune WORKSIZE= and SYMSIZE= values for an application. However, the default WORKSIZE= and SYMSIZE= values should be appropriate in most applications.

Do not specify a very large value in the WORKSIZE= and SYMSIZE= options unless absolutely necessary. Many of the native functions and all of the DATA step functions used are dynamically loaded at execution time. If you use a large amount of the memory for symbolspace and workspace, there may not be enough remaining to load these functions, resulting in the error message

> **Unable to load module** *module-name.*

Should you run into this problem, issue a SHOW SPACE command to examine current usage. You may be able to adjust the SYMSIZE= or WORKSIZE= values.

The amount of memory your system can provide depends on the capacity of your computer and on the products installed. The following techniques for efficient memory use are recommended when memory is at a premium:

- Free matrices as they are no longer needed using the FREE command.
- Store matrices you will need later in external library storage using the STORE command, and then FREE their values. You can restore the matrices later using the LOAD command. See Chapter 14, "Storage Features."
- Plan your work to use smaller matrices.

Accuracy

All numbers are stored and all arithmetic is done in double-precision. The algorithms used are generally very accurate numerically. However, when many operations are performed or when the matrices are ill-conditioned, matrix operations should be used in a numerically responsible way because numerical errors add up.

Error Diagnostics

When an error occurs, several lines of messages are printed. The error description, the operation being performed, and the line and column of the source for that operation are printed. The names of the operation's arguments are also printed. Matrix names beginning with a pound sign (#) or an asterisk (*) may appear; these are temporary names assigned by the IML procedure.

If an error occurs while you are in immediate mode, the operation is not completed and nothing is assigned to the result. If an error occurs while executing statements inside a module, a PAUSE command is automatically issued. You can correct the error and resume execution of module statements with a RESUME statement.

The most common errors are described below:

- referencing a matrix that has not been set to a value, that is, referencing a matrix that has no value associated with the matrix name

- making a subscripting error, that is, trying to refer to a row or column not present in the matrix

- performing an operation with nonconformable matrix arguments, for example, multiplying two matrices together that do not conform, or using a function that requires a special scalar or vector argument

- referencing a matrix that is not square for operations that require a square matrix (for example, INV, DET, or SOLVE)

- referencing a matrix that is not symmetric for operations that require a symmetric matrix (for example, GENEIG)

- referencing a matrix that is singular for operations that require a nonsingular matrix (for example, INV and SOLVE)

- referencing a matrix that is not positive definite or positive semidefinite for operations that require such matrices (for example, ROOT and SWEEP)

- not enough memory (see "Memory and Workspace" earlier in this chapter) to perform the computations and produce the result matrices.

These errors result from the actual dimensions or values of matrices and are caught only after a statement has begun to execute. Other errors, such as incorrect number of arguments or unbalanced parentheses, are syntax errors and resolution errors and are detected before the statement is executed.

Efficiency

The Interactive Matrix Language is an interpretive language executor that can be characterized as follows:

- efficient and inexpensive to compile

- inefficient and expensive for the number of operations executed

- efficient and inexpensive within each operation.

Therefore, you should try to substitute matrix operations for iterative loops. There is a high overhead involved in executing each instruction; however, within the instruction IML runs very efficiently.

Consider four methods of summing the elements of a matrix:

```
s=0;                        /* method 1 */
do i=1 to m;
   do j=1 to n;
      s=s+x[i,j];
   end;
end;
s=j[1,m]*x*j[n,1];       /* method 2 */
s=x[+,+];                /* method 3 */
s=sum(x);                /* method 4 */
```

Method 1 is the least efficient, method 2 is more efficient, method 3 is more efficient yet, and method 4 is the most efficient. The greatest advantage of using IML is reducing human programming labor.

Missing Values

An IML numeric element can have a special value called a *missing value* that indicates that the value is unknown or unspecified. (A matrix with missing values should not be confused with an empty or unvalued matrix, that is, a matrix with 0 rows and 0 columns.) A numeric matrix can have any mixture of missing and nonmissing values.

SAS/IML software supports missing values in a limited way. The operators listed below recognize missing values and propagate them. Most matrix operators and functions do not support missing values. For example, matrix multiplication or exponentiation involving a matrix with missing values is not meaningful. Also, the inverse of a matrix with missing values has no meaning.

Missing values are coded in the bit pattern of very large negative numbers, as an I.E.E.E. "NAN" code, or as a special string, depending on the host system.

In literals, a numeric missing value is specified as a single period. In data processing operations, you can add or delete missing values. All operations that move values around move missing values properly. The following arithmetic operators propagate missing values.

addition (+)   subtraction (−)
multiplication (#) division (/)
maximum (<>)   minimum (><)
modulo (MOD)   exponentiation (##)

The comparison operators treat missing values as large negative numbers. The logical operators treat missing values as zeros. The operators SUM, SSQ, MAX, and MIN check for and exclude missing values.

The subscript reduction operators exclude missing values from calculations. If all of a row or column that is being reduced is missing, then the operator returns the result indicated in the table below.

Operator	Result If All Missing
addition (+)	0
multiplication (#)	1
maximum (<>)	large negative value
minimum (><)	large positive value
sum squares (##)	0
index maximum (<:>)	1
index minimum (>:<)	1
mean (:)	missing value

Also note that, unlike the SAS DATA step, IML does not distinguish between special and generic missing values; it treats all missing values alike.

Principles of Operation

This section presents various technical details on the operation of SAS/IML software. Statements in IML go through three phases:

- The parsing phase includes text acquisition, word scanning, recognition, syntactical analysis, and enqueuing on the statement queue. This is performed immediately as IML reads the statements.

- The resolution phase includes symbol resolution, label and transfer resolution, and function and call resolution. Symbol resolution connects the symbolic names in the statement with their descriptors in the symbol table. New symbols can be added or old ones recognized. Label and transfer resolution connects statements and references affecting the flow of control. This connects LINK and GOTO statements with labels; it connects IF with THEN and ELSE clauses; it connects DO with END. Function-call resolution identifies functions and call routines and loads them if necessary. Each reference is checked with respect to the number of arguments allowed. The resolution phase begins after

a module definition is finished or a DO group is ended. For all other statements outside of any module or DO group, resolution begins immediately after parsing.

- The execution phase occurs when the statements are interpreted and executed. There are two levels of execution: statement and operation. Operation-level execution involves the evaluation of expressions within a statement.

Operation-Level Execution

Operations are executed from a chain of operation elements created at parse-time and resolved later. For each operation, the interpreter performs the following steps:

1. Prints a record of the operation if the FLOW option is on.

2. Looks at the operands to make sure they have values. Only certain special operators are allowed to tolerate operands that have not been set to a value. The interpreter checks whether any argument has character values.

3. Inspects the operator and gives control to the appropriate execution routine. A separate set of routines is invoked for character values.

4. Checks the operands to make sure they are valid for the operation. Then the routine allocates the result matrix and any extra workspace needed for intermediate calculations. Then the work is performed. Extra workspace is freed. A return code notifies IML if the operation was successful. If unsuccessful, it identifies the problem. Control is passed back to the interpreter.

5. Checks the return code. If the return code is nonzero, diagnostic routines are called to explain the problem to the user.

6. Associates the results with the result arguments in the symbol table. By keeping results out of the symbol table until this time, the operation does not destroy the previous value of the symbol if an error has occurred.

7. Prints the result if RESET PRINT or RESET PRINTALL is specified. The PRINTALL option prints intermediate results as well as end results.

8. Moves to the next operation.

Chapter 20
Language Reference

Chapter Contents

Chapter 20
Language Reference

Overview

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

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

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

Scalar Functions

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

Reduction Functions

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

Matrix Inquiry Functions

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

Matrix Sorting And By-Group Processing Calls

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

Character Functionality

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

Random Number Generation Functionality

RANDGEN Call	generates random numbers from specified distributions
RANDSEED Call	initializes seed for subsequent RANGEN calls

Statistical Functionality

BRANKS Function	computes bivariate ranks
CUSUM Function	calculates cumulative sums

DESIGN Function	creates a design matrix
DESIGNF Function	creates a full-rank design matrix
IPF Call	performs an iterative proportional fit of a contingency table
LAV Call	performs linear least absolute value regression by solving the L_1 norm minimization problem
LMS Call	performs robust regression
LTS Call	performs robust regression
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
MVE Call	finds the minimum volume ellipsoid estimator
OPSCAL Function	rescales qualitative data to be a least-squares fit to qualitative data
RANK Function	ranks elements of a matrix
RANKTIE Function	ranks matrix elements using tie-averaging
SEQSCALE Call	perform discrete sequential tests
SEQSHIFT Call	perform discrete sequential tests
SEQTESTS Calls	perform discrete sequential tests
SWEEP Function	sweeps a matrix

Time Series Functionality

ARMACOV Call	computes an autocovariance sequence for an ARMA model
ARMALIK Call	computes the log-likelihood and residuals for an ARMA model
ARMASIM Function	simulates an ARMA series
CONVEXIT Function	calculates convexity of a noncontingent cash-flow
COVLAG Function	computes autocovariance estimates for a vector time series
DURATION Function	calculates modified duration of a noncontingent cash-flow
FARMACOV Call	computes the auto-covariance function for an ARFIMA(p, d, q) process
FARMAFIT Call	estimate 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	obtain a fractionally differenced process
FORWARD Function	calculates forward rates
KALCVF Call	computes the one-step prediction $z_{t+1|t}$ and the filtered estimate $z_{t|t}$, as well as their covariance matrices. The call uses forward recursions, and you can also use it to obtain k-step estimates.
KALCVS Call	uses backward recursions to compute the smoothed estimate $z_{t|T}$ and its covariance matrix, $P_{t|T}$, where T is the number of observations in the complete data set.

<table>
<tr><td>KALDFF Call</td><td>computes the one-step forecast of state vectors in an SSM using the diffuse Kalman filter. The call estimates the conditional expectation of $\mathbf{z}_t$, and it also estimates the initial random vector, δ, and its covariance matrix.</td></tr>
<tr><td>KALDFS Call</td><td>computes the smoothed state vector and its mean square error matrix from the one-step forecast and mean square error matrix computed by KALDFF.</td></tr>
<tr><td>PV Function</td><td>calculates present value</td></tr>
<tr><td>RATES Function</td><td>converts interest rates from one base to another</td></tr>
<tr><td>SPOT Function</td><td>calculates spot rates</td></tr>
<tr><td>TSBAYSEA Call</td><td>performs Bayesian seasonal adjustment modeling</td></tr>
<tr><td>TSDECOMP Call</td><td>analyzes nonstationary time series by using smoothness priors modeling</td></tr>
<tr><td>TSMLOCAR Call</td><td>analyzes nonstationary or locally stationary time series by using the minimum AIC procedure</td></tr>
<tr><td>TSMLOMAR Call</td><td>analyzes nonstationary or locally stationary multivariate time series by using the minimum AIC procedure</td></tr>
<tr><td>TSMULMAR Call</td><td>estimates VAR processes by using the minimum AIC procedure</td></tr>
<tr><td>TSPEARS Call</td><td>analyzes periodic AR models with the minimum AIC procedure</td></tr>
<tr><td>TSPRED Call</td><td>provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are input</td></tr>
<tr><td>TSROOT Call</td><td>calculates AR and MA coefficients from the characteristic roots of the model or calculates the characteristic roots of the model from the AR and MA coefficients</td></tr>
<tr><td>TSTVCAR Call</td><td>analyzes time series that are nonstationary in the covariance function</td></tr>
<tr><td>TSUNIMAR Call</td><td>determines the order of an AR process with the minimum AIC procedure and estimates the AR coefficients</td></tr>
<tr><td>VARMACOV Call</td><td>computes the theoretical cross-covariance matrices for a stationary VARMA(p, q) model</td></tr>
<tr><td>VARMALIK Call</td><td>computes the log-likelihood function for a VARMA(p, q) model</td></tr>
<tr><td>VARMASIM Call</td><td>generates VARMA(p,q) time series</td></tr>
<tr><td>VNORMAL Call</td><td>generates multivariate normal random series</td></tr>
<tr><td>VTSROOT Call</td><td>computes the characteristic roots for a VARMA(p, q) model</td></tr>
<tr><td>YIELD Function</td><td>calculates yield-to-maturity of a cash-flow stream</td></tr>
</table>

Numerical Analysis Functionality

<table>
<tr><td>FFT Function</td><td>performs the finite Fourier transform</td></tr>
<tr><td>IFFT Function</td><td>computes the inverse finite Fourier transform</td></tr>
<tr><td>JROOT Function</td><td>computes the first nonzero roots of a Bessel function of the first kind and the derivative of the Bessel function at each root</td></tr>
</table>

ODE Call	performs numerical integration of vector differential equations of the form
ORPOL Function	generates orthogonal polynomials on a discrete set of points
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	fit a cubic spline to points
SPLINEC Call	fit a cubic spline to points
SPLINEV Function	evaluatesa 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 Functionality

APPCORT CALL	complete orthogonal decomposition
COMPORT Call	complete orthogonal decomposition by Householder transformations
CVEXHULL Function	finds a convex hull of a set of planar points
DET Function	computes the determinant of a square matrix
ECHELON Function	reduces a matrix to row-echelon normal form
EIGEN Call	computes eigenvalues and eigenvectors
EIGVAL Function	computes eigenvalues
EIGVEC Function	computes eigenvectors
GENEIG Call	computes eigenvalues and eigenvectors of a generalized eigenproblem
GINV Function	computes the generalized inverse
GSORTH Call	computes the Gram-Schmidt orthonormalization
HALF Function	computes Cholesky decomposition
HANKEL Function	generates a Hankel matrix
HDIR Function	performs a horizontal direct product
HERMITE Function	reduces a matrix to Hermite normal form
HOMOGEN Function	solves homogeneous linear systems
INV Function	produces the inverse
INVUPDT Function	updates a matrix inverse
ITBICG Call	solves a sparse general linear system by iteration
ITCGRAD Call	solves a sparse symmetric positive-definite system by iteration
ITMINRES Call	solves a sparse symmetric system by iteration

Optimization Subroutines

Set Functions

Control Statements

Dataset and File Functionality

Graphics and Window Functions

Wavelet Analysis Calls

Genetic Algorithm Functionality

Operators

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

Addition Operator: **+**

adds corresponding matrix elements

matrix1 **+** *matrix2*

matrix **+** *scalar*

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

For example, the statements

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

produce the matrix **C**.

```
            C              2 rows      2 cols    (numeric)

                                  2          3
                                  4          5
```

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

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

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

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

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

Comparison Operators: < > = <= >= ˆ=

compare matrix elements

matrix1<matrix2

matrix1<=matrix2

matrix1>matrix2

matrix1>=matrix2

matrix1=matrix2

matrix1ˆ =matrix2

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

For example, let

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

and

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

Evaluation of the expression

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

results in the matrix of values

```
          C              2 rows       3 cols      (numeric)

                            1            0            1
                            1            1            1
```

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

For example the expression

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

yields the result

```
        D              2 rows      3 cols      (numeric)

                          0           1           1
                          1           1           1
```

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

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

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

Consider the following statement:

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

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

Concatenation Operator, Horizontal: ||

concatenates matrices horizontally

   matrix1||matrix2

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

For example, the statements

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

result in

```
          C              2 rows       6 cols       (numeric)

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

Also, if

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

and

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

then

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

results in

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

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

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

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

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

Concatenation Operator, Vertical: //

concatenates matrices vertically

   matrix1//matrix2

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

For example, the statements

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

result in

```
          C          4 rows      3 cols     (numeric)

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

Also let

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

and

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

Then the statement

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

produces the new matrix

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

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

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

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

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

Direct Product Operator: @

takes the direct product of two matrices

> *matrix1@matrix2*

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

For example, the statements

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

result in

```
         C              2 rows      4 cols     (numeric)

                   0           2           0           4
                   0           6           0           8
```

The statement

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

results in

```
         D              2 rows      4 cols     (numeric)

                   0           0           2           4
                   0           0           6           8
```

Division Operator: /

performs elementwise division

> *matrix1/matrix2*

> *matrix/scalar*

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

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

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

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

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

Element Maximum Operator: <>

selects the larger of two elements

matrix1<>matrix2

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

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

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

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

For example, the statements

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

produce the result

```
         C              2 rows       3 cols      (numeric)

                          2             9            6
                         20            11           40
```

Element Minimum Operator: ><

selects the smaller of two elements

matrix1><matrix2

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

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

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

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

For example, the statements

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

produce the result

```
        C              2 rows       3 cols      (numeric)

                          1            4           2
                         10           10          12
```

Index Creation Operator: :

creates an index vector

  value1:*value2*

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

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

results in

```
        I              1 row        4 cols      (numeric)

                   7          8           9           10
```

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

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

results in the row vector

```
           R               1 row       5 cols     (numeric)

           10          9           8           7          6
```

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

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

results in

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

                 var1   var2   var3   var4   var5
```

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

Logical Operators: & | ˆ

perform elementwise logical comparisons

> *matrix1***&***matrix2*
>
> *matrix***&***scalar*
>
> *matrix1***|***matrix2*
>
> *matrix***|***scalar*
>
> *ˆ matrix*

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

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

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

The following statements illustrate the use of these logical operators:

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

Multiplication Operator, Elementwise:

performs elementwise multiplication

> *matrix1#matrix2*
>
> *matrix#scalar*
>
> *matrix#vector*

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

For example, the statements

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

result in the matrix

```
              C              2 rows      2 cols     (numeric)

                               4           16
                               0           20
```

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

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

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

produces the matrix

```
            AD               2 rows      2 cols     (numeric)

                              10           20
                             300          400
```

whereas the statements

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

produce the matrix

```
          AD              2 rows       2 cols     (numeric)

                              10          200
                              30          400
```

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

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

Multiplication Operator, Matrix: *

performs matrix multiplication

*matrix1*matrix2*

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

For example, the statements

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

result in

```
          C               1 row        2 cols     (numeric)

                               7           10
```

and the statement

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

results in

```
          D               2 rows       1 col      (numeric)

                               5
                              11
```

Power Operator, Elementwise:

raises each element to a power

> *matrix1##matrix2*

> *matrix##scalar*

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

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

For example, the statements

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

result in

```
            B              1 row       2 cols      (numeric)

                              1          8         27
```

The statement

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

results in

```
            B               1 row        3 cols     (numeric)

                     1 1.4142136 1.7320508
```

Power Operator, Matrix: **

raises a matrix to a power

> *matrix**scalar*

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

For example, the statements

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

result in

```
          C          2 rows      2 cols    (numeric)

                      3          4
                      2          3
```

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

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

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

The matrix power operater does not support missing values.

Sign Reverse Operator: −

reverses the signs of elements

  −matrix

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

For example, the statements

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

result in the matrix

```
          B          2 rows      3 cols    (numeric)

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

Subscripts: [ ]

select submatrices

> *matrix*[*rows,columns*]

> *matrix*[*elements*]

Subscripts are used with matrices to select submatrices, where *rows* and *columns* are expressions that evaluate to scalars or vectors. If these expressions are numeric, they contain valid subscript values of rows and columns in the argument matrix. If a row or column expression is a character matrix, then it refers to columns or rows in the argument matrix assigned corresponding labels by a MATTRIB statement or READ statement. A subscripted matrix can appear on the left side of the equal sign. The dimensions of the target submatrix must conform to the dimensions of the source matrix. See "Using Matrix Expressions" for further information.

For example, the statements

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

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

```
            M              1 row       1 col     (numeric)

                                         6
```

The statements

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

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

```
            M              1 row       3 cols    (numeric)

                             4           5          6
```

the statements

```
x={1 2 3,
   4 5 6,
   7 8 9};
MATTRIB x colname = {'col1' 'col2' 'col3'}
           rowname = {'row1' 'row2' 'row3'};
a='col3';
m=x['row2',a];
```

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

```
        M               1 row      1 col      (numeric)

                                      6
```

Subtraction Operator: –

subtracts corresponding matrix elements

matrix1–matrix2

matrix–scalar

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

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

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

An example of a valid statement follows:

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

Transpose Operator: '

transposes a matrix

matrix'

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

For example, the statements

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

result in

```
          B                2 rows      3 cols    (numeric)

                             1           3          5
                             2           4          6
```

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

Statements, Functions, and Subroutines

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

ABORT Statement

stops execution and exits IML

> **ABORT ;**

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

ABS Function

takes the absolute value

> **ABS(** *matrix***)**

where *matrix* is a numeric matrix or literal.

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

```
a = { -1 2 -3, 0 -1 2 };
c=abs(a);
```

ALL Function

checks for all elements nonzero

> **ALL(** *matrix***)**

where *matrix* is a numeric matrix or literal.

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

You can use the ALL function to express the results of a comparison operator as a single 1 or 0. For example, the comparison operation $\mathbf{A} > \mathbf{B}$ yields a matrix

containing elements that can be either ones or zeros. All the elements of the new matrix are ones only if each element of **A** is greater than the corresponding element of **B**.

For example, consider the statement

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

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

The statements

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

and

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

have the same effect.

ANY Function

checks for any nonzero element

ANY(*matrix***)**

where *matrix* is a numeric matrix or literal.

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

For example, consider the statement

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

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

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

However, the following statement prints the message:

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

APPCORT Call

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

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

The inputs to the APPCORT subroutine are:

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

$$\mathbf{A} = \mathbf{Q} \left[\begin{array}{c} \mathbf{R} \\ \mathbf{0} \end{array} \right] \mathbf{\Pi}'\mathbf{P}'\mathbf{\Pi}$$

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

sing   is an optional scalar specifying a singularity criterion.

The APPCORT subroutine returns the following values:

prqb   is an $n \times p$ matrix product

$$\mathbf{P}\mathbf{\Pi} \left[\begin{array}{cc} (\mathbf{L}')^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{array} \right] \mathbf{Q}'\mathbf{B}$$

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

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

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

An example of using the APPCORT subroutine follows:

```
/* 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));
call appcort(Ainv,lindep,A,b);
```

```
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
       print "Pseudoinverse conditions not satisfied";
   else
       print "Pseudoinverse conditions satisfied";

/* compute solution for rank deficient LS:
     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;
```

                                    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
```

                    Pseudoinverse conditions satisfied

                                     X

                                    0.3
                                    0.6

APPEND Statement

adds observations to the end of a SAS data set

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

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

In the preceding statements,

operand   can be specified as one of the following:

- a literal containing variable names

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

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

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

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

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

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

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

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

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

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

The ROWNAME= operand to the FROM clause specifies the name of a character matrix to contain row titles. The first *nrow* values of this matrix become values of a variable with the same name in the output data set; *nrow* is the number of rows in the FROM matrix. The procedure uses the first *nrow* elements in row-major order.

Examples using the APPEND statement follow. The first example shows the use of the FROM clause when creating a new data set. See also the section "CREATE Statement" on page 599

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

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

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

You could achieve the same result with the statements

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

APPLY Function

applies an IML module to its arguments

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

In the preceding statement,

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

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

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

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

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

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

The result is

```
              R           2 rows        3 cols      (numeric)

                        106           107          108
                        109           110          111
```

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

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

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

The results are

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

ARMACOV Call

computes an autocovariance sequence for an ARMA model

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

The inputs to the ARMACOV subroutine are as follows:

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

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

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

The ARMACOV subroutine returns the following values:

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

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

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

The model notation for the ARMACOV and ARMALIK subroutines is the same. The $\mathrm{ARMA}(p, q)$ model is denoted

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

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

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

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

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

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

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

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

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

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

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

```
              AUTO
      3.2533333 2.4266667 1.2133333 0.6066667 0.3033333
```

```
              CROSS                   CONVOL
              2.04        0.8         1.64         0.8
```

ARMALIK Call

computes the log-likelihood and residuals for an ARMA model

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

The inputs to the ARMALIK subroutine are as follows:

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

The ARMALIK subroutine returns the following values:

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

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

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

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

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

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

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

use the following IML code:

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

The printed output is

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

ARMASIM Function

simulates a univariate ARMA series

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

The inputs to the ARMASIM function are As follows:

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

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

mu	is a scalar containing the overall mean of the series.
sigma	is a scalar containing the standard deviation of the innovation series.
n	is a scalar containing n, the length of the series. The value of n must be greater than 0.
seed	is a scalar containing the random number seed. If it is not supplied, the system clock is used to generate the seed. If it is negative, then the absolute value is used as the starting seed; otherwise, subsequent calls ignore the value of *seed* and use the last seed generated internally.

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

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

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

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

returns n standard normal pseudo-random deviates.

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

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

use the following code to produce the result shown:

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

```
                            Y

                      2.3253578
                      0.975835
                     -0.376358
```

```
                                          -0.878433
                                          -2.515351
                                          -3.083021
                                          -1.996886
                                          -1.839975
                                          -0.214027
                                           1.4786717
```

BLOCK Function

forms block-diagonal matrices

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

where *matrix* is a numeric matrix or literal.

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

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

produces a matrix of the form

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

The statements

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

result in the matrix

```
        C              4 rows        4 cols      (numeric)

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

BRANKS Function

computes bivariate ranks

BRANKS(*matrix***)**

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

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

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

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

where

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

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

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

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

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

F	5 rows	3 cols	(numeric)
1	1	1	
3	2	2	
2	5	2	
4	3.5	3	
5	3.5	3.5	

BTRAN Function

computes the block transpose

BTRAN(x, n, m)

The inputs to the BTRAN function are as follows:

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

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

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

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

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

For example, the statements

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

produce the result

```
         Z              4 rows      2 cols     (numeric)


                           1           2
                           5           6
                           3           4
                           7           8
```

BYTE Function

translates numbers to ordinal characters

BYTE(*matrix*)

where *matrix* is a numeric matrix or literal.

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

For example, in the ASCII character set,

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

specifies that

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

The lowercase alphabet can be generated with

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

which produces

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

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

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

CALL Statement

calls a subroutine or function

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

The inputs to the CALL statement are as follows:

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

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

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

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

See also the section on the RUN statement.

CHANGE Call

search and replace text in an array

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

The inputs to the CHANGE call are as follows:

matrix   is a character matrix or quoted literal.

old   is the string to be changed.

new   is the string to replace the *old* string.

numchange   is the number of times to make the change.

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

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

produce

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

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

>% $ [ ] { } $<>-$? * # @ ' '(backquote) ^

CHAR Function

produces a character representation of a numeric matrix

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

The inputs to the CHAR function are as follows:

matrix   is a numeric matrix or literal.

w is the field width.

d is the number of decimal positions.

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

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

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

For example, the statements

```
a={1 2 3 4};
f=char(a,4,1);
```

produce the result

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

                    1.0   2.0   3.0   4.0
```

CHOOSE Function

conditionally chooses and changes elements

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

The inputs to the CHOOSE function are as follows:

condition   is checked for being true or false for each element.

result-for-true   is returned when *condition* is true.

result-for-false is returned when *condition* is false.

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

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

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

result in

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

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

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

```
              X           3 rows        3 cols      (numeric)

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

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

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

```
              X           3 rows        3 cols      (numeric)

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

CLOSE Statement

closes a SAS data set

    CLOSE <*SAS-data-set*>;

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

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

Examples of the CLOSE statement are as follows.

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

CLOSEFILE Statement

closes an input or output file

> **CLOSEFILE** *files***;**

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

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

Examples of the CLOSEFILE statement follow.

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

or

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

or

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

COMPORT Call

provides complete orthogonal decomposition by Householder transformations

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

The COMPORT subroutine returns the following values:

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

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

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 $[\mathbf{A}_{piv[1]} \cdots \mathbf{A}_{piv[n]}]$. The vector *piv* corresponds to an $n \times n$ permutation matrix, $\mathbf{\Pi}$, of the pivoted QR decomposition in the first step of orthogonal decomposition.

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

The inputs to the COMPORT subroutine are as follows:

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

$$\mathbf{A} = \mathbf{Q} \left[\begin{array}{c} \mathbf{R} \\ \mathbf{0} \end{array} \right] \mathbf{\Pi}' \mathbf{P}' \mathbf{\Pi}$$

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

sing   is an optional scalar specifying a singularity criterion.

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

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

$$\mathbf{A} = \mathbf{Q} \left[\begin{array}{c} \mathbf{R} \\ \mathbf{0} \end{array} \right] \mathbf{\Pi}' = \left[\begin{array}{cc} \mathbf{Y} & \mathbf{Z} \end{array} \right] \left[\begin{array}{cc} \mathbf{R}_1 & \mathbf{R}_2 \\ \mathbf{0} & \mathbf{0} \end{array} \right] \mathbf{\Pi}'$$

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

2. Use the transpose $\mathbf{L}_{12}$ of the upper trapezoidal matrix $\mathbf{R} = \left[\begin{array}{cc} \mathbf{R}_1 & \mathbf{R}_2 \end{array} \right]$,

$$\mathbf{L}_{12} = \left[\begin{array}{c} \mathbf{L}_1 \\ \mathbf{L}_2 \end{array} \right] = \mathbf{R}' \in \mathcal{R}^{t \times r}$$

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

$$\mathbf{P}_r \cdots \mathbf{P}_1 \left[\begin{array}{c} \mathbf{L}_1 \\ \mathbf{L}_2 \end{array} \right] = \left[\begin{array}{c} \mathbf{L} \\ \mathbf{0} \end{array} \right]$$

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

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

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

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

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

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

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

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

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

An example of using the APPCORT subroutine follows:

```
/* 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
       print "Pseudoinverse conditions not satisfied";
   else
      print "Pseudoinverse conditions satisfied";
```

                                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

                 Pseudoinverse conditions satisfied

CONCAT Function

performs elementwise string concatenation

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

where *arguments* are character matrices or quoted literals.

The CONCAT function produces a character matrix containing elements that are the concatenations of corresponding element strings from each argument. The CONCAT function accepts up to 15 arguments, where each argument is a character matrix or a scalar. All nonscalar arguments must conform. Any scalar arguments are used repeatedly to concatenate to all elements of the other arguments. The element length of the result equals the sum of the element lengths of the arguments. Trailing blanks of one matrix argument appear before elements of the next matrix argument in the result matrix. For example, if you specify

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

and

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

then the statement

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

produces the new 2×2 matrix

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

                                      ABH  C IJ
                                      DE K FGLM
```

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

CONTENTS Function

obtains the variables in a SAS data set

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

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

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

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

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

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

CONVEXIT Function

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

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

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

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

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

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

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

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

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

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

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

The statements

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

result in the following output:

```
CONVEXIT
199.26229
```

COVLAG Function

computes autocovariance estimates for a vector time series

COVLAG(x, k**)**

The inputs to the COVLAG function are as follows:

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

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

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

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

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

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

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

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

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

produce the matrix

```
              COV            1 row      4 cols    (numeric)

                     33       23.1       13.6        4.9
```

CREATE Statement

creates a new SAS data set

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

> **CREATE** *SAS-data-set* **FROM** *matrix-name*

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

The inputs to the CREATE statement are as follows:

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

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

matrix-name   names a matrix containing the data.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CSHAPE Function

reshapes and repeats character values

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

The inputs to the CSHAPE function are as follows:

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

The CSHAPE function shapes character matrices. See also the description of the SHAPE function, which is used with numeric data.

The dimension of the matrix created by the CSHAPE function is specified by *nrow* (the number of rows), *ncol* (the number of columns), and *size* (the element length). A padding character is specified by *padchar*.

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

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

Some examples follow. The statement

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

results in

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

                                 a b
                                 c d
```

The statement

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

results in

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

                                aaa aaa
```

The statement

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

results in

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

                            abc def
                            ghi jkl
```

The statement

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

results in

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

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

And finally, the statement

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

results in

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

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

CUSUM Function

calculates cumulative sums

   CUSUM(*matrix***)**

where *matrix* is a numeric matrix or literal.

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

For example, the statements

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

produce the result

```
        A             1 row      4 cols     (numeric)

                      1         3         7        12
```

B	2 rows	2 cols	(numeric)
	5	11	
	14	18	

CVEXHULL Function

finds a convex hull of a set of planar points

    CVEXHULL(*matrix***)**

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

The argument for the CVEXHULL function is an $n \times 2$ matrix of (x, y) points. The result matrix is an $n \times 1$ matrix of indices. The indices of points in the convex hull in counter-clockwise order are returned as the first part of the result matrix, and the negative of the indices of the internal points are returned as the remaining elements of the result matrix. Any points that lie on the convex hull but lie on a line segment joining two other points on the convex hull are not included as part of the convex hull.

The result matrix can be split into positive and negative parts using the LOC function. For example, the following statements find the index vector for the convex hull and print the asociated 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;
```

               CONVEXHULL

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

DATASETS Function

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

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

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

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

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

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


    A              6 rows      1 col      (character, size 8)

                                CLASS
                                FITNESS
                                GROWTH
                                HOUSES
                                SASPARM
                                TOBACCO
```

DELETE Call

deletes a SAS data set

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

The inputs to the DELETE subroutine are as follows:

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

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

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

Some examples follow.

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

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

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

DELETE Statement

marks observations for deletion

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

The inputs to the DELETE statement are as follows:

range   specifies a range of observations.

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

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

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

ALL   specifies all observations.

CURRENT   specifies the current observation.

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

AFTER   specifies all observations after the current one.

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

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

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

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

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

In the statement above,

variable   is a variable in the SAS data set.

comparison-op   is one of the following comparison operators:

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

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

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

   ^ = ^ ? < <= > >=

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

   = ? = : = *

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

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

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

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

Here are several examples of DELETE statements:

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

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

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

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

DESIGN Function

creates a design matrix

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

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

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

For example, the statements

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

produce the design matrix

```
          A              6 rows      3 cols     (numeric)

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

DESIGNF Function

creates a full-rank design matrix

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

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

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

For example, the statements

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

produce the following design matrix.

```
        B              6 rows       2 cols      (numeric)

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

DET Function

computes the determinant of a square matrix

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

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

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

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

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

produce the matrix **C** containing the determinant:

```
        C              1 row        1 col       (numeric)

                          2
```

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

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

where *MACHEPS* is the relative machine precision.

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

Previously, a much smaller singularity criterion was used, which caused algebraic operations to be performed on values that were essentially floating point error. This occasionally yielded numerically unstable results. The new criterion is much more conservative, and it generates far fewer erroneous results. In some cases, you may need to scale the data to avoid singular matrices. If you think the new criterion is too strong,

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

If A is an $n \times n$ matrix, then the DET function temporarily allocates an n^2 array in order to compute the determinant.

DIAG Function

creates a diagonal matrix

   DIAG(*argument***)**

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

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

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

For example, the statements

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

result in

```
            C            2 rows       2 cols     (numeric)

                              4            0
                              0            1
```

The statements

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

result in

```
        D            3 rows      3 cols     (numeric)

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

DISPLAY Statement

displays fields in display windows

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

The inputs to the DISPLAY statement are as follows:

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

group-options   can be any of the following:

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

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

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

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

Following are several examples of using the DISPLAY statement:

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

DO Function

produces an arithmetic series

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

The inputs to the DO function are as follows:

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

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

For example, the statement

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

yields the result

```
        I               1 row         6 cols     (numeric)

        3          6           9          12          15          18
```

The statement

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

yields the result

```
        J               1 row         5 cols     (numeric)

        3          2           1           0          -1
```

DO and END Statements

groups statements as a unit

  DO ;

    statements

  END ;

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

For example, consider the following statements:

```
if x=y then
    do;
        i=i+1;
        print x;
    end;
print y;
```

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

```
print y;
```

DO groups can be nested. Any number of nested DO groups is allowed.

Here is an example of nested DO groups:

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

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

DO Statement, Iterative

iteratively executes a DO group

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

The inputs to the DO statement are as follows:

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

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

The *start*, *stop*, and *increment* values should be scalars or expressions with evalua-
tions that yield scalars. The *variable* is given a new value for each repetition of the

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

For example, the statements

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

produce the output

```
                                        I
                THE VALUE OF I IS:      1


                                        I
                THE VALUE OF I IS:      3


                                        I
                THE VALUE OF I IS:      5
```

DO DATA Statement

repeats a loop until an end of file occurs

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

The inputs to the DO DATA statement are as follows:

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

The DO DATA statement is used for repetitive DO loops that need to be exited upon the occurrence of an end of file for an INPUT, READ, or other I/O statement. This form is common for loops that read data from either a sequential file or a SAS data set.

When an end of file is reached inside the DO DATA group, IML immediately jumps from the group and starts executing the statement following the END statement. DO DATA groups can be nested, where each end of file causes a jump from the most local DO DATA group. The DO DATA loop simulates the end-of-file behavior of the SAS DATA step. You should avoid using GOTO and LINK statements to jump out of a DO DATA group.

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

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

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

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

DO Statement with an UNTIL Clause

conditionally executes statements iteratively

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

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

The inputs to the DO UNTIL statement are as follows:

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

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

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

DO Statement with a WHILE Clause

conditionally executes statements iteratively

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

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

The inputs to the DO WHILE statement are as follows:

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

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

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

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

The next example increments the starting value by 2:

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

DURATION Function

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

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

The DURATION function returns the modified duration of a noncontingent cash-flow as a scalar.

times	is an n-dimensional column vector of times. Elements should be non-negative.
flows	is an n-dimensional column vector of cash-flows.
ytm	is the per-period yield-to-maturity of the cash-flow stream. This is a scalar and should be positive.

Duration of a security is generally defined as

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

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

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

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

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

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

For example, the statements below

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

produces the output

```
DURATION
0.9090909
```

ECHELON Function

reduces a matrix to row-echelon normal form

ECHELON(*matrix*)

where *matrix* is a numeric matrix or literal.

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

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

The resulting matrix is

```
            E              3 rows       3 cols      (numeric)

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

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

EDIT Statement

opens a SAS data set for editing

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

The inputs to the EDIT statement are as follows:

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

operand   selects a set of variables.

expression   selects observations conditionally.

name   names a variable to contain the number of observations.

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

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

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

  ALL   for all variables

> _CHAR_   for all character variables
>
> _NUM_   for all numeric variables

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

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

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

The general form of the WHERE clause is

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

In the statement above,

variable   is a variable in the SAS data set.

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

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

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

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

> ^= ^? < <= > >=

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

> = ? =: =*

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

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

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

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

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

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

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

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

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

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

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

Following is an example using the NOBS option.

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

EIGEN Call

computes eigenvalues and eigenvectors

> **CALL EIGEN(** *eigenvalues, eigenvectors,* **A)** *<VECL="vl">*;

where **A** is an arbitrary square numeric matrix for which eigenvalues and eigenvectors are to be calculated.

The EIGEN call returns the following values:

eigenvalues   a matrix to contain the eigenvalues of the input matrix.

eigenvectors   names a matrix to contain the right eigenvectors of the input matrix.

vl is an optional $n \times n$ matrix containing the left eigenvectors of **A** in the same manner that *eigenvectors* contains the right eigenvectors.

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

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

The eigenvalues of a matrix **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

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

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

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 $\text{Re}(\lambda) + i\text{Im}(\lambda)$, then the conjugate complex value $\text{Re}(\lambda) - i\text{Im}(\lambda)$ is also an eigenvalue of **A**.

- The right and left eigenvectors corresponding to a real eigenvalue of **A** are real. The right and left eigenvectors corresponding 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 **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} with the square root of the machine precision ϵ. (The value of ϵ is the largest value stored in double precision that, when added to 1 in double precision, still results in 1.)

3. The matrix **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 *symmetric*, the result of the statement

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

has the properties

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

that is,

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

so that

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

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

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

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

The computation can be checked by forming the residuals:

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

The values in **R** should be of the order of round-off error.

The following code computes 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) levec='lvec';
```

The sorted eigenvalues of this matrix are

```
            VAL

          1  1.7320508
          1 -1.732051
          1          0
  0.2087788          0
  0.0222025          0
  0.0026187          0
         -1          2
         -1         -2
```

We can verify the correctness of the left and right eigenvector computation using the following code:

```
/* verify 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 code can be written to verify the left eigenvectors, using the fact that the left eigenvectors of **A** are the same as the complex conjugate right eigenvectors of **A**′.

```
  /* verify 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

computes eigenvalues

> **EIGVAL(** A**)**

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

The EIGVAL function returns a column vector of the eigenvalues of **A**. See the description of the EIGEN subroutine for more details.

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

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

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

The matrix produced containing the eigenvalues is

```
        VAL


        75        100
        75       -100
        25          0
```

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

A symmetric example follows:

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

The matrix produced containing the eigenvalues is

```
        A              2 rows      1 col      (numeric)

                       33.401219
                        0.5987805
```

EIGVEC Function

computes right eigenvectors

EIGVEC(A**)**

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

The EIGVEC function creates a matrix containing the right eigenvectors of **A**. You can obtain the left eigenvectors by first transposing **A**. See the description of the EIGEN subroutine for more details.

An example calculating the eigenvectors of a symmetric matrix follows:

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

The matrix produced containing the eigenvectors is

```
                A           2 rows      2 cols      (numeric)

                          0.3220062 0.9467376
                          0.9467376 -0.322006
```

END Statement

ends a DO loop or DO statement

END:

See the description of the DO and END statements.

EXECUTE Call

executes SAS statements immediately

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

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

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

Following are examples of valid EXECUTE subroutines:

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

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

EXP Function

calculates the exponential

> **EXP(** *matrix***)**

where *matrix* is a numeric matrix or literal.

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

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

```
                   A              1 row        4 cols      (numeric)

                2.7182818  7.3890561 20.085537   54.59815
```

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

FARMACOV Call

computes the auto-covariance function for an ARFIMA(p, d, q) **process**

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

The inputs 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 containing 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 containing 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 auto-covariance 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 auto-covariance 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 containing the auto-covariance function of an ARFIMA(p, d, q) process.

To compute the auto-covariance of an ARFIMA($1, 0.3, 1$) process

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

where $\epsilon_t \sim NID(0, 1.2)$, you can specify

```
d    = 0.3;
phi  = 0.5;
theta= -0.1;
sigma= 1.2;
call farmacov(cov, d, phi, theta, sigma) lag=5;
print cov;
```

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 auto-covariance 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 auto-correlation function

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

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

The FARMACOV subroutine computes the auto-covariance function of an ARFIMA(p, d, 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 $\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p$ and $\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \cdots - \theta_q B^q$ 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 auto-covariance function of $\{x_t\}$ and γ_k^z be the auto-covariance function of $\{z_t\}$.

Then the auto-covariance 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 auto-covariance function of $\{y_t\}$ is given by Sowell (1992, p. 175).

FARMAFIT Call

estimate the parameters of an ARFIMA(p, d, q) model

CALL FARMAFIT(*d, phi, theta, sigma, series <, p, q, opt>* **) ;**

The inputs 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 is *q=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.

   opt=0   requests the conditional sum of squares function. This is the default.

   opt=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 containing a fractional differencing order.
phi	is a vector containing the autoregressive coefficients.
theta	is a vector containing the moving-average coefficients.
sigma	is a scalar containing a variance of the innovation series.

To estimate parameters of an ARFIMA$(1, 0.3, 1)$ model

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

where $\epsilon_t \sim NID(0, 1)$, you can specify

```
d    = 0.3;
phi  = 0.5;
theta= -0.1;
call farmasim(yt, d, phi, theta);
call farmafit(d, ar, ma, sigma, yt) p=1 q=1;
print d ar ma sigma;
```

The FARMAFIT subroutine estimates parameters d, $\phi(B)$, $\theta(B)$, and σ_ϵ^2 of an ARFIMA(p, d, q) model. The log-likelihood function needs to be solved by iterative numerical procedures such as the quasi-Newton optimization. The starting value d is obtained by the approach of Geweke and Poter-Hudak (1983); the starting value of the AR and MA parameters are obtained from the least squares estimates.

FARMALIK Call

computes the log-likelihood function of an ARFIMA(p, d, q) model

CALL FARMALIK(*lnl, series, d <, phi, theta, sigma, p, q, opt>***)** ;

The inputs 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 containing 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 containing 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.

         opt=0   requests the conditional sum of squares function. This is the default.

         opt=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 3-dimensional vector. *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]). If the *opt=0* is specified, only the weighted sum of squares of residuals returns in *lnl*[1].

To compute the log-likelihood function of an ARFIMA$(1, 0.3, 1)$ model

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

where $\epsilon_t \sim NID(0, 1.2)$, you can specify

```
d    = 0.3;
phi  = 0.5;
theta= -0.1;
sigma= 1.2;
call farmasim(yt, d, phi, theta, sigma);
call farmalik(lnl, yt, d, phi, theta, sigma);
print lnl;
```

The FARMALIK subroutine computes a log-likelihood function of the ARFIMA(p, d, q) model. The exact log-likelihood function is worked by Sowell (1992); the conditional sum of squares function is worked 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

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

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 $\Sigma = [\gamma_{i-j}]$ 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

generates an ARFIMA(p, d, q) process

> **CALL FARMASIM(** *series, d <, phi, theta, mu, sigma, n, p, q, initial, seed>* **) ;**

The inputs 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 containing 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 containing 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 specifies the random number seed. If it is not supplied, the system clock is used to generate the seed. If it is negative, then the absolute value is used as the starting seed; otherwise, subsequent calls ignore the value of *seed* and use the last seed generated internally.

The FARMASIM subroutine returns the following value:

series   is an n vector containing the generated ARFIMA(p, d, q) process.

To generate an ARFIMA($1, 0.3, 1$) process

$$(1 - 0.5B)(1 - B)^{0.3}(y_t - 10) = (1 + 0.1B)\epsilon_t$$

where $\epsilon_t \sim NID(0, 1.2)$, you can specify

```
d    = 0.3;
phi  = 0.5;
theta= -0.1;
mu   = 10;
sigma= 1.2;
call farmasim(yt, d, phi, theta, mu, sigma, 100);
print yt;
```

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 using covariance matrices obtained from FARMACOV. If the process is nonstationary, the time series is recursively generated 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

obtain a fractionally differenced process

> **CALL FDIF(** *out, series, d***) ;**

The inputs 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 containing the fractionally differenced process.

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). To get the filtered series z_t, you can specify

```
d    = 0.3;
phi  = 0.5;
theta= -0.1;
call farmasim(yt, d, phi, theta) n=100;
call fdif(zt, yt, d);
print zt;
```

FFT Function

performs the finite Fourier transform

FFT(x **)**

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

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

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

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

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

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

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

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

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

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

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

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

FILE Statement

opens or points to an external file

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

The inputs to the FILE statement are as follows:

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

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

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

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

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 it will be read properly.

Following are several valid uses of FILE statement.

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

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

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

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

FIND Statement

finds observations

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

The inputs to the FIND statement are as follows:

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

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

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

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

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

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

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

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

In the preceding statement,

variable	is a variable in the SAS data set.

comparison-op   is   one   of   the   following   comparison   operators:

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

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

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

   ˆ = ˆ ? < <= > >=

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

   = ? = : = *

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

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

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

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

Following are some valid examples of the FIND statement:

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

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

FINISH Statement

denotes the end of a module

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

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

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

Some examples follow.

```
finish;
finish mod1;
```

FORCE Statement

see the description of the SAVE statement

FORWARD Function

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

> **FORWARD(** *times,spot_rates***)**

The FORWARD function returns an $n \times 1$ vector of (per-period) forward rates.

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

$$f_1 = s_1$$

$$f_i = \left(\frac{(1 + s_i)^{t_i}}{(1 + s_{t_{i-1}})^{t_{i-1}}} \right)^{\frac{1}{t_i - t_{i-1}}} - 1; \quad i = 2, \dots, n$$

For example, the following statements

```
time=T(do(1,5,1));
spot=T(do(0.05,0.09,0.01));
forward=forward(time,spot);
print forward;
```

produce the following output:

```
    FORWARD

        0.05
 0.0700952
 0.0902839
 0.1105642
 0.1309345
```

FREE Statement

free matrix storage space

> **FREE** *matrices*;

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

where *matrices* are names of matrices.

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

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

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

```
free a b c;
```

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

```
free / d e;
```

For more information, see the discussion of workspace storage in Chapter 19, "Further Notes."

GAEND Call (Experimental)

end a genetic algorithm optimization and free memory resources

> **CALL GAEND(** *id* **);**

The inputs 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 (Experimental)

get members of the current solution population for a genetic algorithm optimization

CALL GAGETMEM(*members, values, id*<, *index* >**);**

The GAGETMEM call returns the following values:

members   is a matrix representing the members of the current solution population specified by the *index* parameter.

values   is a matrix of objective function values, with the value at each row corresponding to the solution in *members*.

The inputs 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 will have the most optimal objective function values of the population, and those *elite* members will be 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 will be returned as-is in *members*. If more than one member is requested in a GAGETMEM call, each row of *members* will have one solution, shaped into a row vector. If solutions are not of fixed length, then the number of columns of *members* will equal the number of elements of the largest solution, and rows representing solutions with fewer elements will have the extra elements filled in with missing values.

See the GASETUP function for an example.

GAGETVAL Call (Experimental)

get current solution objective function values for a genetic algorithm optimization

    CALL GAGETVAL(*values, id*<*, index* >**);**

The GAGETVAL call returns the following values:

values   is a matrix of objective function values for solutions in the current population 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 inputs 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 will have the most optimal objective function values of the population, and those *elite* members will be 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 (Experimental)

create and initialize a solution population for a genetic algorithm optimization

    CALL GAINIT(*id, popsize* <*,* < *bounds* > <*, modname* > >**);**

The inputs 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 specifying the lower and upper bounds for each element of a solution matrix. It is only used for integer and real fixed-length vector problem encoding.

modname   is the name of a user-written module to be called from GAINIT when generating 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 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, the user controls 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 may be supplied as a 2 x *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 resulting from all IML-supplied crossover and mutation operators will be checked to ensure they are within the upper and lower bounds, and any solution components violating the bounds will be 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 will be 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 will call 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 (Experimental)

re-evaluate the objective function values for a solution population of a genetic algorithm optimization

> **CALL GAREEVAL(*id*);**

The inputs 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 (Experimental)

replace the current solution population by applying selection, crossover, and mutation for a genetic algorithm optimization problem

> **CALL GAREGEN(** *id***);**

The inputs 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 probabilty and mutation operator. The mutation probabilty and operator are either specified in the corresonding GASETMUT call, or if no such call was made, a default value of 0.05 will be assigned to the mutation probabilty, and a default mutation operator will be assigned based on the problem encoding (see the GASETMUT call). The offspring are then transfered to the new population. If the no crossover option was specified in the GASETCRO call, then only mutation is applied to the nonelite 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 (Experimental)

set the crossover operator for a genetic algorithm optimization

> **CALL GASETCRO(** *id, crossprob, type* $<$ *, parm* $>$ **);**

The inputs 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 will 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 that IML provides, as explained below.

parm   is a matrix whose interpretation depends on the value of *type*, as described below.

The GASETCRO call enables the user to specify the crossover operator to be used in the genetic algorithm optimization problem. The following options can be specified by the user with the *type* parameter:

type = -1:   specifies that no crossover operator is to be applied, and the new population will be generated by applying the mutation operator to the old population, according to the mutation probabilty.

type = 0:   specifies that a user-written module, whose name is passed in the *parm* parameter, is to be used as the crossover operator. This module should be an IML subroutine with four parameters. The module should return the new offspring solutions in the first two parameters based on the input parent solutions, which will be selected by the genetic algorithm and passed into the module in the last two parameters. The module will be called once for each crossover operation within the GAREGEN call to create a new generation of solutions.

type = 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:

```
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, the user may 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.

type = 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 *k1* and *k2* within the vector are chosen at random, such that $1 \leq k1 < k2 < n$. Element values between those positions are swapped between parents. For parents *p1* and *p2* the offspring are:

```
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, the user may 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.

type = 3: specifies the arithmetic operator, defined for real and integer fixed-length vector encoding. This operator computes offspring of parents *p1* and *p2* as:

$$c1 = a * p1 + (1-a) * p2;$$

$$c2 = a * p2 + (1-a) * p1;$$

where a is a random number between 0 and 1. For integer encoding, each component is rounded off to the nearest integer. It has the advantage that it will always produce feasible offspring for a convex solution space. A disadvantage of this operator is that it will tend to produce offspring toward the interior of the search region, so that it may be less effective if the optimum lies on or near the search region boundary.

type = 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:

$$c1 = a * (p2 - p1) + p2;$$

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:

$$c2 = (1 - a) * p1 + a * p2;$$

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 were specified in the GAINIT call, the offspring will be checked against the bounds, and any component outside its bound will be set equal to that bound.

type = 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 cutpoints as indicated:

```
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 (. indicates positions yet to be determined):

```
c1 = {. . 9 3 4 1 . . .};
c2 = {. . 3 4 5 6 . . .};
```

Next, define a mapping according to the two selected segments:

```
9-3, 3-4, 4-5, 1-6
```

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 will tend 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.

type = 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:

```
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 cutpoint, the elements of *p2* in order are (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*, we have:

```
2 8 7 9 1
```

Placing these back in order starting at the second cutpoint yields:

```
c1 = {9 1 3 4 5 6 2 8 7};
```

Applying this logic to *c2* yields:

```
c2 = {5 6 9 3 4 1 7 8 2};
```

This operator maintains the similarity of the relative order, or adjacency, of the sequence elements of the parents. It is especially effective for circular path-oriented optimizations, such as the Traveling Salesman Problem.

type = 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, for 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:

```
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:

```
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* paramter in the GASETUP call, but for fixed-length vector and sequence encoding, a default crossover operator will be 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 (Experimental)

set the mutation operator for a genetic algorithm optimization

> **CALL GASETMUT(** *id, mutprob* $<$ *, type,* $<$ *, parm* $>$ $>$ **);**

The inputs 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 probabilty 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 that IML provides, as explained below.

parm
is a matrix whose interpretation depends on the value of *type*, as described below.

The GASETMUT call enables the user 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. The following mutation operators may be specified by the user with the *type* parameter:

type = 0:
specifies that a user-written module, whose name is passed in the *parm* parameter, is to be used as the mutation operator. This module should be an IML subroutine with one parameter, which receives the solution to be mutated. The module will be 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 will not be checked by IML.

type = 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 may prove especially useful in early stages of the optimization, since it will tend 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 hone in on an optimum, the uniform operator may hinder the optimization.

type = 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 integer in *parm*. To apply the mutation, a randomly chosen element *k* of the solution vector *v* is modified such that

```
v[k] = v[k] + delta[k]; /* with probability 0.5 */
        or
v[k] = v[k] - delta[k];
```

If there are bounds specified for the problem in the GAINIT call, then *v*[*k*] is adjusted as necessary to fit within the bounds. This operator gives the user the ability 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.

type = 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 value. It is the default mutation operator for sequence encoding, except for when the Traveling Salesman option is specified in the GASETOBJ call. The user may 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.

type = 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. IML allows more than one GASETMUT call to 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. The user may 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 call, but when no GASETMUT call is made for fixed-length vector and sequence encoding, a default value of 0.05 will be set for *mutprob*, and a default mutation operator will be used in the GAREGEN call. The mutation operator will

default to the uniform operator for fixed-length vector encoding with bounds speci-
fied 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 (Experimental)

set the objective function for a genetic algorithm optimization

> **CALL GASETOBJ(*id, type* <, *parm* >);**

The inputs 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 de-
     scribed below.

IML enables the specification of a user-written module to be used to compute the
value of the objective function to be optimized, or a standard preset function may be
specified. This is specified with the *type* and *parm* parameters.

type = 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 representing a given solution,
     and return a scalar numeric value for the objective function.

type = 1: specifies that a user-written function module is to be maximized. The
     name of the module is supplied in the *parm* parameter. The specified
     module should take a single parameter representing a given solution,
     and return a scalar numeric value for the objective function.

type = 2: specifies an objective function from the Traveling Salesman Problem,
     which will be minimized. This option is only valid if sequence encod-
     ing is specified in the corresponding GASETUP function call, and the
     solution vector is to be interpreted as a circular route, with each ele-
     ment representing 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 will be called once for each solution, to evaluate the
objective values for the GAREGEN call, GAINIT call, and the GAREEVAL call.
Also, the objective values for the current solution population will be re-evaluated if
GASETOBJ is called after a GAINIT call.

See the GASETUP function for an example.

GASETSEL Call (Experimental)

set the selection parameters for a genetic algorithm optimization

> **CALL GASETSEL(** *id, elite, type, parm* **);**

The inputs 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 will be 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 will be used in the GAREGEN call to select solutions for the crossover operation. IML currently allows you to 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* = 1, the group size is set to 2, and the best solution is chosen with probability specified by by *parm*. If *parm* is 1 the best solution is always picked, and a *parm* value of 0.5 is equivalent to pure random selection. The *parm* value must be between 0.5 and 1. The first variation, *type* = 0, will always produce a selective pressure greater than for *type* = 1. Higher selective pressure will lead 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* will generally lead to faster convergence of the algorithm, but will increase the chances of premature convergence to a local optimum. If GASETSEL is not called, default values of *elite* = 1, *type* = 1 and *parm* = 2 will be used in the optimization.

See the GASETUP function for an example.

GASETUP Function (Experimental)

set up the problem encoding for a genetic algorithm optimization problem

> **GASETUP(** *encoding, size* <*, seed* > **)**

The GASETUP function returns a scalar number identifying the genetic algorithm optimization problem, to be used in subsequent calls in setting up and executing the optimization.

The inputs 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 is used to indicate a numeric matrix of arbitrary dimensions, 1 to indicate a fixed-length floating point row vector, 2 to indicate a fixed-length integer row vector, and 3 to indicate 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 will be 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 to use to initialize 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 optimization problem should be terminated with a call to GAEND to free up resources associated with the genetic algorithm.

The following example demonstrates use of many 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) );

   /* calculate 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);
```

GBLKVP Call

defines a blanking viewport

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

The inputs to the GBLKVP subroutine are as follows:

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

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

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

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

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

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

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

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

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

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

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

GBLKVPD Call

deletes the blanking viewport

> **CALL GBLKVPD;**

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

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

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

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

GCLOSE Call

closes the graphics segment

> **CALL GCLOSE;**

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

GDELETE Call

deletes a graphics segment

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

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

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

An example of a valid statement follows.

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

The segment can also be specified as a quoted literal:

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

GDRAW Call

draws a polyline

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

The inputs to the GDRAW subroutine are as follows:

x is a vector containing the x coordinates of points used to draw a sequence of lines.

y is a vector containing the y coordinates of points used to draw a sequence of lines.

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

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

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

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

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

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

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

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

GDRAWL Call

draws individual lines

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

The inputs to the GDRAWL subroutine are as follows:

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

<div style="margin-left:2em">{*minimum-x minimum-y maximum-x maximum-y*}</div>

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

<div style="margin-left:2em">{*minimum-x minimum-y maximum-x maximum-y*}</div>

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

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

GENEIG Call

computes eigenvalues and eigenvectors of a generalized eigenproblem

CALL GENEIG(*eigenvalues, eigenvectors, symmetric-matrix1,*

*symmetric-matrix2***);**

The inputs to the GENEIG subroutine are as follows:

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

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

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

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

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

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

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

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

The matrices produced are as follows.

```
             M
             1.49235
             1.10928
             0.94385
             0.66366
             0.43278

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

GGRID Call

draws a grid

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

The inputs to the GGRID subroutine are as follows:

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

$$\{\textit{minimum-x minimum-y maximum-x maximum-y}\}$$

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

$$\{\textit{minimum-x minimum-y maximum-x maximum-y}\}$$

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

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

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

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

GINCLUDE Call

includes a graphics segment

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

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

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

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

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

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

GINV Function

computes the generalized inverse

GINV(*matrix***)**

where *matrix* is a numeric matrix or literal.

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

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

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

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

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

Least-squares regression for the model

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

can be performed by using

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

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

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

The program below 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;
```

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
```

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 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

opens a graphics segment

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

The inputs to the GOPEN subroutine are as follows:

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

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

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

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

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

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

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

GOTO Statement

jumps to a new statement

> **GOTO** *label*;

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

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

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

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

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

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

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

GPIE Call

draws pie slices

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

The inputs to the GPIE subroutine are AS FOLLOWS:

x and y are numeric scalars (or possibly vectors) defining the center (or centers) of the pie (or pies).

r is a scalar or vector giving the radii of the pie slices.

angle1   is a scalar or vector giving the start angles. It defaults to 0.

angle2   is a scalar or vector giving the terminal angles. It defaults to 360.

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

outline   is an index indicating the side of the slice to draw. The default is 3.

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

window   is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form
                {*minimum-x minimum-y maximum-x maximum-y*}

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

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

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

<0   uses absolute value as the line style and draws no line segment from center to arc.

0   draws no line segment from center to arc.

1   draws an arc and line segment from the center to the starting angle point.

2   draws an arc and line segment from the center to the ending angle point.

3   draws all sides of the slice. This is the default.

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

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

GPIEXY Call

converts from polar to world coordinates

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

The inputs to the GPIEXY subroutine are as follows:

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

$$\{minimum\text{-}x \ minimum\text{-}y \ maximum\text{-}x \ maximum\text{-}y\}$$

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

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

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

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

GPOINT Call

plots points

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

The inputs to the GPOINT subroutine are as follows:

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

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

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

```
x=30:80;
y=x;
call gpoint(x,y);
call gshow;
```

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

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

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

GPOLY Call

draws and fills a polygon

CALL GPOLY(*x, y* <*, style*><*, ocolor*><*, pattern*><*, color*>

<*, window*><*, viewport*>**);**

The inputs to the GPOLY subroutine are as follows.

x	is a vector defining the x coordinates of the corners of the polygon.
y	is a vector defining the y coordinates of the corners of the polygon.
style	is a numeric matrix or literal that specifies an index corresponding to a valid line style.
ocolor	is a matrix or literal specifying a valid outline color. The *ocolor* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
pattern	is a character matrix or quoted literal that specifies the pattern to fill the interior of a closed curve.
color	is a valid SAS color used in filling the polygon. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}
viewport	is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form {*minimum-x minimum-y maximum-x maximum-y*}

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

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

GPORT Call

defines a viewport

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

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

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

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

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

The default values for viewport are 0 0 100 100.

GPORTPOP Call

pops the viewport

> **CALL GPORTPOP;**

The GPORTPOP subroutine deletes the top viewport from the stack.

GPORTSTK Call

stacks the viewport

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

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

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

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

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

GSCALE Call

calculates round numbers for labeling axes

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

The inputs to the GSCALE subroutine are as follows:

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

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

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

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

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

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

GSCRIPT Call

writes multiple text strings with special fonts

CALL GSCRIPT(x, y, *text<, angle><, rotate><, height><, font>*

*<, color><, window><, viewport>***);**

The inputs to the GSCRIPT subroutine are as follows:

x	is a scalar or vector containing the x coordinates of the lower left starting position of the text string's first character.
y	is a scalar or vector containing the y coordinates of the lower left starting position of the text string's first character.
text	is a character vector of text strings.
angle	is the slant of each text string.
rotate	is the rotation of individual characters.
height	is a real number specifying the character height.
font	is a character matrix or quoted literal that specifies a valid font name.
color	is a valid SAS color. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form

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

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

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

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

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

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

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

GSET Call

sets attributes for a graphics segment

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

The inputs to the GSET subroutine are as follows:

attribute   is a graphics attribute. The *attribute* argument can be a character matrix or quoted literal.

value   is the value to which the attribute is set. The *value* argument is specified as a matrix or quoted literal.

The GSET subroutine enables you to change the following attributes for the current graphics segment.

aspect   a numeric matrix or literal that specifies the aspect ratio (width relative to height) for characters.

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

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

height   a numeric matrix or literal that specifies the character height.

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

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

thick   an integer specifying line thickness.

To reset the IML default value for any one of the attributes, omit the second argument. Attributes are reset back to the default with a call to the GOPEN or GSTART subroutines. Single or double quotes can be used around this argument. For more information on the attributes, see Chapter 12, "Graphics Examples."

Examples of valid statements follow:

```
call gset('pattern','m1n45');
call gset('font','simplex');

f='font';
s='simplex';
call gset(f,s);
```

For example, the statement

```
call gset("color");
```

resets *color* to its default.

GSHOW Call

shows a graph

> **CALL GSHOW** $<$**(***segment-name***)**$>$**;**

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

If you do not specify *segment-name*, the GSHOW subroutine displays the current graph. If the current graph is active at the time that the GSHOW subroutine is called, it remains active after the call; that is, graphics primitives can still be added to the segment. On the other hand, if you specify *segment-name*, the GSHOW subroutine closes any active graphics segment, searches the current catalog for a segment with the given name, and then displays that graph. Examples of valid statements follow.

```
call gshow;
call gshow("plot_a5");

seg={myplot};
call gshow(seg);
```

GSORTH Call

computes the Gram-Schmidt orthonormalization

> **CALL GSORTH(** p, t, *lindep*, a**);**

The inputs to the GSORTH subroutine are as follows:

p	is an $m \times n$ column-orthonormal output matrix.
t	is an upper triangular $n \times n$ output matrix.
lindep	is a flag with a value of 0 if columns of a are independent and a value of 1 if they are dependent. The *lindep* argument is an output scalar.
a	is an input $m \times n$ matrix.

The GSORTH subroutine computes the Gram-Schmidt orthonormal factorization of the $m \times n$ matrix $\mathbf{A}$, where m is greater than or equal to n; that is, the GSORTH subroutine computes the column-orthonormal $m \times n$ matrix $\mathbf{P}$ and the upper triangular $n \times n$ matrix $\mathbf{T}$ such that

$$\mathbf{A} = \mathbf{P} * \mathbf{T} .$$

If the columns of $\mathbf{A}$ are linearly independent (that is, rank$(\mathbf{A}) = n$), then $\mathbf{P}$ is full-rank column-orthonormal: $\mathbf{P}'\mathbf{P} = \mathbf{I}_w$, $\mathbf{T}$ is nonsingular, and the value of *lindep* (a scalar) is set to 0. If the columns of $\mathbf{A}$ are linearly dependent (say rank$(\mathbf{A}) = k < n$) then $n - k$ columns of $\mathbf{P}$ are set to 0, the corresponding rows of $\mathbf{T}$ are set to 0 ($\mathbf{T}$ is singular), and *lindep* is set to 1. The pattern of zero columns in $\mathbf{P}$ corresponds to the pattern of linear dependencies of the columns of $\mathbf{A}$ when columns are considered in left-to-right order.

The GSORTH subroutines implements an algorithm described by Golub (1969).

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.

If *lindep* is 1, you can rearrange the columns of $\mathbf{P}$ and rows of $\mathbf{T}$ so that the zero columns of $\mathbf{P}$ are right-most, that is, $\mathbf{P} = (\mathbf{P}(,1), \mathbf{P}(,k), 0, \ldots, 0)$, where k is the column rank of $\mathbf{A}$ and $\mathbf{A} = \mathbf{P} * \mathbf{T}$ is preserved. The following statements make this rearrangement:

```
d=rank((ncol(t)-(1:ncol(t))`)#(vecdiag(t)=0));
temp=p;
p[,d]=temp;
temp=t;
t[,d]=temp;
```

An example of a valid GSORTH call follows:

```
x={1 1 1, 1 2 4, 1 3 9};
xpx=x`*x;
call gsorth(p, t, l, xpx);
```

These statements produce the output matrices

```
P               3 rows      3 cols     (numeric)

              0.193247 -0.753259 0.6286946
              0.386494 -0.530521 -0.754434
             0.9018193 0.3887787 0.1886084


T               3 rows      3 cols     (numeric)

             15.524175 39.035892 104.99753
                     0 2.0491877 8.4559365
                     0         0 0.1257389


L               1 row       1 col      (numeric)

                                 0
```

GSTART Call

initializes the graphics system

> **CALL GSTART(** <*catalog*><, *replace*>**);**

The inputs to the GSTART subroutine are as follows:

catalog   is a character matrix or quoted literal specifying the SAS catalog for saving the graphics segments.

replace   is a numeric argument.

The GSTART subroutine activates the graphics system the first time it is called. A catalog is opened to capture the graphics segments to be generated in the session. If you do not specify a catalog, IML uses the temporary catalog WORK.GSEG.

The *replace* argument is a flag; a nonzero value indicates that the new segment should replace the first found segment with the same name. The *replace* flag set by the GSTART subroutine is a global flag, as opposed to the *replace* flag set by the GOPEN subroutine. When set by GSTART, this flag is applied to all subsequent segments created for this catalog, whereas with GOPEN, the *replace* flag is applied only to the segment that is being created. The GSTART subroutine sets the *replace* flag to 0 when the *replace* argument is omitted. The *replace* option can be very inefficient for a catalog with many segments. In this case, it is better to create segments with different names (if necessary) than to use the *replace* option.

The GSTART subroutine must be called at least once to load the graphics subsystem. Any subsequent GSTART calls are generally to change graphics catalogs or reset the global *replace* flag.

The GSTART subroutine resets the defaults for all graphics attributes that can be changed by the GSET subroutine. It does not reset GOPTIONS back to their defaults unless the GOPTION corresponds to a GSET parameter. The GOPEN subroutine also resets GSET parameters.

An example of a valid statement follows:

```
call gstart;
```

GSTOP Call

deactivates the graphics system

CALL GSTOP;

The GSTOP subroutine deactivates the graphics system. The graphics subsystem is disabled until the GSTART subroutine is called again.

GSTRLEN Call

finds the string length

CALL GSTRLEN(*length, text*<*, height*><*, font*><*, window*>**);**

The inputs to the GSTRLEN subroutine are as follows:

length   is a matrix of lengths specified in world coordinates.

text   is a matrix of text strings.

height   is a numeric matrix or literal specifying the character height.

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

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

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

The GSTRLEN subroutine returns in world coordinates the graphics text lengths in a given font and for a given character height. The *length* argument is the returned matrix. It has the same shape as the matrix *text*. Thus, if *text* is an $n \times m$ matrix of text strings, then *length* will be an $n \times m$ matrix of lengths in world coordinates. If you do not specify *font*, the default font is assumed. If you do not specify *height*, the default height is assumed. An example using the GSTRLEN subroutine follows.

```
    /* centers text strings about coordinate    */
    /* points (50, 90) assume font=simplex       */
ht=2;
x=30;
y=90;
str='Nonparametric Cluster Analysis';
call gstrlen(len, str, ht, 'simplex');
call gscript(x-(len/2), y, str, ,,ht,'simplex');
```

GTEXT and GVTEXT Calls

place text horizontally or vertically on a graph

> **CALL GTEXT(** x, y, *text*<, *color*><, *window*><, *viewport*>**);**

> **CALL GVTEXT(** x, y, *text*<, *color*><, *window*><, *viewport*>**);**

The inputs to the GTEXT and GVTEXT subroutines are as follows:

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

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

text is a vector of text strings

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

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

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

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

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

The GTEXT subroutine places text horizontally across a graph; the GVTEXT subroutine places text vertically on a graph. Both subroutines use hardware characters when possible. The number of text strings drawn is the maximum dimension of the first three vectors. The *color* argument can have more than one element. Hardware characters cannot always be obtained if you change the HEIGHT or ASPECT parameters (using GSET or GOPTIONS) or if you use a viewport. The coordinates in use for this graphics command are world coordinates.

Examples of the GTEXT and GVTEXT subroutines follow:

```
call gopen;
call gport({0 0 50 50});
call gset('height',4); /* shrink to a 4th of the screen */
call gtext(50,50,'Testing GTEXT: This will start in the
                   center of the viewportp ');
call gshow;
call gopen;
call gvtext(.35,4.6,'VERTICAL STRING BY GVTEXT',
            'white',{0.2 -1,1.5 6.5},{0 0,100 100});
call gshow;
```

GWINDOW Call

defines the data window

CALL GWINDOW(*window*);

where *window* is a numeric matrix or literal specifying a window. The rectangular area's boundary is given in world coordinates, where you specify the lower left and upper right corners in the form

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

The GWINDOW subroutine sets up the window for scaling data values in subsequent graphics primitives. It is in effect until the next GWINDOW call or until the segment is closed. The coordinates in use for this graphics command are world coordinates. An example using the GWINDOW subroutine follows:

```
ydata={2.358,0.606,3.669,1.000,0.981,1.192,0.926,1.590,
        1.806,1.962,4.028,3.148,1.836,2.845,1.013,0.414};
xdata={1.215,0.930,1.152,1.138,0.061,0.696,0.686,1.072,
        1.074,0.934,0.808,1.071,1.009,1.142,1.229,0.595};

    /* WD shows the actual range of the data */
wd=(min(xdata)||min(ydata))//(max(xdata)||max(ydata));
call gwindow(wd);
```

GXAXIS and GYAXIS Calls

draw a horizontal or vertical axis

CALL GXAXIS(*starting-point, length, nincr* <*, nminor*><*, noticklab*>

<*, format*><*, height*><*, font*><*, color*><*, fixed-end*>

<*, window*><*, viewport*>**);**

CALL GYAXIS(*starting-point, length, nincr* <*, nminor*><*, noticklab*>

<*, format*><*, height*><*, font*><*, color*><*, fixed-end*>

<*, window*><*, viewport*>**);**

The inputs to the GXAXIS and GYAXIS subroutines are as follows:

starting-point	is the (x, y) starting point of the axis, specified in world coordinates.
length	is a numeric scalar giving the length of the axis.
nincr	is a numeric scalar giving the number of major tick marks on the axis.
nminor	is an integer specifying the number of minor tick marks between major tick marks.
noticklab	is a flag that is nonzero if the tick marks are not labeled. The default is to label tick marks.
format	is a character scalar that gives a valid SAS numeric format used in formatting the tick-mark labels. The default format is 8.2.
height	is a numeric matrix or literal that specifies the character height. This is used for the tick-mark labels.
font	is a character matrix or quoted literal that specifies a valid font name. This is used for the tick-mark labels.
color	is a valid color. The *color* argument can be specified as a quoted text string (such as 'RED'), the name of a character matrix containing a valid color as an element, or a color number (such as 1). A color number n refers to the nth color in the color list.
fixed-end	allows one end of the scale to be held fixed. **U** fixes the upper end; **L** fixes the lower end; **X** and allows both ends to vary from the data values. In addition, you may specify **N**, which causes the axis routines to bypass the scaling routine. The interval between tick marks is *length* divided by (*nincr*−1). The default is **X**.
window	is a numeric matrix or literal specifying a window. This is given in world coordinates and has the form
	{*minimum-x minimum-y maximum-x maximum-y*}

> *viewport*   is a numeric matrix or literal specifying a viewport. This is given in normalized coordinates and has the form
>
> {*minimum-x minimum-y maximum-x maximum-y*}

The GXAXIS subroutine draws a horizontal axis; the GYAXIS subroutine draws a vertical axis. The first three arguments are required.

The *starting-point* argument is a matrix of two numbers given in world coordinates. The matrix is the (x, y) starting point of the axis.

The *length* argument is a scalar value giving the length of the x axis or y axis in world coordinates along the x or y direction.

The *nincr* argument is a scalar value giving the number of major tick marks shown on the axis. The first tick mark will be on the starting point as specified.

The axis routines use the same scaling algorithm as the GSCALE subroutine. For example, if the x starting point is 10 and the length of the axis is 44, and if you call the GSCALE subroutine with the x vector containing the two elements, 10 and 44, the scale obtained should be the same as that obtained by the GXAXIS subroutine. Sometimes, it may be helpful to use the GSCALE subroutine in conjunction with the axis routines to get more precise scaling and labeling.

For example, suppose you want to draw the axis for $-2 \leq X \leq 2$ and $-2 \leq Y \leq 2$. The code below draws these axes. Each axis is 4 units long. Note that the x axis begins at the point $(-2, 0)$ and the y axis begins at the point $(0, -2)$. The tick marks can be set at each integer value, with minor tick marks in between the major tick marks. The *noticklab* option is turned off, so that the tick marks are not labeled.

```
call gport({20 20 80 80});
call gwindow({-2 -2 2 2});
call gxaxis({-2,0},4,5,2,1);
call gyaxis({0,-2},4,5,2,1);
```

HALF Function

computes Cholesky decomposition

> **HALF(** *matrix***)**

where *matrix* is a numeric matrix or literal.

The HALF function is the same as the ROOT function. See the description of the ROOT function for Cholesky decomposition.

HANKEL Function

generates a Hankel matrix

    HANKEL(*matrix***)**

where *matrix* is a numeric matrix or literal.

The HANKEL function generates a Hankel matrix from a vector, or a block Hankel matrix from a matrix. A block Hankel matrix has the property that all matrices on the reverse diagonals are the same. The argument matrix is an $(np) \times p$ or $p \times (np)$ matrix; the value returned is the $(np) \times (np)$ result.

The Hankel function uses the first $p \times p$ submatrix $\mathbf{A}_1$ of the argument matrix as the blocks of the first reverse diagonal. The second $p \times p$ submatrix $\mathbf{A}_2$ of the argument matrix forms the second reverse diagonal. The remaining reverse diagonals are formed accordingly. After the values in the argument matrix have all been placed, the rest of the matrix is filled in with 0. If $\mathbf{A}$ is $(np) \times p$, then the first p columns of the returned matrix, $\mathbf{R}$, will be the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, then the first p rows of $\mathbf{R}$ will be the same as $\mathbf{A}$. The HANKEL function is especially useful in time-series applications, where the covariance matrix of a set of variables representing the present and past and a set of variables representing the present and future is often assumed to be a block Hankel matrix. If

$$\mathbf{A} = [\mathbf{A}_1 | \mathbf{A}_2 | \mathbf{A}_3 | \cdots | \mathbf{A}_n]$$

and if $\mathbf{R}$ is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & | & \mathbf{A}_2 & | & \mathbf{A}_3 & | & \cdots & | & \mathbf{A}_n \\ \mathbf{A}_2 & | & \mathbf{A}_3 & | & \mathbf{A}_4 & | & \cdots & | & \mathbf{0} \\ \mathbf{A}_3 & | & \mathbf{A}_4 & | & \mathbf{A}_5 & | & \cdots & | & \mathbf{0} \\ \vdots & & & & & & & & \\ \mathbf{A}_n & | & \mathbf{0} & | & \mathbf{0} & | & \cdots & | & \mathbf{0} \end{bmatrix}$$

If

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

and if $\mathbf{R}$ 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 IML code

```
r=hankel({1 2 3 4 5});
```

results in

```
     R              5 rows      5 cols     (numeric)

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

The statement

```
r=hankel({1 2 ,
         3 4 ,
         5 6 ,
         7 8});
```

returns the matrix

```
     R              4 rows      4 cols     (numeric)

                  1         2         5         6
                  3         4         7         8
                  5         6         0         0
                  7         8         0         0
```

And the statement

```
r=hankel({1 2 3 4 ,
         5 6 7 8});
```

returns the result

```
     R              4 rows      4 cols     (numeric)

                  1         2         3         4
                  5         6         7         8
                  3         4         0         0
                  7         8         0         0
```

HDIR Function

performs a horizontal direct product

HDIR(*matrix1, matrix2***)**

where *matrix1* and *matrix2* are numeric matrices or literals.

The HDIR function performs a direct product on all rows of *matrix1* and *matrix2* and creates a new matrix by stacking these row vectors into a matrix. This operation is useful in constructing design matrices of interaction effects. The *matrix1* and *matrix2* arguments must have the same number of rows. The result has the same number of rows as *matrix1* and *matrix2*. The number of columns is equal to the product of the number of columns in *matrix1* and *matrix2*.

For example, the statements

```
a={1 2,
   2 4,
   3 6};
b={0  2,
   1  1,
   0 -1};
c=hdir(a,b);
```

produce a matrix containing the values

```
        C           3 rows      4 cols    (numeric)

                    0      2      0      4
                    2      2      4      4
                    0     -3      0     -6
```

The HDIR function is useful for constructing crossed and nested effects from main effect design matrices in ANOVA models.

HERMITE Function

reduces a matrix to Hermite normal form

HERMITE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The HERMITE function uses elementary row operations to reduce a matrix to Hermite normal form. For square matrices this normal form is upper-triangular and idempotent.

If the argument is square and nonsingular, the result will be the identity matrix. In general the result satisfies the following four conditions (Graybill 1969, p. 120):

- It is upper-triangular.
- It has only values of 0 and 1 on the diagonal.
- If a row has a 0 on the diagonal, then every element in that row is 0.
- If a row has a 1 on the diagonal, then every off-diagonal element is 0 in the column in which the 1 appears.

Consider the following example (Graybill 1969, p. 288):

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

These statements produce

```
H                 3 rows        3 cols      (numeric)

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

If the argument is a square matrix, then the Hermite normal form can be transformed into the row echelon form by rearranging rows in which all values are 0.

HOMOGEN Function

solves homogeneous linear systems

> **HOMOGEN(** *matrix***)**

where *matrix* is a numeric matrix or literal.

The HOMOGEN function solves the homogeneous system of linear equations $\mathbf{A} * \mathbf{X} = \mathbf{0}$ for $\mathbf{X}$. For at least one solution vector $\mathbf{X}$ to exist, the $m \times n$ matrix $\mathbf{A}$, $m \geq n$, has to be of rank $r < n$. The HOMOGEN function computes an $n \times (n - r)$ column orthonormal matrix $\mathbf{X}$ with the property $\mathbf{A} * \mathbf{X} = \mathbf{0}$, $\mathbf{X}'\mathbf{X} = \mathbf{I}$. If $\mathbf{A}'\mathbf{A}$ is ill conditioned, rounding-error problems can occur in determining the correct rank of $\mathbf{A}$ and in determining the correct number of solutions $\mathbf{X}$. Consider the following example (Wilkinson and Reinsch 1971, p. 149):

```
a={22  10   2   3   7,
   14   7  10   0   8,
   -1  13  -1 -11   3,
   -3  -2  13  -2   4,
    9   8   1  -2   4,
    9   1  -7   5  -1,
    2  -6   6   5   1,
    4   5   0  -2   2};
x=homogen(a);
```

These statements produce the solution

```
X                 5 rows        2 cols      (numeric)

              -0.419095              0
               0.4405091      0.4185481
              -0.052005       0.3487901
               0.6760591       0.244153
               0.4129773     -0.802217
```

In addition, this function could be used to determine the rank of an $m \times n$ matrix $\mathbf{A}$, $m \geq n$.

If X 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 perfoming its computation.

I Function

creates an identity matrix

> **I(** *dimension***)**

where *dimension* specifies the size of the identity matrix.

The I function creates an identity matrix with *dimension* rows and columns. The diagonal elements of an identity matrix are 1s; all other elements are 0s. The value of *dimension* must be an integer greater than or equal to 1. Noninteger operands are truncated to their integer part.

For example, the statement

```
a=I(3);
```

yields the result

```
            A
            1   0   0
            0   1   0
            0   0   1
```

IF-THEN/ELSE Statement

conditionally executes statements

> **IF** *expression* **THEN** *statement1***;**
>
> **ELSE** *statement2***;**

The inputs to the IF-THEN/ELSE statements are

expression	is an expression that is evaluated for being true or false.
statement1	is a statement executed when *expression* is true.
statement2	is a statement executed when *expression* is false.

The IF statement contains an expression to be evaluated, the keyword THEN, and an action to be taken when the result of the evaluation is true.

The ELSE statement optionally follows the IF statement and gives an action to be taken when the IF expression is false. The expression to be evaluated is often a comparison, for example,

```
if max(a)<20 then p=0;
else p=1;
```

The IF statement results in the evaluation of the condition MAX(A)<20. If the largest value found in matrix **A** is less than 20, **P** is set to 0. Otherwise, **P** is set to 1. See the description of the MAX function for details.

When the condition to be evaluated is a matrix expression, the result of the evaluation is another matrix. If all values of the result matrix are nonzero and nonmissing, the condition is true; if any element in the result matrix is 0 or missing, the condition is false. This evaluation is equivalent to using the ALL function.

For example, writing

```
if x<y then
   do;
```

produces the same result as writing

```
if all(x<y) then
   do;
```

IF statements can be nested within the clauses of other IF or ELSE statements. Any number of nesting levels is allowed. Below is an example.

```
if x=y then if abs(y)=z then
   do;
```

CAUTION: Execution of THEN clauses occurs as if you were using the ALL function.

The statements

```
if a^=b then do;
```

and

```
if ^(a=b) then do;
```

are both valid, but the THEN clause in each case is only executed when all corresponding elements of **A** and **B** are unequal; that is, when none of the corresponding elements are equal.

Evaluation of the statement

```
if any(a^=b) then do;
```

requires only one element of **A** and **B** to be unequal for the expression to be true.

IFFT Function

computes the inverse finite Fourier transform

    IFFT(*f*)

where f is an $np \times 2$ numeric matrix.

The IFFT function expands a set of sine and cosine coefficients into a sequence equal to the sum of the coefficients times the sine and cosine functions. The argument f is an $np \times 2$ matrix; the value returned is an $n \times 1$ vector.

Note: If the element in the last row and second column of f is exactly 0, then n is $2np - 2$; otherwise, n is $2np - 1$.

The inverse finite Fourier transform of a two column matrix $\mathbf{F}$, denoted by the vector $\mathbf{x}$ is

$$x_i = F_{1,1} + 2 \sum_{j=2}^{np} \left(F_{j,1} \cos\left(\frac{2\pi}{n}(j-1)(i-1) \right) + F_{j,2} \sin\left(\frac{2\pi}{n}(j-1)(i-1) \right) \right) + q_i$$

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.

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

IFFT(FFT(X)) returns n times $\mathbf{x}$, where n is the dimension of $\mathbf{x}$. If f is not the Fourier transform of a real sequence, then the vector generated by the IFFT function is not a true inverse Fourier transform. However, applications exist where the FFT and IFFT functions may be used for operations on multidimensional or complex data (Gentleman and Sande 1966; Nussbaumer 1982).

The convolution of two vectors $\mathbf{x}$ ($n \times 1$) and $\mathbf{y}$ ($m \times 1$) can be accomplished using the following statements:

```
a=fft(x//j(nice-nrow(x),1,0));
b=fft(y//j(nice-nrow(y),1,0));
z=(a#b)[,+];
b[,2]=-b[,2];
z=ifft(z||((a#(b[,2])))[,+]);
```

where NICE is a number chosen to allow efficient use of the FFT and IFFT functions and also is greater than $n + m$.

Windowed spectral estimates and inverse autocorrelation function estimates can also be readily obtained.

INDEX Statement

indexes a variable in a SAS data set

> **INDEX** *variables*|**NONE**

where *variables* are the names of variables for which indexes are to be built.

You can use the INDEX statement to create an index for the named variables in the current input SAS data set. An index is created for each variable listed if it does not already have an index. Current retrieval is set to the last variable indexed. Subsequent I/O operations such as LIST, READ, FIND, and DELETE may use this index to retrieve observations from the data set if IML determines that indexed retrieval will be faster. The indices are automatically updated when a data set is edited with the APPEND, DELETE, or REPLACE statements. Only one index is in effect at any given time. The SHOW *contents* command indicates which index is in use.

For example, the following statement creates indexes for the SAS data set CLASS in the order of NAME and the order of SEX:

```
    index name sex;
```

Current retrieval is set to use SEX. A LIST*all* statement would list females before males.

An INDEX *none* statement can be used to set retrieval back to physical order.

When a WHERE clause is being processed, IML automatically determines which index to use, if any. The decision is based on the variables and operators involved in the WHERE clause, and the decision criterion is based on the efficiency of retrieval.

INFILE Statement

opens a file for input

> **INFILE** *operand* <*options*>;

The inputs to the INFILE statement are as follows:

operand   is either a predefined filename or a quoted string containing the filename or character expression in parentheses referring to the filepath.

options   are explained below.

You can use the INFILE statement to open an external file for input or, if the file is already open, to make it the current input file so that subsequent INPUT statements read from it.

The options available for the INFILE statement are described as follows.

LENGTH=variable
specifies a variable in which the length of a record will be stored as IML reads it in.

RECFM=N

specifies that the file is to be read in as a pure binary file rather than as a file with record separator characters. To do this, you must use the byte operand (<) on the INPUT statement to get new records rather than using separate input statements or the new line (/) operator.

The following options control how IML behaves when an INPUT statement tries to read past the end of a record. The default is STOPOVER.

FLOWOVER

allows the INPUT statement to go to the next record to obtain values for the variables.

MISSOVER

tolerates attempted reading past the end of the record by assigning missing values to variables read past the end of the record.

STOPOVER

treats going past the end of a record as an error condition, which triggers an end-of-file condition.

Several examples of INFILE statements are given below:

```
filename in1 'student.dat';  /* specify filename IN1   */
infile in1;                  /* infile filepath        */

infile 'student.dat';        /* path by quoted literal */

infile 'student.dat' missover;    /* using options   */
```

See Chapter 7, "File Access," for further information.

INPUT Statement

inputs data

> **INPUT** <*variables*> <*informats*> <*record-directives*> <*positionals*>;

where the clauses and options are explained below.

You can use the INPUT statement to input records from the current input file, placing the values into IML variables. The INFILE statement sets up the current input file. See Chapter 7, "File Access," , for details.

The INPUT statement contains a sequence of arguments that include the following:

variables	specify the variable or variables you want to read from the current position in the record. Each variable can be followed immediately by an input format specification.
informats	specify an input format. These are of the form $w.d$ or $\$w.$ for standard numeric and character informats, respectively, where w is the width of the field and d is the decimal parameter, if any. You can also use a SAS format of the form *NAMEw.d*,

where *NAME* is the name of the format. Also, you can use a single $ or & for list input applications. If the width is unspecified, the informat uses list-input rules to determine the length by searching for a blank (or comma) delimiter. The special format $RECORD. is used for reading the rest of the record into one variable. For more information on formats, refer to *SAS Language Reference: Dictionary*.

Record holding is always implied for RECFM=N binary files, as if the INPUT statement has a trailing @ sign. For more information, see Chapter 7, "File Access."

Examples of valid INPUT statements are shown below:

```
input x y;
input @1 name $ @20 sex $ @(20+2) age 3.;

eight=8;
input >9 <eight  number2 ib8.;
```

Below is an example using binary input:

```
file 'out2.dat' recfm=n ;
number=499; at=1;
do i = 1 to 5;
   number=number+1;
   put >at number ib8.; at=at+8;
end;
closefile 'out2.dat';

infile 'out2.dat' recfm=n;
size=8;      /* 8 bytes */
do pos=1 to 33 by size;
   input >pos number ib8.;
   print number;
end;
```

record-directives are used to advance to a new record. *Record-directives* are the following:

holding @ sign is used at the end of an INPUT statement to instruct IML to hold the current record so that you can continue to read from the record with later INPUT statements. Otherwise, IML automatically goes to the next record for the next INPUT statement.

/ advances to the next record.

> *operand* specifies that the next record to be read starts at the indicated byte position in the file (for RECFM= N files only). The *operand* is a literal number, a variable name, or an expression in parentheses.

<table>
<tr><td>< *operand*</td><td>instructs IML to read the indicated number of bytes as the next record. The record directive must be specified for binary files (RECFM=N). The *operand* is a literal number, a variable name, or an expression in parentheses.</td></tr>
</table>

positionals	instruct PROC IML to go to a specific column on the record. The *positionals* are the following:

@ *operand*	instructs IML to go to the indicated column, where *operand* is a literal number, a variable name, or an expression in parentheses. For example, @30 means to go to column 30. The operand can also be a character operand when pattern searching is needed. For more information, see Chapter 7, "File Access."
+ *operand*	specifies to skip the indicated number of columns. The *operand* is a literal number, a variable name, or an expression in parentheses.

INSERT Function

inserts one matrix inside another

> **INSERT(** x, y, *row*<, *column*>**)**

The inputs to the INSERT function are as follows:

x	is the target matrix. It can be either numeric or character.
y	is the matrix to be inserted into the target. It can be either numeric or character, depending on the type of the target matrix.
row	is the row where the insertion is to be made.
column	is the column where the insertion is to be made.

The INSERT function returns the result of inserting the matrix y inside the matrix x at the place specified by the *row* and *column* arguments. This is done by splitting x either horizontally or vertically before the row or column specified and concatenating y between the two pieces. Thus, if x is m rows by n columns, *row* can range from 0 to $m + 1$ and *column* can range from 0 to $n + 1$. However, it is not possible to insert in both dimensions simultaneously, so either *row* or *column* must be 0, but not both. The *column* argument is optional and defaults to 0. Also, the matrices must conform in the dimension in which they are joined.

For example, the statements

```
a={1 2, 3 4};
b={5 6, 7 8};
c=insert(a, b, 2, 0);
d=insert(a, b, 0, 3);
```

produce the result

```
              C              4 rows       2 cols      (numeric)

                                 1            2
                                 5            6
                                 7            8
                                 3            4

              D              2 rows       4 cols      (numeric)

                    1            2            5            6
                    3            4            7            8
```

C shows the result of insertion in the middle, while **D** shows insertion on an end.

INT Function

truncates a value

INT(*matrix***)**

where *matrix* is a numeric matrix or literal.

The INT function truncates the decimal portion of the value of the argument. The integer portion of the value of the argument remains. The INT function takes the integer value of each element of the argument matrix.

An example using the INT function follows:

```
c=2.8;
b=int(c);
```

```
              B              1 row        1 col       (numeric)

                                 2
```

In the next example, notice that a value of 11 is returned. This is because of the maximal machine precision. If the difference is less than $1E{-}12$, the INT function rounds up.

```
x={12.95  10.9999999999999,
   -30.5  1e-6};
b=int(x);
```

```
              B              2 rows       2 cols      (numeric)

                                12           11
                               -30            0
```

INV Function

computes the matrix inverse

> **INV(** *matrix***)**

where *matrix* is a square nonsingular matrix.

The INV function produces a matrix that is the inverse of *matrix*, which must be square and nonsingular.

For $\mathbf{G} = \text{INV}(\mathbf{A})$ 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 statement

```
x=inv(a)*b;
```

However, the SOLVE function is more accurate and efficient for this task.

An example of valid usage is

```
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 using a more efficient algorithm */
x2 = solve(A,b);
```

which produces the output

	X1	X2
	3	3
	1	1
	4	4
	1	1
	5	5

The INV function uses an LU decomposition followed by backsubstitution to solve for the inverse, as described in Forsythe, Malcolm, and Moler (1967).

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

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

where *MACHEPS* is the relative machine precision.

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

Previously, a much smaller singularity criterion was used, which caused algebraic operations to be performed on values that were essentially floating point error. This occasionally yielded numerically unstable results. The new criterion is much more conservative, and it generates far fewer erroneous results. In some cases, you may need to scale the data to avoid singular matrices. If you think the new criterion is too strong,

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

If A is an $n \times n$ matrix, the INV function allocates an 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

updates a matrix inverse

INVUPDT(*matrix, vector<, scalar>***)**

The inputs to the INVUPDT function are as follows:

matrix   is an $n \times n$ nonsingular matrix. In most applications *matrix* will be 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 is used primarily to update a matrix inverse. The 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 calling the INVUPDT function as

```
R=invupdt(M,X,w);
```

computes the matrix

$$\mathbf{R} = \mathbf{M} - w\mathbf{M}\mathbf{X}(\mathbf{I} + w\mathbf{X}'\mathbf{M}\mathbf{X})^{-1}\mathbf{X}'\mathbf{M}.$$

which 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

```
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;
```

The output is

<div align="center">

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

</div>

IPF Call

performs an iterative proportional fit of a contingency table

CALL IPF(*fit, status, dim, table, config*$<$*, initab*$><$*, mod*$>$**);**

The inputs 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. 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 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. The *initab* argument is an array of initial values for the iterative procedure. If you do not specify values, 1s are used. For incomplete tables, *initab* is set to 1 if the cell is included in the design, and 0 if it is not.

mod is an input matrix. The *mod* argument is a two-element vector specifying the stopping criteria. If *mod*= {*maxdev*, *maxit*}, then the procedure iterates until either the maximum difference between estimates of the last two iterations is less than *maxdev*, or until *maxit* iterations are completed. Default values are *maxdev*=0.25 and *maxit*=15.

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 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 algorithm 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 showing 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 may 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 will be 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 will have 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 much of the marginals as possible to the first cell, then assign as much of the remaining marginals 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 program below.

```
/* 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 given below. The same results are obtained if the *table* parameter is formed using the "independent algorithm."

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 below. 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. If the margins are specified as

```
NewVotes   = {100 100 100};
NewSection = {100,100,100};
```

then the program for marital status by age can be easily rewritten to adjust the votes into a standardized form. The result is

```
                   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 1-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 below. Afterwards the *table* parameter can be reshaped 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 1-way. For example, the three-dimensional table below is similar to one appearing 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 corresponding 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 program below encodes this table, uses the MARG call to compute a 2-way marginal table by summing over the third variable and a 1-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. Lastly, 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,

/* Prote-  Hi */ 359    233  109   197    90,
/* stant   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;
```

The results of this program are shown below.

```
     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 IPF (but see the comment on structural zeros below).

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 IPF. The variables are nested across columns, then across rows.

Crabmeat:	Y E S		N O	
Potato Salad:	Yes	No	Yes	No
Ill	120	4	22	0
Not Ill	80	31	24	23

The program below defines the variables and observations, 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 crab meat. The next model excludes the interaction between potato salad and illness. This model is rejected with a large chi-square value, so the data supports 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};

/* Crabmeat:   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 (Crabmeat) */
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 */
```

The associated output is

```
                 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


                 HYPOTH

Hypothesis: no interaction between Illness and Potatoes

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 interaction between Illness and Crab

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

A few additional comments on the examples are in order.

Structural versus Random Zeros In the marriage-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 will retain a zero in that category. By contrast, in the potato-crab-illness example, the *table* parameter contained a zero for number of illnesses observed among those who did not eat either crab meat or potato salad. This is a sampling (or *random*) zero. Some models may preserve that zero, others may 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. Refer to Bishop, Fienberg, and Holland (1975) or the documentation for the CATMOD procedure in SAS/STAT 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 table below 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 are

allowed. 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 on "Generating a Table with Given Marginals" 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 indicating the typical number of years spent in school: "<12," "12," "<16," "16," and ">16."

The table would be encoded as

```
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,

/* Prote-  Hi */ 359    233   109   197    90,
/* stant   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 get

```
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:

```
table={
/*      CATHOLIC              ...          PROTESTANT
   High          Low          ...       High        Low
<12 ... >16 <12 ... >16   ...   <12 ... >16 <12 ... >16
*/
575 ...   51 267 ...   19   ...    359 ... 90 159 ... 32
      };
```

ITBICG Call

solves a sparse linear system by biconjugate gradient iteration

> **CALL ITBICG(** *x, error, iter, A, b, <precon>, <tol>, <maxiter>, <start>,
> <history>***)**;

The ITBICG 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 inputs to the ITCBICG call are as follows:

A	is the sparse coefficient matrix in the equation $Ax = b$.
b	is a column vector, the right side of the equation $Ax = b$.
precon	is the name of a preconditioning technique to use.
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 ITCBICG call solves a sparse linear system by iterative methods, which involve updating a trial solution over successive iterations to minimize the error. The ITCBICG call uses a biconjugate gradient algorithm to update the solution. The algorithm does not require that the A matrix be positive-definite or symmetric.

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. 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

$$error = \|Ax - b\|_2 / (\|b\|_2 + \epsilon)$$

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 $maxiter$ are not specified in the call, then default values of 10^{-7} will be used for tol and 100000 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, which can take one of the following values:

'*NONE*': no preconditioning

'*MILU*': modified incomplete LU factorization

If $precon$ is not specified, no preconditioning will be applied.

A starting trial solution can be specified with the $start$ parameter, otherwise the ITBICG call will generate an all-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.

Your IML program should always check the returned $error$ and $iter$ parameters to verify that the desired relative error tolerance was reached. If not, your program might continue the solution process with another ITBICG call, with $start$ set to the latest result. You might also try a different $precon$ option to enhance convergence.

For example, solve the system:

$$
\begin{bmatrix}
3 & 2 & 0 & 0 \\
1.1 & 4 & 1 & 3.2 \\
0 & 1 & -10 & 0 \\
0 & 3.2 & 0 & 3
\end{bmatrix}
x =
\begin{bmatrix}
1 \\
1 \\
1 \\
1
\end{bmatrix}
$$

```
/* 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 itbicg(x, error, iter, A, b, 'milu', tol, maxiter, start, hist);
print x;
print iter error;
print hist;
```

The results are:

```
                   X

             0.2040816
             0.1938776
            -0.080612
             0.1265306


          ITER      ERROR

             3 3.494E-16


             HIST

          0.0254375
          0.0080432
          3.494E-16
                  .
                  .
                  .
                  .
                  .
                  .
                  .
```

ITCGRAD Call

solves a sparse symmetric positive-definite system by conjugate gradient iteration

> **CALL ITCGRAD(** *x, error, iter, A, b, <precon>, <tol>, <maxiter>, <start>, <history>***);**

The ITCGRAD 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 inputs to the ITCGRAD call are as follows:

A	is the sparse coefficient matrix in the equation $Ax = b$.
b	is a column vector, the right side of the equation $Ax = b$.
precon	is the name of a preconditioning technique to use.
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 ITCGRAD call solves a sparse symmetric positive-definite system by iterative methods, which involve updating a trial solution over successive iterations to minimize the error. The ITCGRAD call uses a conjugate gradient algorithm to update the solution. The algorithm requires that the A matrix be positive-definite and symmetric (the general symmetric case can be solved with an ITMINRES call, and the nonsymmetric case with the ITBICG call).

The input matrix A represents the coefficient matrix in sparse symmetric format. Only the diagonal and lower triangular nonzero elements are specified, and it is an error to specify an element in the upper triangular portion of A. A must be an n by 3 matrix, where n is the number of nonzero elements specified. The first column contains the nonzero values, while the second and third columns contain the row and column locations for the nonzero elements, respectively. 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

$$error = \|Ax - b\|_2 / (\|b\|_2 + \epsilon)$$

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 $maxiter$ are not specified in the call then default values of 10^{-7} will be used for tol and 100000 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, which can take one of the following values:

'*NONE*': no preconditioning

'*IC*': incomplete Cholesky factorization

'*DIAG*': diagonal Jacobi preconditioner

If *precon* is not specified, no preconditioning will be applied.

A starting trial solution can be specified with the *start* parameter, otherwise the ITCGRAD call will generate an all-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.

Your IML program should always check the returned *error* and *iter* parameters to verify that the desired relative error tolerance was reached. If not, your program might continue the solution process with another ITCGRAD call, with *start* set to the latest result. You might also try a different *precon* option to enhance convergence.

For example, 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 itcgrad(x, error, iter, A, b);

print x, iter, error;
```

The results are

```
                   X
```

```
                    2.68
                    -6.4
                    0.74
                    7.16

               ITER

                   4

              ERROR

             5.77E-15
```

Here is the same example, with all the inputs specified.

```
/* 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};
maxiter = 10;
hist = j(maxiter,1,.);
start = {1,1,1,1};
tol = 1e-10;
call itcgrad(x, error, iter, A, b, 'ic', tol, maxiter, start, hist);
print x;
print iter error;
print hist;
```

The results are

```
                    X

                   2.68
                   -6.4
                   0.74
                   7.16


        ITER      ERROR

           3 1.572E-16
```

```
              HIST

         0.5265479
          0.063117
         1.572E-16
                  .
                  .
                  .
                  .
                  .
                  .
                  .
```

ITMINRES Call

solves a sparse symmetric linear system by minimum residual iteration

> **CALL ITMINRES(** *x, error, iter, A, b, <precon>, <tol>, <maxiter>, <start>,*
> *<history>***);**

The ITMINRES 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 inputs to the ITMINRES call are as follows:

A	is the sparse coefficient matrix in the equation $Ax = b$.
b	is a column vector, the right side of the equation $Ax = b$.
precon	is the name of a preconditioning technique to use.
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 ITMINRES call solves a sparse symmetric linear system by iterative methods, which involve updating a trial solution over successive iterations to minimize the error. The ITMINRES call seeks to minimize the Euclidean norm of the residuals at each iteration. The algorithm requires that the A matrix be symmetric (the nonsymmetric case can be solved with the ITBICG call).

The input matrix A represents the coefficient matrix in sparse symmetric format. Only the diagonal and lower triangular nonzero elements are specified, and it is an error to specify an element in the upper triangular portion of A. A must be an n by

3 matrix, where n is the number of nonzero elements specified. The first column contains the nonzero values, while the second and third columns contain the row and column locations for the nonzero elements, respectively. 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

$$error = \|Ax - b\|_2/(\|b\|_2 + \epsilon)$$

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 $maxiter$ are not specified in the call then default values of 10^{-7} will be used for tol and 100000 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, which can take one of the following values:

'*NONE*': no preconditioning

'*IC*': incomplete Cholesky factorization

'*DIAG*': diagonal Jacobi preconditioner

If $precon$ is not specified, no preconditioning will be applied.

A starting trial solution can be specified with the $start$ parameter, otherwise the ITMINRES call will generate an all-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.

Your IML program should always check the returned $error$ and $iter$ parameters to verify that the desired relative error tolerance was reached. If not, your program might continue the solution process with another ITMINRES call, with $start$ set to the latest result. You might also try a different $precon$ option to enhance convergence.

The following example illustrates the use of ITMINRES to solve a system that is not positive definite:

$$\begin{bmatrix} 3 & 1.1 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & -10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

```
/* 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};
maxiter = 10;
hist = j(maxiter,1,.);
start = {1,1,1,1};
tol = 1e-10;
call itminres(x, error, iter, A, b, 'ic', tol, maxiter, start, hist);
print x;
print iter error;
print hist;
```

The results are

```
            X
      0.7647059
      -1.176471
      -0.217647
      1.5882353


      ITER      ERROR

        3 5.987E-17


          HIST

      0.4598724
      0.0856374
      5.987E-17
              .
              .
              .
              .
              .
              .
              .
```

J Function

creates a matrix of identical values

J(*nrow<, ncol<, value>>***)**

The inputs to the J function are as follows:

nrow   is a numeric matrix or literal giving the number of rows.

ncol   is a numeric matrix or literal giving the number of columns.

> *value*　　　　　is a numeric or character matrix or literal for filling the rows and
> 　　　　　　　columns of the matrix.

The J function creates a matrix with *nrow* rows and *ncol* columns with all elements
equal to *value*. If *ncol* is not specified, it defaults to *nrow*. If *value* is not specified,
it defaults to 1. The REPEAT and SHAPE functions can also perform this operation,
and they are more general.

Examples of the J function are as follows.

```
b=j(3);

              B                3 rows       3 cols      (numeric)

                                1            1           1
                                1            1           1
                                1            1           1


r=j(5,2,'xyz');

              R             5 rows        2 cols     (character, size 3)

                                xyz xyz
                                xyz xyz
                                xyz xyz
                                xyz xyz
                                xyz xyz
```

JROOT Function

> **computes the first nonzero roots of a Bessel function of the first kind and the
> derivative of the Bessel function at each root**
>
> **JROOT(ν, n)**

The JROOT function returns an $n \times 2$ matrix with the calculated roots in the first
column and the derivatives in the second column.

The inputs to the JROOT function are as follows:

ν　　　　　is a scalar denoting the order of the Bessel function, with $\nu > -1$.

n　　　　　is a positive integer denoting the number of roots.

The JROOT function returns a matrix in which the first column contains the first n
roots of the Bessel function; these roots are the solutions to the equation

$$J_\nu(x_i) = 0, \quad i = 1, \ldots, n$$

The second column of this matrix contains the derivatives $J'_\nu(x_i)$ of the Bessel function at each of the roots x_i. The expression $J_\nu(x)$ is a solution to the differential equation

$$x^2 \frac{d^2 J_\nu}{dx^2} + x \frac{dJ_\nu}{dx} + (x^2 - \nu^2)J_\nu = 0$$

One of the expressions for such a function is given by the series

$$J_\nu(x) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu + k + 1)}$$

where $\Gamma(\cdot)$ is the gamma function. Refer to Abramowitz and Stegun (1972) for more details concerning the Bessel and gamma functions. The algorithm is a Newton method coupled with a reasonable initial guess. For large values of n or ν, the algorithm could fail due to machine limitations. In this case, JROOT returns a matrix with zero rows and zero columns. The values that cause the algorithm to fail are machine dependent.

The following code provides an example:

```
x = jroot(1,4);
print x;
```

To obtain only the roots, you can use the following statement, which extracts the first column of the returned matrix:

```
x = jroot(1,4)[,1];
```

KALCVF Call

CALL KALCVF(*pred, vpred, filt, vfilt, data, lead,* a, f, b, h,

var $<$, *z0, vz0* $>$**);**

The KALCVF call computes the one-step prediction $z_{t+1|t}$ and the filtered estimate $z_{t|t}$, as well as their covariance matrices. The call uses forward recursions, and you can also use it to obtain k-step estimates.

The inputs to the KALCVF subroutine are as follows:

data	is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.
lead	is the number of steps to forecast after the end of the data.
a	is an $N_z \times 1$ vector for a time-invariant input vector in the transition equation, or a $(T + lead)N_z \times 1$ vector containing input vectors in the transition equation.
f	is an $N_z \times N_z$ matrix for a time-invariant transition matrix in the transition equation, or a $(T + lead)N_z \times N_z$ matrix containing transition matrices in the transition equation.

b	is an $N_y \times 1$ vector for a time-invariant input vector in the measurement equation, or a $(T + lead)N_y \times 1$ vector containing input vectors in the measurement equation.
h	is an $N_y \times N_z$ matrix for a time-invariant measurement matrix in the measurement equation, or a $(T + lead)N_y \times N_z$ matrix containing measurement matrices in the measurement equation.
var	is an $(N_y + N_z) \times (N_y + N_z)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + lead)(N_y + N_z) \times (N_y + N_z)$ matrix containing variance matrices for the error in the transition equation and the error in the measurement equation, that is, $(\eta'_t, \epsilon'_t)'$.
$z0$	is an optional $1 \times N_z$ initial state vector $\mathbf{z}'_{1|0}$.
$vz0$	is an optional $N_z \times N_z$ covariance matrix of an initial state vector $\mathbf{P}_{1|0}$.

The KALCVF call returns the following values:

pred	is a $(T + lead) \times N_z$ matrix containing one-step predicted state vectors $(\mathbf{z}_{1|0}, \cdots, \mathbf{z}_{T+1|T}, \mathbf{z}_{T+2|T}, \cdots, \mathbf{z}_{T+lead|T})'$.
vpred	is a $(T+lead)N_z \times N_z$ matrix containing mean square errors of predicted state vectors $(\mathbf{P}_{1|0}, \cdots, \mathbf{P}_{T+1|T}, \mathbf{P}_{T+2|T}, \cdots, \mathbf{P}_{T+lead|T})'$.
filt	is a $T \times N_z$ matrix containing filtered state vectors $(\mathbf{z}_{1|1}, \cdots, \mathbf{z}_{T|T})'$.
vfilt	is a $TN_z \times N_z$ matrix containing mean square errors of filtered state vectors $(\mathbf{P}_{1|1}, \cdots, \mathbf{P}_{T|T})'$.

The KALCVF call computes the conditional expectation of the state vector $\mathbf{z}_t$ given the observations, assuming that the mean and the variance of the initial state vector are known. The filtered value is the conditional expectation of the state vector $\mathbf{z}_t$ given the observations up to time t. For k-step forecasting where $k > 0$, the conditional expectation at time $t + k$ is computed given observations up to t. For notation, $\mathbf{V}_t$ and $\mathbf{R}_t$ are variances of η_t and ϵ_t, respectively, and $\mathbf{G}_t$ is a covariance of η_t and ϵ_t. $\mathbf{A}^-$ stands for the generalized inverse of $\mathbf{A}$. The filtered value and its covariance matrix are denoted $\mathbf{z}_{t|t}$ and $\mathbf{P}_{t|t}$, respectively. For $k > 0$, $\mathbf{z}_{t+k|t}$ and $\mathbf{P}_{t+k|t}$ stand for the k-step forecast of $\mathbf{z}_{t+k}$ and its mean square error. The Kalman filtering algorithm for one-step prediction and filtering is given as follows:

$$
\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t \mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{R}_t \\
\mathbf{z}_{t|t} &= \mathbf{z}_{t|t-1} + \mathbf{P}_{t|t-1} \mathbf{H}'_t \mathbf{D}^-_t \hat{\epsilon}_t \\
\mathbf{P}_{t|t} &= \mathbf{P}_{t|t-1} - \mathbf{P}_{t|t-1} \mathbf{H}'_t \mathbf{D}^-_t \mathbf{H}_t \mathbf{P}_{t|t-1} \\
\mathbf{K}_t &= (\mathbf{F}_t \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{G}_t) \mathbf{D}^-_t
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{z}_{t+1|t} &= \mathbf{a}_t + \mathbf{F}_t \mathbf{z}_{t|t-1} + \mathbf{K}_t \hat{\epsilon}_t \\
\mathbf{P}_{t+1|t} &= \mathbf{F}_t \mathbf{P}_{t|t-1} \mathbf{F}'_t + \mathbf{V}_t - \mathbf{K}_t \mathbf{D}_t \mathbf{K}'_t
\end{aligned}
$$

And for k-step forecasting for $k > 1$,

$$
\begin{aligned}
\mathbf{z}_{t+k|t} &= \mathbf{a}_{t+k-1} + \mathbf{F}_{t+k-1} \mathbf{z}_{t+k-1|t} \\
\mathbf{P}_{t+k|t} &= \mathbf{F}_{t+k-1} \mathbf{P}_{t+k-1|t} \mathbf{F}'_{t+k-1} + \mathbf{V}_{t+k-1}
\end{aligned}
$$

When you use the alternative transition equation

$$
\mathbf{z}_t = \mathbf{a}_t + \mathbf{F}_t \mathbf{z}_{t-1} + \eta_t
$$

the forward recursion algorithm is written

$$
\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t \mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{H}_t \mathbf{G}_t + \mathbf{G}'_t \mathbf{H}'_t + \mathbf{R}_t \\
\mathbf{z}_{t|t} &= \mathbf{z}_{t|t-1} + (\mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{G}_t) \mathbf{D}_t^- \hat{\epsilon}_t \\
\mathbf{P}_{t|t} &= \mathbf{P}_{t|t-1} - (\mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{G}_t) \mathbf{D}_t^- (\mathbf{H}_t \mathbf{P}_{t|t-1} + \mathbf{G}'_t) \\
\mathbf{K}_t &= (\mathbf{F}_{t+1} \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{G}_t) \mathbf{D}_t^- \\
\mathbf{z}_{t+1|t} &= \mathbf{a}_{t+1} + \mathbf{F}_{t+1} \mathbf{z}_{t|t-1} + \mathbf{K}_t \hat{\epsilon}_t \\
\mathbf{P}_{t+1|t} &= \mathbf{F}_{t+1} \mathbf{P}_{t|t-1} \mathbf{F}'_{t+1} + \mathbf{V}_{t+1} - \mathbf{K}_t \mathbf{D}_t \mathbf{K}'_t
\end{aligned}
$$

And for k-step forecasting $(k > 1)$,

$$
\begin{aligned}
\mathbf{z}_{t+k|t} &= \mathbf{a}_{t+k} + \mathbf{F}_{t+k} \mathbf{z}_{t+k-1|t} \\
\mathbf{P}_{t+k|t} &= \mathbf{F}_{t+k} \mathbf{P}_{t+k-1|t} \mathbf{F}'_{t+k} + \mathbf{V}_{t+k}
\end{aligned}
$$

You can use the KALCVF call when you specify the alternative transition equation and $\mathbf{G}_t = \mathbf{0}$.

The initial state vector and its covariance matrix of the time invariant Kalman filters are computed under the stationarity condition

$$
\begin{aligned}
\mathbf{z}_{1|0} &= (\mathbf{I} - \mathbf{F})^- \mathbf{a} \\
\mathbf{P}_{1|0} &= (\mathbf{I} - \mathbf{F} \otimes \mathbf{F})^- \mathrm{vec}(\mathbf{V})
\end{aligned}
$$

where $\mathbf{F}$ and $\mathbf{V}$ are the time invariant transition matrix and the covariance matrix of transition equation noise, and $\mathrm{vec}(\mathbf{V})$ is an $N_z^2 \times 1$ column vector that is constructed by the stacking N_z columns of matrix $\mathbf{V}$. Note that all eigenvalues of the matrix $\mathbf{F}$ are inside the unit circle when the SSM is stationary. When the preceding formula

cannot be applied, the initial state vector estimate $z_{1|0}$ is set to a_1 and its covariance matrix $P_{1|0}$ is given by $10^6 I$. Optionally, you can specify initial values.

The KALCVF call allows missing values in observations. If there is a missing observation, the filtered state vector for the missing observation is given by the one-step forecast.

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];
```

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 call uses backward recursions to compute the smoothed estimate $\mathbf{z}_{t|T}$ and its covariance matrix, $\mathbf{P}_{t|T}$, where T is the number of observations in the complete data set.

The inputs to the KALCVS subroutine are as follows.

data	is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.
a	is an $N_z \times 1$ vector for a time-invariant input vector in the transition equation, or a $TN_z \times 1$ vector containing input vectors in the transition equation.
f	is an $N_z \times N_z$ matrix for a time-invariant transition matrix in the transition equation, or a $TN_z \times N_z$ matrix containing T transition matrices.
b	is an $N_y \times 1$ vector for a time-invariant input vector in the measurement equation, or a $TN_y \times 1$ vector containing input vectors in the measurement equation.
h	is an $N_y \times N_z$ matrix for a time-invariant measurement matrix in the measurement equation, or a $TN_y \times N_z$ matrix containing T time variant $\mathbf{H}_t$ matrices in the measurement equation.
var	is an $(N_y + N_z) \times (N_y + N_z)$ covariance matrix for the errors in the transition and the measurement equations, or a $T(N_y + N_z) \times (N_y + N_z)$ matrix containing covariance matrices in the transition equation and measurement equation noises, that is, $(\eta_t', \epsilon_t')'$.
pred	is a $T \times N_z$ matrix containing one-step forecasts $(\mathbf{z}_{1|0}, \cdots, \mathbf{z}_{T|T-1})'$.
vpred	is a $TN_z \times N_z$ matrix containing mean square error matrices of predicted state vectors $(\mathbf{P}_{1|0}, \cdots, \mathbf{P}_{T|T-1})'$.
un	is an optional $1 \times N_z$ vector containing $\mathbf{u}_T$. The returned value is $\mathbf{u}_0$.
vun	is an optional $N_z \times N_z$ matrix containing $\mathbf{U}_T$. The returned value is $\mathbf{U}_0$.

The KALCVS call returns the following values:

sm	is a $T \times N_z$ matrix containing smoothed state vectors $(\mathbf{z}_{1|T}, \cdots, \mathbf{z}_{T|T})'$.
vsm	is a $TN_z \times N_z$ matrix containing covariance matrices of smoothed state vectors $(\mathbf{P}_{1|T}, \cdots, \mathbf{P}_{T|T})'$.

When the Kalman filtering is performed in the KALCVF call, the KALCVS call computes smoothed state vectors and their covariance matrices. The fixed-interval smoothing state vector at time t is obtained by the conditional expectation given all observations.

The smoothing algorithm uses one-step forecasts and their covariance matrices, which are given in the KALCVF call. For notation, $\mathbf{z}_{t|T}$ is the smoothed value of the state vector $\mathbf{z}_t$, and the mean square error matrix is denoted $\mathbf{P}_{t|T}$. For smoothing,

$$
\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t \mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{R}_t \\
\mathbf{K}_t &= (\mathbf{F}_t \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{G}_t) \mathbf{D}_t^- \\
\mathbf{L}_t &= \mathbf{F}_t - \mathbf{K}_t \mathbf{H}_t \\
\mathbf{u}_{t-1} &= \mathbf{H}'_t \mathbf{D}_t^- \hat{\epsilon}_t + \mathbf{L}'_t \mathbf{u}_t \\
\mathbf{U}_{t-1} &= \mathbf{H}'_t \mathbf{D}_t^- \mathbf{H}_t + \mathbf{L}'_t \mathbf{U}_t \mathbf{L}_t \\
\mathbf{z}_{t|T} &= \mathbf{z}_{t|t-1} + \mathbf{P}_{t|t-1} \mathbf{u}_{t-1} \\
\mathbf{P}_{t|T} &= \mathbf{P}_{t|t-1} - \mathbf{P}_{t|t-1} \mathbf{U}_{t-1} \mathbf{P}_{t|t-1}
\end{aligned}
$$

where $t = T, T-1, \ldots, 1$. The initial values are $\mathbf{u}_T = \mathbf{0}$ and $\mathbf{U}_T = \mathbf{0}$.

When the SSM is specified using the alternative transition equation

$$
\mathbf{z}_t = \mathbf{a}_t + \mathbf{F}_t \mathbf{z}_{t-1} + \eta_t
$$

the fixed-interval smoothing is performed using the following backward recursions:

$$
\begin{aligned}
\hat{\epsilon}_t &= \mathbf{y}_t - \mathbf{b}_t - \mathbf{H}_t \mathbf{z}_{t|t-1} \\
\mathbf{D}_t &= \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}'_t + \mathbf{R}_t \\
\mathbf{K}_t &= \mathbf{F}_{t+1} \mathbf{P}_{t|t-1} \mathbf{H}'_t \mathbf{D}_t^- \\
\mathbf{L}_t &= \mathbf{F}_{t+1} - \mathbf{K}_t \mathbf{H}_t \\
\mathbf{u}_{t-1} &= \mathbf{H}'_t \mathbf{D}_t^- \hat{\epsilon}_t + \mathbf{L}'_t \mathbf{u}_t \\
\mathbf{U}_{t-1} &= \mathbf{H}'_t \mathbf{D}_t^- \mathbf{H}_t + \mathbf{L}'_t \mathbf{U}_t \mathbf{L}_t \\
\mathbf{z}_{t|T} &= \mathbf{z}_{t|t-1} + \mathbf{P}_{t|t-1} \mathbf{u}_{t-1} \\
\mathbf{P}_{t|T} &= \mathbf{P}_{t|t-1} - \mathbf{P}_{t|t-1} \mathbf{U}_{t-1} \mathbf{P}_{t|t-1}
\end{aligned}
$$

where it is assumed that $\mathbf{G}_t = \mathbf{0}$.

You can use the KALCVS call regardless of the specification of the transition equation when $\mathbf{G}_t = \mathbf{0}$. Harvey (1989) gives the following fixed-interval smoothing formula, which produces the same smoothed value:

$$
\begin{aligned}
\mathbf{z}_{t|T} &= \mathbf{z}_{t|t} + \mathbf{P}_t^*(\mathbf{z}_{t+1|T} - \mathbf{z}_{t+1|t}) \\
\mathbf{P}_{t|T} &= \mathbf{P}_{t|t} + \mathbf{P}_t^*(\mathbf{P}_{t+1|T} - \mathbf{P}_{t+1|t}) \mathbf{P}_t^{*\prime}
\end{aligned}
$$

where

$$\mathbf{P}_t^* = \mathbf{P}_{t|t}\mathbf{F}_t'\mathbf{P}_{t+1|t}^-$$

under the shifted transition equation, but

$$\mathbf{P}_t^* = \mathbf{P}_{t|t}\mathbf{F}_{t+1}'\mathbf{P}_{t+1|t}^-$$

under the alternative transition equation.

The KALCVS call is accompanied by the KALCVF call, as shown in the following code. Note that you do not need to specify UN and VUN.

```
call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var);
call kalcvs(sm,vsm,y,a,f,b,h,var,pred,vpred);
```

You can also compute the smoothed estimate and its covariance matrix on an observation-by-observation basis. When the SSM is time invariant, the following example performs smoothing. In this situation, you should initialize UN and VUN as matrices of value 0.

```
call kalcvf(pred,vpred,filt,vfilt,y,0,a,f,b,h,var);
n = nrow(y);
nz = ncol(f);
un = j(1,nz,0);
vun = j(nz,nz,0);
do i = 1 to n;
  y_i = y[n-i+1,];
  pred_i  = pred[n-i+1,];
  vpred_i = vpred[(n-i)*nz+1:(n-i+1)*nz,];
  call kalcvs(sm_i,vsm_i,y_i,a,f,b,h,var,pred_i,vpred_i,un,vun);
  sm  = sm_i // sm;
  vsm = vsm_i // vsm;
end;
```

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 call computes the one-step forecast of state vectors in an SSM 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.

The inputs to the KALDFF subroutine are as follows:

data	is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.
lead	is the number of steps to forecast after the end of the data set.
int	is an $(N_y + N_z) \times N_\beta$ matrix for a time-invariant fixed matrix, or a $(T+lead)(N_y+N_z) \times N_\beta$ matrix containing fixed matrices for the time-variant model in the transition equation and the measurement equation, that is, $(\mathbf{W}'_t, \mathbf{X}'_t)'$.
coef	is an $(N_y + N_z) \times N_z$ matrix for a time-invariant coefficient, or a $(T + lead)(N_y + N_z) \times N_z$ matrix containing coefficients at each time in the transition equation and the measurement equation, that is, $(\mathbf{F}'_t, \mathbf{H}'_t)'$.
var	is an $(N_y + N_z) \times (N_y + N_z)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + lead)(N_y + N_z) \times (N_y + N_z)$ matrix containing covariance matrices for the error in the transition equation and the error in the measurement equation, that is, $(\eta'_t, \epsilon'_t)'$.
intd	is an $(N_z + N_\beta) \times 1$ vector containing the intercept term in the equation for the initial state vector $\mathbf{z}_0$ and the mean effect β, that is, $(\mathbf{a}', \mathbf{b}')'$.
coefd	is an $(N_z + N_\beta) \times N_\delta$ matrix containing coefficients for the initial state δ in the equation for the initial state vector $\mathbf{z}_0$ and the mean effect β, that is, $(\mathbf{A}', \mathbf{B}')'$.
n0	is an optional scalar including an initial denominator. If $n0 > 0$, the denominator for $\hat{\sigma}^2_t$ is $n0$ plus the number n_t of elements of $(\mathbf{y}_1, \cdots, \mathbf{y}_t)'$. If $n0 \leq 0$ or $n0$ is not specified, the denominator for $\hat{\sigma}^2_t$ is n_t. With $n0 \geq 0$, the initial values, $\mathbf{A}_1, \mathbf{M}_1$, and $\mathbf{Q}_1$, are assumed to be known and, hence, at, mt, and qt are used for input containing the initial values. If the value of $n0$ is negative or $n0$ is not specified, the initial values for at, mt, and qt are computed. The value of $n0$ is updated as $\max(n0, 0) + n_t$ after the KALDFF call.
at	is an optional $kN_z \times (N_\delta + 1)$ matrix. If $n0 \geq 0$, at contains $(\mathbf{A}'_1, \cdots, \mathbf{A}'_k)'$. However, only the first matrix $\mathbf{A}_1$ is used as input. When you specify the KALDFF call, at returns $(\mathbf{A}'_{T-k+lead+1}, \cdots, \mathbf{A}'_{T+lead})'$. If $n0$ is negative or the matrix $\mathbf{A}_1$ contains missing values, $\mathbf{A}_1$ is automatically computed.
mt	is an optional $kN_z \times N_z$ matrix. If $n0 \geq 0$, mt contains $(\mathbf{M}_1, \cdots, \mathbf{M}_k)'$. However, only the first matrix $\mathbf{M}_1$ is used as input. If $n0$ is negative or the matrix $\mathbf{M}_1$ contains missing values, mt is used for output, and it contains $(\mathbf{M}_{T-k+lead+1}, \cdots, \mathbf{M}_{T+lead})'$. Note that the matrix $\mathbf{M}_1$ can be used as an input matrix if either of the off-diagonal elements is not missing. The missing element $\mathbf{M}_1(i, j)$ is replaced by the nonmissing element $\mathbf{M}_1(j, i)$.

qt	is an optional $k(N_\delta + 1) \times (N_\delta + 1)$ matrix. If $n0 \geq 0$, qt contains $(\mathbf{Q}_1, \cdots, \mathbf{Q}_k)'$. However, only the first matrix $\mathbf{Q}_1$ is used as input. If $n0$ is negative or the matrix $\mathbf{Q}_1$ contains missing values, qt is used for output and contains $(\mathbf{Q}_{T-k+lead+1}, \cdots, \mathbf{Q}_{T+lead})'$. The matrix $\mathbf{Q}_1$ can also be used as an input matrix if either of the off-diagonal elements is not missing since the missing element $\mathbf{Q}_1(i, j)$ is replaced by the nonmissing element $\mathbf{Q}_1(j, i)$.

The KALDFF call returns the following values:

pred	is a $(T + lead) \times N_z$ matrix containing estimated predicted state vectors $(\hat{\mathbf{z}}_{1|0}, \cdots, \hat{\mathbf{z}}_{T+1|T}, \hat{\mathbf{z}}_{T+2|T}, \cdots, \hat{\mathbf{z}}_{T+lead|T})'$.
vpred	is a $(T + lead)N_z \times N_z$ matrix containing estimated mean square errors of predicted state vectors $(\mathbf{P}_{1|0}, \cdots, \mathbf{P}_{T+1|T}, \mathbf{P}_{T+2|T}, \cdots, \mathbf{P}_{T+lead|T})'$.
initial	is an $N_d \times (N_d + 1)$ matrix containing an estimate and its variance for initial state δ, that is, $(\hat{\delta}_T, \hat{\Sigma}_{\delta,T})$.
s2	is a scalar containing the estimated variance $\hat{\sigma}_T^2$.

The KALDFF call computes the one-step forecast of state vectors in an SSM using the diffuse Kalman filter. The SSM for the diffuse Kalman filter is written

$$
\begin{aligned}
\mathbf{y}_t &= \mathbf{X}_t\beta + \mathbf{H}_t\mathbf{z}_t + \epsilon_t \\
\mathbf{z}_{t+1} &= \mathbf{W}_t\beta + \mathbf{F}_t\mathbf{z}_t + \eta_t \\
\mathbf{z}_0 &= \mathbf{a} + \mathbf{A}\delta \\
\beta &= \mathbf{b} + \mathbf{B}\delta
\end{aligned}
$$

where $\mathbf{z}_t$ is an $N_z \times 1$ state vector, $\mathbf{y}_t$ is an $N_y \times 1$ observed vector, and

$$
\begin{bmatrix} \eta_t \\ \epsilon_t \end{bmatrix} \sim N\left(\mathbf{0}, \sigma^2 \begin{bmatrix} \mathbf{V}_t & \mathbf{G}_t \\ \mathbf{G}_t' & \mathbf{R}_t \end{bmatrix}\right)
$$
$$
\delta \sim N(\mu, \sigma^2\Sigma)
$$

It is assumed that the noise vector $(\eta_t', \epsilon_t')'$ is independent and δ is independent of the vector $(\eta_t', \epsilon_t')'$. The matrices, $\mathbf{W}_t$, $\mathbf{F}_t$, $\mathbf{X}_t$, $\mathbf{H}_t$, $\mathbf{a}$, $\mathbf{A}$, $\mathbf{b}$, $\mathbf{B}$, $\mathbf{V}_t$, $\mathbf{G}_t$, and $\mathbf{R}_t$, are assumed to be known. The KALDFF call estimates the conditional expectation of the state vector $\mathbf{z}_t$ given the observations. The KALDFF subroutine also produces the estimates of the initial random vector δ and its covariance matrix. For k-step forecasting where $k > 0$, the estimated conditional expectation at time $t + k$ is computed with observations given up to time t. The estimated k-step forecast and its estimated MSE are denoted $\mathbf{z}_{t+k|t}$ and $\mathbf{P}_{t+k|t}$ (for $k > 0$). $\mathbf{A}_{t+k(\delta)}$ and $\mathbf{E}_{t(\delta)}$ are last-column-deleted

submatrices of $\mathbf{A}_{t+k}$ and $\mathbf{E}_t$, respectively. The algorithm for one-step prediction is given as follows:

$$
\begin{aligned}
\mathbf{E}_t &= (\mathbf{X}_t\mathbf{B},\ \mathbf{y}_t - \mathbf{X}_t\mathbf{b}) - \mathbf{H}_t\mathbf{A}_t \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{R}_t \\
\mathbf{Q}_{t+1} &= \mathbf{Q}_t + \mathbf{E}_t'\mathbf{D}_t^-\mathbf{E}_t \\
&= \begin{bmatrix} \mathbf{S}_t & \mathbf{s}_t \\ \mathbf{s}_t' & q_t \end{bmatrix} \\
\hat{\sigma}_t^2 &= (q_t - \mathbf{s}_t'\mathbf{S}_t^-\mathbf{s}_t)/n_t \\
\hat{\delta}_t &= \mathbf{S}_t^-\mathbf{s}_t \\
\hat{\Sigma}_{\delta,t} &= \hat{\sigma}_t^2\mathbf{S}_t^- \\
\mathbf{K}_t &= (\mathbf{F}_t\mathbf{M}_t\mathbf{H}_t' + \mathbf{G}_t)\mathbf{D}_t^-
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{A}_{t+1} &= \mathbf{W}_t(-\mathbf{B}, \mathbf{b}) + \mathbf{F}_t\mathbf{A}_t + \mathbf{K}_t\mathbf{E}_t \\
\mathbf{M}_{t+1} &= (\mathbf{F}_t - \mathbf{K}_t\mathbf{H}_t)\mathbf{M}_t\mathbf{F}_t' + \mathbf{V}_t - \mathbf{K}_t\mathbf{G}_t' \\
\mathbf{z}_{t+1|t} &= \mathbf{A}_{t+1}(-\hat{\delta}_t', 1)' \\
\mathbf{P}_{t+1|t} &= \hat{\sigma}_t^2\mathbf{M}_{t+1} + \mathbf{A}_{t+1(\delta)}\hat{\Sigma}_{\delta,t}\mathbf{A}_{t+1(\delta)}'
\end{aligned}
$$

where n_t is the number of elements of $(\mathbf{y}_1, \cdots, \mathbf{y}_t)'$ plus $\max(n0, 0)$. Unless initial values are given and $n0 \geq 0$, initial values are set as follows:

$$
\begin{aligned}
\mathbf{A}_1 &= \mathbf{W}_1(-\mathbf{B}, \mathbf{b}) + \mathbf{F}_1(-\mathbf{A}, \mathbf{a}) \\
\mathbf{M}_1 &= \mathbf{V}_1 \\
\mathbf{Q}_1 &= \mathbf{0}
\end{aligned}
$$

For k-step forecasting where $k > 1$,

$$
\begin{aligned}
\mathbf{A}_{t+k} &= \mathbf{W}_{t+k-1}(-\mathbf{B}, \mathbf{b}) + \mathbf{F}_{t+k-1}\mathbf{A}_{t+k-1} \\
\mathbf{M}_{t+k} &= \mathbf{F}_{t+k-1}\mathbf{M}_{t+k-1}\mathbf{F}_{t+k-1}' + \mathbf{V}_{t+k-1} \\
\mathbf{D}_{t+k} &= \mathbf{H}_{t+k}\mathbf{M}_{t+k}\mathbf{H}_{t+k}' + \mathbf{R}_{t+k} \\
\mathbf{z}_{t+k|t} &= \mathbf{A}_{t+k}(-\hat{\delta}_t', 1)' \\
\mathbf{P}_{t+k|t} &= \hat{\sigma}_t^2\mathbf{M}_{t+k} + \mathbf{A}_{t+k(\delta)}\hat{\Sigma}_{\delta,t}\mathbf{A}_{t+k(\delta)}'
\end{aligned}
$$

Note that if there is a missing observation, the KALDFF call computes the one-step forecast for the observation following the missing observation as the two-step forecast from the previous observation.

An example of using 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>***);**

KALDFS computes the smoothed state vector and its mean square error matrix from the one-step forecast and mean square error matrix computed by KALDFF.

The inputs to the KALDFS subroutine are as follows:

data	is a $T \times N_y$ matrix containing data $(\mathbf{y}_1, \cdots, \mathbf{y}_T)'$.
int	is an $(N_y + N_z) \times N_\beta$ vector for a time-invariant intercept, or a $(T + lead)(N_y + N_z) \times N_\beta$ vector containing fixed matrices for the time-variant model in the transition equation and the measurement equation, that is, $(\mathbf{W}'_t, \mathbf{X}'_t)'$.
coef	is an $(N_y + N_z) \times N_z$ matrix for a time-invariant coefficient, or a $(T + lead)(N_y + N_z) \times N_z$ matrix containing coefficients at each time in the transition equation and the measurement equation, that is, $(\mathbf{F}'_t, \mathbf{H}'_t)'$.
var	is an $(N_y + N_z) \times (N_y + N_z)$ matrix for a time-invariant variance matrix for transition equation noise and the measurement equation noise, or a $(T + lead)(N_y + N_z) \times (N_y + N_z)$ matrix containing covariance matrices for the transition equation and measurement equation errors, that is, $(\eta'_t, \epsilon'_t)'$.
bvec	is an $N_\beta \times 1$ constant vector for the intercept for the mean effect β.
bmat	is an $N_\beta \times N_\delta$ matrix for the coefficient for the mean effect β.
initial	is an $N_\delta \times (N_\delta + 1)$ matrix containing an initial random vector estimate and its covariance matrix, that is, $(\hat{\delta}_T, \hat{\Sigma}_{\delta,T})$.
at	is a $TN_z \times (N_\delta + 1)$ matrix containing $(\mathbf{A}'_1, \cdots, \mathbf{A}'_T)'$.
mt	is a $(TN_z) \times N_z$ matrix containing $(\mathbf{M}_1, \cdots, \mathbf{M}_T)'$.
s2	is the estimated variance in the end of the data set, $\hat{\sigma}^2_T$.
un	is an optional $N_z \times (N_\delta + 1)$ matrix containing $\mathbf{u}_T$. The returned value is $\mathbf{u}_0$.
vun	is an optional $N_z \times N_z$ matrix containing $\mathbf{U}_T$. The returned value is $\mathbf{U}_0$.

The KALDFS call returns the following values:

sm	is a $T \times N_z$ matrix containing smoothed state vectors $(\mathbf{z}_{1|T}, \cdots, \mathbf{z}_{T|T})'$.
vsm	is a $TN_z \times N_z$ matrix containing mean square error matrices of smoothed state vectors $(\mathbf{P}_{1|T}, \cdots, \mathbf{P}_{T|T})'$.

Given the one-step forecast and mean square error matrix in the KALDFF call, the KALDFS call computes a smoothed state vector and its mean square error matrix. Then the KALDFS subroutine produces an estimate of the smoothed state vector at time t, that is, the conditional expectation of the state vector $\mathbf{z}_t$ given all observations. Using the notations and results from the KALDFF section, the backward recursion algorithm for smoothing is denoted for $t = T, T - 1, \ldots, 1$,

$$
\begin{aligned}
\mathbf{E}_t &= (\mathbf{X}_t\mathbf{B}, \ \mathbf{y}_t - \mathbf{X}_t\mathbf{b}) - \mathbf{H}_t\mathbf{A}_t \\
\mathbf{D}_t &= \mathbf{H}_t\mathbf{M}_t\mathbf{H}'_t + \mathbf{R}_t \\
\mathbf{L}_t &= \mathbf{F}_t - (\mathbf{F}_t\mathbf{M}_t\mathbf{H}'_t + \mathbf{G}_t)\mathbf{D}_t^-\mathbf{H}_t \\
\mathbf{u}_{t-1} &= \mathbf{H}'_t\mathbf{D}_t^-\mathbf{E}_t + \mathbf{L}'_t\mathbf{u}_t \\
\mathbf{U}_{t-1} &= \mathbf{H}'_t\mathbf{D}_t^-\mathbf{H}_t + \mathbf{L}'_t\mathbf{U}_t\mathbf{L}_t \\
\mathbf{z}_{t|T} &= (\mathbf{A}_t + \mathbf{M}_t\mathbf{u}_{t-1})(-\hat{\delta}'_T, 1)' \\
\mathbf{C}_t &= \mathbf{A}_t + \mathbf{M}_t\mathbf{u}_{t-1} \\
\mathbf{P}_{t|T} &= \hat{\sigma}_T^2(\mathbf{M}_t - \mathbf{M}_t\mathbf{R}_{t-1}\mathbf{M}_t) + \mathbf{C}_{t(\delta)}\hat{\Sigma}_{\delta,T}\mathbf{C}'_{t(\delta)}
\end{aligned}
$$

where the initial values are $\mathbf{u}_T = \mathbf{0}$ and $\mathbf{U}_T = \mathbf{0}$, and $\mathbf{C}_{t(\delta)}$ is the last-column-deleted submatrix of $\mathbf{C}_t$. Refer to De Jong (1991) for details on smoothing in the diffuse Kalman filter.

The KALDFS call is accompanied by the KALDFF call as shown in the following code:

```
ny = ncol(y);
nz = ncol(coef);
nb = ncol(int);
nd = ncol(coefd);
at = j(nz,nd+1,.);
mt = j(nz,nz,.);
qt = j(nd+1,nd+1,.);
n0 = -1;
call kaldff(pred,vpred,initial,s2,y,0,int,coef,var,intd,coefd,
            n0,at,mt,qt);
bvec = intd[nz+1:nz+nb,];
bmat = coefd[nz+1:nz+nb,];
call kaldfs(sm,vsm,x,int,coef,var,bvec,bmat,initial,at,mt,s2);
```

You can also compute the smoothed estimate and its covariance matrix observation by observation. When the SSM is time invariant, the following code performs smoothing. You should initialize UN and VUN as matrices of value **0**.

```
n  = nrow(y);
ny = ncol(y);
nz = ncol(coef);
nb = ncol(int);
nd = ncol(coefd);
at = j(nz,nd+1,.);
mt = j(nz,nz,.);
qt = j(nd+1,nd+1,.);
n0 = -1;
call kaldff(pred,vpred,initial,s2,y,0,int,coef,var,intd,coefd,
            n0,at,mt,qt);
bvec = intd[nz+1:nz+nb,];
bmat = coefd[nz+1:nz+nb,];
un  = j(nz,nd+1,0);
vun = j(nz,nz,0);
do i = 1 to n;
   call kaldfs(sm_i,vsm_i,y[n-i+1],int,coef,var,bvec,bmat,
               initial,at,mt,s2,un,vun);
   sm  = sm_i // sm;
   vsm = vsm_i // vsm;
end;
```

LAV Call

performs linear least absolute value regression by solving the L_1 norm minimization problem

CALL LAV(*rc, xr, a, b <, <x0><, opt>>* **);**

The LAV subroutine returns the following values:

rc is a scalar return code indicating the reason for optimization termination.

rc	**Termination**
0	Successful
1	Successful, but approximate covariance matrix and standard errors cannot be computed
−1 or −3	Unsuccessful: error in the input arguments
−2	Unsuccessful: matrix **A** is rank deficient $(\mathrm{rank}(\mathbf{A}) < n)$
−4	Unsuccessful: maximum iteration limit exceeded
−5	Unsuccessful: no solution found for ill-conditioned problem

xr specifies a vector or matrix with n columns. If the optimization process is not successfully completed, *xr* is a row vector with n missing values. If termination is successful and the *opt*[3] option is not set, *xr* is the vector with the optimal estimate, x^*. If termination is successful and

the *opt*[3] option is specified, *xr* is an $(n+2) \times n$ matrix that contains the optimal estimate, x^*, in the first row, the asymptotic standard errors in the second row, and the $n \times n$ covariance matrix of parameter estimates in the remaining rows.

The inputs to the LAV subroutine are as follows:

a

specifies an $m \times n$ matrix $\mathbf{A}$ with $m \geq n$ and full column rank, $\mathrm{rank}(\mathbf{A}) = n$. If you want to include an intercept in the model, you must include a column of ones in the matrix $\mathbf{A}$.

b

specifies the $m \times 1$ vector $\mathbf{b}$.

x0

specifies an optional $n \times 1$ vector that specifies the starting point of the optimization.

opt

is an optional vector used to specify options.

opt[1] specifies the maximum number *maxi* of outer iterations (this corresponds to the number of changes of the Huber parameter γ). The default is $maxi = \min(100, 10n)$. (The number of inner iterations is restricted by an internal threshold. If the number of inner iterations exceeds this threshold, a new outer iteration is started with an increased value of γ.)

opt[2] specifies the amount of printed output. Higher values request additional output and include the output of lower values.

opt[2]	**Termination**
0	no output is printed
1	error and warning messages are printed
2	the iteration history is printed (this is the default)
3	the n least-squares (L_2 norm) estimates are printed if no starting point is specified; the L_1 norm estimates are printed; if *opt*[3] is set, the estimates are printed together with the asymptotic standard errors
4	the $n \times n$ approximate covariance matrix of parameter estimates is printed if *opt*[3] is set
5	the residual and predicted values for all m rows (equations) of $\mathbf{A}$ are printed

opt[3] specifies which estimate of the variance of the median of nonzero residuals is to be used as a factor for the approximate covariance matrix of parameter estimates and for the approximate standard errors (ASE). If $opt[3] = 0$, the McKean-Schrader (1987) estimate is used, and if $opt[3] > 0$, the Cox-Hinkley (1974) estimate, with $v = opt[3]$, is used. The default is $opt[3] = -1$ or $opt[3] = .$, which means that the covariance matrix is not computed.

opt[4] specifies whether a computationally expensive test for necessary and sufficient optimality of the solution x is executed. The default is *opt*[4] = 0 or *opt*[4] = ., which means that the convergence test is not performed.

Missing values are not permitted in the a or b argument. The $x0$ argument is ignored if it contains any missing values. Missing values in the *opt* argument cause the default value to be used.

The Least Absolute Values (LAV) subroutine is designed for solving the unconstrained linear L_1 norm minimization problem,

$$\min_{\mathbf{x}} L_1(\mathbf{x}) \quad \text{where} \quad L_1(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 = \sum_{i=1}^{m} \left| \sum_{j=1}^{n} a_{ij} x_j - b_i \right|$$

for m equations with n (unknown) parameters $\mathbf{x} = (x_1, \ldots, x_n)$. This is equivalent to estimating the unknown parameter vector, $\mathbf{x}$, by least absolute value regression in the model

$$\mathbf{b} = \mathbf{A}\mathbf{x} + \boldsymbol{\epsilon}$$

where $\mathbf{b}$ is the vector of n observations, $\mathbf{A}$ is the design matrix, and $\boldsymbol{\epsilon}$ is a random error term.

An algorithm by Madsen and Nielsen (1993) is used, which can be faster for large values of m and n than the Barrodale and Roberts (1974) algorithm. The current version of the algorithm assumes that $\mathbf{A}$ has full column rank. Also, constraints cannot be imposed on the parameters in this version.

The L_1 norm minimization problem is more difficult to solve than the least-squares (L_2 norm) minimization problem because the objective function of the L_1 norm problem is not continuously differentiable (the first derivative has jumps). A function that is continuous but not continuously differentiable is called *nonsmooth*. Using PROC NLP and the IML nonlinear optimization subroutines, you can obtain the estimates in linear and nonlinear L_1 norm estimation (even subject to linear or nonlinear constraints) as long as the number of parameters, n, is small. Using the nonlinear optimization subroutines, there are two ways to solve the nonlinear L_p-norm, $p \geq 1$, problem:

- For small values of n, you can implement the Nelder-Mead simplex algorithm with the NLPNMS subroutine to solve the minimization problem in its original specification. The Nelder-Mead simplex algorithm does not assume a smooth objective function, does not take advantage of any derivatives, and therefore does not require continuous differentiability of the objective function. See the "NLPNMS Call" section on page 798 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 808 for details on the NLPQN subroutine.

Both approaches are successful only for a small number of parameters and good initial estimates. If you cannot supply good initial estimates, the optimal results of the corresponding nonlinear least-squares (L_2 norm) estimation may provide fairly good initial estimates.

Gonin and Money (1989, pp. 44–45) show that the nonlinear L_1 norm estimation problem

$$\min_{\mathbf{x}} \sum_{i=1}^{m} |f_i(\mathbf{x})|$$

can be reformulated as a linear optimization problem with nonlinear constraints in the following ways.

- $$\min_{\mathbf{x}} \sum_{i=1}^{m} u_i \quad \text{subject to} \quad \left. \begin{array}{rcl} f_i(\mathbf{x}) - u_i & \leq & 0 \\ f_i(\mathbf{x}) + u_i & \geq & 0 \\ u_i & \geq & 0 \end{array} \right\} \quad i = 1, \ldots, m$$

  is a linear optimization problem with $2m$ nonlinear inequality constraints in $m + n$ variables u_i and x_j.

- $$\min_{\mathbf{x}} \sum_{i=1}^{m} (y_i + z_i) \quad \text{subject to} \quad \left. \begin{array}{rcl} f_i(\mathbf{x}) + y_i - z_i & = & 0 \\ y_i & \geq & 0 \\ z_i & \geq & 0 \end{array} \right\} \quad i = 1, \ldots, m$$

  is a linear optimization problem with $2m$ nonlinear equality constraints in $2m + n$ variables y_i, z_i, and x_j.

For linear functions $f_i(\mathbf{x}) = \sum_{j=1}^{n} (a_{ij}x_j - b_i)$, $i = 1, \ldots, m$, you obtain linearly constrained linear optimization problems, for which the number of variables and constraints is on the order of the number of observations, m. The advantage that the algorithm by Madsen and Nielsen (1993) has over the Barrodale and Roberts (1974) algorithm is that its computational cost increases only linearly with m, and it can be faster for large values of m.

In addition to computing an optimal solution $\mathbf{x}^*$ that minimizes $L_1(\mathbf{x})$, you can also compute approximate standard errors and the approximate covariance matrix of $\mathbf{x}^*$. The standard errors may be used to compute confidence limits.

The following example is the same one used for illustrating the LAV procedure by Lee and Gentle (1986). $\mathbf{A}$ and $\mathbf{b}$ are as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & 2 \\ 1 & 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ -1 \\ 2 \\ 4 \end{bmatrix}$$

The following code specifies the matrix A, the vector B, and the options vector OPT. The options vector specifies that all output is printed ($opt[2] = 5$), that the asymptotic standard errors and covariance matrix are computed based on the McKean-Schrader (1987) estimate λ of the variance of the median ($opt[3] = 0$), and that the convergence test should be performed ($opt[4] = 1$).

```
a = { 0,  1, -1, -1,  2,  2 };
m = nrow(a);
a = j(m,1,1.) || a;
b = { 1,  2,  1, -1,  2,  4 };

opt= { . 5  0 1 };
call lav(rc,xr,a,b,,opt);
```

The first part of the printed output refers to the least-squares solution, which is used as the starting point. The estimates of the largest and smallest nonzero eigenvalues of $\mathbf{A'A}$ give only an idea of the magnitude of these values, and they can be very crude approximations.

The second part of the printed output shows the iteration history.

The third part of the printed output shows the L_1 norm solution (first row) together with asymptotic standard errors (second row) and the asymptotic covariance matrix of parameter estimates (the ASEs are the square roots of the diagonal elements of this covariance matrix).

The last part of the printed output shows the predicted values and residuals, as in Lee and Gentle (1986).

LCP Call

solves the linear complementarity problem

> **CALL LCP(** *rc, w, z, m, q* <*, epsilon*> **);**

The inputs to the LCP subroutine are as follows:

m	is an $m \times m$ matrix.
q	is an $m \times 1$ matrix.
epsilon	is a scalar defining virtual zero. The default value of *epsilon* is 1.0E−8.
rc	returns one of the following scalar return codes:

0	solution found
1	no solution possible
5	solution is numerically unstable
6	subroutine could not obtain enough memory.

w and z	return the solution in an m-element column vector.

The LCP subroutine solves the linear complementarity problem:

$$
\begin{aligned}
\mathbf{w} &= \mathbf{Mz} + \mathbf{q} \\
\mathbf{w'z} &= 0 \\
\mathbf{w}, \mathbf{z} &\geq 0
\end{aligned}
$$

Consider the following example:

```
q={1, 1};
m={1 0,
   0 1};
call lcp(rc,w,z,m,q);
```

The result is

```
         RC              1 row        1 col      (numeric)

                                      0


         W               2 rows       1 col      (numeric)

                                      1
                                      1


         Z               2 rows       1 col      (numeric)

                                      0
                                      0
```

The next example shows the relationship between quadratic programming and the linear complementarity problem. Consider the linearly constrained quadratic program:

$$
\begin{aligned}
\min \mathbf{c'x} &+ \frac{1}{2}\mathbf{x'Hx} \\
\text{st. } \mathbf{Gx} &\geq \mathbf{b} \quad \text{(QP)} \\
\mathbf{x} &\geq 0
\end{aligned}
$$

If $\mathbf{H}$ is positive semidefinite, then a solution to the Kuhn-Tucker conditions solves QP. The Kuhn-Tucker conditions for QP are

$$
\begin{aligned}
\mathbf{c} + \mathbf{Hx} &= \mu + \mathbf{G'}\lambda \\
\lambda'(\mathbf{Gx} - \mathbf{b}) &= 0 \\
\mu'\mathbf{x} &= 0 \\
\mathbf{Gx} &\geq \mathbf{b} \\
\mathbf{x}, \mu, \lambda &\geq 0
\end{aligned}
$$

In the linear complementarity problem, let

$$\mathbf{M} \;=\; \begin{bmatrix} H & -G' \\ G & 0 \end{bmatrix}$$

$$\mathbf{w}' \;=\; (\mu' \mathbf{s}')$$

$$\mathbf{z}' \;=\; (\mathbf{x}' \lambda')$$

$$\mathbf{q}' \;=\; (\mathbf{c}' - \mathbf{b})$$

Then the Kuhn-Tucker conditions are expressed as finding $\mathbf{w}$ and $\mathbf{z}$ that satisfy

$$\mathbf{w} \;=\; \mathbf{Mz} + \mathbf{q}$$

$$\mathbf{w}'\mathbf{z} \;=\; 0$$

$$\mathbf{w}, \mathbf{z} \;\geq\; 0$$

From the solution $\mathbf{w}$ and $\mathbf{z}$ to this linear complementarity problem, the solution to QP is obtained; namely, $\mathbf{x}$ is the primal structural variable, $\mathbf{s} = \mathbf{Gx} - \mathbf{b}$ the surpluses, and μ and λ are the dual variables. Consider a quadratic program with the following data:

$$\mathbf{C}' \;=\; (1245) \quad \mathbf{B}' = (11)$$

$$\mathbf{H} \;=\; \begin{bmatrix} 100 & 10 & 1 & 0 \\ 10 & 100 & 10 & 1 \\ 1 & 10 & 100 & 10 \\ 0 & 1 & 10 & 100 \end{bmatrix}$$

$$\mathbf{G} \;=\; \begin{bmatrix} 1 & 2 & 3 & 4 \\ 10 & 20 & 30 & 40 \end{bmatrix}$$

This problem is solved using the LCP subroutine in PROC IML as follows:

```
   /*---- Data for the Quadratic Program -----*/
c={1,2,3,4};
h={100 10 1 0, 10 100 10 1, 1 10 100 10, 0 1 10 100};
g={1 2 3 4, 10 20 30 40};
b={1, 1};

   /*----- Express the Kuhn-Tucker Conditions as an LCP ----*/
m=h||-g`;
m=m//(g||j(nrow(g),nrow(g),0));
q=c//-b ;

   /*----- Solve for a Kuhn-Tucker Point --------*/
call lcp(rc,w,z,m,q);

   /*------ Extract the Solution to the Quadratic Program ----*/
x=z[1:nrow(h)];
print rc x;
```

The printed solution is

```
                    RC              1 row        1 col       (numeric)

                                                   0

                    X               4 rows       1 col       (numeric)

                                              0.0307522
                                              0.0619692
                                              0.0929721
                                              0.1415983
```

LENGTH Function

finds the lengths of character matrix elements

> **LENGTH(** *matrix***)**

where *matrix* is a character matrix or quoted literal.

The LENGTH function takes a character matrix as an argument and produces a numeric matrix as a result. The result matrix has the same dimensions as the argument and contains the lengths of the corresponding string elements in *matrix*. The length of a string is equal to the position of the rightmost nonblank character in the string. If a string is entirely blank, its length value is set to 1. An example of the LENGTH function follows:

```
c={'Hello' 'My name is Jenny'};
b=length(c);
```

```
                    B               1 row        2 cols      (numeric)

                                                   5           16
```

See also the description of the NLENG function.

LINK and RETURN Statements

jump to another statement

> **LINK** *label***;**
>
>   *statements*
>
> *label:statements*
>
> **RETURN;**

The LINK statement, like the GOTO statement, directs IML to jump to the statement with the specified label. Unlike the GOTO statement, however, IML remembers where the LINK statement was issued and returns to that point when a RETURN

statement is executed. This statement can only be used inside modules and DO groups.

The LINK statement provides a way of calling sections of code as if they were subroutines. The LINK statement calls the routine. The routine begins with the label and ends with a RETURN statement. LINK statements can be nested within other LINK statements to any level. A RETURN statement without a LINK statement is executed the same as the STOP statement.

Any time you use a LINK statement, you may consider using a RUN statement and a module defined using the START and FINISH statements instead.

An example using the LINK statement is shown below:

```
start a;
   x=1;
   y=2;
   link sum1;
   print z;
   stop;
   sum1:
      z=x+y;
   return;
finish a;
run a;
```

```
            Z              1 row       1 col     (numeric)

                                        3
```

LIST Statement

displays observations of a data set

> **LIST** <*range*> <**VAR** *operand*> <**WHERE(***expression***)**>;

The inputs to the LIST statement are as follows:

range	specifies a range of observations
operand	specifies a set of variables
expression	is an expression evaluated to be true or false.

The LIST statement prints selected observations of a data set. If all data values for variables in the VAR clause fit on a single line, values are displayed in columns headed by the variable names. Each record occupies a separate line. If the data values do not fit on a single line, values from each record are grouped into paragraphs. Each element in the paragraph has the form *name=value*.

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

ALL	all observations
CURRENT	the current observation (this is the default for the LIST statement)
NEXT <*number*>	the next observation or the next *number* of observations
AFTER	all observations after the current one
POINT *operand*	observations specified by number, where *operand* can be one of the following:

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

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

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

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

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

Below are examples showing each possible way you can use the VAR clause:

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

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

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

In the statement above,

variable   is a variable in the SAS data set.

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

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

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

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

   ^= ^? < <= > >=

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

   = ? =: =*

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

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

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

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

Below are several examples on using the LIST statement:

```
list all;                              /* lists whole data set */
list;                             /* lists current observation */
list var{name addr};     /* lists NAME and ADDR in current obs */
list all where(age>30); /* lists all obs where condition holds */
list next;                            /* lists next observation */
list point 24;                          /* lists observation 24 */
list point (10:15);        /* lists observations 10 through 15 */
```

LMS Call

performs robust regression

> **CALL LMS(** *sc, coef, wgt, opt,* $y <, < x ><, sorb>>$**);**

The Least Median of Squares (LMS) performs *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 value of h can be specified, but in most applications the default value works just fine and the results seem to be quite stable toward different choices of h.

In the following discussion, N is the number of observations and n is the number of regressors. The inputs 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.
>
> > *opt*[2]=0   prints no output except error messages.
> >
> > *opt*[2]=1   prints all output except (1) arrays of $O(N)$, such as weights, residuals, and diagnostics; (2) the history of the optimization process; and (3) subsets that result in singular linear systems.
> >
> > *opt*[2]=2   additionally prints arrays of $O(N)$, such as weights, residuals, and diagnostics; also prints the case numbers of the observations in the best subset and some basic history of the optimization process.
> >
> > *opt*[2]=3   additionally prints subsets that result in singular linear systems.
> >
> > The default is *opt*[2]=0.
>
> *opt*[3]   specifies whether only LMS is computed or whether, additionally, least-squares (LS) and weighted least-squares (WLS) regression are computed.

opt[3]=0 computes only LMS.

opt[3]=1 computes, in addition to LMS, weighted least-squares regression on the observations with *small* LMS residuals (where *small* is defined by opt[8]).

opt[3]=2 computes, in addition to LMS, unweighted least-squares regression.

opt[3]=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 consisting of n observations out of N observations is

$$N_{\text{tot}} = \binom{N}{n} = \frac{\prod_{j=1}^{n}(N - j + 1)}{\prod_{j=1}^{n} j}$$

where n is the number of parameters including the intercept.

Due to computer time restrictions, not all subset combinations of n observations out of N can be inspected for larger values of N and n. Specifying a value of $N_{\text{Rep}} < N_{\text{tot}}$ enables you to save computer time at the expense of computing a suboptimal solution.

If *opt*[5] is zero or missing, the default number of subsets is taken from the following table.

n	1	2	3	4	5	6	7	8	9	10
N_{lower}	500	50	22	17	15	14	0	0	0	0
N_{upper}	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 indicating 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.

      opt[7]=0 *sorb* contains a given subset index.
      opt[7]=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.

      opt[8]=0 does not compute covariance matrix and ASEs.
      opt[8]=1 computes covariance matrix and ASEs but prints neither of them.
      opt[8]=2 computes the covariance matrix and ASEs but prints only the ASEs.
      opt[8]=3 computes and prints both the covariance matrix and the ASEs.

      The default is *opt*[8]=0.

y refers to an N response vector **y**.

x refers to an $N \times n$ matrix **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 **X**. If the matrix **X** is not specified, **y** is analyzed as a univariate data set.

sorb   refers to an n vector containing either of the following:

- n observation numbers of a subset for which the objective function should be evaluated; this subset can be the start for a pairwise exchange algorithm if *opt*[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 containing 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^2 (*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^2 value for LS or WLS
sc[14]	F value for LS or WLS

For *opt*[3]=1 or *opt*[3]=3, these rows correspond to WLS estimates; for *opt*[3]=2, these rows correspond to LS estimates.

coef   is a matrix with n columns containing the following results in its rows:

coef[1,]	LMS 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, to LS estimates.

wgt   is a matrix with N columns containing the following results in its rows:

wgt[1]	weights (=1 for small, =0 for large residuals)
wgt[2]	residuals $r_i = y_i - \mathbf{x}_i \mathbf{b}$
wgt[3]	resistant diagnostic u_i (note that the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider results for Brownlee's (1965) stackloss data. The three explanatory variables correspond to measurements for a plant oxidizing ammonia to nitric acid:

- x_1 air flow to the plant
- x_2 cooling water inlet temperature
- x_3 acid concentration

on 21 consecutive days. The response variable y_i gives the permillage of ammonia lost (stackloss). The data are also given by Rousseeuw and Leroy (1987, p. 76) and Osborne (1985, p. 267). Rousseeuw and Leroy (1987, p. 76) 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 outlier.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5985 different subsets of 4 observations out of 21. If you decide not to specify `optn[5]`, the LMS subroutine chooses $N_{rep} = 2000$ random sample subsets. Since there is a large number of subsets with singular linear systems, which you do not want to print, choose `optn[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];
  optn = j(8,1,.);
  optn[2]= 2;    /* ipri */
  optn[3]= 3;    /* ilsq */
  optn[8]= 3;    /* icov */

  CALL LMS(sc,coef,wgt,optn,b,a);
```

```
       LMS: The 13th ordered squared residual will be minimized.


                          Median and Mean

                            Median                Mean

          VAR1                   58         60.428571429
          VAR2                   20         21.095238095
          VAR3                   87         86.285714286
          Intercep                1                    1
          Response               15         17.523809524


                 Dispersion and Standard Deviation

                         Dispersion              StdDev

          VAR1          5.930408874         9.1682682584
          VAR2          2.965204437          3.160771455
          VAR3          4.4478066555        5.3585712381
          Intercep                0                    0
          Response      5.930408874        10.171622524
```

The following are the results of LS regression:

```
                 Unweighted Least-Squares Estimation


                      LS Parameter Estimates

                           Approx               Pr >
     Variable    Estimate  Std Err    t Value    |t|

     VAR1        0.715640  0.134858     5.31    <.0001
     VAR2        1.295286  0.368024     3.52    0.0026
     VAR3       -0.152123  0.156294    -0.97    0.3440
     Intercep  -39.919674 11.895997    -3.36    0.0038

     Variable   Lower WCI   Upper WCI

     VAR1        0.451323   0.979957
     VAR2        0.573972   2.016600
     VAR3       -0.458453   0.154208
     Intercep   -63.2354   -16.603949



              Sum of Squares = 178.8299616
                Degrees of Freedom = 17
              LS Scale Estimate = 3.2433639182
```

```
                  Cov Matrix of Parameter Estimates

                VAR1         VAR2         VAR3        Intercep

VAR1          0.018187    -0.036511     0.007144     0.287587
VAR2         -0.036511     0.135442     0.000010    -0.651794
VAR3         -0.007144     0.000011     0.024428    -1.676321
Intercep      0.287587    -0.651794     1.676321   141.514741


             R-squared = 0.9135769045
          F(3,17) Statistic = 59.9022259
             Probability = 3.0163272E-9
```

These are the LMS results for the 2,000 random subsets:

```
                  Random Subsampling for LMS




                               Best
         Subset    Singular   Criterion     Percent

           500         23     0.163262         25
          1000         55     0.140519         50
          1500         79     0.140519         75
          2000        103     0.126467        100


         Minimum Criterion= 0.1264668282
          Least Median of Squares (LMS) Method
        Minimizing 13th Ordered Squared Residual.
      Highest Possible Breakdown Value =  42.86 %
            Random Selection of 2103 Subsets
          Among 2103 subsets 103 are singular.


              Observations of Best Subset

      15            11            19            10

                Estimated Coefficients

      VAR1          VAR2          VAR3        Intercep

      0.75          0.5            0           -39.25


          LMS Objective Function = 0.75
           Preliminary LMS Scale = 1.0478510755
                Robust R Squared = 0.96484375
               Final LMS Scale = 1.2076147288
```

For LMS observations, 1, 3, 4, and 21 have scaled residuals larger than 2.5 (table not shown) and are considered outliers. These are the corresponding WLS results:

```
                 Weighted Least-Squares Estimation

              RLS Parameter Estimates Based on LMS

                               Approx                 Pr >
     Variable      Estimate    Std Err   t Value      |t|

     VAR1          0.797686   0.067439    11.83      <.0001
     VAR2          0.577340   0.165969     3.48      0.0041
     VAR3         -0.067060   0.061603    -1.09      0.2961
     Intercep    -37.652459   4.732051    -7.96      <.0001


                        Lower WCI     Upper WCI

                         0.665507      0.929864
                         0.252047      0.902634
                        -0.187800      0.053680
                       -46.927108    -28.37781


                Weighted Sum of Squares = 20.400800254
                       Degrees of Freedom = 13
                  RLS Scale Estimate = 1.2527139846

                 Cov Matrix of Parameter Estimates

                  VAR1         VAR2         VAR3        Intercep

     VAR1       0.004548    -0.007921    -0.001199     0.001568
     VAR2      -0.007921     0.027546    -0.000463    -0.065018
     VAR3      -0.001199    -0.000463     0.003795    -0.246102
     Intercep   0.001568    -0.065018    -0.246102    22.392305

                 Weighted R-squared = 0.9750062263
                 F(3,13) Statistic = 169.04317954
                      Probability = 1.158521E-10
               There are 17 points with nonzero weight.
                     Average Weight = 0.8095238095
```

LOAD Statement

loads modules and matrices from library storage

> **LOAD** <**MODULE=(***module-list***)**> <*matrix-list*>;

The inputs to the LOAD statement are as follows:

module-list   is a list of modules.

matrix-list   is a list of matrices

The LOAD statement loads modules or matrix values from the current library storage into the current workspace. For example, to load three modules A, B, and C and one matrix X, specify the statement

```
load module=(A B C) X;
```

The special operand _ALL_ can be used to load all matrices or all modules. For example, if you want to load all matrices, specify

```
load _all_;
```

If you want to load all modules, specify

```
load module=_all_;
```

To load all matrices and modules stored in the library storage, you can enter the LOAD command without any arguments:

```
load;
```

The storage library can be specified using a RESET *storage* command. The default library is WORK.IMLSTOR. For more information, see Chapter 14, "Storage Features," and the descriptions of the STORE, REMOVE, RESET, and SHOW statements.

LOC Function

finds nonzero elements of a matrix

    LOC(*matrix***)**

where *matrix* is a numeric matrix or literal.

The LOC function creates a $1 \times n$ row vector, where n is the number of nonzero elements in the argument. Missing values are treated as zeros. The values in the resulting row vector are the locations of the nonzero elements in the argument (in row-major order, like subscripting). For example, the statements

```
a={1 0 2 3 0};
b=loc(a);
```

result in the row vector

```
                 B               1 row       3 cols     (numeric)

                                     1          3          4
```

since the first, third, and fourth elements of **A** are nonzero. 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, suppose you want to create a matrix **Y** containing the rows of **X** that have a positive element in the diagonal of **X**. Specify the following statements.

```
x={1  1  0,
   0 -2  2,
   0  0  3};
y=x[loc(vecdiag(x)>0),];
```

The result is

$$Y = X[\{13\},]$$

or the matrix

```
          Y          2 rows     3 cols     (numeric)

                        1          1          0
                        0          0          3
```

since the first and third rows of **X** have positive elements on the diagonal of **X**.

The next example selects all positive elements of a column vector **A**:

```
a={0,
   -1,
    2,
    0};
y=a[loc(a>0),];
```

The result is

$$Y = A[3,]$$

or the scalar

```
          Y          1 row      1 col      (numeric)

                        2
```

LOG Function

takes the natural logarithm

LOG(*matrix***)**

where *matrix* is a numeric matrix or literal.

The LOG function is the scalar function that takes the natural logarithm of each element of the argument matrix. An example of a valid statement is shown below:

```
c = { 1 2 3 };
b=log(c);
print b;
```

```
          B

  0 0.6931472 1.0986123
```

LP Call

solves the linear programming problem

    CALL LP(*rc, x, dual, a, b <, cntl><, u><, l><, basis>***);**

The inputs to the LP subroutine are as follows:

a is an $m \times n$ vector specifying the technological coefficients, where m is less than or equal to n.

b is an $m \times 1$ vector specifying the right-side vector.

cntl   is an optional row vector with 1 to 5 elements. If CNTL=(*indx, nprimal, ndual, epsilon, infinity*), then

         indx   is the subscript of nonzero objective coefficient.

         nprimal   is the maximum number of primal iterations.

         ndual   is the maximum number of dual iterations.

         epsilon   is the value of virtual zero.

         infinity   is the value of virtual infinity.

     The default values are as follows: *indx* equals n, *nprimal* equals 999999, *ndual* equals 999999, *epsilon* equals 1.0E−8, and *infinity* is machine dependent. If you specify *ndual* or *nprimal* or both, then on return they contain the number of iterations actually performed.

u is an optional array of dimension n specifying upper bounds on the decision variables. If you do not specify u, the upper bounds are assumed to be *infinity*.

l is an optional array of dimension n specifying lower bounds on the decision variables. If l is not given, then the lower bounds are assumed to be 0 for all the decision variables. This includes the decision variable associated with the objective value, which is specified by the value of *indx*.

basis   is an optional array of dimension n specifying the current basis. This is given by identifying which columns are explicitly in the basis and which columns are at their upper bound, as given in u. The absolute value of the elements in this vector is a permutation of the column indices. The columns specified in the first m elements of *basis* are considered the explicit basis. The absolute value of the last $n - m$ elements of *basis* are the indices of the nonbasic variables. Any of the last $n - m$ elements of *basis* that are negative indicate that that nonbasic variable is at its upper bound. On return from the LP subroutine, the *basis* vector contains the final basis encountered. If you do not specify *basis*, then the subroutine assumes that an initial basis is in the last m columns of **A** and that no nonbasic variables are at their upper bound.

rc   returns one of the following scalar return codes:

    0   solution is optimal
    1   solution is primal infeasible and dual feasible
    2   solution is dual infeasible and primal feasible
    3   solution is neither primal nor dual feasible
    4   singular basis encountered
    5   solution is numerically unstable
    6   subroutine could not obtain enough memory
    7   number of iterations exceeded

    x returns the current primal solution in a column vector of length n.

    dual   returns the current dual solution in a row vector of length m.

The LP subroutine solves the linear program:

$$\max(0, \ldots, 0, 1, 0, \ldots, 0)\mathbf{x}$$
$$\text{st. } \mathbf{A}\mathbf{x} = \mathbf{b}$$
$$l \leq \mathbf{x} \leq \mathbf{u}$$

The subroutine first inverts the initial basis. If the **BASIS** vector is given, then the initial basis is the $m \times m$ submatrix identified by the first m elements in **BASIS**; otherwise, the initial basis is defined by the last m columns of **A**. If the initial basis is singular, the subroutine returns with RC=4. If the basis is nonsingular, then the current dual and primal solutions are evaluated. If neither is feasible, then the subroutine returns with RC=3. If the primal solution is feasible, then the primal algorithm iterates until either a dual feasible solution is encountered or the number of NPRIMAL iterations is exceeded. If the dual solution is feasible, then the dual algorithm iterates until either a primal feasible solution is encountered or the number of NDUAL iterations is exceeded. When a basis is identified that is both primal and dual feasible, then the subroutine returns with RC=0.

Note that care must be taken when solving a sequence of linear programs and using the NPRIMAL or NDUAL control parameters or both. Because the LP subroutine resets the NPRIMAL and NDUAL parameters to reflect the number of iterations executed, subsequent invocations of the LP subroutine will have the number of iterations limited to the number used by the last LP subroutine executed. In these cases you should consider resetting these parameters prior to each LP call.

Consider the following example to maximize X_1 subject to the constraints $X_1 + X_2 \leq 10$ and $X_1 \geq 0$. The problem is solved as follows:

```
    /* the problem data */
obj={1 0};
coef={1 -1};
b={0, 10};

    /* embed the objective function */
    /* in the coefficient matrix    */
a=obj//coef;
a=a||{-1, 0};

    /* solve the problem */
call lp(rc,x,dual,a,b);
```

The result is

```
            RC              1 row        1 col      (numeric)

                                          0


            X               3 rows       1 col      (numeric)

                                         10
                                          0
                                         10


            DUAL            1 row        2 cols     (numeric)

                                   -1            1
```

LTS Call

performs robust regression

> **CALL LTS(** *sc, coef, wgt, opt,* $y <, < x ><,$ *sorb>>***);**

A robust (resistant) regression method, defined by minimizing the sum of the h smallest squared residuals.

The Least Trimmed Squares (LTS) subroutine performs *robust* regression (sometimes called *resistant* regression). It is able to detect outliers and perform a least-squares regression on the remaining observations. Beginning with SAS/IML Version 8.1,

the LTS subroutine implements a new algorithm, FAST-LTS, given by Rousseeuw and Van Driessen (1998). The new algorithm is set as the default. The algorithm in previous versions is temporarily available, but will be phased out. See *opt*[9] for detail.

The value of h may be specified, but for many applications the default value works just fine 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 inputs 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]$\neq$ 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.

      opt[2]=0   prints no output except error messages.

      opt[2]=1   prints all output except (1) arrays of $O(N)$, such as weights, residuals, and diagnostics; (2) the history of the optimization process; and (3) subsets that result in singular linear systems.

      opt[2]=2   additionally prints arrays of $O(N)$, such as weights, residuals, and diagnostics; also prints the case numbers of the observations in the best subset and some basic history of the optimization process.

      opt[2]=3   additionally prints subsets that result in singular linear systems.

      The default is *opt*[2]=0.

   opt[3]   specifies whether only LTS is computed or whether, additionally, least-squares (LS) and weighted least-squares (WLS) regression are computed:

      opt[3]=0   computes only LTS.

      opt[3]=1   computes, in addition to LTS, weighted least-squares regression on the observations with *small* LTS residuals (where *small* is defined by opt[8]).

      opt[3]=2   computes, in addition to LTS, unweighted least-squares regression.

      opt[3]=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 consisting of n observations out of N observations is

$$N_{\mathrm{tot}} = \binom{N}{n} = \frac{\prod_{j=1}^{n}(N - j + 1)}{\prod_{j=1}^{n} j}$$

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 $N_{\mathrm{Rep}} < N_{\mathrm{tot}}$ enables you to save computer time at the expense of computing a suboptimal solution.

If *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.

For the default FAST-LTS algorithm with $N < 600$ or the previous algorithm (before V8.1), 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 V8.1 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 indicating 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.

> *opt*[7]=0 *sorb* contains a given subset index.
> *opt*[7]=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.

> *opt*[8]=0 does not compute covariance matrix and ASEs.
> *opt*[8]=1 computes covariance matrix and ASEs but prints neither of them.
> *opt*[8]=2 computes the covariance matrix and ASEs but prints only the ASEs.
> *opt*[8]=3 computes and prints both the covariance matrix and the ASEs.

> The default is *opt*[8]=0.

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 refers to an N response vector **y**.

x refers to an $N \times n$ matrix **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 **X**. If the matrix **X** is not specified, **y** is analyzed as a univariate data set.

sorb refers to an n vector containing either of the following:

- n observation numbers of a subset for which the objective function should be evaluated; this subset can be the start for a pairwise exchange algorithm if *opt*[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 containing 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^2 (*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^2 value for LS or WLS
sc[14]	F value for LS or WLS

For *opt*[3]=1 or *opt*[3]=3, these rows correspond to WLS estimates; for *opt*[3]=2, these rows correspond to LS estimates.

coef	is a matrix with n columns containing the following results in its rows:

coef[1,]	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, to LS estimates.

wgt	is a matrix with N columns containing the following results in its rows:

wgt[1]	weights (=1 for small, =0 for large residuals)
wgt[2]	residuals $r_i = y_i - \mathbf{x}_i\mathbf{b}$
wgt[3]	resistant diagnostic u_i (note that the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider Brownlee's (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 optn[5], the FAST-LTS algorithm chooses 500 random sample subsets:

```
        /* 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];
 optn = j(8,1,.);
 optn[2]= 1;     /* ipri */
 optn[3]= 3;     /* ilsq */
 optn[8]= 3;     /* icov */

 CALL LTS(sc,coef,wgt,optn,b,a);
```

The preceding program produces the following output:

```
      Least Trimmed Squares (LTS) Method
  Minimizing Sum of 13 Smallest Squared Residuals.
    Highest Possible Breakdown Value =  42.86 %
       Random Selection of 523 Subsets
        Among 523 subsets 23 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 = 0.973976868
         Final LTS Scale = 1.0360272594
```

For LTS observations, 1, 2, 3, 4, 13, and 21 have scaled residuals larger than 2.5 (table not shown) and are considered outliers. These are the corresponding WLS results:

Weighted Least-Squares Estimation

RLS Parameter Estimates Based on LMS

Variable	Estimate	Approx Std Err	t Value	Pr > |t|
VAR1	0.756940	0.078607	9.63	<.0001
VAR2	0.453530	0.136050	3.33	0.0067
VAR3	-0.05211	0.054637	-0.95	0.3607
Intercep	-34.0575	3.828818	-8.90	<.0001

Lower WCI	Upper WCI
0.602872	0.911008
0.186876	0.720184
-0.15919	0.054977
-41.5618	-26.5531

```
         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	0.0061791	-0.005776	-0.002300	-0.034290
VAR2	-0.005776	0.0185096	0.0002582	-0.069740

```
VAR3        -0.002300     0.0002582     0.0029852     -0.131487
Intercep    -0.034290    -0.069740     -0.131487      14.659852

            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
```

See the entry for the LMS subroutine for details.

LUPDT Call

provides updating and downdating for rank deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses

> **CALL LUPDT(** *lup, bup, sup, l, z <, b, y <, ssq>>***);**

The LUPDT subroutine returns the following values:

lup   is an $n \times n$ lower triangular matrix $\mathbf{L}$ that is updated or downdated by using the q rows in $\mathbf{Z}$.

bup   is an $n \times p$ matrix $\mathbf{B}$ of right-hand sides that is updated or downdated by using the q rows in $\mathbf{Y}$. If b is not specified, *bup* is not accessible.

sup   is a p vector of square roots of residual sum of squares that is updated or downdated by using the q rows in $\mathbf{Y}$. If *ssq* is not specified, *sup* is not accessible.

The inputs to the LUPDT subroutine are as follows:

l   specifies an $n \times n$ lower triangular matrix $\mathbf{L}$ to be updated or downdated by q row vectors z stored in the $q \times n$ matrix $\mathbf{Z}$. Only the lower triangle of l is used; the upper triangle may contain any information.

z   is a $q \times n$ matrix $\mathbf{Z}$ used rowwise to update or downdate the matrix $\mathbf{L}$.

b   specifies an optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with $\mathbf{L}$. If b is specified, the argument y must be specified.

y   specifies an optional $q \times p$ matrix $\mathbf{Y}$ used rowwise to update or downdate the right-hand-side matrix $\mathbf{B}$.

ssq   specifies an optional $p \times 1$ vector that, if b is specified, specifies the square root of the error sum of squares that should be updated or downdated simultaneously with $\mathbf{L}$ and b.

The relevant formula for the LUPDT call is $\tilde{\mathbf{L}}\tilde{\mathbf{L}}' = \mathbf{L}\mathbf{L}' + \mathbf{Z}\mathbf{Z}'$. See the example in the documentation for the RZLIND call.

MAD Function

finds the univariate (scaled) median-absolute-deviation

> **MAD(** *(x <, spt >)***)**

where

x   is an $n \times p$ input data matrix.

spt   is an optional string argument with the following values:

> **"MAD"** for computing the MAD (which is the default)
>
> **"NMAD"** for computing the normalized version of MAD
>
> **"SN"** for computing S_n
>
> **"QN"** for computing Q_n

The MAD function treats the input matrix x as univariate data by appending each row to the previous row to make a single row vector with elements $x_{11}, \ldots, x_{1p}, x_{21}, \ldots, x_{2p}, \ldots, x_{n1}, \ldots, x_{np}$. In the following description, the notation x_i means the ith element of x when thought of as a row vector.

The MAD function can be used for computing one of the following three robust scale estimates:

- Median Absolute Deviation (MAD) or normalized form of MAD:

$$MAD_n = b * med_i^n |x_i - med_j^n x_j|$$

  where $b = 1$ is the unscaled default and $b = 1.4826$ is used for the scaled version (consistency with the Gaussian distribution).

- S_n, which is a more efficient alternative to MAD:

$$S_n = c_n * med_i \, med_{j \neq i} |x_i - x_j|$$

  where the outer median is a low median (order statistic of rank $\left[\frac{n+1}{2}\right]$) and the inner median is a high median (order statistic of rank $\left[\frac{n}{2} + 1\right]$), and where c_n is a scalar depending on sample size n.

- Q_n is another efficient alternative to MAD. It is based on the kth order statistic of the $\binom{n}{2}$ 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:

$$c_n = 1.1926* \begin{cases} .743 & \text{for n=2} \\ 1.851 & \text{for n=3} \\ .954 & \text{for n=4} \\ 1.351 & \text{for n=5} \\ .993 & \text{for n=6} \\ 1.198 & \text{for n=7} \\ 1.005 & \text{for n=8} \\ 1.131 & \text{for n=9} \\ n/(n-0.9) & \text{odd n} \\ 1.0 & \text{otherwise} \end{cases} \qquad d_n = 2.2219* \begin{cases} .399 & \text{for n=2} \\ .994 & \text{for n=3} \\ .512 & \text{for n=4} \\ .844 & \text{for n=5} \\ .611 & \text{for n=6} \\ .857 & \text{for n=7} \\ .669 & \text{for n=8} \\ .872 & \text{for n=9} \\ n/(n+1.4) & \text{uneven n} \\ n/(n+3.8) & \text{even n} \end{cases}$$

Example

This example uses the univariate data set of Barnett and Lewis (1978) that is used above to illustrate the univariate LMS and LTS estimates:

```
b = { 3, 4, 7, 8, 10, 949, 951 };

rmad1 = mad(b);
rmad2 = mad(b,"mad");
rmad3 = mad(b,"nmad");
rmad4 = mad(b,"sn");
rmad5 = mad(b,"qn");
print "Default MAD=" rmad1,
      "Common MAD =" rmad2,
      "MAD*1.4826 =" rmad3,
      "Robust S_n =" rmad4,
      "Robust Q_n =" rmad5;
```

This program produces the following:

```
Default MAD=          4
Common MAD =          4
MAD*1.4826 = 5.9304089
Robust S_n =  7.143674
Robust Q_n = 5.7125049
```

MARG Call

evaluates marginal totals in a multiway contingency table

CALL MARG(*locmar, marginal, dim, table, config***);**

The inputs to the MARG subroutine are as follows:

locmar   is a returned matrix containing 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 on specifying *table*.

For example, consider the three-dimensional table discussed in the IPF call, based on data appearing 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 program below encodes this table, uses the MARG call to compute a 2-way marginal table by summing over the third variable and a 1-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,

/* Prote-  Hi */ 359    233   109   197   90,
/* stant   Lo */ 159    173    47    82   32
       };


config = { 1 3,
           2 0 };
call marg(locmar, marginal, dim, table, config);
print locmar, marginal;

/* 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)];
```

The results of this program are below.

```
   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

VAR12_MARG

```
   1051        723        276        361        203
    474        361        105        131         64
```

```
VAR3_MARG

1707
 561
1481
```

The first marginal total is contained in locations 1 through 10 of the *marginal* vector. 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.

MATTRIB Statement

associates printing attributes with matrices

> **MATTRIB** *name* <**ROWNAME=***row-name*>
>
> <**COLNAME=***column-name*> <**LABEL=***label*> <**FORMAT=***format*>;

The inputs to the MATTRIB subroutine are as follows:

name	is a character matrix or quoted literal giving the name of a matrix.
row-name	is a character matrix or quoted literal specifying row names.
column-name	is a character matrix or quoted literal specifying column names.
label	is a character matrix or quoted literal associating a label with the matrix. The *label* argument has a maximum length of 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 is used whenever the matrix is printed to label the rows and columns, respectively. The statement is written as the keyword MATTRIB followed by a list of one or more names and attribute associations. It is not necessary to specify all attributes. The attribute associations are applied to the previous *name*. Thus, the following statement gives a row name RA and a column name CA to **A**, and a column name CB to **B**:

```
mattrib a rowname=ra colname=ca b colname=cb;
```

You cannot group names; although the following statement is valid, it does not associate anything with **A**.

```
mattrib a b rowname=n;
```

The values of the associated matrices are not looked up until they are needed. Thus, they need not have values at the time the MATTRIB statement is specified. They can be specified later when the object matrix is printed. The attributes continue to bind with the matrix until reassigned with another MATTRIB statement. To eliminate an attribute, specify EMPTY as the name, for example, ROWNAME=EMPTY. Labels can be up to 40 characters long. Longer labels are truncated. Use the SHOW *names statement* to view current matrix attributes.

An example using the MATTRIB statement follows:

```
rows='xr1':'xr5';
print rows;
```

```
                              ROWS
                              xr1 xr2 xr3 xr4 xr5
```

```
cols='cl1':'cl5';
print cols;
```

```
                              COLS
                              cl1 cl2 cl3 cl4 cl5
```

```
x={1 1 1 1,2 2 2 2,3 3 3 3};
mattrib x rowname=(rows [1:3 ])
           colname=(cols [1:4])
           label={'matrix,x'}
           format=5.2;
print x;
```

```
                            matrix,x
                       cl1    cl2    cl3    cl4

                xr1    1.00   1.00   1.00   1.00
                xr2    2.00   2.00   2.00   2.00
                xr3    3.00   3.00   3.00   3.00
```

MAX Function

finds the maximum value of matrix

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

where *matrix* is a numeric or character matrix or literal.

The MAX function produces a single numeric value (or a character string value) that is the largest element (or highest character string value) in all arguments. There can be as many as 15 argument matrices. The function checks for missing numeric values and does not include them in the result. If all arguments are missing, then the machine's most negative representable number is the result.

If you want to find the elementwise maximums of the corresponding elements of two matrices, use the maximum operator (<>).

For character arguments, the size of the result is the size of the largest of all arguments.

An example using the MAX function follows:

```
c = { 1 -123 13 56 128 -81 12 };
b=max(c);
print b;

        B

        128
```

MAXQFORM Call

computes the subsets of a matrix system that maximize the quadratic form

> **CALL MAXQFORM(** *rc, maxq, V, b <, best>***);**

If $\mathbf{V}$ and $\mathbf{b}$ are an $n \times n$ matrix and an $n \times 1$ vector, respectively, then the MAXQFORM function computes the subsets of components s such that $\mathbf{b}'[s]\mathbf{V}^{-1}[s, s]\mathbf{b}[s]$ is maximized.

The MAXQFORM subroutine returns the following values:

rc   is one of the following scalar return codes:

> 0   normal return
> 1   error: the number of elements of $\mathbf{b}$ is too large to process
> 2   error: $\mathbf{V}$ is not positive semidefinite

maxq   is an $m \times (n + 2)$ matrix, where m is the total number of subsets computed and n is the number of elements in $\mathbf{b}$. The value of m depends on the value of *best* and is equal to $2^n - 1$ if *best* is not specified. Each row of *maxq* contains information for a selected subset of $\mathbf{V}$ and $\mathbf{b}$. The first element of the row is the number of components in the

subset. The second element is the value of the quadratic form. The following elements of the row are either 0 or 1, to indicate whether the corresponding components of **V** and **b** are included in the subset.

The inputs to the MAXQFORM subroutine are as follows:

V

specifies an $n \times n$ positive semidefinite matrix. Often this is generated as a crossproduct matrix, $\mathbf{X}'\mathbf{X}$, where $\mathbf{X}$ is a $k \times n$ matrix.

b

specifies an $n \times 1$ vector. Often this arises as $\mathbf{X}'\mathbf{y}$, where $\mathbf{X}$ is a $k \times n$ matrix, and $\mathbf{y}$ is a $k \times 1$ vector.

best

specifies an optional scalar. If *best* is specified with the value p, then the p subsets with the largest value for the quadratic form are returned for each subset size.

The leaps and bounds algorithm by Furnival and Wilson (1974) computes the maximum value of quadratic forms for subsets of components. Many statistics computed as a quadratic form can then be used as the criterion for the method of subset selection. These include the regression sum of squares, Wald statistics, and score statistics.

Consider the following fitness data, which consists of observations with values for age measured in years, weight measured in kilograms, time to run 1.5 miles measured in minutes, heart rate while resting, heart rate while running, maximum heart rate recorded while running, and oxygen intake rate while running measured in milliliters per kilogram of body weight per minute.

```
fit = {
  44  89.47  11.37  62  178  182  44.609,
  40  75.07  10.07  62  185  185  45.313,
  44  85.84   8.65  45  156  168  54.297,
  42  68.15   8.17  40  166  172  59.571,
  38  89.02   9.22  55  178  180  49.874,
  47  77.45  11.63  58  176  176  44.811,
  40  75.98  11.95  70  176  180  45.681,
  43  81.19  10.85  64  162  170  49.091,
  44  81.42  13.08  63  174  176  39.442,
  38  81.87   8.63  48  170  186  60.055,
  44  73.03  10.13  45  168  168  50.541,
  45  87.66  14.03  56  186  192  37.388,
  45  66.45  11.12  51  176  176  44.754,
  47  79.15  10.60  47  162  164  47.273,
  54  83.12  10.33  50  166  170  51.855,
  49  81.42   8.95  44  180  185  49.156,
  51  69.63  10.95  57  168  172  40.836,
  51  77.91  10.00  48  162  168  46.672,
  48  91.63  10.25  48  162  164  46.774,
  49  73.37  10.08  67  168  168  50.388,
  57  73.37  12.63  58  174  176  39.407,
  54  79.38  11.17  62  156  165  46.080,
```

```
52   76.32    9.63   48   164   166   45.441,
50   70.87    8.92   48   146   155   54.625,
51   67.25   11.08   48   172   172   45.118,
54   91.63   12.88   44   168   172   39.203,
51   73.71   10.47   59   186   188   45.790,
57   59.08    9.93   49   148   155   50.545,
49   76.32    9.40   56   186   188   48.673,
48   61.24   11.50   52   170   176   47.920,
52   82.78   10.50   53   170   172   47.467 };
```

Use the following IML statement to center the data.

```
fit = fit - j(31,1,1) * fit[:,];
```

Now compute the crossproduct matrices, as follows:

```
x = fit[,1:6];
y = fit[,7];
xpx = x`*x;
xpy = x`*y;
```

The following statements compute the best three regression sums of squares for each size of regressor set:

```
call maxqform( rc, maxq, xpx, xpy, 3 );
print maxq;
```

MCD Call

finds the minimum covariance determinant estimator

> **CALL MCD(** *sc, coef, dist, opt,* x**);**

The MCD call is the robust (resistent) 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).

The MCD subroutine computes the minimum covariance determinant estimator. These robust locations and covariance matrices can be used to detect multivariate outliers and leverage points. For this purpose, the 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 inputs 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.

        opt[1]=0 prints no output except error messages.
        opt[1]=1 prints most of the output.
        opt[1]=2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.
        opt[1]=3 additionally prints how many subsets result in singular linear systems.

    The default is *opt*[1]=0.

opt[2]   specifies whether the classical, initial, and final robust covariance matrices are printed. The default is *opt*[2]=0. Note that the final robust covariance matrix is always returned in *coef*.

opt[3]   specifies whether the classical, initial, and final robust correlation matrices are printed or returned:

        opt[3]=0 does not return or print.
        opt[3]=1 prints the robust correlation matrix.
        opt[3]=2 returns the final robust correlation matrix in *coef*.
        opt[3]=3 prints and returns the final robust correlation matrix.

opt[4]   specifies the quantile h used in the objective function. The default is *opt*[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 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 $N > 600$, construct up to five disjoint random subsets with sizes as equal as possible, but not to exceed 300. Inside each subset, choose $500/5 = 100$ subset combinations of n observations.

    If $N < 600$, 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	11	12	13	14	15
N_{lower}	0	0	0	0	0

If the number of cases (observations) N is smaller than N_{lower}, then all possible subsets are used; otherwise, 500 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 indicating how many subsets exist.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors.

Missing values are not permitted in x. Missing values in *opt* cause the default value to be used.

The MCD subroutine returns the following values:

sc   is a column vector containing the following scalar information:

sc[1]	the quantile h used in the objective function
sc[2]	number of subsets generated
sc[3]	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 containing the following results in its rows:

coef[1]	location of ellipsoid center
coef[2]	eigenvalues of final robust scatter matrix
coef[3:2+n]	the final robust scatter matrix for *opt*[2]=1 or *opt*[2]=3
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 containing the following results in its rows:

dist[1]	Mahalanobis distances
dist[2]	robust distances based on the final estimates
dist[3]	weights (=1 for small, =0 for large robust distances)

Example

Consider Brownlee's (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 optn[5], the MCD algorithm chooses 500 random sample subsets:

```
      /* 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];
  optn = j(8,1,.);
  optn[1]= 2;                    /* ipri */
  optn[2]= 1;                    /* pcov: print COV */
  optn[3]= 1;                    /* pcor: print CORR */

  CALL MCD(sc,xmcd,dist,optn,a);
```

The first part of the output of this program is a summary of the MCD algorithm and the final h points selected:

```
            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 -
```

**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

The second part of the output is the MCD estimators of the location, scatter matrix, and correlation matrix:

MCD Location Estimate

VAR1	VAR2	VAR3
59.5	20.833333333	87.333333333

Average of 12 Selected Points

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

Determinant = 238.07387929
Covariance Matrix of 12 Selected Points

MCD Correlation Matrix

	VAR1	VAR2	VAR3
VAR1	1	0.7674714142	0.4745347313
VAR2	0.7674714142	1	0.4192963398
VAR3	0.4745347313	0.4192963398	1

**The MCD scatter matrix is multiplied by a factor to make it
consistent when all the data come from a single Gaussian
distribution.**

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

Determinant = 397.77668436

The final output presents a table containing the classical Mahalanobis distances, the robust distances, and the weights identifying the outlying observations (that is, leverage points when explaining y with these three regressor variables):

```
        Classical Distances and Robust (Rousseeuw) Distances
                 Unsquared Mahalanobis Distance and
            Unsquared Rousseeuw Distance of Each Observation
                   Mahalanobis              Robust
           N         Distances            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.
```

MIN Function

finds the smallest element of a matrix

    MIN(*matrix1<, matrix2,..., matrix15>***)**

where *matrix* is a numeric or character matrix or literal.

The MIN function produces a single numeric value (or a character string value) that is the smallest element (lowest character string value) in all arguments. There can be as many as 15 argument matrices. The function checks for missing numeric values

and excludes them from the result. If all arguments are missing, then the machine's largest representable number is the result.

If you want to find the elementwise minimums of the corresponding elements of two matrices, use the element minimum operator (><).

For character arguments, the size of the result is the size of the largest of all arguments.

An example using the MIN function is shown below.

```
c = { 1 -123 13 56 128 -81 12 };
b=min(c);
print b;
```

```
          B

           -123
```

MOD Function

computes the modulo (remainder)

MOD(*value, divisor***)**

The inputs to the MOD function are as follows:

value   is a numeric matrix or literal giving the dividend.

divisor   is a numeric matrix or literal giving the divisor.

The MOD function is the scalar function returning the remainder of the division of elements of the first argument by elements of the second argument.

Unlike the MOD function in base SAS, the IML MOD function does not perform any numerical "fuzzing" to return an exact zero when the result would otherwise be very small. Thus the results of the IML MOD function is more similar to the MODZ function in base SAS.

An example of a valid statement follows:

```
c = { -7 14 20 -81 23 };
b=mod(c,4);
print b;
```

```
                      B

        -3          2          0         -1          3
```

MVE Call

finds the minimum volume ellipsoid estimator

CALL MVE(*sc, coef, dist, opt,* $x <, s >$ **);**

The MVE call is the robust (resistent) estimation of multivariate location and scatter, defined by minimizing the volume of an ellipsoid containing h points.

The MVE 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 inputs 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.

        opt[1]=0 prints no output except error messages.
        opt[1]=1 prints most of the output.
        opt[1]=2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.
        opt[1]=3 additionally prints how many subsets result in singular linear systems.

        The default is *opt*[1]=0.

    opt[2]   specifies whether the classical, initial, and final robust covariance matrices are printed. The default is *opt*[2]=0. Note that the final robust covariance matrix is always returned in *coef*.

    opt[3]   specifies whether the classical, initial, and final robust correlation matrices are printed or returned:

        opt[3]=0 does not return or print.
        opt[3]=1 prints the robust correlation matrix.
        opt[3]=2 returns the final robust correlation matrix in *coef*.
        opt[3]=3 prints and returns the final robust correlation matrix.

    opt[4]   specifies the quantile h used in the objective function. The default is $opt[5] = h = \left\lceil \frac{N+n+1}{2} \right\rceil$. If the value of h is specified outside the range $\frac{N}{2} + 1 \leq h \leq \frac{3N}{4} + \frac{n+1}{4}$, it is reset to the closest boundary of this region.

    opt[5]   specifies the number N_{Rep} of subset generations. This option is the same as described previously for the LMS and LTS subroutines. Due to computer time restrictions, not all subset combinations can be inspected for larger values of N and n. If *opt*[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 indicating how many subsets exist.

x refers to an $N \times n$ matrix **X** of regressors.

s refers to an $n + 1$ vector containing $n + 1$ 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 containing the observation numbers contained in s.

Missing values are not permitted in x. Missing values in *opt* cause the default value to be used.

The MVE subroutine returns the following values:

sc   is a column vector containing the following scalar information:

       sc[1]   the quantile h used in the objective function

       sc[2]   number of subsets generated

       sc[3]   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 containing the following results in its rows:

coef[1]	location of ellipsoid center
coef[2]	eigenvalues of final robust scatter matrix
coef[3:2+n]	the final robust scatter matrix for *opt*[2]=1 or *opt*[2]=3
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 containing the following results in its rows:

dist[1]	Mahalanobis distances
dist[2]	robust distances based on the final estimates
dist[3]	weights (=1 for small, =0 for large robust distances)

Example

Consider results for Brownlee's (1965) stackloss data. The three explanatory variables correspond to measurements for a plant oxidizing ammonia to nitric acid on 21 consecutive days:

- x_1 air flow to the plant

- x_2 cooling water inlet temperature

- x_3 acid concentration

The response variable y_i gives the permillage of ammonia lost (stackloss). These data are also given by Rousseeuw and Leroy (1987, p. 76).

```
        /* 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, p. 76) 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:

```
a = aa[,2:4];
optn = j(8,1,.);
optn[1]= 2;               /* ipri */
optn[2]= 1;               /* pcov: print COV */
optn[3]= 1;               /* pcor: print CORR */
optn[5]= -1;              /* nrep: use all subsets */

CALL MVE(sc,xmve,dist,optn,a);
```

The first part of the output shows the classical scatter and correlation matrix:

```
             Minimum Volume Ellipsoid (MVE) Estimation
              Consider Ellipsoids Containing 12 Cases.

                    Classical Covariance Matrix

                    VAR1                 VAR2                 VAR3

     VAR1    84.057142857       22.657142857       24.571428571
     VAR2    22.657142857        9.9904761905       6.6214285714
     VAR3    24.571428571        6.6214285714      28.714285714

                    Classical Correlation Matrix

                    VAR1                 VAR2                 VAR3

      VAR1               1        0.781852333        0.5001428749
      VAR2    0.781852333                  1        0.3909395378
      VAR3    0.5001428749      0.3909395378                  1

                          Classical Mean

                    VAR1    60.428571429
                    VAR2    21.095238095
                    VAR3    86.285714286

          There are 5985 subsets of 4 cases out of 21 cases.
                All 5985 subsets will be considered.
```

The second part of the output shows the results of the optimization (complete subset sampling):

Complete Enumeration for MVE

Subset	Singular	Best Criterion	Percent
1497	22	253.312431	25
2993	46	224.084073	50
4489	77	165.830053	75
5985	156	165.634363	100

Minimum Criterion= 165.63436284

Among 5985 subsets 156 are singular.

Observations of Best Subset

7	10	14	20

Initial MVE Location
Estimates

VAR1	58.5
VAR2	20.25
VAR3	87

Initial MVE Scatter Matrix

	VAR1	VAR2	VAR3
VAR1	34.829014749	28.413143611	62.32560534
VAR2	28.413143611	38.036950318	58.659393261
VAR3	62.32560534	58.659393261	267.63348175

The third part of the output shows the optimization results after local improvement:

Final MVE Estimates (Using Local Improvement)

Number of Points with Nonzero Weight=17

Robust MVE Location
Estimates

VAR1	56.705882353
VAR2	20.235294118
VAR3	85.529411765

```
                      Robust MVE Scatter Matrix

                  VAR1                VAR2                VAR3

  VAR1     23.470588235         7.5735294118        16.102941176
  VAR2      7.5735294118        6.3161764706         5.3676470588
  VAR3     16.102941176         5.3676470588        32.389705882


                      Eigenvalues of Robust
                         Scatter Matrix

                  VAR1      46.597431018
                  VAR2      12.155938483
                  VAR3       3.423101087


                    Robust Correlation Matrix

                  VAR1                VAR2                VAR3

  VAR1              1          0.6220269501        0.5840361335
  VAR2     0.6220269501                  1         0.375278187
  VAR3     0.5840361335        0.375278187                  1
```

The final output presents a table containing the classical Mahalanobis distances, the robust distances, and the weights identifying the outlying observations (that is leverage points when explaining y with these three regressor variables):

```
       Classical Distances and Robust (Rousseeuw) Distances
               Unsquared Mahalanobis Distance and
           Unsquared Rousseeuw Distance of Each Observation
               Mahalanobis              Robust
       N        Distances             Distances          Weight

       1          2.253603             5.528395               0
       2          2.324745             5.637357               0
       3          1.593712             4.197235               0
       4          1.271898             1.588734        1.000000
       5          0.303357             1.189335        1.000000
       6          0.772895             1.308038        1.000000
       7          1.852661             1.715924        1.000000
       8          1.852661             1.715924        1.000000
       9          1.360622             1.226680        1.000000
      10          1.745997             1.936256        1.000000
      11          1.465702             1.493509        1.000000
      12          1.841504             1.913079        1.000000
      13          1.482649             1.659943        1.000000
      14          1.778785             1.689210        1.000000
      15          1.690241             2.230109        1.000000
      16          1.291934             1.767582        1.000000
      17          2.700016             2.431021        1.000000
      18          1.503155             1.523316        1.000000
      19          1.593221             1.710165        1.000000
      20          0.807054             0.675124        1.000000
```

```
      21          2.176761          3.657281                    0

               Distribution of Robust Distances

            MinRes               1st Qu.               Median

        0.6751244996         1.5084120761         1.7159242054

             Mean                3rd Qu.               MaxRes

        2.2282960174         2.0831826658         5.6373573538

              Cutoff Value = 3.0575159206

    The cutoff value is the square root of
      the 0.975 quantile of the chi square
     distribution with 3 degrees of freedom.

  There are 4 points with large robust distances receiving
  zero weights. These may include boundary cases.
  Only points whose robust distances are substantially larger
  than the cutoff value should be considered outliers.
```

NAME Function

lists the names of arguments

> **NAME(** *arguments***);**

where *arguments* are the names of existing matrices.

The NAME function returns the names of the arguments in a column vector. In the following example, **N** is a 3×1 character matrix of element size 8 containing the character values A, B, and C:

```
n=name(a,b,c);
```

The main use of the NAME function is with macros when you want to use an argument for both its name and its value.

NCOL Function

finds the number of columns of a matrix

> **NCOL(** *matrix***)**

where *matrix* is a numeric or character matrix.

The NCOL function returns a single numeric value that is the number of columns in *matrix*. If the matrix has not been given a value, the NCOL function returns a value of 0.

For example, to let B contain the number of columns of matrix **S**, use the statement

```
b=ncol(s);
```

NLENG Function

finds the size of an element

> **NLENG(** *matrix***)**

where *matrix* is a numeric or character matrix.

The NLENG function returns a single numeric value that is the size in bytes of each element in *matrix*. All matrix elements have the same size. If the matrix does not have a value, then the NLENG function returns a value of 0. This function is different from the LENGTH function, which returns the size of each element of a character matrix, omitting the trailing blanks.

The following statement returns the value 7:

```
a=nleng({"ab " "ijklm  ",
         "x"   " "});
```

Nonlinear Optimization and Related Subroutines

Table 20.1. Nonlinear Optimization and Related Subroutines

Optimization Subroutines
Conjugate Gradient Optimization Method   **CALL NLPCG(** *rc, xr, "fun", x0* <, *opt, blc, tc, par, "ptit", "grd"*>**);**
Double Dogleg Optimization Method   **CALL NLPDD(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd"*>**);**
Nelder-Mead Simplex Optimization Method   **CALL NLPNMS(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "nlc"*>**);**
Newton-Raphson Optimization Method   **CALL NLPNRA(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd", "hes"*>**);**
Newton-Raphson Ridge Optimization Method   **CALL NLPNRR(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd", "hes"*>**);**
(Dual) Quasi-Newton Optimization Method   **CALL NLPQN(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd", "nlc", "jacnlc"*>**);**
Quadratic Optimization Method   **CALL NLPQUA(** *rc, xr, quad, x0* <,*opt, blc, tc, par, "ptit", lin*>**);**
Trust-Region Optimization Method   **CALL NLPTR(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd", "hes"*>**);**

Least-Squares Subroutines

Hybrid Quasi-Newton Least-Squares Methods
   CALL NLPHQN(*rc, xr, "fun", x0, opt* <*,blc, tc, par, "ptit", "jac"*>**);**

Levenberg-Marquardt Least-Squares Method
   CALL NLPLM(*rc, xr, "fun", x0, opt* <*,blc, tc, par, "ptit", "jac"*>**);**

Supplementary Subroutines

Approximate Derivatives by Finite Differences
   CALL NLPFDD(*f, g, h, "fun", x0* <*,par, "grd"*>**);**

Feasible Point Subject to Constraints
   CALL NLPFEA(*xr, x0, blc* <*,par*>**);**

Note: The names of the optional arguments can be used as keywords. For example, the following statements are equivalent:

```
call nlpnrr(rc,xr,"fun",x0,,,ter,,,"grad");
call nlpnrr(rc,xr,"fun",x0) tc=ter grd="grad";
```

All the optimization subroutines require at least two input arguments.

- The NLPQUA subroutine requires the *quad* matrix argument, which specifies the symmetric matrix **G** of the quadratic problem. The input can be dense or sparse. Other optimization subroutines require the *fun* module argument, which specifies an IML module that defines the objective function or functions. For least-squares subroutines, the FUN module must return a column vector of length m that corresponds to the values of the m functions $f_1(x), \ldots, f_m(x)$, each evaluated at the point $x = (x_1, \ldots, x_n)$. For other subroutines, the FUN module must return the value of the objective function $f = f(x)$ evaluated at the point x.

- The argument *x0* specifies a row vector that defines the number of parameters n. If *x0* is a feasible point, it represents a starting point for the iterative optimization process. Otherwise, a linear programming algorithm is called at the start of each optimization subroutine to replace the input *x0* by a feasible starting point.

The other arguments that can be used as input are described in the following list. As indicated in Table 20.1, not all input arguments apply to each subroutine.

Note that you can specify optional arguments with the *keyword=argument* syntax.

- The *opt* argument indicates an options vector that specifies details of the optimization process, such as particular updating techniques and whether the objective function is to be maximized instead of minimized. See "Options Vector" for details.

- The *blc* argument specifies a constraint matrix that defines lower and upper bounds for the n parameters as well as general linear equality and inequality constraints. For details, see "Parameter Constraints."

- The *tc* argument specifies a vector of thresholds corresponding to the termination criteria tested in each iteration. See "Termination Criteria" for details.

- The *par* argument specifies a vector of control parameters that can be used to modify the algorithms if the default settings do not complete the optimization process successfully. For details, see "Control Parameters Vector."

- The *"ptit"* module argument specifies an IML module that replaces the subroutine used to print the iteration history and test the termination criteria. If the *"ptit"* module is specified, the matrix specified by the *tc* argument has no effect. See "Termination Criteria" for details.

- The *"grd"* module argument specifies an IML module that computes the gradient vector, $g = \nabla f$, at a given input point x. See "Objective Function and Derivatives" for details.

- The *"hes"* module argument specifies an IML module that computes the $n \times n$ Hessian matrix, $\mathbf{G} = \nabla^2 f$, at a given input point x. See "Objective Function and Derivatives" for details.

- The *"jac"* module argument specifies an IML module that computes the $m \times n$ Jacobian matrix, $\mathbf{J} = (\nabla f_i)$, of the m least-squares functions at a given input point x. See "Objective Function and Derivatives" for details.

- The *"nlc"* module argument specifies an IML module that allows the computation of general equality and inequality constraints. This is the method by which nonlinear constraints must be specified. For details, see "Parameter Constraints."

- The *"jacnlc"* module argument specifies an IML module that computes the Jacobian matrix of first-order derivatives of the equality and inequality constraints specified by the NLC module. For details, see "Parameter Constraints".

- The *lin* argument specifies the linear part of the quadratic optimization problem. See "NLPQUA Call" for details.

The modules that can be used as input arguments for the subroutines (*"fun", "grd", "hes", "jac", "ptit", "nlc",* and *"jacnlc"*) allow only a single input parameter $x = (x_1, \ldots, x_n)$. You can provide more input parameters for these modules by using the GLOBAL clause. See "Using the GLOBAL Clause" for an example.

All the optimization subroutines return the following results:

- The scalar return code *rc* indicates the reason for the termination of the optimization process. A return code $rc > 0$ indicates successful termination corresponding to one of the specified termination criteria. A return code $rc < 0$

indicates unsuccessful termination, that is, that the result *xr* is unreliable. See "Definition of Return Codes" for more details.

- The row vector *xr*, which has length n, the number of parameters, contains the optimal point when $rc > 0$.

NLPCG Call

nonlinear optimization by conjugate gradient method

CALL NLPCG(*rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPCG subroutine requires function and gradient calls; it does not need second-order derivatives. The gradient vector contains the first derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$g(x) = \nabla f(x) = \left(\frac{\partial f}{\partial x_j} \right)$$

If you do not specify an IML module with the *"grd"* argument, the first-order derivatives are approximated by finite difference formulas using only function calls. The NLPCG algorithm can require many function and gradient calls, but it requires less memory than other subroutines for unconstrained optimization. In general, many iterations are needed to obtain a precise solution, but each iteration is computationally inexpensive. You can specify one of four update formulas for generating the conjugate directions with the fourth element of the *opt* input argument.

Value of *opt*[4]	Update Method
1	Automatic restart method of Powell (1977) and Beale (1972). This is the default.
2	Fletcher-Reeves update (Fletcher 1987)
3	Polak-Ribiere update (Fletcher 1987)
4	Conjugate-descent update of Fletcher (1987)

The NLPCG subroutine is useful for optimization problems with large n. For the unconstrained or boundary constrained case, the NLPCG method needs only order n bytes of working memory, whereas the other optimization methods require order n^2 bytes of working memory. During n successive iterations, uninterrupted by restarts or changes in the working set, the conjugate gradient algorithm computes a cycle of n conjugate search directions. In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step size α that satisfies the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit for the step size. You can specify other line-search algorithms with the fifth element of the *opt* argument.

For an example of the NLPCG subroutine, see "Constrained Betts Function".

NLPDD Call

nonlinear optimization by double dogleg method

    CALL NLPDD(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The double dogleg optimization method combines the ideas of the quasi-Newton and trust-region methods. In each iteration, the algorithm computes the step, $s^{(k)}$, as a linear combination of the steepest descent or ascent search direction, $s_1^{(k)}$, and a quasi-Newton search direction, $s_2^{(k)}$, as follows:

$$s^{(k)} = \alpha_1 s_1^{(k)} + \alpha_2 s_2^{(k)}.$$

The step $s^{(k)}$ must remain within a specified trust-region radius (refer to Fletcher 1987). Hence, the NLPDD subroutine uses the dual quasi-Newton update but does not perform a line search. You can specify one of two update formulas with the fourth element of the *opt* input argument.

Value of *opt*[**4**]	Update Method
1	Dual BFGS update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual DFP update of the Cholesky factor of the Hessian matrix

The double dogleg optimization technique works well for medium to moderately large optimization problems, in which the objective function and the gradient are much faster to compute than the Hessian. The implementation is based on Dennis and Mei (1979) and Gay (1983), but it is extended for boundary and linear constraints. The NLPDD subroutine generally needs more iterations than the techniques that require second-order derivatives (NLPTR, NLPNRA, and NLPNRR), but each of the NLPDD iterations is computationally inexpensive. Furthermore, the NLPDD subroutine needs only gradient calls to update the Cholesky factor of an approximate Hessian.

In addition to the standard iteration history, the NLPDD routine prints the following information:

- The heading *lambda* refers to the parameter λ of the double dogleg step. A value of 0 corresponds to the full (quasi-) Newton step.

- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero.

The following statements invoke the NLPDD subroutine to solve the constrained Betts optimization problem (see "Constrained Betts Function").

```
    start F_BETTS(x);
       f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
       return(f);
    finish F_BETTS;

    con = {  2. -50.  .   .,
            50.  50.  .   .,
            10.  -1. 1. 10.};
    x = {-1. -1.};
    optn = {0 1};
    call nlpdd(rc,xres,"F_BETTS",x,optn,con);
```

The preceding statements produce the following iteration history.

                      Double Dogleg Optimization

          Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

                         Without Parameter Scaling
                  Gradient Computed by Finite Differences

                  Parameter Estimates   2
                  Lower Bounds   2
                  Upper Bounds   2
                  Linear Constraints   1

                        Optimization Start

Active Constraints   0 Objective Function   -98.5376
Max Abs Gradient Element   2 Radius   1

                    Function   Active   Objective
   Iter   Restarts   Calls   Constraints   Function

     1   0   2   0   -99.54678
     2   0   3   0   -99.59120
     3   0   5   0   -99.90252
     4   0   6   1   -99.96000
     5   0   7   1   -99.96000
     6   0   8   1   -99.96000

              Objective   Max Abs   Slope of
               Function   Gradient   Search
   Iter   Change   Element   Lambda   Direction

     1   1.0092   0.1346   6.012   -1.805
     2   0.0444   0.1279   0   -0.0228
     3   0.3113   0.0624   0   -0.209
     4   0.0575   0.00432   0   -0.0975
     5   4.66E-6   0.000079   0   -458E-8
     6   1.559E-9   0   0   -16E-10
```

```
                        Optimization Results

Iterations                        6  Function Calls               9
Gradient Calls                    8  Active Constraints           1
Objective Function           -99.96  Max Abs Gradient Element     0
Slope of Search Direction -1.56621E-9  Radius                     1


    GCONV convergence criterion satisfied.
```

NLPFDD Call

approximates derivatives by finite differences method

CALL NLPFDD(*f, g, h, "fun", x0,* <*,par, "grd"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP sub-routines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPFDD subroutine can be used for the following tasks:

- If the module *"fun"* returns a scalar, the NLPFDD subroutine computes the function value *f*, the gradient vector *g*, and the Hessian matrix *h*, all evaluated at the point *x0*.

- If the module *"fun"* returns a column vector of m function values, the sub-routine assumes that a least-squares function is specified, and it computes the function vector *f*, the Jacobian matrix **J**, and the crossproduct of the Jacobian matrix $\mathbf{J}'\mathbf{J}$ at the point *x0*. Note that in this case, you must set the first element of the *par* argument to m.

If any of the results cannot be computed, the subroutine returns a missing value for that result.

You can specify the following input arguments with the NLPFDD subroutine:

- The *"fun"* argument refers to an IML module that returns either a scalar value or a column vector of length m. This module returns the value of the objective function or, for least-squares problems, the values of the m functions that comprise the objective function.

- The *x0* argument is a vector of length n that defines the point at which the functions and derivatives should be computed.

- The *par* argument is a vector that defines options and control parameters. Note that the *par* argument in the NLPFDD call is different from the one used in the optimization subroutines.

- The *"grd"* argument is optional and refers to an IML module that returns a vector defining the gradient of the function at *x0*. If the *"fun"* argument returns a vector of values instead of a scalar, the *"grd"* argument is ignored.

If the *"fun"* module returns a scalar, the subroutine returns the following values:

- *f* is the value of the function at the point *x0*.

- *g* is a vector containing the value of the gradient at the point *x0*. If you specify the *"grd"* argument, the gradient is computed from that module. Otherwise, the approximate gradient is computed by a finite difference approximation using calls of the function module in a neighborhood of *x0*.

- *h* is a matrix containing a finite difference approximation of the value of the Hessian at the point *x0*. If you specify the *"grd"* argument, the Hessian is computed by calls of that module in a neighborhood of *x0*. Otherwise, it is computed by calls of the function module in a neighborhood of *x0*.

If the *"fun"* module returns a vector, the subroutine returns the following values:

- *f* is a vector containing the values of the m functions comprising the objective function at the point *x0*.

- *g* is the $m \times n$ Jacobian matrix $\mathbf{J}$, which contains the first-order derivatives of the functions with respect to the parameters, evaluated at *x0*. It is computed by finite difference approximations in a neighborhood of *x0*.

- *h* is the $n \times n$ crossproduct of the Jacobian matrix, $\mathbf{J}^T\mathbf{J}$. It is computed by finite difference approximations in a neighborhood of *x0*.

The *par* argument is a vector of length 3.

- *par*[1] corresponds to the *opt*[1] argument in the optimization subroutines. This argument is relevant only to least-squares optimization methods, in which case it specifies the number of functions returned by the module *"fun"*. If *par*[1] is missing or is smaller than 1, it is set to 1.

- *par*[2] corresponds to the *opt*[8] argument in the optimization subroutines. It determines what type of approximation is to be used and how the finite difference interval, h, is to be computed. See "Finite Difference Approximations of Derivatives" for details.

- *par*[3] corresponds to the *par*[8] argument in the optimization subroutines. It specifies the number of accurate digits in evaluating the objective function. The default is $-\log_{10}(\epsilon)$, where ϵ is the machine precision.

If you specify a missing value in the *par* argument, the default value is used.

The NLPFDD subroutine is particularly useful for checking your analytical derivative specifications of the *"grd"*, *"hes"*, and *"jac"* modules. You can compare the results of the modules with the finite difference approximations of the derivatives of *f* at the point *x0* to verify your specifications.

In the unconstrained Rosenbrock problem (see "Unconstrained Rosenbrock Function"), the objective function is

$$f(x) = 50(x_2 - x_1^2)^2 + \frac{1}{2}(1 - x_1)^2$$

Then the gradient and the Hessian, evaluated at the point $x = (2, 7)$, are

$$g' = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 200x_1^3 - 200x_1x_2 + x_1 - 1 \\ -100x_1^2 + 100x_2 \end{bmatrix} = \begin{bmatrix} -1199 \\ 300 \end{bmatrix}$$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 600x_1^2 - 200x_2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix} = \begin{bmatrix} 1001 & -400 \\ -400 & 100 \end{bmatrix}$$

The following statements define the Rosenbrock function and use the NLPFDD call
to compute the gradient and the Hessian.

```
start F_ROSEN(x);
   y1 = 10. * (x[2] - x[1] * x[1]);
   y2 = 1. - x[1];
   f  = .5 * (y1 * y1 + y2 * y2);
   return(f);
finish F_ROSEN;
x = {2 7};
CALL NLPFDD(crit,grad,hess,"F_ROSEN",x);
print grad;
print hess;
```

                                        GRAD

                               -1199 300.00001

                                        HESS

                           1000.9998 -400.0018
                           -400.0018 99.999993

If the Rosenbrock problem is considered from a least-squares perspective, the two
functions are

$$f_1(x) = 10(x_2 - x_1^2)$$
$$f_2(x) = 1 - x_1$$

Then the Jacobian and the crossproduct of the Jacobian, evaluated at the point $x = (2, 7)$, are

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -40 & 10 \\ -1 & 0 \end{bmatrix}$$

$$\mathbf{J}^T\mathbf{J} = \begin{bmatrix} 400x_1^2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix} = \begin{bmatrix} 1601 & -400 \\ -400 & 100 \end{bmatrix}$$

The following statements define the Rosenbrock problem in a least-squares framework and use the NLPFDD call to compute the Jacobian and the crossproduct matrix. Since the value of the PARMS variable, which is used for the *par* argument, is 2, the NLPFDD subroutine allocates memory for a least-squares problem with two functions, $f_1(x)$ and $f_2(x)$.

```
start F_ROSEN(x);
   y = j(2,1,0.);
   y[1] = 10. * (x[2] - x[1] * x[1]);
   y[2] = 1. - x[1];
   return(y);
finish F_ROSEN;
x      = {2 7};
parms = 2;
CALL NLPFDD(fun,jac,crpj,"F_ROSEN",x,parms);
print jac;
print crpj;
```

The finite difference approximations for Jacobian follow.

```
              JAC

        -40          10
         -1           0



            CRPJ

      1601        -400
      -400         100
```

NLPFEA Call

computes feasible points subject to constraints

> **CALL NLPFEA(** *xr, x0, blc* <,*par*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPFEA subroutine tries to compute a point that is feasible subject to a set of boundary and linear constraints. You can specify boundary and linear constraints that define an empty feasible region, in which case the subroutine will return missing values.

You can specify the following input arguments with the NLPFEA subroutine:

- *x0* is a row vector defining the coordinates of a point that is not necessarily feasible for a set of linear and boundary constraints.

- *blc* is an $m \times n$ matrix defining a set of m boundary and linear constraints. See "Parameter Constraints" for details.

- *par* is a vector of length two. The argument is different from the one used in the optimization subroutines. The first element sets the LCEPS parameter, which controls how precisely the returned point must satisfy the constraints. The second element sets the LCSING parameter, which specifies the criterion for deciding when constraints are considered linearly dependent. For details, see "Control Parameters Vector."

The NLPFEA subroutine returns the *xr* argument. The result is a vector containing either the n coordinates of a feasible point, which indicates that the subroutine was successful, or missing values, which indicates that the subroutine could not find a feasible point.

The following statements call the NLPFEA subroutine with the constraints from the Betts problem (see "Constrained Betts Function") and an initial infeasible point $x_0 = (-17, -61)$. The subroutine returns the feasible point $(2, -50)$ as the vector XFEAS.

```
con = {  2. -50.  .   .,
        50.  50.  .   .,
        10.  -1. 1. 10.};
x = {-17. -61};
call nlpfea(xfeas,x,con);
```

NLPHQN Call

calculates hybrid quasi-Newton least squares

> **CALL NLPHQN(** *rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "jac"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPHQN subroutine uses one of the Fletcher and Xu (1987) hybrid quasi-Newton methods. Refer also to Al-Baali and Fletcher (1985, 1986). In each iteration, the subroutine uses a criterion to decide whether a Gauss-Newton or a dual quasi-Newton search direction is appropriate. You can choose one of three criteria (HY1, HY2, or HY3) proposed by Fletcher and Xu (1987) with the sixth element of the *opt* vector. The default is HY2. The subroutine computes the crossproduct Jacobian (for the Gauss-Newton step), updates the Cholesky factor of an approximate Hessian (for the quasi-Newton step), and performs a line search to compute an approximate minimum along the search direction. The default line-search technique used by the NLPHQN method is designed for least-squares problems (refer to Lindström and Wedin 1984, and Al-Baali and Fletcher 1986), but you can specify a different line-search algorithm with the fifth element of the *opt* argument. See "Options Vector" for details.

You can specify two update formulas with the fourth element of the *opt* argument as indicated in the following table.

Value of $opt[4]$	Update Method
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.

The NLPHQN subroutine needs approximately the same amount of working memory as the NLPLM subroutine, and in most applications, the latter seems to be superior. Hence, the NLPHQN method is recommended only when the NLPLM method encounters problems.

Note: In least-squares subroutines, you must set the first element of the *opt* vector to m, the number of functions.

In addition to the standard iteration history, the NLPHQN subroutine prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that, on the basis of the Fletcher and Xu (1987) criterion, the subroutine used a Gauss-Newton search direction instead of a quasi-Newton search direction.

- The heading *alpha* is the step size, α, computed with the line-search algorithm.

- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The following statements use the NLPHQN call to solve the unconstrained Rosenbrock problem (see "Unconstrained Rosenbrock Function").

```
title 'Test of NLPHQN subroutine: No Derivatives';
start F_ROSEN(x);
   y = j(1,2,0.);
   y[1] = 10. * (x[2] - x[1] * x[1]);
   y[2] = 1. - x[1];
   return(y);
finish F_ROSEN;

x = {-1.2 1.};
optn = {2 2};
call nlphqn(rc,xr,"F_ROSEN",x,optn);
```

The iteration history for the subroutine follows.

```
                     Optimization Start
                     Parameter Estimates
                                                    Gradient
                                                    Objective
           N Parameter              Estimate         Function

           1 X1                   -1.200000       -107.799999
           2 X2                    1.000000        -44.000000


              Value of Objective Function = 12.1



              Hybrid Quasi-Newton LS Minimization


     Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
             Version HY2 of Fletcher & Xu (1987)
           Gradient Computed by Finite Differences
          CRP Jacobian Computed by Finite Differences


           Parameter Estimates                2
           Functions (Observations)           2

                     Optimization Start

   Active Constraints                     0  Objective Function     12.1
   Max Abs Gradient Element       107.7999987
```

Iter	Restarts	Function Calls	Active Constraints	Objective Function
1	0	3	0	7.22423
2*	0	5	0	0.97090
3*	0	7	0	0.81911
4	0	9	0	0.69103
5	0	19	0	0.47345
6*	0	21	0	0.35906
7*	0	22	0	0.23342
8*	0	24	0	0.14799
9*	0	26	0	0.00948
10*	0	28	0	1.98834E-6
11*	0	30	0	7.0768E-10
12*	0	32	0	2.0246E-21

Iter	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
1	4.8758	56.9322	0.0616	-628.8
2*	6.2533	2.3017	0.266	-14.448
3*	0.1518	3.7839	0.119	-1.942
4	0.1281	5.5103	2.000	-0.144
5	0.2176	8.8638	11.854	-0.194
6*	0.1144	9.8734	0.253	-0.947
7*	0.1256	10.1490	0.398	-0.718
8*	0.0854	11.6248	1.346	-0.467
9*	0.1385	2.6275	1.443	-0.296
10*	0.00947	0.00609	0.938	-0.0190
11*	1.988E-6	0.000748	1.003	-398E-8

```
12*      7.08E-10   1.82E-10    1.000     -14E-10
```

                         Optimization Results

```
Iterations                        12  Function Calls                    33
Jacobian Calls                    13  Gradient Calls                    19
Active Constraints                 0  Objective Function      2.024612E-21
Max Abs Gradient Element  1.816863E-10  Slope of Search Direction  -1.415366E-9
```

    ABSGCONV convergence criterion satisfied.

                         Optimization Results
                         Parameter Estimates
                                         Gradient
                                         Objective
          N Parameter   Estimate   Function

          1 X1   1.000000   1.816863E-10
          2 X2   1.000000 -1.22069E-10

      Value of Objective Function = 2.024612E-21

NLPLM Call

calculates Levenberg-Marquardt least squares

   CALL NLPLM(*rc, xr, "fun", x0, opt, blc, tc, par, "ptit", "jac"*>**)**;

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP sub-routines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPLM subroutine uses the Levenberg-Marquardt method, which is an efficient modification of the trust-region method for nonlinear least-squares problems and is implemented as in Moré (1978). This is the recommended algorithm for small to medium least-squares problems. Large least-squares problems can often be processed more efficiently with other subroutines, such as the NLPCG and NLPQN methods. In each iteration, the NLPLM subroutine solves a quadratically-constrained quadratic minimization problem that restricts the step to the boundary or interior of an n-dimensional elliptical trust region.

The m functions $f_1(x), \ldots, f_m(x)$ are computed by the module specified with the *"fun"* module argument. The $m \times n$ Jacobian matrix, **J**, contains the first-order derivatives of the m functions with respect to the n parameters, as follows:

$$\mathbf{J}(x) = (\nabla f_1, \ldots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

You can specify **J** with the *"jac"* module argument; otherwise, the subroutine will compute it with finite difference approximations. In each iteration, the subroutine

computes the crossproduct of the Jacobian matrix, $\mathbf{J}^T\mathbf{J}$, to be used as an approximate Hessian.

Note: In least-squares subroutines, you must set the first element of the *opt* vector to m, the number of functions.

In addition to the standard iteration history, the NLPLM subroutine also prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that the computed Hessian approximation was singular and had to be ridged with a positive value.

- The heading *lambda* represents the Lagrange multiplier, λ. This has a value of zero when the optimum of the quadratic function approximation is inside the trust region, in which case a trust-region-scaled Newton step is performed. It is greater than zero when the optimum is at the boundary of the trust region, in which case the scaled Newton step is too long to fit in the trust region and a quadratically-constrained optimization is done. Large values indicate optimization difficulties, and as in Gay (1983), a negative value indicates the special case of an indefinite Hessian matrix.

- The heading *rho* refers to ρ, the ratio between the achieved and predicted difference in function values. Values that are much smaller than one indicate optimization difficulties. Values close to or larger than one indicate that the trust region radius can be increased.

The iteration history for the solution of the unconstrained Rosenbrock problem follows. See the section Unconstrained Rosenbrock Function for the statements that generate this output.

```
                      Optimization Start
                      Parameter Estimates


                                        Gradient
                                        Objective
        N Parameter          Estimate    Function

        1 X1               -1.200000   -107.799999
        2 X2                1.000000    -44.000000


            Value of Objective Function = 12.1



            Levenberg-Marquardt Optimization

         Scaling Update of More (1978)
         Gradient Computed by Finite Differences
         CRP Jacobian Computed by Finite Differences

            Parameter Estimates                     2
            Functions (Observations)                2
```

Optimization Start

Active Constraints	0	Objective Function	12.1
Max Abs Gradient Element	107.7999987	Radius	2626.5613171

Iter	Restarts	Function Calls	Active Constraints	Objective Function
1	0	4	0	2.18185
2	0	6	0	1.59370
3	0	7	0	1.32848
4	0	8	0	1.03891
5	0	9	0	0.78943
6	0	10	0	0.58838
7	0	11	0	0.34224
8	0	12	0	0.19630
9	0	13	0	0.11626
10	0	14	0	0.0000396
11	0	15	0	2.4652E-30

Iter	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	9.9181	17.4704	0.00804	0.964
2	0.5881	3.7015	0.0190	0.988
3	0.2652	7.0843	0.00830	0.678
4	0.2896	6.3092	0.00753	0.593
5	0.2495	7.2617	0.00634	0.486
6	0.2011	7.8837	0.00462	0.393
7	0.2461	6.6815	0.00307	0.524
8	0.1459	8.3857	0.00147	0.469
9	0.0800	9.3086	0.00016	0.409
10	0.1162	0.1781	0	1.000
11	0.000040	4.44E-14	0	1.000

Optimization Results

Iterations	11	Function Calls	16
Jacobian Calls	12	Active Constraints	0
Objective Function	2.46519E-30	Max Abs Gradient Element	4.440892E-14
Lambda	0	Actual Over Pred Change	1
Radius	0.0178062912		

ABSGCONV convergence criterion satisfied.

```
              Optimization Results
              Parameter Estimates

                                     Gradient
                                     Objective
    N Parameter          Estimate    Function

    1 X1                 1.000000   -4.44089E-14
    2 X2                 1.000000    2.220446E-14

 Value of Objective Function = 2.46519E-30
```

NLPNMS Call

nonlinear optimization by Nelder-Mead simplex method

CALL NLPNMS(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "nlc"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The Nelder-Mead simplex method is one of the subroutines that can solve optimization problems with nonlinear constraints. It does not use any derivatives, and it does not assume that the objective function has continuous derivatives. However, the objective function must be continuous. The NLPNMS technique uses a large number of function calls, and it may be unable to generate precise results when $n > 40$.

The NLPNMS subroutine uses the following simplex algorithms:

- For unconstrained or only boundary-constrained problems, the original Nelder-Mead simplex algorithm is implemented and extended to boundary constraints. This algorithm does not compute the objective for infeasible points, and it is invoked if the *"nlc"* module argument is not specified and the *blc* argument contains at most two rows (corresponding to lower and upper bounds).

- For linearly or nonlinearly constrained problems, a slightly modified version of Powell's (1992) Constrained Optimization BY Linear Approximations (COBYLA) implementation is used. This algorithm is invoked if the *"nlc"* module argument is specified or if at least one linear constraint is specified with the *blc* argument.

The original Nelder-Mead algorithm cannot be used for general linear or nonlinear constraints, but in the unconstrained or boundary-constrained cases, it can be faster. It changes the shape of the simplex by adapting the nonlinearities of the objective function; this contributes to an increased speed of convergence.

Powell's COBYLA Algorithm

Powell's COBYLA algorithm is a sequential trust-region algorithm that tries to maintain a regularly-shaped simplex throughout the iterations. The algorithm uses a monotone-decreasing radius, ρ, of a spheric trust region. The modification implemented in the NLPNMS call permits an increase of the trust-region radius ρ in special situations. A sequence of iterations is performed with a constant trust-region radius ρ until the computed function reduction is much less than the predicted reduction. Then, the trust-region radius ρ is reduced. The trust-region radius is increased only if the computed function reduction is relatively close to the predicted reduction and if the simplex is well-shaped. The start radius, ρ_{beg}, can be specified with the second element of the *par* argument, and the final radius, ρ_{end}, can be specified with the ninth element of the *tc* argument. Convergence to small values of ρ_{end}, or high-precision convergence, may require many calls of the function and constraint modules and may result in numerical problems. The main reasons for the slow convergence of the COBYLA algorithm are as follows:

- Linear approximations of the objective and constraint functions are used locally.
- Maintaining the regularly-shaped simplex and not adapting its shape to non-linearities yields very small simplexes for highly nonlinear functions, such as fourth-order polynomials.

To allocate memory for the vector returned by the *"nlc"* module argument, you must specify the total number of nonlinear constraints with the tenth element of the *opt* argument. If any of the constraints are equality constraints, the number of equality constraints must be specified by the eleventh element of the *opt* argument. See "Parameter Constraints" for details.

For more information on the special sets of termination criteria used by the NLPNMS algorithms, see "Termination Criteria."

In addition to the standard iteration history, the NLPNMS subroutine prints the following information. For unconstrained or boundary-constrained problems, the subroutine also prints

- *difcrit*, which, in this subroutine, refers to the difference between the largest and smallest function values of the $n + 1$ simplex vertices
- *std*, which is the standard deviation of the function values of the simplex vertices
- *deltax*, which is the vertex length of a restarted simplex. If there are convergence problems, the algorithm restarts the iteration process with a simplex of smaller vertex length.
- *size*, which is the average L_1 distance of the simplex vertex with the smallest function value to the other simplex vertices

For linearly and nonlinearly constrained problems, the subroutine prints the following:

- *conmax* is the maximum constraint violation.

- *meritf* is the value of the merit function, Φ.

- *difmerit* is the difference between adjacent values of the merit function.

- *ρ* is the trust-region radius.

The following code uses the NLPNMS call to solve the Rosen-Suzuki problem (see "Rosen-Suzuki Problem"), which has three nonlinear constraints:

```
start F_HS43(x);
   f = x*x` + x[3]*x[3] - 5*(x[1] + x[2]) - 21*x[3] + 7*x[4];
   return(f);
finish F_HS43;
start C_HS43(x);
   c = j(3,1,0.);
   c[1] = 8 - x*x` - x[1] + x[2] - x[3] + x[4];
   c[2] = 10 - x*x` - x[2]*x[2] - x[4]*x[4] + x[1] + x[4];
   c[3] = 5 - 2.*x[1]*x[1] - x[2]*x[2] - x[3]*x[3]
            - 2.*x[1] + x[2] + x[4];
   return(c);
finish C_HS43;
x = j(1,4,1);
optn= j(1,11,.); optn[2]= 3; optn[10]= 3; optn[11]=0;
call nlpnms(rc,xres,"F_HS43",x,optn,,,,,"C_HS43");
```

Part of the output produced by the preceding code follows.

```
            Optimization Start
            Parameter Estimates
      N Parameter          Estimate

      1 X1                 1.000000
      2 X2                 1.000000
      3 X3                 1.000000
      4 X4                 1.000000

   Value of Objective Function = -19



            Values of Nonlinear Constraints

      Constraint          Residual

      [          1 ]       4.0000
      [          2 ]       6.0000
      [          3 ]       1.0000


      Nelder-Mead Simplex Optimization

   COBYLA Algorithm by M.J.D. Powell (1992)
```

```
Minimum Iterations                                         0
Maximum Iterations                                      1000
Maximum Function Calls                                  3000
Iterations Reducing Constraint Violation                   0
ABSFCONV Function Criterion                                0
FCONV Function Criterion                        2.220446E-16
FCONV2 Function Criterion                                1E-6
FSIZE Parameter                                            0
ABSXCONV Parameter Change Criterion                   0.0001
XCONV Parameter Change Criterion                           0
XSIZE Parameter                                            0
ABSCONV Function Criterion                      -1.34078E154
Initial Simplex Size (INSTEP)                            0.5
Singularity Tolerance (SINGULAR)                         1E-8
        Nelder-Mead Simplex Optimization
```

```
   COBYLA Algorithm by M.J.D. Powell (1992)


            Parameter Estimates             4
            Nonlinear Constraints           3


            Optimization Start

Objective Function   -29.5  Maximum Constraint Violation   4.5
```

Iter	Restarts	Function Calls	Objective Function	Maximum Constraint Violation
1	0	12	-52.80342	4.3411
2	0	17	-39.51475	0.0227
3	0	53	-44.02098	0.00949
4	0	62	-44.00214	0.000833
5	0	72	-44.00009	0.000033
6	0	79	-44.00000	1.783E-6
7	0	90	-44.00000	1.363E-7
8	0	94	-44.00000	1.543E-8

Iter	Merit Function	Merit Function Change	Between Actual and Predicted Change
1	-42.3031	12.803	1.000
2	-39.3797	-2.923	0.250
3	-43.9727	4.593	0.0625
4	-43.9977	0.0249	0.0156
5	-43.9999	0.00226	0.0039
6	-44.0000	0.00007	0.0010
7	-44.0000	1.74E-6	0.0002
8	-44.0000	5.33E-7	0.0001

```
                                     Optimization Results

Iterations                         8  Function Calls                  95
Restarts                           0  Objective Function  -44.00000003
Maximum Constraint Violation  1.543059E-8  Merit Function      -43.99999999
Actual Over Pred Change        0.0001


   ABSXCONV convergence criterion satisfied.

WARNING: The point x is feasible only at the LCEPSILON= 1E-7 range.


             Optimization Results
             Parameter Estimates
        N Parameter          Estimate

        1 X1            -0.000034167
        2 X2                1.000004
        3 X3                2.000023
        4 X4               -0.999971

 Value of Objective Function = -44.00000003




             Values of Nonlinear Constraints

         Constraint          Residual

         [          1 ]    -1.54E-8    *?*
         [          2 ]      1.0000
         [          3 ]    -1.5E-8     *?*
```

NLPNRA Call

nonlinear optimization by Newton-Raphson method

CALL NLPNRA(*rc, xr, "fun", x0* <*,opt, blc, tc, par, "ptit", "grd", "hes"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPNRA algorithm uses a pure Newton step at each iteration when both the Hessian is positive definite and the Newton step successfully reduces the value of the objective function. Otherwise, it performs a combination of ridging and line-search to compute successful steps. If the Hessian is not positive definite, a multiple of the identity matrix is added to the Hessian matrix to make it positive definite (refer to Eskow & Schnabel 1991).

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix

$\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$, and it requires continuous first- and second-order derivatives of the objective function inside the feasible region. If second-order derivatives are computed efficiently and precisely, the NLPNRA method does not need many function, gradient, and Hessian calls, and it may perform well for medium to large problems.

Note that using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and may contain significant rounding errors. If you use the *"grd"* input argument to specify a module that computes first-order derivatives analytically, you can reduce drastically the computation time for numerical second-order derivatives. The computation of the finite difference approximation for the Hessian matrix generally uses only n calls of the module that specifies the gradient.

In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation. You can specify other line-search algorithms with the fifth element of the *opt* argument. See "Options Vector" for details.

In unconstrained and boundary constrained cases, the NLPNRA algorithm can take advantage of diagonal or sparse Hessian matrices that are specified by the input argument *"hes"*. To use sparse Hessian storage, the value of the ninth element of the *opt* argument must specify the number of nonzero Hessian elements returned by the Hessian module. See "Objective Function and Derivatives" for more details.

In addition to the standard iteration history, the NLPNRA subroutine prints the following information:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The following statements invoke the NLPNRA subroutine to solve the constrained Betts optimization problem (see "Constrained Betts Function"). The iteration history follows.

```
start F_BETTS(x);
   f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
   return(f);
finish F_BETTS;

con = {  2. -50.  .    .,
        50.  50.  .    .,
        10.  -1. 1. 10.};
x = {-1. -1.};
optn = {0 2};
call nlpnra(rc,xres,"F_BETTS",x,optn,con);
quit;
```

```
                        Optimization Start
                        Parameter Estimates

                           Gradient     Lower        Upper
                           Objective    Bound        Bound
   N Parameter    Estimate  Function   Constraint   Constraint
   1 X1           6.800000  0.136000   2.000000     50.000000
   2 X2          -1.000000 -2.000000  -50.000000    50.000000


             Value of Objective Function = -98.5376



                        Linear Constraints

   1    59.00000 :       10.0000  <=   +   10.0000 * X1        -     1.0000 * X2
```

```
                 Newton-Raphson Optimization with Line Search

                        Without Parameter Scaling
                 Gradient Computed by Finite Differences
                CRP Jacobian Computed by Finite Differences
                     Parameter Estimates          2
                     Lower Bounds                  2
                     Upper Bounds                  2
                     Linear Constraints            1

                        Optimization Start

    Active Constraints       0  Objective Function    -98.5376
    Max Abs Gradient Element  2
```

```
                    Function    Active       Objective
      Iter  Restarts   Calls  Constraints    Function

       1        0       2          0         -98.81551
       2*       0       3          0         -99.40840
       3*       0       4          1         -99.87504
       4        0       5          1         -99.96000
       5        0       6          1         -99.96000
```

```
            Objective    Max Abs              Slope of
            Function    Gradient    Step      Search
      Iter   Change     Element     Size     Direction

       1     0.2779     1.8000      0.100     -2.925
       2*    0.5929     1.2713      0.294     -2.365
       3*    0.4666     0.5829      0.542     -1.181
       4     0.0850    0.000039     1.000     -0.170
       5    3.9E-10    9.537E-7     1.000     -76E-11
```

```
                        Optimization Results

    Iterations                        5  Function Calls              7
    Hessian Calls                     6  Active Constraints          1
    Objective Function           -99.96  Max Abs Gradient Element    0
    Slope of Search Direction -7.64376E-10  Ridge                    0
```

```
   GCONV convergence criterion satisfied.


                  Optimization Results
                  Parameter Estimates

                                Gradient     Active
                                Objective     Bound
     N Parameter       Estimate  Function   Constraint

     1 X1              2.000000  0.040000    Lower BC
     2 X2          -0.000000196         0

          Value of Objective Function = -99.96



     Linear Constraints Evaluated at Solution

  1     10.00000  =  -10.0000 + 10.0000 * X1 - 1.0000 * X2
```

NLPNRR Call

nonlinear optimization by Newton-Raphson ridge method

CALL NLPNRR(*rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit", "grd", "hes"*>**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPNRR algorithm uses a pure Newton step when both the Hessian is positive definite and the Newton step successfully reduces the value of the objective function. Otherwise, a multiple of the identity matrix is added to the Hessian matrix.

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix

$\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$, and it requires continuous first- and second-order derivatives of the objective function inside the feasible region.

Note that using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and may contain significant rounding errors. If you use the *"grd"* input argument to specify a module that computes first-order derivatives analytically, you can reduce drastically the computation time for numerical second-order derivatives. The computation of the finite difference approximation for the Hessian matrix generally uses only n calls of the module that specifies the gradient.

The NLPNRR method performs well for small to medium-sized problems, and it does not need many function, gradient, and Hessian calls. However, if the gradient is not specified analytically by using the *"grd"* module argument, or if the computation

of the Hessian module specified with the *"hes"* argument is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms may be more efficient.

In addition to the standard iteration history, the NLPNRR subroutine prints the following information:

- The heading *ridge* refers to the value of the nonnegative ridge parameter. A value of zero indicates that a Newton step is performed. A value greater than zero indicates either that the Hessian approximation is zero or that the Newton step fails to reduce the optimization criterion. A large value can indicate optimization difficulties.

- The heading *rho* refers to ρ, the ratio of the achieved difference in function values and the predicted difference, based on the quadratic function approximation. A value that is much smaller than one indicates possible optimization difficulties.

The following statements invoke the NLPNRR subroutine to solve the constrained Betts optimization problem (see "Constrained Betts Function"). The iteration history follows.

```
start F_BETTS(x);
   f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
   return(f);
finish F_BETTS;

con = {  2. -50.  .   .,
        50.  50.  .   .,
        10.  -1. 1. 10.};
x = {-1. -1.};
optn = {0 2};
call nlpnrr(rc,xres,"F_BETTS",x,optn,con);
quit;
```

```
                    Optimization Start
                    Parameter Estimates

                             Gradient     Lower        Upper
                             Objective    Bound        Bound
 N Parameter     Estimate    Function     Constraint   Constraint

 1 X1            6.800000     0.136000    2.000000     50.000000
 2 X2           -1.000000    -2.000000   -50.000000    50.000000

          Value of Objective Function = -98.5376



                      Linear Constraints

 1    59.00000 :       10.0000  <=   +    10.0000 * X1         -    1.0000 * X2
```

```
              Newton-Raphson Ridge Optimization


                  Without Parameter Scaling
             Gradient Computed by Finite Differences
           CRP Jacobian Computed by Finite Differences


             Parameter Estimates              2
             Lower Bounds                     2
             Upper Bounds                     2
             Linear Constraints              1




                      Optimization Start

Active Constraints         0  Objective Function    -98.5376
Max Abs Gradient Element    2


\end{jverbatim}

\begin{kverbatim}


                       Function      Active       Objective
      Iter    Restarts     Calls   Constraints     Function

       1          0         2           1         -99.87337
       2          0         3           1         -99.96000
       3          0         4           1         -99.96000


                                              Ratio
                                             Actual
              Objective    Max Abs             and
              Function     Gradient         Predicted
      Iter     Change      Element   Ridge    Change

       1       1.3358      0.5887      0       0.706
       2       0.0866     0.000040     0       1.000
       3       4.07E-10        0       0       1.014



                     Optimization Results

     Iterations               3  Function Calls               5
     Hessian Calls            4  Active Constraints           1
     Objective Function  -99.96  Max Abs Gradient Element     0
     Ridge                    0  Actual Over Pred Change   1.0135158294


     GCONV convergence criterion satisfied.

                    Optimization Results
                    Parameter Estimates
```

```
                                 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

nonlinear optimization by quasi-Newton method

CALL NLPQN(*rc, xr, "fun", x0* <,*opt, blc, tc, par, "ptit",*

"grd", "nlc", "jacnlc">**);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPQN subroutine uses (dual) quasi-Newton optimization techniques, and it is one of the two subroutines available that can solve problems with nonlinear constraints. These techniques work well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian matrix. The NLPQN subroutine does not need to compute second-order derivatives, but it generally requires more iterations than the techniques that compute second-order derivatives.

The two categories of problems solved by the NLPQN subroutine are unconstrained or linearly constrained problems and nonlinearly constrained problems. Unconstrained or linearly constrained problems do not use the *"nlc"* or *"jacnlc"* module arguments, whereas nonlinearly constrained problems use the arguments to specify the nonlinear constraints and the Jacobian matrix of their first-order derivatives, respectively.

The type of optimization problem specified determines the algorithm that the subroutine invokes. The algorithms are very different, and they use different sets of termination criteria. For more details, see "Termination Criteria."

Unconstrained or Linearly Constrained QN Optimization

The NLPQN subroutine invokes this algorithm if you do not specify the *"nlc"* argument. Using the fourth element of the *opt* argument, you can specify two update formulas for either the original quasi-Newton algorithm or the dual quasi-Newton algorithm, as indicated in the following table:

Value of *opt*[4]	Update Method
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.
3	Original Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update of the inverse Hessian matrix.
4	Original Davidon, Fletcher, and Powell (DFP) update of the inverse Hessian matrix.

In each iteration, a line search is performed along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step size that satisfies the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit of the step size. Violating the left-side Goldstein condition can affect the positive definiteness of the quasi-Newton update. In these cases, either the update is skipped or the iterations are restarted with an identity matrix resulting in the steepest descent or ascent search direction. You can specify line-search algorithms different from the default method with the fifth element of the *opt* argument.

Note: In Release 6.08, the DBFGS and DDFP updates were implemented with the NLPDQN subroutine. In Release 6.09 and in later releases, these updates are specified with the NLPQN subroutine, and the NLPDQN subroutine is not permitted.

The following statements invoke the NLPQN subroutine to solve the Rosenbrock problem (see "Unconstrained Rosenbrock Function"):

```
start F_ROSEN(x);
   y1 = 10. * (x[2] - x[1] * x[1]);
   y2 = 1. - x[1];
   f  = .5 * (y1 * y1 + y2 * y2);
   return(f);
finish F_ROSEN;
x = {-1.2 1.};
optn = {0 2 . 2};
call nlpqn(rc,xr,"F_ROSEN",x,optn);
```

Since OPTN[4] = 2, the DDFP update is performed. The gradient is approximated by finite differences since no module is specified that computes the first-order derivatives. Part of the iteration history follows. In addition to the standard iteration history, the NLPQN subroutine prints the following information for unconstrained or linearly constrained problems:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly

smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

```
                    Optimization Start
                    Parameter Estimates


                                    Gradient
                                    Objective
     N Parameter          Estimate   Function

     1 X1               -1.200000  -107.799989
     2 X2                1.000000   -43.999999


             Value of Objective Function = 12.1



              Dual Quasi-Newton Optimization


      Dual Davidon - Fletcher - Powell Update (DDFP)
        Gradient Computed by Finite Differences

         Parameter Estimates              2

                    Optimization Start

  Active Constraints                 0  Objective Function  12.1
  Max Abs Gradient Element  107.79998927




                     Function      Active      Objective
        Iter  Restarts   Calls   Constraints    Function

          1       0        4           0         2.06405
          2       0        7           0         1.92035
          3       0       10           0         1.78089
          4       0       13           0         1.33331
          5       0       17           0         1.13400
          6       0       22           0         0.93915
          7       0       24           0         0.84821
          8       0       30           0         0.54334
          9       0       32           0         0.46593
         10       0       37           0         0.35322
         12       0       41           0         0.20282
         12       0       41           0         0.20282
         13       0       46           0         0.11714
         14       0       51           0         0.07149
         15       0       53           0         0.04746
         16       0       58           0         0.02759
         17       0       60           0         0.01625
         18       0       62           0         0.00475
         19       0       66           0         0.00167
         20       0       70           0        0.0005952
         21       0       72           0        0.0000771
         23       0       78           0        2.39914E-8
         23       0       78           0        2.39914E-8
         24       0       80           0        5.0936E-11
```

```
        25          0        119              0      3.9538E-11
```

	Objective Function	Max Abs Gradient	Step	Slope of Search
Iter	Change	Element	Size	Direction
1	10.0359	0.7917	0.0340	-628.8
2	0.1437	8.6301	6.557	-0.0363
3	0.1395	11.0943	8.193	-0.0288
4	0.4476	7.6069	33.376	-0.0269
5	0.1993	0.9386	15.438	-0.0260
6	0.1948	3.5290	11.537	-0.0233
7	0.0909	4.8308	8.124	-0.0193
8	0.3049	4.1770	35.143	-0.0186
9	0.0774	0.9479	8.708	-0.0178
10	0.1127	2.5981	10.964	-0.0147
11	0.0894	3.3028	13.590	-0.0121
12	0.0610	0.6451	10.000	-0.0116
13	0.0857	1.6603	11.395	-0.0102
14	0.0456	2.4050	11.559	-0.0074
15	0.0240	0.5628	6.868	-0.0071
16	0.0199	1.3282	5.365	-0.0055
17	0.0113	1.9246	5.882	-0.0035
18	0.0115	0.6357	8.068	-0.0032
19	0.00307	0.4810	2.336	-0.0022
20	0.00108	0.6043	3.287	-0.0006
21	0.000518	0.0289	2.329	-0.0004
22	0.000075	0.0365	1.772	-0.0001
23	1.897E-6	0.00158	1.159	-331E-8
24	2.394E-8	0.000016	0.967	-46E-9
25	1.14E-11	7.962E-7	1.061	-19E-13

Optimization Results

Iterations	25	Function Calls	120
Gradient Calls	107	Active Constraints	0
Objective Function	3.953804E-11	Max Abs Gradient Element	7.9622469E-7
Slope of Search Direction	-1.88032E-12		

ABSGCONV convergence criterion satisfied.

Optimization Results
Parameter Estimates

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 QN Optimization

The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's (1978, 1982) Variable Metric Constrained WatchDog (VMCWD) algorithm. A similar but older algorithm (VF02AD) is part of the Harwell library. Both the VMCWD and VF02AD algorithms use Fletcher's VE02AD algorithm, which is also part of the Harwell library, for positive definite quadratic programming. This NLPQN implementation uses a quadratic programming subroutine that updates and downdates the Cholesky factor when the active set changes (refer to Gill, Murray, Saunders, and Wright 1984). The nonlinear NLPQN algorithm is not a feasible point algorithm, and the value of the objective function is not required to decrease monotonically. Instead, the algorithm tries to reduce a linear combination of objective function and constraint violations.

The following are similarities and differences between this algorithm and Powell's VMCWD algorithm:

- You can use the sixth element of the *opt* argument to modify the algorithm used by the NLPQN subroutine. If you specify $opt[6] = 2$, which is the default, the evaluation of the Lagrange vector μ is performed the same way as described in Powell (1982b). Note, however, that the VMCWD program seems to have a bug in the implementation of formula (4.4) in Powell (1982b). If you specify $opt[6] = 1$, the original update of μ used in the VF02AD algorithm in Powell (1978a) is performed.

- Instead of updating an approximate Hessian matrix, this algorithm uses the dual BFGS or dual DFP update that updates the Cholesky factor of an approximate Hessian. If the condition of the updated matrix gets too bad, a restart is done with a positive diagonal matrix. At the end of the first iteration after each restart, the Cholesky factor is scaled.

- The Cholesky factor is loaded into the quadratic programming subroutine, which ensures positive definiteness of the problem. During the quadratic programming step, the Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ is updated simultaneously with QT decomposition when the active set changes. Refer to Gill, Murray, Saunders, and Wright (1984) for more information.

- The line-search strategy is very similar to that of Powell's algorithm, but this algorithm does not call for derivatives during the line search. For that reason, this algorithm generally needs fewer derivative calls than function calls, whereas the VMCWD algorithm always requires the same number of derivative calls as function calls. Also, Powell's line-search method sometimes uses steps that are too long during the early iterations. In those cases, you can use the second element of the *par* argument to restrict the step length α in the first five iterations. See "Control Parameters Vector" for more details.

- The watchdog strategy is also similar to that of Powell's algorithm. However, this algorithm does not return automatically after a fixed number of iterations to a previous, more optimal point. A return to such a point is further delayed if

the observed function reduction is close to the expected function reduction of the quadratic model.

- Although Powell's termination criterion, the FTOL2 criterion, can still be used, the NLPQN implementation uses, by default, two other termination criteria (GTOL and ABSGTOL).

This algorithm is automatically invoked if the *"nlc"* argument is specified. The module specified with the *"nlc"* argument must return a vector of length nc, where nc is the total number of constraints. Letting nec be the number of equality constraints, the constraints must be of the following form:

$$
\begin{array}{rcll}
c_i(x) & = & 0, & i = 1, \ldots, nec \\
c_i(x) & \geq & 0, & i = nec + 1, \ldots, nc
\end{array}
$$

The first nec elements of the returned vector contain the c_i for the equality constraints, and the remaining elements contain the c_i for the inequality constraints.

Note: You must specify the total number of constraints with the tenth element of the *opt* argument, and if there are any equality constraints, you must specify that number, nec, with the eleventh element of the *opt* argument.

The nonlinear NLPQN algorithm requires the Jacobian matrix of the first-order derivatives of the nc constraints returned by the module specified by the *"nlc"* argument. You can provide these derivatives by specifying a module with the *"jacnlc"* argument. This module must return the Jacobian matrix **J** of first-order partial derivatives. That is, **J** is an $nc \times n$ matrix such that the entry in the ith row and j^{th} column is given by

$$
\mathbf{J}(i,j) = \frac{\partial c_i}{\partial x_j}
$$

If you specify an *"nlc"* module without specifying a *"jacnlc"* argument, finite difference approximations of the first-order derivatives of the constraints are used. You can use the ninth element of the *par* argument to specify the number of accurate digits used in evaluating the constraints.

You can specify two update formulas with the fourth element of the *opt* argument as indicated in the following table:

Value of *opt*[4]	Update Method
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.

This algorithm uses its own line-search technique. None of the options and parameters that control the line search in the other algorithms apply in the nonlinear NLPQN

algorithm, with the exception of the second element of the *par* vector, which can be used to restrict the length of the step size in the first five iterations.

See Example 11.8 for an example where you need to specify a value for the second element of the *par* argument. The values of the fourth, fifth, and sixth elements of the *par* vector, which control the processing of linear and boundary constraints, are valid only for the quadratic programming subroutine used in each iteration of the NLPQN call. For a simple example of the NLPQN subroutine, see "Rosen-Suzuki Problem".

NLPQUA Call

nonlinear optimization by quadratic method

> **CALL NLPQUA(** *rc, xr, quad, x0 <,opt, blc, tc, par, "ptit", lin>***);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP subroutines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPQUA subroutine uses a fast algorithm for maximizing or minimizing the quadratic objective function

$$\frac{1}{2}x^T G x + g^T x + con$$

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 (refer to Gill, Murray, Saunders, and Wright, 1984). If n_f is the number of free parameters (that is, n minus the number of active boundary constraints), and n_a is the number of active linear constraints, then Q is an $n_f \times n_f$ orthogonal matrix containing null space Z in its first $n_f - n_a$ columns and range space Y in its last n_a columns. The matrix T is an $n_a \times n_a$ triangular matrix of the form $t_{ij} = 0$ for $i < n - j$. The Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ is updated simultaneously with the QT decomposition when the active set changes.

The objective function is specified by the input arguments *quad* and *lin*, as follows:

- The *quad* argument specifies the symmetric $n \times n$ Hessian matrix, G, of the quadratic term. The input can be in dense or sparse form. In dense form, all n^2 entries of the *quad* matrix must be specified. If $n \leq 3$, the dense specification must be used. The sparse specification can be useful when G has many zero elements. You can specify an $nn \times 3$ matrix in which each row represents one of the nn nonzero elements of G. The first column specifies the row location

in G, the second column specifies the column location, and the third column specifies the value of the nonzero element.

- The *lin* argument specifies the linear part of the quadratic optimization problem. It must be a vector of length n or $n + 1$. If *lin* is a vector of length n, it specifies the vector g of the linear term, and the constant term *con* is considered zero. If *lin* is a vector of length $n + 1$, then the first n elements of the argument specify the vector g and the last element specifies the constant term *con* of the objective function.

As in the other optimization subroutines, you can use the *blc* argument to specify boundary and general linear constraints, and you must provide a starting point *x0* to determine the number of parameters. If *x0* is not feasible, a feasible initial point is computed by linear programming, and the elements of *x0* can be missing values.

Assuming nonnegativity constraints $x \geq 0$, the quadratic optimization problem solved with the LCP call, which solves the linear complementarity problem. Refer to *SAS/IML Software: Usage and Reference, Version 6, First Edition* for details.

Choosing a sparse (or dense) input form of the *quad* argument does not mean that the algorithm used in the NLPQUA subroutine is necessarily sparse (or dense). If the following conditions are satisfied, the NLPQUA algorithm will store and process the matrix G as sparse:

- No general linear constraints are specified.

- The memory needed for the sparse storage of G is less than 80% of the memory needed for dense storage.

- G is not a diagonal matrix. If G is diagonal, it is stored and processed as a diagonal matrix.

The sparse NLPQUA algorithm uses a modified form of minimum degree Cholesky factorization (George and Liu 1981).

In addition to the standard iteration history, the NLPNRA subroutine prints the following information:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.

- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The Betts problem (see "Constrained Betts Function") can be expressed as a quadratic problem in the following way:

$$x = \left[\begin{array}{c} x_1 \\ x_2 \end{array} \right], \quad G = \left[\begin{array}{cc} 0.02 & 0 \\ 0 & 2 \end{array} \right], \quad g = \left[\begin{array}{c} 0 \\ 0 \end{array} \right], \quad con = -100$$

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.};
optn = {0 2};
CALL NLPQUA(rc,xres,quad,x,optn,c,,,,lin);
```

The *quad* argument specifies the *G* matrix, and the *lin* argument specifies the *g* vector with the value of *con* appended as the last element. The matrix C specifies the boundary constraints and the general linear constraint.

The iteration history follows.

Optimization Start
Parameter Estimates

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
```

Null Space Method of Quadratic Problem

Parameter Estimates	2
Lower Bounds	2
Upper Bounds	2
Linear Constraints	1
Using Sparse Hessian	—

Optimization Start

```
Active Constraints         0   Objective Function   -98.5376
Max Abs Gradient Element   2
```

		Function	Active	Objective
Iter	Restarts	Calls	Constraints	Function
1	0	2	1	-99.87349
2	0	3	1	-99.96000

	Objective	Max Abs		Slope of
	Function	Gradient	Step	Search
Iter	Change	Element	Size	Direction
1	1.3359	0.5882	0.706	-2.925
2	0.0865	0	1.000	-0.173

Optimization Results

Iterations	2	Function Calls	4
Gradient Calls	3	Active Constraints	1
Objective Function	-99.96	Max Abs Gradient Element	0
Slope of Search Direction	-0.173010381		

ABSGCONV convergence criterion satisfied.

Optimization Results
Parameter Estimates

| | | Gradient | Active |
| | | Objective | Bound |
N Parameter	Estimate	Function	Constraint
1 X1	2.000000	0.040000	Lower BC
2 X2	0	0	

Value of Objective Function = -99.96

Linear Constraints Evaluated at Solution

1   10.00000 = -10.0000 + 10.0000 * X1 - 1.0000 * X2

NLPTR Call

nonlinear optimization by trust region method

CALL NLPTR(*rc, xr, "fun", x0 <,opt, blc, tc, par, "ptit", "grd", "hes">***);**

See "Nonlinear Optimization and Related Subroutines" for a listing of all NLP sub-routines. See Chapter 11, "Nonlinear Optimization Examples," for a description of the inputs to and outputs of all NLP subroutines.

The NLPTR subroutine is a trust-region method that uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$. It requires that the objective function $f = f(x)$ has continuous first- and second-order derivatives inside the feasible region.

The $n \times n$ Hessian matrix $\mathbf{G}$ 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 non-linear objective function within a hyperelliptic trust region. This trust region has a radius, Δ, that constrains the step size corresponding to the quality of the quadratic approximation. The method is implemented using Dennis, Gay, and Welsch (1981), Gay (1983), and Moré and Sorensen (1983).

Note that finite difference approximations for second-order derivatives using only function calls are computationally very expensive. If you specify first-order deriva-tives analytically with the *"grd"* module argument, you can drastically reduce the computation time for numerical second-order derivatives. Computing the finite dif-ference 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 analytically by using the *"grd"* argument or if the computation of the Hessian module, as specified by the *"hes"* module argument, is computationally ex-pensive, one of the (dual) quasi-Newton or conjugate gradient algorithms may be more efficient.

In addition to the standard iteration history, the NLPTR subroutine prints the follow-ing information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number in-dicates 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 re-gion, in which case the scaled Newton step is too long to fit in the trust region and a quadratically-constrained optimization is done. Large values indicate optimization difficulties, and as in Gay (1983), a negative value indicates the special case of an indefinite Hessian matrix.

- The heading *radius* refers to Δ, the radius of the trust region. Small values of the radius combined with large values of λ in subsequent iterations indicate optimization problems.

For an example of the use of the NLPTR subroutine, see "Unconstrained Rosenbrock Function".

NORMAL Function

generates a pseudo-random normal deviate

> **NORMAL(** *seed***)**

where *seed* is a numeric matrix or literal. The *seed* argument can be any integer value up to $2^{31} - 1$.

The NORMAL function is a scalar function that returns a pseudo-random number having a normal distribution with a mean of 0 and a standard deviation of 1. The NORMAL function returns a matrix with the same dimensions as the argument. The first argument on the first call is used for the seed (or if that is 0, the system clock is used for the seed). This function is synonymous with the DATA step function RANNOR. The Box-Muller transformation of the UNIFORM function deviates is used to generate the numbers.

```
seed = 123456;
c = j(10,1,seed);
b = normal(c);
print b;

         B

    -0.109483
    -0.348785
    1.1202546
    -2.513766
    1.3630022
```

NROW Function

finds the number of rows of a matrix

> **NROW(** *matrix***)**

where *matrix* is a numeric or character matrix.

The NROW function returns a single numeric value that is the number of rows in matrix. If the matrix has not been given a value, the NROW function returns a value of 0.

For example, to let J contain the number of rows of the matrix **S**, use the statement

```
j=nrow(s);
```

NUM Function

produces a numeric representation of a character matrix

NUM(*matrix***)**

where *matrix* is a character matrix or a quoted literal.

The NUM function takes as an argument a character matrix with elements that are character numerics; and produces a numeric matrix with dimensions that are the same as the dimensions of the argument and with elements that are the numeric representations (double-precision floating-point) of the corresponding elements of the argument.

An example using the NUM function is shown below:

```
c={'1' '2' '3'};
j=num(c);
```

```
        C          1 row        3 cols     (character, size 1)

                                  1 2 3


        J          1 row        3 cols     (numeric)
                        1           2          3
```

See also the description of the CHAR function, which does the reverse conversion.

ODE Call

performs numerical integration of vector differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}(t)) \quad \text{with } \mathbf{y}(0) = \mathbf{c}$$

CALL ODE(*r, "dername", c, t, h* <, **J=***"jacobian"*><, **EPS=***eps*><, *"data"*>**);**

The ODE subroutine returns the following values:

r is a numeric matrix that contains the results of the integration over connected subintervals. The number of columns in r is equal to the number of subintervals of integration as defined by the argument t. In case of any error in the integration on any subinterval, partial results will not be reported in r.

The inputs to the ODE subroutine are as follows:

"dername"   specifies the name of an IML module used to evaluate the integrand.

c specifies an initial value vector for the variable $\mathbf{y}$.

t	specifies a sorted vector that describes the limits of integration over connected subintervals. The simplest form of the vector t contains only the limits of the integration on one interval. The first component of t should contain the initial value, and the second component should be the final value of the independent variable. For more advanced usage of the ODE subroutine, the vector t can contain more than two components. The components of the vector must be sorted in ascending order. Two consecutive components of the vector t are interpreted as a subinterval. The ODE call reports the final result of integration at the right endpoint of each subinterval. This information is vital if $f(\cdot)$ has internal points of discontinuity. To produce accurate solutions, it is essential that you provide the location of these points in the variable t, since the continuity of the forcing function is vital to the internal control of error.
h	specifies a numeric vector that contains three components: the minimum allowable step size, *hmin*; the maximum allowable step size, *hmax*; and the initial step size to start the integration process, *hinit*.
"jacobian"	optionally specifies the name of an IML module that is used to evaluate the Jacobian analytically. The Jacobian is the matrix J, with

$$J_{ij} = \frac{\partial f_i}{\partial y_j}$$

If the *"jacobian"* module is not specified, the ODE call uses a finite difference method to approximate the Jacobian. The keyword for this option is J.

eps	specifies a scalar indicating the required accuracy. It has a default value of 1E−4. The keyword for this option is EPS.
data	(scalar, optional, character, input) a valid predefined SAS Dataset name that is used to save the successful independent and dependent variables of the integration at each step.

The ODE subroutine is an adaptive, variable order, variable step-size, stiff integrator based on implicit backward-difference methods. Refer to Aiken (1985), Bickart and Picel (1973), Donelson and Hansen (1971), Gaffney (1984), and Shampine (1978). The integrator is an implicit predictor-corrector method that locally attempts to maintain the prescribed precision *eps* relative to

$$d = \max_{0 \le t \le T} (\|y(t)\|_\infty, 1)$$

As you can see from the expression, this quantity is dynamically updated during the integration process and can help you to understand the validity of the results reported by the subroutine.

Consider the differential equation

$$\frac{dy}{dt} = -ty \text{ with } y = 0.5 \text{ at } t = 0$$

The following statements attempt to find the solution at $t = 1$:

```
/* 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(r,"FUN",c,t,h);
print r[format=E21.14];
```

In this case, the integration is carried out over $(0, 1)$ to give the value of y at $t = 1$. The optional parameter *eps* has not been specified, so it is internally set to 1E−4. Also, the optional parameter *"jacobian"* has not been specified, so finite difference methods are used to estimate the Jacobian. The accuracy of the answer can be increased by specifying *eps*. For example, set *eps*=1E−7 as follows.

```
/* 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(r,"FUN",c,t,h) eps=1E-7;

print r[format =E21.14];
```

Compare this value to $0.5e^{-0.5} = 3.03265329856310\text{E}-01$ and observe that the result is correct through the sixth decimal digit and has an error relative to 1 that is $O(1\text{E}-7)$.

If the solution was desired at 1 and 2 with an accuracy of 1E−7, you would use the following statements:

```
/* Define the integrand */
start fun(t,y);
    v = -t*y;
    return(v);
finish;

/* Call ODE */
c   = 0.5;
t   = { 0 1 2};
```

```
h    = { 1E-12 1. 1E-5};
call ode(r,"FUN",c,t,h) eps=1E-7;

print r[format=E21.14];
```

Note that R contains the solution at $t = 1$ in the first column and at $t = 2$ in the second column.

Now consider the smoothness of the forcing function $f(\cdot)$. For the purpose of estimating errors, adaptive methods require some degree of smoothness in the function $f(\cdot)$. If this smoothness is not present in $f(\cdot)$ over the interior and including the left endpoint of the subinterval, the reported result will not have the desired accuracy. The function $f(\cdot)$ must be at least continuous. If the function does not meet this requirement, you should specify the discontinuity as an intermediate point. For example, consider the differential equation

$$\frac{dy}{dt} = \begin{cases} t & \text{if } t < 1 \\ 0.5t^2 & \text{if } t \geq 1 \end{cases}$$

To find the solution at $t = 2$, use the following statements:

```
/* Define the integrand */
start fun(t,y);
    if t < 1 then v = t;
    else    v = .5*t*t;
    return(v);
finish;

/* Call ODE */
c    = 0;
t    = { 0 2};
h    = { 1E-12 1. 1E-5};
call ode(r,"FUN",c,t,h) eps = 1E-12;
print r[format =E21.14];
```

In the preceding case, the integration is carried out over a single interval, $(0, 2)$. The optional parameter *eps* is specified to be 1E$-$12. The optional parameter *jacobian* is not specified, so finite difference methods are used to estimate the Jacobian.

Note that the value of R does not have the required accuracy (it should contain a 12 decimal-place representation of 5/3), although no error message is produced. The reason is that the function is not continuous at the point $t = 1$. Even the lowest-order method cannot produce a local reliable error estimate near the point of discontinuity. To avoid this problem, you can create subintervals so that the integration is carried out first over $(0, 1)$ and then over $(1, 2)$. The following code implements this method:

```
/* Define the integrand */
start fun(t,y);
    if t < 1 then v = t;
    else            v = .5*t*t;
```

```
      return(v);
   finish;

   /* Call ODE */
   c    = 0;
   t    = { 0 1 2};
   h    = { 1E-12 1. 1E-5};
   call ode(r,"FUN",c,t,h) eps=1E-12;
   print r[format=E21.14];
```

The variable R contains the solutions at both $t = 1$ and $t = 2$, and the errors are of the specified order. Although there is no interest in the solution at the point $t = 1$, the advantage of specifying subintervals with no discontinuities is that the function $f(\cdot)$ is infinitely differentiable in each subinterval.

When $f(\cdot)$ is continuous, the ODE subroutine can compute the integration to the specified precision, even if the function is defined piecewise. Consider the differential equation

$$\frac{dy}{dt} = \begin{cases} t & \text{if } t < 1 \\ t^2 & \text{if } t \geq 1 \end{cases}$$

The following code finds the solution at $t = 2$: Since the function $f(\cdot)$ is continuous, the requirements for error control are satisfied.

```
   /* Define the integrand */
   start fun(t,y);
       if t < 1 then v = t;
       else            v = t*t;
       return(v);
   finish;

   /* Call ODE */
   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];
```

This example compares the ODE call to an eigenvalue decomposition for stiff-linear systems. In the problem

$$\frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y} \ \text{ with } \ \mathbf{y}(0) = \mathbf{c}$$

where $\mathbf{A}$ is a symmetric constant matrix, the solution can be written in terms of the eigenvalue decomposition, as follows:

$$\mathbf{y}(t) = \mathbf{U}e^{\mathbf{D}t}\mathbf{U}'\mathbf{c}$$

where **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 using the ODE call and the other 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);
   v = a;
   return(v);
finish;

a = {-1000 -1 -2 -3,
        -1 -2  3 -1,
        -2  3 -4 -3,
        -3 -1 -3 -5 };

/* Call ODE */
count = 0;
t = { 0 1 2};
h = {1E-12  1 1E-5};
eps = 1E-9;
c = {1, 0, 0, 0 };
call ode(z,"FUN",c,t,h)  eps=eps j="JAC";
print z[format=E21.14];


 print count;

/* Do the eigenvalue decomposition */
start eval(t) global(d,u,c);
   v = u*diag(exp(d*t))*u`*c;
   return(v);
finish;

call eigen(d,u,a);
free z1;
do i = 1 to nrow(t)*ncol(t)-1;
  z1 = z1 || (eval(t[i+1]));
end;
print z1[format=E21.14];
```

The question now is whether this is an $O(1\mathrm{E}-9)$ result. Note that for this problem

$$d = \max_{0 \le t \le T} (\|y(t)\|_\infty, 1) = 1$$

with the $1E-6$ result, the ODE call should attempt to maintain an accuracy of $1E-9$ relative to 1. Therefore, the $1E-6$ result should have almost three correct decimal places. At $t = 2$, the first component of **Z** is 6.58597048842310E-06, while its more accurate value is 6.58580950203220E-06, showing an $O(1E-10)$ error.

The ODE subroutine may fail for problems with unusual qualitative properties, such as finite escape time in the interval of integration (that is, the solution goes towards infinity at some finite time). In such cases, try testing with different subintervals and different levels of accuracy to gain some qualitative information about the behavior of the solution of the differential equation.

OPSCAL Function

rescales qualitative data to be a least-squares fit to qualitative data

OPSCAL(*mlevel, quanti$<$, qualit$>$***)**

The inputs to the OPSCAL function are as follows:

mlevel	specifies a scalar that has one of two values. When *mlevel* is 1 the *qualit* matrix is at the nominal measurement level; when *mlevel* is 2 it is at the ordinal measurement level.
quanti	specifies an $m \times n$ matrix of quantitative information assumed to be at the interval level of measurement.
qualit	specifies an $m \times n$ matrix of qualitative information whose level of measurement is specified by *mlevel*. When *qualit* is omitted, *mlevel* must be 2. When omitted, a temporary *qualit* is constructed that contains the integers from 1 to n in the first row, from $n + 1$ to $2n$ in the second row, from $2n + 1$ to $3n$ in the third row, and so forth, up to the integers $(m - 1)n$ to mn in the last(mth) row. Note that you cannot specify *qualit* as a character matrix.

The result of the OPSCAL function is the optimal scaling transformation of the qualitative (nominal or ordinal) data in *qualit*. The optimal scaling transformation result

- is a least-squares fit to the quantitative data in *quanti*
- preserves the qualitative measurement level of *qualit*

When *qualit* is at the nominal level of measurement, the optimal scaling transformation result is a least-squares fit to *quanti*, given the restriction that the category structure of *qualit* must be preserved. If element i of *qualit* is in category c, then element i of the optimum scaling transformation result is the mean of all those elements of *quanti* that correspond to elements of *qualit* that are in category c.

For example, consider these statements:

```
quanti={5   4   6   7   4   6   2   4   8   6};
qualit={6   6   2  12   4  10   4  10   8   6};
os=opscal(1,quanti,qualit);
```

The resulting vector **OS** has the following values:

```
            OS         1 row      10 cols     (numeric)

      5       5       6       7       3       5       3
:     5       8       5
```

The optimal scaling transformation result is said to preserve the nominal measurement level of *qualit* because wherever there was a *qualit* category c, there is now a result category label v. The transformation is least squares because the result element v is the mean of appropriate elements of *quanti*. This is Young's (1981) discrete-nominal transformation.

When *qualit* is at the ordinal level of measurement, the optimal scaling transformation result is a least-squares fit to *quanti*, given the restriction that the ordinal structure of *qualit* must be preserved. This is done by determining blocks of elements of *qualit* so that if element i of *qualit* is in block b, then element i of the result is the mean of all those *quanti* elements corresponding to block b elements of *qualit* so that the means are (weakly) in the same order as the elements of *qualit*. For example, consider these statements:

```
quanti={5   4   6   7   4   6   2   4   8   6};
qualit={6   6   2 12   4 10   4 10   8   6};
os=opscal(2,quanti,qualit);
```

The resulting vector **OS** has the following values:

```
            OS         1 row     10 cols   (numeric)

      5       5       4       7       4       6       4
:     6       6       5
```

This transformation preserves the ordinal measurement level of *qualit* because the elements of *qualit* and the result are (weakly) in the same order. It is least-squares because the result elements are the means of appropriate elements of *quanti*. By comparing this result to the nominal one, you see that categories whose means are incorrectly ordered have been merged together to form correctly ordered blocks. This is known as Kruskal's (1964) least-squares monotonic transformation. Consider the following statements:

```
quanti={5   3   6   7   5   7   8   6   7   8};
os=opscal(2,quanti);
```

These statements imply that

```
qualit={ 1   2   3   4   5   6   7   8   9 10} ;
```

which means that the resulting vector has the values

```
            OS        1 row   10 cols    (numeric)

        4     4       6       6        6      7     7
     :  7     7       8
```

ORPOL Function

generates orthogonal polynomials on a discrete set of points

ORPOL(*x<, maxdegree<, weights>>***)**

The inputs 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, *maxdegree must* also be specified.

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* using the algorithm of Emerson (1968). The result is a column-orthonormal matrix $\mathbf{P}$ with n rows and *maxdegree*+1 columns such that $\mathbf{P}'\text{diag}(weights)\mathbf{P} = \mathbf{I}$. 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,

$$\mathbf{P}'\text{diag}(weights)\mathbf{P} = \begin{bmatrix} I(r) & 0 \\ 0 & 0 \end{bmatrix}$$

The statement below results in a matrix with three orthogonal columns:

```
x = T(1:5);
P = orpol(x,2);
```

```
                              P

              0.4472136 -0.632456 0.5345225
              0.4472136 -0.316228 -0.267261
              0.4472136         0 -0.534522
              0.4472136 0.3162278 -0.267261
              0.4472136 0.6324555 0.5345225
```

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 $\mathbf{P}$ 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 the following code shows.

```
start InnerProduct(f,g,w);
   h = f#g#w;
   return (h[+]);
finish;

/* Verify orthonormal */
reset fuzz;           /* print tiny numbers as zero */
w = j(ncol(x),1,1); /* default weight is all ones */
do i = 1 to 3;
   do j = 1 to i;
      InnerProd = InnerProduct(P[,i], P[,j], w );
      print i j InnerProd;
   end;
end;
```

Some reference books on orthogonal polynominals do not normalize the columns of the matrix that represents the orthogonal polynominals. For example, a textbook may give

```
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 };
```

as a fourth-degree polynomial evaluated on evenly spaced data. To compare this representation to the normalized representation that ORPOL 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;
```

```
        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 $\mathbf{P}$ 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 program below, 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> */
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"}];
```

The results of this program are given below.

```
  FITRESULTS
Degree        SSE

     0 3.1733014
     1 4.6716722
     2 1.3345326
     3 1.3758639
     4 0.8644558
```

Testing Linear Hypotheses

ORPOL can also be used to test linear hypotheses. Suppose you have an experimental design with k factor levels. (The factor levels may be equally or unequally spaced.) At the ith level, you record n_k observation, $i = 1 \ldots k$. 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 acount, 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 program below 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 program calls ORPOL to generate the orthogonal polynomial matrix $\mathbf{P}$, and uses it to form the Type I hypothesis matrix $\mathbf{L}$. The program then uses the DESIGN function to generate $\mathbf{X}$, the design matrix associated with the experiment. The program then computes $\mathbf{b}$, the estimated parameters of the linear model.

Since $\mathbf{L}$ was expressed in terms of the orthogonal polynomial matrix $\mathbf{P}$, 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 is 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 will be the mean of the y's
   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)

   so 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"];
```

The resulting output is

```
          Type I SS

    Intercept 331.8783538
    Linear    173.4756050
    Quadratic   0.4612604
    Cubic       0.0752106
```

which indicates that most of the variation in the data can be explained by a first-degree polynomial.

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" familes 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 ORPOL are *different* from these named families because ORPOL uses a different inner product. There are no IML functions that can automatically generate these families, however, you can write an IML 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" familes, 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 program below 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

provides columnwise orthogonalization by the Gram-Schmidt process and step-wise QR decomposition by the Gram-Schmidt process

    CALL ORTVEC(w, r, ρ, *lindep*, v <, q>);

The ORTVEC subroutine returns the following values:

w If the Gram-Schmidt process converges (*lindep*=0), w is the $m \times 1$ vector $\mathbf{w}$ orthonormal to the columns of $\mathbf{Q}$, which is assumed to have $n \leq m$ (nearly) orthonormal columns. If the Gram-Schmidt process does not converge (*lindep*=1), w is a vector of missing values. For stepwise QR decomposition, w is the $(n+1)^{\text{th}}$ orthogonal column of the matrix $\mathbf{Q}$. If there is no matrix $\mathbf{Q}$, that is, if the q argument is not specified, w is the normalized value of the vector $\mathbf{v}$,

$$\mathbf{w} = \frac{\mathbf{v}}{\sqrt{\mathbf{v}'\mathbf{v}}}$$

r If the Gram-Schmidt process converges (*lindep*=0), r specifies the $n \times 1$ vector $\mathbf{r}$ of Fourier coefficients. If the Gram-Schmidt process does not converge (*lindep*=1), r is a vector of missing values. If the q argument is not specified, r is a vector with zero dimension. For stepwise QR decomposition, r contains the n upper triangular elements of the $(n+1)^{\text{th}}$ column of $\mathbf{R}$.

ρ If the Gram-Schmidt process converges (*lindep*=0), ρ specifies the distance from $\mathbf{w}$ to the range of $\mathbf{Q}$. Even if the Gram-Schmidt process converges, if ρ is sufficiently small, the vector $\mathbf{v}$ may be linearly dependent on the columns of $\mathbf{Q}$. If the Gram-Schmidt process does not converge (*lindep*=1), ρ is set to 0. For stepwise QR decomposition, ρ contains the diagonal element of the $(n+1)^{\text{th}}$ column of $\mathbf{R}$.

lindep   returns a value of 1 if the Gram-Schmidt process does not converge in 10 iterations. In most cases, if *lindep*=1, the input vector **v** is linearly dependent on the n columns of the input matrix **Q**. In that case, ρ is set to 0, and the results w and r contain missing values. If *lindep*=0, the Gram-Schmidt process did converge, and the results w, r, and ρ are computed.

The inputs to the ORTVEC subroutine are as follows:

v specifies an $m \times 1$ vector **v** that is to be orthogonalized to the n columns of **Q**. For stepwise QR decomposition of a matrix, **v** is the $(n+1)^{\text{th}}$ 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

$$\mathbf{v} = \mathbf{Q}\mathbf{r} + \rho\mathbf{w}$$

Assuming that the $m \times n$ matrix **Q** has n (nearly) orthonormal columns, the ORTVEC subroutine orthogonalizes the vector **v** to the columns of **Q**. The vector **r** is the array of Fourier coefficients, 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 $\mathbf{w}'\mathbf{w} = 1$.
- 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, ORTVEC can be called to normalize **v** only, that is, to compute $\mathbf{w} = \mathbf{v}/\sqrt{\mathbf{v}'\mathbf{v}}$ and $\rho = \sqrt{\mathbf{v}'\mathbf{v}}$ only. There are two ways of using the ORTVEC call for this reason:

- Omit the last argument q, as in `call ortvec(w,r,rho,lindep,v);`.
- Provide a matrix **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 $\mathbf{Q}$ and $\mathbf{R}$, so that $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is column orthonormal, $\mathbf{Q}'\mathbf{Q} = \mathbf{I}$, and $\mathbf{R}$ is upper triangular. The jth step is applied to the jth column, $\mathbf{v}$, of $\mathbf{A}$, and it computes the jth column $\mathbf{w}$ of $\mathbf{Q}$ and the jth column, $(\mathbf{r} \; \rho \; 0)'$, of $\mathbf{R}$.

- computing the $m \times (m - n)$ null space matrix, $\mathbf{Q}_2$, corresponding to an $m \times n$ range space matrix, $\mathbf{Q}_1$ $(m > n)$, by the following stepwise process: set $\mathbf{v} = \mathbf{e}_i$ (where $\mathbf{e}_i$ is the ith unit vector) and try to make it orthogonal to all column vectors of $\mathbf{Q}_1$ and the already generated $\mathbf{Q}_2$, if the subroutine is successful, append $\mathbf{w}$ to $\mathbf{Q}_2$; otherwise, try $\mathbf{v} = \mathbf{e}_{i+1}$.

The 4×3 matrix $\mathbf{Q}$ contains the unit vectors $\mathbf{e}_1, \mathbf{e}_3$, and $\mathbf{e}_4$. The column vector $\mathbf{v}$ is pairwise linearly independent with the three columns of $\mathbf{Q}$. As expected, the ORTVEC call computes the vector $\mathbf{w}$ as the unit vector $\mathbf{e}_2$ with $\mathbf{u} = (1, 1, 1)$ and $\rho = 1$.

```
q = { 1   0   0,
      0   0   0,
      0   1   0,
      0   0   1 };
v = { 1, 1, 1, 1 };
call ortvec(w,u,rho,lindep,v,q);
print rho u w;
```

You can perform the QR decomposition of the linearly independent columns of an $m \times n$ matrix $\mathbf{A}$ with the following statements:

```
a = { . . . enter matrix A here . . . };
nind = 0;  ndep = 0;   dmax = 0.;
n = ncol(a);   m = nrow(a);
free q;
do j = 1 to n;
   v = a[ ,j];
   call ortvec(w,u,rho,lindep,v,q);
   aro = abs(rho);
   if aro > dmax then dmax = aro;
   if aro <= 1.e-10 * dmax then lindep = 1;
   if lindep = 0 then do;
      nind = nind + 1;
      q = q || w;
      if nind = n then r = r || (u // rho);
      else r = r || (u // rho // j(n-nind,1,0.));
   end;
   else do;
      print "Column " j " is linearly dependent.";
      ndep = ndep + 1; ind[ndep] = j;
   end;
end;
```

Next, process the remaining columns of $\mathbf{A}$:

```
do j = 1 to ndep;
   k = ind[ndep-j+1];
   v = a[ ,k];
   call ortvec(w,u,rho,lindep,v,q);
   if lindep = 0 then do;
      nind = nind + 1;
      q = q || w;
      if nind = n then r = r || (u // rho);
      else r = r || (u // rho // j(n-nind,1,0.));
   end;
end;
```

Now compute the null space in the last columns of **Q**:

```
do i = 1 to m;
   if nind < m then do;
     v = j(m,1,0.); v[i] = 1.;
     call ortvec(w,u,rho,lindep,v,q);
     aro = abs(rho);
     if aro > dmax then dmax = aro;
     if aro <= 1.e-10 * dmax then lindep = 1;
     if lindep = 0 then do;
       nind = nind + 1;
       q = q || w;
     end;
     else print "Unit vector" i "linearly dependent.";
   end;
end;
if nind < m then do;
   print "This is theoretically not possible.";
end;
```

PAUSE Statement

interrupts module execution

    PAUSE *<expression> <*>*;

The inputs to the PAUSE statement are as follows:

expression   is a character matrix or quoted literal giving a message to print.

*   suppresses any messages.

The PAUSE statement stops execution of a module, saves the calling chain so that execution can resume later (by a RESUME statement), prints a pause message that you can specify, and puts you in immediate mode so you can enter more statements.

You can specify an operand in the PAUSE statement to supply a message to be printed for the pause prompt. If no operand is specified, the default message,

    paused in module XXX

is printed, where *XXX* is the name of the module containing the pause. If you want to suppress all messages in a PAUSE statement, use an asterisk as the operand:

```
pause *;
```

The PAUSE statement should only be specified in modules. It generates a warning if executed in immediate mode.

When an error occurs while executing inside a module, IML automatically behaves as though a PAUSE statement was issued. PROC IML prints a note saying

```
paused in module
```

and IML puts you in immediate mode within the module environment, where you can correct the error. You can then resume execution from the statement following the one where the error occurred by issuing a RESUME command.

IML supports pause processing of both subroutine and function modules. See also the description of the SHOW statement using the PAUSE option.

PGRAF Call

produces scatter plots

    CALL PGRAF(xy *<, id><, xlabel><, ylabel><, title>***);**

The inputs to the PGRAF subroutine are as follows:

xy is an $n \times 2$ matrix of (x, y) points.

id   is an $n \times 1$ character matrix of labels for each point. The PGRAF subroutine uses up to 8 characters per point. If *id* is a scalar (1×1), then the same label is used for all of the points. The label is centered over the actual point location. If you do not specify *id*, **x** is the default character for labeling the points.

xlabel   is a character scalar or quoted literal that labels the x axis (centered below the x axis).

ylabel   is a character scalar or quoted literal that labels the y axis (printed vertically to the left of the y axis).

title   is a character scalar or quoted literal printed above the graph.

The PGRAF subroutine produces a scatter plot suitable for display on a line printer or similar device.

The statements below specify a plotting symbol, axis labels, and a title to produce the plot shown.

```
xy={1 2, 3 3, 5 4, 6 2};
call pgraf(xy,'*','X','Y','Plot of X vs Y');
```

```
                                    Plot of X vs Y

       |
   4 + |                                    *
       |
       |
       |
 Y     |
   3 + |                   *
       |
       |
       |
       |
   2 + *                                            *
     --+------+------+------+------+------+-
        1.0    2.0    3.0    4.0    5.0    6.0
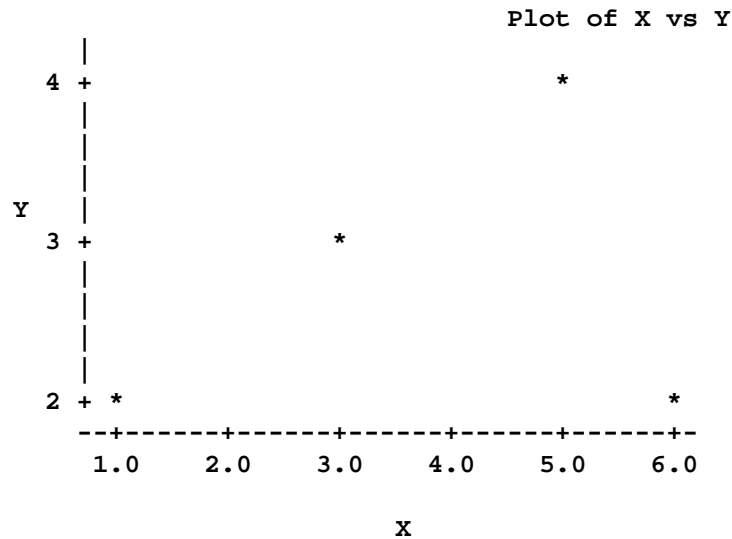
                       X
```

POLYROOT Function

finds zeros of a real polynomial

POLYROOT(*vector*)

where *vector* is an $n \times 1$ (or $1 \times n$) vector containing the coefficients of an $(n - 1)$ degree polynomial with the coefficients arranged in order of decreasing powers. The POLYROOT function returns the array r, which is an $(n - 1) \times 2$ matrix containing the roots of the polynomial. The first column of r contains the real part of the complex roots and the second column contains the imaginary part. If a root is real, the imaginary part will be 0.

The POLYROOT function finds the real and complex roots of a polynomial with real coefficients.

The POLYROOT function uses an algorithm proposed by Jenkins and Traub (1970) to find the roots of the polynomial. The algorithm is not guaranteed to find all roots of the polynomial. An appropriate warning message is issued when one or more roots cannot be found. The POLYROOT algorithm produces roots within the precision allowed by the hardware. If r is given as a root of the polynomial $P(x)$, then $1 + P(R) = 1$ based on the roundoff error of the computer that is employed.

For example, to find the roots of the polynomial

$$P(x) = 0.2567x^4 + 0.1570x^3 + 0.0821x^2 - 0.3357x + 1$$

use the following IML code to produce the result shown.

```
p={0.2567 0.1570 0.0821 -0.3357 1};
r=polyroot(p);
```

```
       R              4 rows        2 cols      (numeric)

                          0.8383029 0.8514519
                          0.8383029 -0.851452
                          -1.144107 1.1914525
                          -1.144107 -1.191452
```

The polynomial has two conjugate pairs of roots that, within machine precision, are given by $r = 0.8383029 \pm 0.8514519i$ and $r = -1.144107 \pm 1.1914525i$.

PRINT Statement

prints matrix values

PRINT *<matrices> <(expression)> <"message">*

<pointer-controls> <[options]>;

The inputs to the PRINT statement are as follows:

matrices	are the names of matrices.
(*expression*)	is an expression in parentheses that is evaluated. The result of the evaluation is printed. The evaluation of a subscripted matrix used as an expression results in printing the submatrix.
"*message*"	is a message in quotes.
pointer-controls	control the pointer for printing. For example, using a comma (,) skips a single line and using a slash (/) skips to a new page.
[*options*	] are described below.

The PRINT statement prints the specified matrices or message. The options below can appear in the PRINT statement. They are specified in brackets after the matrix name to which they apply.

COLNAME=*matrix*

specifies the name of a character matrix whose first *ncol* elements are to be used for the column labels of the matrix to be printed, where *ncol* is the number of columns in the matrix. (You can also use the RESET *autoname* statement to automatically label columns as COL1, COL2, and so on.)

FORMAT=*format*

specifies a valid SAS or user-defined format to use in printing the values of the matrix, for example,

```
print x[format=5.3];
```

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 RESET *autoname*
statement to automatically label rows as ROW1, ROW2, and so on.)

```
reset autoname;
```

For example, you can use the statement below to print a matrix called $\mathbf{X}$ in format
12.2 with columns labeled AMOUNT and NET PAY, and rows labeled DIV A and
DIV B:

```
x={45.125 50.500,
   75.375 90.825};
r={"DIV A"  "DIV B"};
c={"AMOUNT" "NET PAY"};

print x[rowname=r colname=c format=12.2];
```

The output is

```
              X           AMOUNT        NET PAY

           DIV A           45.13          50.50
           DIV B           75.38          90.83
```

To permanently associate the above options with a matrix name, refer to the descrip-
tion of the MATTRIB statement.

If there is not enough room to print all the matrices across the page, then one or
more matrices are printed out in the next group. If there is not enough room to print
all the columns of a matrix across the page, then the columns are folded, with the
continuation lines identified by a colon(:).

The spacing between adjacent matrices can be controlled by the SPACES= option
of the RESET statement. The FW= option of the RESET statement can be used to
control the number of print positions used to print each numeric element. For more
print-related options, including the PRINTADV option, see the description of the
RESET statement. The example below shows how to print part of a matrix:

```
y=1:10;
    /* prints first five elements of y*/
print (y[1:5]) [format=5.1];
```

PRODUCT Function

multiplies matrices of polynomials

> **PRODUCT(** a, b <, *dim*>**)**

The inputs to the PRODUCT function are as follows:

a is an $m \times (ns)$ numeric matrix. The first $m \times n$ submatrix contains the constant terms of the polynomials, the second $m \times n$ submatrix contains the first order terms, and so on.

b is an $n \times (pt)$ matrix. The first $n \times p$ submatrix contains the constant terms of the polynomials, the second $n \times p$ submatrix contains the first order terms, and so on.

dim   is a 1×1 matrix, with value $p > 0$. The value of this matrix is used to set p above. If omitted, the value of p is set to 1.

The PRODUCT function multiplies matrices of polynomials. The value returned is the $m \times (p(s + t - 1))$ matrix of the polynomial products. The first $m \times p$ submatrix contains the constant terms, the second $m \times p$ submatrix contains the first order terms, and so on.

Note: The PRODUCT function can be used to multiply the matrix operators employed in a multivariate time-series model of the form

$$\Phi_1(B)\Phi_2(B)\mathbf{Y}_t = \Theta_1(B)\Theta_2(B)\epsilon_t$$

where $\Phi_1(B)$, $\Phi_2(B)$, $\Theta_1(B)$, and $\Theta_2(B)$ are matrix polynomial operators whose first matrix coefficients are identity matrices. Often $\Phi_2(B)$ and $\Theta_2(B)$ represent seasonal components that are isolated in the modeling process but multiplied with the other operators when forming predictors or estimating parameters. The RATIO function is often employed in a time series context as well.

For example, the statement

```
r=product({1 2 3 4,
           5 6 7 8},
          {1 2 3,
           4 5 6}, 1);
```

produces the result

```
            R            2 rows      4 cols     (numeric)

                  9          31          41          33
                 29          79         105          69
```

PURGE Statement

removes observations marked for deletion and renumbers records

   PURGE;

The PURGE data processing statement is used to remove observations marked for deletion and to renumber the remaining observations. This closes the gaps created by deleted records. Execution of this statement may be time consuming because it involves rewriting the entire data set.

CAUTION: Any indexes associated with the data set are lost after a purge.

IML does not do an automatic purge for you at quit time.

In the example that follows, a data set named A is created. Then, you begin an IML session and edit A. You delete the fifth observation, list the data set, and issue a PURGE statement to delete the fifth observation and renumber the remaining observations.

```
data a;
   do i=1 to 10;
      output;
   end;
run;

proc iml;
   edit a;
   delete point 5;
   list all;
   purge;
   list all;
```

PUSH Call

pushes SAS statements into the command input stream

   CALL PUSH(*argument1*<*, argument2,..., argument15*>**);**

where *argument* is a character matrix or quoted literal containing valid SAS statements.

The PUSH subroutine pushes character arguments containing valid SAS statements (usually SAS/IML statements or global statements) to the input command stream. You can specify up to 15 arguments. Any statements pushed to the input command queue get executed when the module is put in a hold state. This is usually induced by one of the following:

- an execution error within a module
- an interrupt
- a pause command

The string pushed is read before any other lines of input. If you call the PUSH subroutine several times, the strings pushed each time are ahead of the less recently pushed strings. If you would rather place the lines after others in the input stream, then use the QUEUE command instead.

The strings you push do not appear on the log.

CAUTION: Do not push too much code at one time.

Pushing too much code at one time, or getting into infinite loops of pushing, causes problems that may result in exiting the SAS system.

For details, see Chapter 15, "Using SAS/IML Software to Generate IML Statements."

An example using the PUSH subroutine is shown below:

```
start;
   code='reset pagesize=25;';
   call push(code,'resume;');
   pause;
      /* show that pagesize was set to 25 during   */
      /* a PAUSE state of a module                 */
   show options;
finish;
run;
```

PUT Statement

writes data to an external file

> **PUT** <*operand*> <*record-directives*> <*positionals*> <*format*>;

The inputs to the PUT statement are as follows:

operand	specifies the value you want to output to the current position in the record. The *operand* can be either a variable name, a literal value, or an expression in parentheses. The *operand* can be followed immediately by an output format specification.
record-directives	start new records. There are three types:

	holding @	at the end of a PUT statement, instructs IML to put a hold on the current record so that IML can write more to the record with later PUT statements. Otherwise, IML automatically begins the next record for the next PUT statement.
	/	writes out the current record and begins forming a new record.

> *operand*   specifies that the next record written will start at the indicated byte position in the file (for RECFM=N files only). The *operand* is a literal number, a variable name, or an expression in parentheses, for example,

```
put >3 x 3.2;
```

positionals   specify the column on the record to which the PUT statement should go. There are two types of positionals:

@ *operand*   specifies to go to the indicated column, where *operand* is a literal number, a variable name, or an expression in parentheses. For example, @30 means to go to column 30.

+ *operand*   specifies that the indicated number of columns are to be skipped, where *operand* is a literal number, a variable name, or an expression in parentheses.

format   specifies a valid SAS or user-defined output format. These are of the form $w.d$ or $\$w.$ for standard numeric and character formats, respectively, where w is the width of the field and d is the decimal parameter, if any. They can also be a named format of the form *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 IML variables. The statement is described in detail in Chapter 7, "File Access."

The PUT statement is a sequence of positionals and record directives, variables, and formats. An example using the PUT statement is shown below:

```
/* output variable A in column 1 using SAS format 6.4. */
/* Skip 3 columns and output X using format 8.4        */

put @1 a 6.4 +3 x 8.4;
```

PV Function

calculates the present value of a vector of cash-flows and returns a scalar

PV(*times,flows,freq,rates***)**

The PV function returns a scalar containing the present value of the cash-flows based on the specified frequency and rates.

times   is an $n \times 1$ 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 allowed.

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 k-th cash-flow in periods from the present, and $D(t)$ is the discount function for time t.
With per-unit-time-period discount factors d_t:

$$D(t) = d_t^t$$

With continuous 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 code presents an example of the PV function.

```
timesn=T(do(1,100,1));
flows=repeat(10,100);
freq=50;
rate=repeat(0.10,100);
pv=pv(timesn,flows,freq,rate);
print pv;
```

The result is

```
PV
266.4717
```

QR Call

produces the QR decomposition of a matrix by Householder transformations

CALL QR(*q, r, piv, lindep, a* <, *ord*><, *b*>**);**

The QR subroutine returns the following values:

q　　　　　specifies an orthogonal matrix $\mathbf{Q}$ that is the product of the Householder transformations applied to the $m \times n$ matrix $\mathbf{A}$, if the b argument is not specified. In this case, the $\min(m, n)$ Householder transformations are applied, and q is an $m \times m$ matrix. If the b argument is specified, q is the $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$ that has the transposed Householder transformations $\mathbf{Q}'$ applied on the p columns of the argument matrix $\mathbf{B}$.

r　　　　　specifies a $\min(m, n) \times n$ upper triangular matrix $\mathbf{R}$ that is the upper part of the $m \times n$ upper triangular matrix $\widetilde{\mathbf{R}}$ of the QR decomposition of the matrix $\mathbf{A}$. The matrix $\widetilde{\mathbf{R}}$ of the QR decomposition can be obtained by vertical concatenation (using the IML operator //) of the $(m - \min(m, n)) \times n$ zero matrix to the result matrix $\mathbf{R}$.

piv　　　　specifies an $n \times 1$ vector of permutations of the columns of $\mathbf{A}$; that is, on return, the QR decomposition is computed, not of $\mathbf{A}$, but of the permuted matrix whose columns are $[\mathbf{A}_{piv[1]} \cdots \mathbf{A}_{piv[n]}]$. The vector *piv* corresponds to an $n \times n$ permutation matrix $\mathbf{\Pi}$.

lindep　　is the number of linearly dependent columns in matrix $\mathbf{A}$ detected by applying the $\min(m, n)$ Householder transformations in the order specified by the argument vector *piv*.

The inputs to the QR subroutine are as follows:

a　　　　　specifies an $m \times n$ matrix $\mathbf{A}$ that is to be decomposed into the product of the orthogonal matrix $\mathbf{Q}$ and the upper triangular matrix $\widetilde{\mathbf{R}}$.

ord　　　　specifies an optional $n \times 1$ vector that specifies the order of Householder transformations applied to matrix $\mathbf{A}$, as follows:

　　　　$ord[j] > 0$　　　　Column j of $\mathbf{A}$ is an *initial column*, meaning it has to be processed at the start in increasing order of $ord[j]$.

　　　　$ord[j] = 0$　　　　Column j of $\mathbf{A}$ is allowed to be permuted in order of decreasing residual Euclidean norm (pivoting).

　　　　$ord[j] < 0$　　　　Column j of $\mathbf{A}$ is a *final column*, meaning it has to be processed at the end in decreasing order of $ord[j]$.

　　　　The default is $ord[j] = j$, in which case the Householder transformations are done in the same order that the columns are stored in matrix $\mathbf{A}$ (without pivoting).

b　　　　　specifies an optional $m \times p$ matrix $\mathbf{B}$ that is to be multiplied by the transposed $m \times m$ matrix $\mathbf{Q}'$. If b is specified, the result q contains the $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$. If b is not specified, the result q contains the $m \times m$ matrix $\mathbf{Q}$.

The QR subroutine decomposes an $m \times n$ matrix $\mathbf{A}$ into the product of an $m \times m$ orthogonal matrix $\mathbf{Q}$ and an $m \times n$ upper triangular matrix $\widetilde{\mathbf{R}}$, so that

$$\mathbf{A}\mathbf{\Pi} = \mathbf{Q}\widetilde{\mathbf{R}}, \quad \mathbf{Q}'\mathbf{Q} = \mathbf{Q}\mathbf{Q}' = \mathbf{I}_m$$

by means of $\min(m, n)$ Householder transformations.

The $m \times m$ orthogonal matrix $\mathbf{Q}$ is computed only if the last argument b is not specified, as follows:

```
call qr(q,r,piv,lindep,a,ord);
```

In many applications, the number of rows, m, is very large. In these cases, the explicit computation of the $m \times m$ matrix $\mathbf{Q}$ can require too much memory or time.

In the usual case where $m > n$,

$$\mathbf{A} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

$$\widetilde{\mathbf{R}} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}$$

$$\mathbf{Q} = [\mathbf{Q}_1 \ \mathbf{Q}_2], \qquad \widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

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 **A** on which the Householder transformations are applied. When you specify the *ord* argument, the columns of **A** can be divided into the following groups:

- *ord*[*j*] > 0: Column *j* of **A** is an *initial column*, meaning it has to be processed at the start in increasing order of *ord*[*j*]. This specification defines the first nl columns of **A** that are to be processed.

- *ord*[*j*] = 0: Column *j* of **A** is a *pivot column*, meaning it is to be processed in order of decreasing residual Euclidean norms. The *pivot columns* of **A** are processed after the nl *initial columns* and before the nu *final columns*.

- *ord*[*j*] < 0: Column *j* of **A** is a *final column*, meaning it has to be processed at the end in decreasing order of *ord*[*j*]. This specification defines the last nu columns of **A** that are to be processed. If $n > m$, some of these columns will not be processed at all.

There are two special cases:

- If you do not specify the *ord* argument, the default values *ord*[*j*] = *j* are used. In this case, Householder transformations are done in the same order in which the columns are stored in **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 **A**.

The resulting $n \times 1$ vector *piv* specifies the permutation of the columns of **A** on which the Householder transformations are applied; that is, on return, the QR decomposition is computed, not of **A**, but of the matrix with columns that are permuted in the order $\mathbf{A}_{piv[1]}, \ldots, \mathbf{A}_{piv[n]}$.

To check the QR decomposition, use the following statements to compute the three residual sum of squares, represented by the variables SS0, SS1, and SS2, which should be close to zero:

```
m = nrow(a); n = ncol(a);
call qr(q,r,piv,lindep,a,ord);
ss0 = ssq(a[ ,piv] - q[,1:n] * r);
ss1 = ssq(q * q` - i(m));
ss2 = ssq(q` * q - i(m));
```

If the QR subroutine detects linearly dependent columns while processing matrix **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 size of the singularity criterion used; currently it is specified as 1E−8.

Solving the linear system $\mathbf{Rx} = \mathbf{Q'b}$ with an upper triangular matrix $\mathbf{R}$ whose columns are permuted corresponding to the result vector *piv* leads to a solution $\mathbf{x}$ with permuted components. You can reorder the components of $\mathbf{x}$ by using the index vector *piv* at the left-hand side of an expression, as follows:

```
call qr(qtb,r,piv,lindep,a,ord,b);
x[piv] = inv(r) * qtb[1:n,1:p];
```

The following example solves the full-rank linear least-squares problem. Specify the argument b as an $m \times p$ matrix $\mathbf{B}$, as follows:

```
call qr(q,r,piv,lindep,a,ord,b);
```

When you specify the b argument, the QR call computes the matrix $\mathbf{Q'B}$ (instead of $\mathbf{Q}$) as the result q. Now you can compute the p least-squares solutions $\mathbf{x}_k$ of an overdetermined linear system with an $m \times n, m > n$ coefficient matrix $\mathbf{A}$, rank$(\mathbf{A})$ $= n$, and p right-hand sides $\mathbf{b}_k$ stored as the columns of the $m \times p$ matrix $\mathbf{B}$:

$$\min_{\mathbf{x}_k} \|\mathbf{A}\mathbf{x}_k - \mathbf{b}_k\|^2, \quad k = 1, \dots, p$$

where $\| \cdot \|$ is the Euclidean vector norm. This is accomplished by solving the p upper triangular systems with back-substitution:

$$\mathbf{x}_k = \mathbf{\Pi'R}^{-1}\mathbf{Q}_1'\mathbf{b}_k, \quad k = 1, \dots, p$$

For most applications, m, the number of rows of $\mathbf{A}$, is much larger than n, the number of columns of $\mathbf{A}$, or p, the number of right-hand sides. In these cases, you are advised not to compute the large $m \times m$ matrix $\mathbf{Q}$ (which can consume too much memory and time) if you can solve your problem by computing only the smaller $m \times p$ matrix $\mathbf{Q'B}$ implicitly. For an example, use the first five columns of the 6×6 Hilbert matrix $\mathbf{A}$.

```
a= {   36       -630       3360      -7560       7560      -2772,
      -630      14700     -88200     211680    -220500      83160,
      3360     -88200     564480   -1411200    1512000    -582120,
     -7560     211680   -1411200    3628800   -3969000    1552320,
      7560    -220500    1512000   -3969000    4410000   -1746360,
     -2772      83160    -582120    1552320   -1746360     698544 };
b= { 463, -13860, 97020, -258720, 291060, -116424};
n = 5; aa = a[,1:n];
call qr(qtb,r,piv,lindep,aa,,b);
if lindep=0 then x=inv(r)*qtb[1:n];
print x;
```

Note that you are using only the first n rows, $\mathbf{Q}'_1 \mathbf{B}$, of QTB. The IF-THEN statement of the preceding code may be replaced by the more efficient TRISOLV function, as follows:

```
if lindep=0 then x=trisolv(1,r,qtb[1:n],piv);
print x;
```

Both cases produce the following output:

```
          X
          1
        0.5
  0.3333333
       0.25
        0.2
```

For information on solving rank-deficient linear least-squares problems, see the RZLIND call.

QUAD Call

performs numerical integration of scalar functions in one dimension over infinite, connected semi-infinite, and connected finite intervals

> **CALL QUAD(** r, *"fun"*, *points* <, **EPS=**_eps_><, **PEAK=**_peak_>
>
> <, **SCALE=**_scale_><, **MSG=**_msg_><, **CYCLES=**_cycles_>**);**

The QUAD subroutine returns the following value:

r is a numeric vector containing the results of the integration. The size of r is equal to the number of subintervals defined by the argument *points*. Should the numerical integration fail on a particular subinterval, the corresponding element of r is set to missing.

The inputs to the QUAD are as follows:

"fun"   specifies the name of an IML module used to evaluate the integrand.

points   specifies a sorted vector that provides the limits of integration over connected subintervals. The simplest form of the vector provides the limits of the integration on one interval. The first element of *points* should contain the left limit. The second element should be the right limit. A missing value of .M in the left limit is interpreted as $-\infty$, and a missing value of .P is interpreted as $+\infty$. For more advanced usage of the QUAD call, *points* can contain more than two elements. The elements of the vector must be sorted in an ascending order. Each two consecutive elements in *points* defines a subinterval, and the subroutine reports

the integration over each specified subinterval. The use of subintervals is important because the presence of internal points of discontinuity in the integrand will hinder the algorithm.

eps   is an optional scalar specifying the desired relative accuracy. It has a default value of $1E-7$. You can specify *eps* with the keyword EPS.

peak   is an optional scalar that is the approximate location of a maximum of the integrand. By default, it has a location of 0 for infinite intervals, a location that is one unit away from the finite boundary for semi-infinite intervals, and a centered location for bounded intervals. You can specify *peak* with the keyword PEAK.

scale is an optional scalar that is the approximate estimate of any scale in the integrand along the independent variable (see the examples). It has a default value of 1. You can specify *scale* with the keyword SCALE.

msg   is an optional character scalar that restricts the number of messages produced by the QUAD subroutine. If *msg* = "NO" then it does not produce any warning messages. You can specify *msg* with the keyword MSG.

cycles is an optional integer indicating the number of refinements allowed to achieve the required accuracy. It has a default value of 8. You can specify *cycles* with the keyword CYCLES.

If the dimensions of any optional argument are 0×0, the QUAD call uses its default value.

The QUAD subroutine quad is a numerical integrator based on adaptive Romberg-type integration techniques. Refer to Rice (1973), Sikorsky (1982), Sikorsky and Stenger (1984), and Stenger (1973a, 1973b, 1978). Many adaptive numerical integration methods (Ralston and Rabinowitz 1978) start at one end of the interval and proceed towards the other end, working on subintervals while locally maintaining a certain prescribed precision. This is not the case with the QUAD call. The QUAD call is an adaptive global-type integrator that produces a quick, rough estimate of the integration result and then refines the estimate until achieving the prescribed accuracy. This gives the subroutine an advantage over Gauss-Hermite and Gauss-Laguerre quadratures (Ralston and Rabinowitz 1978, Squire 1987), particularly for infinite and semi-infinite intervals, because those methods perform only a single evaluation.

Consider the integration

$$\int_0^\infty e^{-t} \, dt$$

The following statements evaluate this integral:

```
/* Define the integrand */
start fun(t);
    v = exp(-t);
    return(v);
```

```
finish;

/* Call QUAD */
a    = { 0   .P };
call quad(z,"fun",a);
print z[format=E21.14];
```

The integration is carried out over the interval $(0, \infty)$, as specified by the variable A. Note that the missing value in the second element of A is interpreted as ∞. The values of *eps*=1E−7, *peak*=1, *scale*=1, and *cycles*=8 are used by default.

The following code performs the integration over two subintervals, as specified by the variable A:

```
/* Define the integrand */
start fun(t);
    v = exp(-t);
    return(v);
finish;

/* Call QUAD */
a    = { 0   3 .P };
call quad(z,"fun",a);
print z[format=E21.14];
```

Note that the elements of A are in ascending order. The integration is carried out over $(0, 3)$ and $(3, \infty)$, and the corresponding results are shown in the output. The values of *eps*=1E−7, *peak*=1, *scale*=1, and *cycles*=8 are used by default. To obtain the results of integration over $(0, \infty)$, use the SUM function on the elements of the vector **Z**, as follows:

```
b = sum(z);
print b[format=E21.14];
```

The purpose of the *peak* and *scale* options is to enable you to avoid analytically changing the variable of the integration in order to produce a well-conditioned integrand that permits the numerical evaluation of the integration.

Consider the integration

$$\int_0^\infty e^{-10000t} \, dt$$

The following statements evaluate this integral:

```
/* Define the integrand */
start fun(t);
    v = exp(-10000*t);
    return(v);
finish;
```

```
/* Call QUAD */
a   = { 0   .P };
/* Either syntax can be used                */
/*   call quad(z,"fun",a,1E-10,0.0001); or */
call quad(z,"fun",a) eps=1E-10 peak=0.0001 ;
print z[format=E21.14];
```

Only one interval exists. The integration is carried out over $(0, \infty)$. The default values of *scale*=1 and *cycles*=8 are used.

If you do not specify a *peak* value, the integration cannot be evaluated to the desired accuracy, a message is printed to the LOG, and a missing value is returned. Note that *peak* can still be set to $1E-7$ and the integration will be successful. The evaluation of the integrand at *peak* must be 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 may lead to an increase in the number of function evaluations per step at a linear rate.

Consider the integration

$$\int_{infty}^{\infty} e^{-100000(t-3)^2} \, dt$$

The following statements evaluate this integral:

```
/* Define the integrand */
start fun(t);
    v = exp(-100000*(t-3)*(t-3));
    return(v);
finish;
/* Call QUAD */
a   = { .M   .P };
call quad(z,"fun",a) eps=1E-10 peak=3 scale=0.001 ;
print z[format=E21.14];
```

Only one interval exists. The integration is carried out over $(-\infty, \infty)$. The default value of *cycles*=8 has been used.

If you use the default value of *scale*, the integral cannot be evaluated to the desired accuracy, and a missing value is returned. The variables *scale* and *cycles* can be used to allow an increase in the number of possible function evaluations; the number of possible function evaluations will increase linearly with the reciprocal of *scale*, but it will potentially increase in an exponential manner when *cycles* is increased. Increasing the number of function evaluations increases execution time.

When you perform double integration, you must separate the variables between the iterated integrals. There should be a clear distinction between the variables of the one-dimensional integration at hand and the parameters to be passed to the integrand.

Posting the correct limits of integration is also an important issue. For example, consider the binormal probability, given by

$$\mathrm{probbnrm}(a, b, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{a} \int_{-\infty}^{b} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}\right) dy\, dx$$

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

$$\mathrm{probbnrm}(a, b, \rho) = \int_{-\infty}^{a} g(x, b, \rho)\, dx$$

with the limits from $-\infty$ to a.

You can write the equation in the form of a function with the parameters a, b, ρ as arguments. The following statements provide an example of this technique:

```
start norpdf2(t) global(yv,rho,omrho2,count);

/*-------------------------------------------------------*/
/* This function is the density function and requires  */
/* the variable T (passed in the argument)             */
/* and a list of parameters, YV, RHO, OMRHO2, COUNT    */
/* (defined in the GLOBAL clause)                      */
/*-------------------------------------------------------*/

   count = count+1;
   q=(t#t-2#rho#t#yv+yv#yv)/omrho2;
   p=exp(-q/2);
   return(p);
finish;


start marginal(v) global(yy,yv,eps);
/*-------------------------------------------------*/
/* The inner integral                            */
/* The limits of integration from .M to YY       */
/* YV is passed as a parameter to the inner integral*/
/*-------------------------------------------------*/
   interval = .M || yy;
   if ( v < -12 ) then return(0);
   yv = v;
   call quad(pm,"NORPDF2",interval) eps=eps;
   return(pm);
finish;
```

```
start norcdf2(a,b,rrho) global(yy,rho,omrho2,eps);
/*------------------------------------------------*/
/* Post some global parameters                    */
/*    YY, RHO, OMRHO2                             */
/* EPS will be set from IML                       */
/* RHO and B cannot be arguments in the GLOBAL    */
/* list at the same time                          */
/*------------------------------------------------*/
   rho = rrho;
   yy = b;
   omrho2 = 1-rho#rho;
   /*------------------------------------------------*/
   /* The outer integral                             */
   /* The limits of integration                      */
   /*------------------------------------------------*/
   interval= .M || a;



   /*------------------------------------------------*/
   /*Note that EPS the keyword = EPS the variable  */
   /*------------------------------------------------*/
   call quad(p,"MARGINAL",interval) eps=eps;

   /*-------------------------*/
   /*  PER will be reset here  */
   /*-------------------------*/
   per = 1/(8#atan(1)#sqrt(omrho2)) * p;
   return(per);
finish;

/*----------------------------------*/
/*First set up the global constants */
/*----------------------------------*/
count = 0;
eps = 1E-11;

/*-----------------------------------*/
/* Do the work and print the results  */
/*-----------------------------------*/
p =  norcdf2(2,1,0.1);
print p[format=E21.14];
print count;
```

The variable COUNT contains the number of times the NORPDF2 module is called. Note that the value computed by the NORCDF2 module is very close to that returned by the PROBBNRM function, as computed by the following statements:

```
/*------------------------------------------------*/
/* Calculate the value with the PROBBNRM function */
/*------------------------------------------------*/
pp = probbnrm(2,1,0.1);
print pp[format=E21.14];
```

Note the following:

- The iterated inner integral cannot have a left endpoint of $-\infty$. For large values of v, the inner integral does not contribute to the answer but still needs to be calculated to the required relative accuracy. Therefore, either cut off the function (when $v \leq -12$), as in the MARGINAL module in the preceding code, or have the intervals start from a reasonable cutoff value. In addition, the QUAD call stops if the integrands appear to be identically 0 (probably caused by underflow) over the interval of integration.

- This method of integration (iterated, one-dimensional integrals) is extremely conservative and requires unnecessary function evaluations. In this example, the QUAD call for the inner integration lacks information about the final value that the QUAD call for the outer integration is trying to refine. The lack of communication between the two QUAD routines can cause useless computations to be performed in the inner integration.

  To illustrate this idea, let the relative error be $1E-11$ and let the answer delivered by the outer integral be 0.8, as in this example. Any computation of the inner execution of the QUAD call that yields $0.8E-11$ or less will not contribute to the final answer of the QUAD call for the outer integral. However, the inner integral lacks this information, and for a given value of the parameter yv, it attempts to compute an answer with much more precision than is necessary. The lack of communication between the two QUAD subroutines prevents the introduction of better cut-offs. Although this method can be inefficient, the final calculations are accurate.

QUEUE Call

queues SAS statements into the command input stream

> **CALL QUEUE(** *argument1$<$, argument2,..., argument15$>$* **);**

where *argument* is a character matrix or quoted literal containing valid SAS statements.

The QUEUE subroutine places character arguments containing valid SAS statements (usually SAS/IML statements or global statements) at the end of the input command stream. You can specify up to 15 arguments. The string queued is read after other lines of input already on the queue. If you want to push the lines in front of other lines already in the queue, use the PUSH subroutine instead. Any statements queued to the input command queue get executed when the module is put in a hold state. This is usually induced by one of the following:

- an execution error within a module
- an interrupt
- a pause command.

The strings you queue do not appear on the log.

CAUTION: Do not queue too much code at one time.

Queuing too much code at one time, or getting into infinite loops of queuing, causes problems that may result in exiting the SAS System.

For more examples, consult Chapter 15, "Using SAS/IML Software to Generate IML Statements."

An example using the QUEUE subroutine follows:

```
start mod(x);
    code='x=0;';
    call queue (code,'resume;');
    pause;
finish;
x=1;
run mod(x);
print(x);
```

produces

```
                              X

                              0
```

QUIT Statement

exits from IML

QUIT;

Use the QUIT statement to exit IML. If a DATA or PROC statement is encountered, QUIT is implied. The QUIT statement is executed immediately; therefore, you cannot use QUIT as an executable statement, that is, as part of a module or conditional clause. (See the description of the ABORT statement.)

PROC IML closes all open data sets and files when a QUIT statement is encountered. Workspace and symbol spaces are freed up. If you need to use any matrix values or any module definitions in a later session, you must store them in a storage library before you quit.

RANDGEN Call

generates random numbers from a specified distribution

CALL RANDGEN(*result, distname<, parm1><, parm2><, parm3>***);**

The inputs to the RANDGEN call are as follows:

result   is a preallocated 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 call generates random numbers using the same numerical method as the RAND function in base SAS, with the efficiency optimized for IML. You can initialize the random number stream used by RANDGEN with the RANDSEED call. The $result$ parameter must be preallocated to receive the number of samples that you want to generate.

The following distributions may 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: $0 \le p \le 1$

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:

$$f(x) = \begin{cases} 1 & \text{for } p = 0, x = 0 \\ \binom{n}{x} p^x(1-p)^{1-x} & \text{for } 0 < p < 1, x = 0, \ldots, n \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

x is in the range: $x = 0, 1, \ldots, n$

p is a success probability, with range: $0 \le p \le 1$

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:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

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 probabililty, with range: $0 < p \leq 1$

Hypergeometric Distribution

The random sample x is from the probability density function:

$$f(x) = \frac{\dbinom{R}{x}\dbinom{N-R}{n-x}}{\dbinom{N}{n}}$$

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:

$$f(x) = \begin{cases} \left(\begin{array}{c} x + k - 1 \\ k - 1 \end{array} \right)(1-p)^x p^k & \text{for } 0 < p < 1, x = 0, 1, \ldots \\ 1 & \text{for } p = 1, x = 0 \end{cases}$$

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:

$$f(x) = \frac{a}{b^a} x^{a-1} e^{-\left(\frac{x}{b}\right)^a}$$

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 20.2. 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'	θ	λ	
Poisson	'POISSON'	m		
T	'T'	df		
Table	'TABLE'	p		
Triangle	'TRIANGLE'	h		
Uniform	'UNIFORM'			
Weibull	'WEIBULL'	a	b	

In practice, *distname* can be in lowercase or uppercase, and you only need to specify enough letters to distinguish one distribution from the others. For example,

```
/* generate 10 samples from a Bernoulli distribution */
r = j(10,1,.);
call randgen(r,'ber',p);
```

Except for the normal distribution, you must specify the parameters listed for each distribution above or IML will report an error. For the normal distribution, default values of $\theta = 0$ and $\lambda = 1$ are used if none 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',10,0.75);
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'); /* default parameters */
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;
print m;
```

The output is

```
                        M

            1          0          1          0
            1  0.9999234  0.9842784  0.9997739
            7          8          5         10
    -1.209834  3.9732282  -0.048339  -1.337284
    30.300691  20.653151  27.301922  26.878221
    10.636299  4.6455449  7.5284821  2.5558646
    0.2449632  2.7656037  4.2254588  0.2866158
    0.7035829  1.2676112  0.9806787  1.4811389
     8.475216  8.8723256  8.2993617  8.0409742
          109          4         33         30
            1          1          2          1
    0.7784513  0.9792472  0.6018993  0.3643607
```

```
          3          2          0          2
   0.0053637 1.4026784 -0.271338 -0.416685
          5         11          8          4
   1.3237918 0.0505162 -0.660845 -0.634447
          2          3          2          3
   0.5270875 0.6909336 0.8607548 0.5450831
   0.4064393 0.7464901 0.3463207 0.2615394
   0.4183405 0.9981923 16.812803 0.0001131
```

RANDSEED Call

sets initial random seed for RANDGEN call

> **CALL RANDSEED(** *seed<, reinit>***);**

The inputs to the RANDSEED call are as follows:

seed   is a number to be used to initialize the RANDGEN random number generator.

reinit   indicates whether to allow the random number stream to be reinitialized after the first initialization, within the same IML session.

The RANDSEED call creates an initial random seed for subsequent RANDGEN calls. If RANDSEED is not called, an initial seed will be generated from the system clock. This call is normally used when it is desirable to reproduce the same random number stream in different IML sessions. The optional *reinit* parameter controls whether the seed will be reinitialized within the same IML session. If it is set to one, identical seeds will produce the same random number sequence, otherwise a second call to RANDSEED within the same 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 IML session.

RANK Function

ranks elements of a matrix

> **RANK(** *matrix***)**

where *matrix* is a numeric matrix or literal.

The RANK function creates a new matrix containing elements that are the ranks of the corresponding elements of *matrix*. The ranks of tied values are assigned arbitrarily rather than averaged. (See the description of the RANKTIE function.)

For example, the statements

```
x={2 2 1 0 5};
y=rank(x);
```

produce the vector

```
                         Y

     3           4           2           1           5
```

The RANK function can be used to sort a vector **x**:

```
  b=x;
  x[,rank(x)]=b;
```

```
                         X

     0           1           2           2           5
```

You can also sort a matrix using the SORT subroutine.

The RANK function can also be used to find anti-ranks of **x**:

```
  r=rank(x);
  i=r;
  i[,r]=1:ncol(x);
```

```
                         I

     4           3           1           2           5
```

While the RANK function only ranks the elements of numerical matrices, you can rank the elements of a character matrix using the UNIQUE function as demonstrated below:

```
  /* Create RANK-like functionality for character matrices */
  start rankc( x );
     /* the unique function returns a sorted list */
     s = unique( x );
     idx = j(nrow(x), ncol(x));
     ctr = 1;  /* there may 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};
  r = rankc( x );
  sortedx=x;
  sortedx[ r ] = x;
  print r, x, sortedx;

  /* note that ranking is in ASCII order, where capital
```

```
      letters precede lower case letters.   To get case-insensitive
      behavior, trasform matrix before comparison */
   x = {'a' 'b' 'X' 'Y' };
   asciiOrder = rankc( x );
   alphaOrder = rankc( upcase( x ) );
   print x, asciiOrder, alphaOrder;
```

IML does not have a function that directly computes the rank of a matrix. You can use the following technique to compute the rank of matrix A:

```
   rank=round(trace(ginv(a)*a));
```

RANKTIE Function

ranks matrix elements using tie-averaging

RANKTIE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The RANKTIE function creates a new matrix containing elements that are the ranks of the corresponding elements of *matrix*. The ranks of tied values are averaged.

For example, the statements

```
   x={2 2 1 0 5};
   y=ranktie(x);
```

produce the vector

```
                          Y

   3.5         3.5            2            1            5
```

The RANKTIE function differs from the RANK function in that RANKTIE averages the ranks of tied values, whereas RANK breaks ties arbitrarily.

While the RANK function only ranks the elements of numerical matrices, you can rank the elements of a character matrix using the UNIQUE function as demonstrated below:

```
   /* Create RANKTIE-like functionality for character matrices */
   start ranktiec( x );
      s = unique( x );
      idx = j(nrow(x), ncol(x));
      ctr = 1; /* there may 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};
    rt = ranktiec( x );
    print x, rt;
```

RATES Function

calculates a column vector of (per-period, e.g per-year) interest rates converted from one base to another

RATES(*rates,oldfreq,newfreq***)**

The RATES function returns an $n \times 1$ vector of interest rates converted from one base to another.

rates   is an $n \times 1$ column vector of rates corresponding 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 allowed.

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 allowed.

Let $D(t)$ be the discount function, which is the present value of a unit amount to be received t periods from now. The discount function can be expressed in three different ways:

with per-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 period. The RATES function converts between these three representations.

For example, the following code uses the RATES function:

```
rates=T(do(0.1,0.3,0.1));
oldfreq=0;
newfreq=0;
rates=rates(rates,oldfreq,newfreq);
print rates;
```

The output is

```
          RATES
           0.1
           0.2
           0.3
```

RATIO Function

divides matrix polynomials

returns a matrix containing the terms of $\Phi(B)^{-1}\Theta(B)$ considered as a matrix of rational functions in B that have been expanded as power series

RATIO(*ar, ma, terms*<, *dim*>)

The inputs to the RATIO function are as follows:

ar   is an $n \times (ns)$ matrix representing a matrix polynomial generating function, $\Phi(B)$, in the variable B. The first $n \times n$ submatrix represents the constant term and must be nonsingular, the second $n \times n$ submatrix represents the first order coefficients, and so on.

ma   is an $n \times (mt)$ matrix representing a matrix polynomial generating function, $\Theta(B)$, in the variable B. The first $n \times m$ submatrix represents the constant term, the second $n \times m$ submatrix represents the first order term, and so on.

terms is a scalar containing the number of terms to be computed, denoted by r in the discussion below. This value must be positive.

dim   is a scalar containing the value of m above. The default value is 1.

The RATIO function multiplies a matrix of polynomials by the inverse of another matrix of polynomials. It is useful for expressing univariate and multivariate ARMA models in pure moving-average or pure autoregressive forms.

The value returned is an $n \times (mr)$ matrix containing the terms of $\Phi(B)^{-1}\Theta(B)$ considered as a matrix of rational functions in B that have been expanded as power series.

Note: The RATIO function can be used to consolidate the matrix operators employed in a multivariate time-series model of the form

$$\Phi(B)\mathbf{Y}_t = \Theta(B)\epsilon_t$$

where $\Phi(B)$ and $\Theta(B)$ are matrix polynomial operators whose first matrix coefficients are identity matrices. The RATIO function can be used to compute a truncated form of $\Psi(B) = \Phi(B)^{-1}\Theta(B)$ for the equivalent infinite order model

$$\mathbf{Y}_t = \Psi(B)\epsilon_t .$$

The RATIO function can also be employed for simple scalar polynomial division, giving a truncated form of $\theta(x)/\phi(x)$ for two scalar polynomials $\theta(x)$ and $\phi(x)$.

The cumulative sum of the elements of a column vector $\mathbf{x}$ can be obtained using

```
ratio({ 1 -1} ,x,ncol(x));
```

Consider the following example for multivariate ARMA(1,1):

```
ar={1 0 -.5  2,
    0 1   3 -.8};
ma={1 0 .9  .7,
    0 1   2 -.4};
psi=ratio(ar,ma,4,2);
```

The matrix produced is

```
        PSI
        1    0   1.4  -1.3   2.7  -1.45  11.35
  :   -9.165

        0    1    -1   0.4    -5   4.22  -12.1
  :    7.726
```

RDODT and RUPDT Calls

downdate and update QR and Cholesky decompositions

CALL RDODT(*def, rup, bup, sup, r, z* <, *b, y* <, *ssq*>>**);**

CALL RUPDT(*rup, bup, sup, r, z* <, *b, y* <, *ssq*>>**);**

The RDODT and RUPDT subroutines return the values:

def   is only used for downdating, and it specifies whether the downdating of matrix $\mathbf{R}$ by using the q rows in argument z has been successful. The result *def*=2 means that the downdating of $\mathbf{R}$ by at least one row of $\mathbf{Z}$ leads to a singular matrix and cannot be completed successfully (since the result of downdating is not unique). In that case, the results *rup*, *bup*, and *sup* contain missing values only. The result *def*=1 means that the residual sum of squares, *ssq*, could not be downdated successfully and the result *sup* contains missing values only. The result *def*=0 means that the downdating of $\mathbf{R}$ by $\mathbf{Z}$ was completed successfully.

rup	is the $n \times n$ upper triangular matrix $\mathbf{R}$ that has been updated or downdated by using the q rows in $\mathbf{Z}$.
bup	is the $n \times p$ matrix $\mathbf{B}$ of right-hand sides that has been updated or downdated by using the q rows in argument y. If the argument b is not specified, *bup* is not computed.
sup	is a p vector of square roots of residual sum of squares that is updated or downdated by using the q rows of argument y. If *ssq* is not specified, *sup* is not computed.

The inputs to the RDODT and RUPDT subroutines are as follows:

r	specifies an $n \times n$ upper triangular matrix $\mathbf{R}$ to be updated or downdated by the q rows in $\mathbf{Z}$. Only the upper triangle of $\mathbf{R}$ is used; the lower triangle can contain any information.
z	specifies a $q \times n$ matrix $\mathbf{Z}$ used rowwise to update or downdate the matrix $\mathbf{R}$.
b	specifies an optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with $\mathbf{R}$. If b is specified, the argument y must also be specified.
y	specifies an optional $q \times p$ matrix $\mathbf{Y}$ used rowwise to update or downdate the right-hand side matrix $\mathbf{B}$. If b is specified, the argument y must also be specified.
ssq	is an optional p vector that, if b is specified, specifies the square root of the error sum of squares that should be updated or downdated simultaneously with $\mathbf{R}$ and $\mathbf{B}$.

The upper triangular matrix $\mathbf{R}$ of the QR decomposition of an $m \times n$ matrix $\mathbf{A}$,

$$\mathbf{A} = \mathbf{QR}, \text{ where } \mathbf{Q}'\mathbf{Q} = \mathbf{QQ}' = \mathbf{I}_m$$

is recomputed efficiently in two cases:

- *update*: An n vector $\mathbf{z}$ is added to matrix $\mathbf{A}$.
- *downdate*: An n vector $\mathbf{z}$ is deleted from matrix $\mathbf{A}$.

Computing the whole QR decomposition of matrix $\mathbf{A}$ by Householder transformations requires $4mn^2 - 4n^3/3$ floating point operations, whereas updating or downdating the QR decomposition (by Givens rotations) of one row vector $\mathbf{z}$ requires only $2n^2$ floating point operations.

If the QR decomposition is used to solve the full-rank linear least-squares problem

$$\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|^2 = ssq$$

by solving the nonsingular upper triangular system

$$\mathbf{x} = \mathbf{R}^{-1}\mathbf{Q}'\mathbf{b}$$

then the RUPDT and RDODT subroutines can be used to update or downdate the p-transformed right-hand sides $\mathbf{Q}'\mathbf{B}$ and the residual sum-of-squares p vector *ssq* provided that for each n vector $\mathbf{z}$ added to or deleted from $\mathbf{A}$ there is also a p vector $\mathbf{y}$ added to or deleted from the $m \times p$ right-hand-side matrix $\mathbf{B}$.

If the arguments z and y of the subroutines RUPDT and RDODT contain $q > 1$ row vectors for which $\mathbf{R}$ (and $\mathbf{Q}'\mathbf{B}$, and eventually *ssq*) is to be updated or downdated, the process is performed stepwise by processing the rows $\mathbf{z}_k$ (and $\mathbf{y}_k$), $k = 1, \ldots, q$, in the order in which they are stored.

The QR decomposition of an $m \times n$ matrix $\mathbf{A}$, $m \geq n$, $\mathrm{rank}(\mathbf{A}) = n$,

$$\mathbf{A} = \mathbf{Q}\mathbf{R}, \text{ where } \mathbf{Q}'\mathbf{Q} = \mathbf{Q}\mathbf{Q}' = \mathbf{I}_m$$

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 $\mathrm{rank}(\mathbf{A}) = n$, 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,

$$\mathbf{A}'\mathbf{A} = (\mathbf{Q}\mathbf{R})'(\mathbf{Q}\mathbf{R}) = \mathbf{R}'\mathbf{R}$$

Adding a row vector $\mathbf{z}$ to matrix $\mathbf{A}$ corresponds to the rank-1 modification of the crossproduct matrix $\mathbf{C}$

$$\widetilde{\mathbf{C}} = \mathbf{C} + \mathbf{z}'\mathbf{z}, \text{ where } \widetilde{\mathbf{C}} = \widetilde{\mathbf{A}}'\widetilde{\mathbf{A}}$$

and the $(m + 1) \times n$ matrix $\widetilde{\mathbf{A}}$ contains all rows of $\mathbf{A}$ with the row $\mathbf{z}$ added.

Deleting a row vector $\mathbf{z}$ from matrix $\mathbf{A}$ corresponds to the rank-1 modification

$$\mathbf{C}^* = \mathbf{C} - \mathbf{z}'\mathbf{z}, \text{ where } \mathbf{C}^* = \mathbf{A}^{*'}\mathbf{A}^*$$

and the $(m - 1) \times n$ matrix $\mathbf{A}^*$ contains all rows of $\mathbf{A}$ with the row $\mathbf{z}$ deleted. Thus, you can also use the subroutines RUPDT and RDODT to update or downdate the Cholesky factor $\mathbf{R}$ of a positive definite crossproduct matrix $\mathbf{C}$ of $\mathbf{A}$.

The process of downdating an upper triangular matrix $\mathbf{R}$ (and eventually a residual sum-of-squares vector *ssq*) is not always successful. First of all, the downdated matrix $\mathbf{R}$ could be rank deficient. Even if the downdated matrix $\mathbf{R}$ is of full rank, the

process of downdating can be ill conditioned and does not work well if the down-
dated matrix is close (by rounding errors) to a rank-deficient one. In these cases, the
downdated matrix **R** is not unique and cannot be computed by subroutine RDODT. If
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

$$ssq_{new} = \sqrt{ssq_{old} - ssq_{dod}}$$

cannot always be computed because, due to rounding errors, the radicand can be
negative. In this case, the result vector *sup* returns missing values, and *def* returns 1.

You can use various methods to compute the p columns $\mathbf{x}_k$ of the $n \times p$ matrix $\mathbf{X}$
that minimize the p linear least-squares problems with an $m \times n$ coefficient matrix
$\mathbf{A}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$, and p right-hand-side vectors $\mathbf{b}_k$ (stored columnwise in
the $m \times p$ matrix $\mathbf{B}$). The first of the following methods solves the *normal equations*
and cannot be applied to the example with the 6×5 Hilbert matrix since too much
rounding error is introduced. Therefore, use the following simple example:

```
a = { 1 3 ,
      2 2 ,
      3 1 };
b = { 1, 1, 1};
m = nrow(a);
n = ncol(a);
p = 1;
```

- Cholesky Decomposition of Crossproduct Matrix:

```
aa = a` * a; ab = a` * b;
r  = root(aa);
x  = trisolv(2,r,ab);
x  = trisolv(1,r,x);
```

- QR Decomposition by Householder Transformations:

```
call qr(qtb,r,piv,lindep,a, ,b);
x = trisolv(1,r[,piv],qtb[1:n,]);
```

- Stepwise Update by Givens Rotations:

```
r = j(n,n,0.); qtb = j(n,p,0.); ssq = j(1,p,0.);
do i = 1 to m;
   z = a[i,];
   y = b[i,];
   call rupdt(rup,bup,sup,r,z,qtb,y,ssq);
   r   = rup;
   qtb = bup;
   ssq = sup;
end;
x = trisolv(1,r,qtb);
```

Or equivalently:

```
r   = j(n,n,0.);
qtb = j(n,p,0.);
ssq = j(1,p,0.);
call rupdt(rup,bup,sup,r,a,qtb,b,ssq);
x   = trisolv(1,rup,bup);
```

- Singular Value Decomposition:

```
call svd(u,d,v,a);
d = diag(1 / d);
x = v * d * u` * b;
```

For the preceding 3×2 example matrix **A**, each method obtains the unique LS estimator:

```
ss = ssq(a * x - b);
print ss x;
```

To compute the (transposed) matrix **Q**, you can use the following specification:

```
r = shape(0,n,n);
y = i(m);
qt = shape(0,n,m);
call rupdt(rup,qtup,sup,r,a,qt,y);
```

READ Statement

reads observations from a data set

> **READ** <*range*> <**VAR** *operand*> <**WHERE(***expression***)**>
>
> > <**INTO** *name* <**[ROWNAME=***row-name*
> >
> > **COLNAME=***column-name***]**>> ;

The inputs to the READ function are as follows:

range	specifies a range of observations.
operand	selects a set of variables.
expression	is evaluated for being true or false.
name	is the name of the target matrix.
row-name	is a character matrix or quoted literal giving descriptive row labels.
column-name	is a character matrix or quoted literal giving descriptive column labels.

The clauses and options are explained below.

Use the READ statement to read variables or records from the current SAS data set into column matrices of the VAR clause or into the single matrix of the INTO clause. When the INTO clause is used, each variable in the VAR clause becomes a column of the target matrix, and all variables in the VAR clause must be of the same type. If you specify no VAR clause, the default variables for the INTO clause are all numeric variables. Read all character variables into a target matrix by using VAR _CHAR_.

You can specify a *range* of observations with a keyword or by record number using the POINT option. You can use any of the following keywords to specify a range:

ALL   all observations

CURRENT   the current observation

NEXT <*number*>   the next observation or the next *number* of observations

AFTER   all observations after the current one

POINT *operand*   observations specified by number, where *operand* can be one of the following.

Operand	**Example**
a single record number	`point 5`
a literal giving several record numbers	`point {2 5 10}`
the name of a matrix containing record numbers	`point p`
an expression in parentheses	`point (p+1)`

If the current data set has an index in use, the POINT option is invalid.

You can specify a set of variables to use with the VAR clause. The *operand* in the VAR clause can be one of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of keywords described below:

  ALL   for all variables
  CHAR   for all character variables
  NUM   for all numeric variables.

Examples showing each possible way you can use the VAR clause follow.

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the *range* specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable   is a variable in the SAS data set.

comparison-op   is one of the following comparison operators:

> | < | less than |
> |---|---|
> | <= | less than or equal to |
> | = | equal to |
> | > | greater than |
> | >= | greater than or equal to |
> | ˆ= | not equal to |
> | ? | contains a given string |
> | ˆ? | does not contain a given string |
> | =: | begins with a given string |
> | =* | sounds like or is spelled similar to a given string |

operand   is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

   ˆ= ˆ? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

   = ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

   clause&clause   (for an AND clause)
   clause|clause   (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

You can specify ROWNAME= and COLNAME= matrices as part of the INTO clause. The COLNAME= matrix specifies the name of a new character matrix to be created. This COLNAME= matrix is created in addition to the target matrix of the INTO clause and contains variable names from the input data set corresponding to columns of the target matrix. The COLNAME= matrix has dimension $1 \times nvar$, where *nvar* is the number of variables contributing to the target matrix.

The ROWNAME= option specifies the name of a character variable in the input data set. The values of this variable are put in a character matrix with the same name as the variable. This matrix has the dimension $nobs \times 1$, where *nobs* is the number of observations in the range of the READ statement.

The *range*, VAR, WHERE, and INTO clauses are all optional and can be specified in any order.

Row 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 IML variables with the same names as the data set variables.

```
read all var{name age};
```

To read all variables for the 23rd observation only, use the statement

```
read point 23;
```

To read the data set variables NAME and ADDR for all observations with a STATE value of `NJ`, use the statement

```
read all var{name addr} where(state="NJ");
```

See Chapter 6, "Working with SAS Data Sets," for further information.

REMOVE Function

discards elements from a matrix

> **REMOVE(** *matrix, indices***)**

The inputs to the REMOVE function are as follows:

matrix   is a numeric or character matrix or literal.

indices   refers to a matrix containing the indices of elements that are removed from *matrix*.

The REMOVE function returns as a row vector elements of the first argument, with elements corresponding to the indices in the second argument discarded and the gaps removed. The first argument is indexed in row-major order, as in subscripting, and the indices must be in the range 1 to the number of elements in the first argument. Non-integer indices are truncated to their integer part. You can repeat the indices, and you can give them in any order. If all elements are removed, the result is a null matrix (zero rows and zero columns).

Thus, the statement

```
a=remove({ 5 6, 7 8} , 3);
```

removes the third element, producing the result shown:

```
         A
         5    6    8
```

The statement

```
a=remove({ 5 6 7 8} , { 3 2 3 1} );
```

causes all but the fourth element to be removed, giving the result shown:

```
         A
         8
```

REMOVE Statement

removes matrices from storage

> **REMOVE** <**MODULE=(***module-list***)** <*matrix-list*>>;

The inputs to the REMOVE statement are as follows:

module-list   specifies a module or modules to remove from storage.

matrix-list   specifies a matrix or matrices to remove from storage.

The REMOVE statement removes matrices or modules or both from the current library storage. For example, the statement below removes the three modules A, B, and C and the matrix X:

```
remove module=(A B C) X;
```

The special operand _ALL_ can be used to remove all matrices or all modules or both. For example, the following statement removes everything:

```
remove _all_ module=_all_;
```

See Chapter 14, "Storage Features," and also the descriptions of the LOAD, STORE, RESET, and SHOW statements for related information.

RENAME Call

renames a SAS data set

> **CALL RENAME(** <*libname,*> *member-name, new-name***);**

The inputs to the RENAME subroutine are as follows:

libname   is a character matrix or quoted literal containing the name of the SAS data library.

member-name   is a character matrix or quoted literal containing the current name of the data set.

new-name   is a character matrix or quoted literal containing the new data set name.

The RENAME subroutine renames a SAS data set in the specified library. All of the arguments can directly be specified in quotes, although quotes are not required. If a one-word data set name is specified, the libname specified by the RESET *deflib statement* is used. Examples of valid statements follow:

```
call rename('a','b');
call rename(a,b);
call rename(work,a,b);
```

REPEAT Function

creates a new matrix of repeated values

> **REPEAT(** *matrix, nrow, ncol***)**

The inputs to the REPEAT function are as follows:

matrix   is a numeric matrix or literal.

nrow   gives the number of times *matrix* is repeated across rows.

ncol   gives the number of times *matrix* is repeated across columns.

The REPEAT function creates a new matrix by repeating the values of the argument matrix *nrow*ncol* times, *ncol* times across the rows, and *nrow* times down the columns. The *matrix* argument can be numeric or character. For example, the following statements result in the matrix $\mathbf{Y}$, repeating the $\mathbf{X}$ matrix twice down and three times across:

```
x={ 1 2 ,
   3 4} ;
y=repeat(x,2,3);

   Y
   1          2          1          2          1          2
   3          4          3          4          3          4
   1          2          1          2          1          2
   3          4          3          4          3          4
```

REPLACE Statement

replaces values in observations and updates observations

> **REPLACE** <*range*> <**VAR** *operand*> <**WHERE(***expression***)**>;

The inputs to the REPLACE statement are as follows:

range　　　　specifies a range of observations.

operand　　　selects a set of variables.

expression　　is evaluated for being true or false.

The REPLACE statement replaces the values of observations in a SAS data set with current values of IML matrices with the same name. Use the *range*, VAR, and WHERE arguments to limit replacement to specific variables and observations. Replacement matrices should be the same type as the data set variables. The REPLACE statement uses matrix elements in row order replacing the value in the ith observation with the ith matrix element. If there are more observations in *range* than matrix elements, the REPLACE statement continues to use the last matrix element.

For example, the statements below cause all occurrences of **ILL** to be replaced by **IL** for the variable STATE:

```
state="IL";
replace all var{state} where(state="ILL");
```

You can specify a *range* of observations with a keyword or by record number using the POINT option. You can use any of the following keywords to specify a range:

ALL　　　　　　　　　all observations

CURRENT　　　　　　the current observation

NEXT <*number*>　　　the next observation or the next *number* of observations

AFTER　　　　　　　all observations after the current one

POINT *operand*　　　observations by number, where *operand* can be one of the following:

Operand	Example
a single record number	**point 5**
a literal giving several record numbers	**point {2 5 10}**
the name of a matrix containing record numbers	**point p**
an expression in parentheses	**point (p+1)**

If the current data set has an index in use, the POINT option is invalid.

You can specify a set of variables to use with the VAR clause. The *variables* argument can have the following values:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of the keywords described below:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

Examples showing each possible way you can use the VAR clause follow.

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the range specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable	is a variable in the SAS data set.
comparison-op	is any one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
^=	not equal to
?	contains a given string
^?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand	is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

ˆ= ˆ? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

= ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

clause&clause   (for an AND clause)
clause|clause   (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data setvariables, and the expression on the right-hand side refers to matrix values.

The code statement below replaces all variables in the current observation:

```
replace;
```

RESET Statement

sets processing options

> **RESET** *<options>*;

where the *options* are described below.

The RESET statement sets processing options. The options described below are currently implemented options. Note that the prefix NO turns off the feature where indicated. For options that take operands, the operand should be a literal, a name of a matrix containing the value, or an expression in parentheses. The SHOW *options statement* displays the current settings of all of the options.

AUTONAME
NOAUTONAME
specifies whether rows are automatically labeled ROW1, ROW2, and so on, and columns are labeled COL1, COL2, and so on, when a matrix is printed. Row-name and column-name attributes specified in the PRINT statement or associated via the MATTRIB statement override the default labels. The AUTONAME option causes the SPACES option to be reset to 4. The default is NOAUTONAME.

CENTER
NOCENTER

specifies whether output from the PRINT statement is centered on the page. The default is CENTER.

CLIP
NOCLIP

specifies whether SAS/IML graphs are automatically clipped outside the viewport; that is, any data falling outside the current viewport is not displayed. NOCLIP is the default.

DEFLIB=*operand*

specifies the default libname for SAS data sets when no other libname is given. This defaults to USER if a USER libname is set up, or WORK if not. The libname operand can be specified with or without quotes.

DETAILS
NODETAILS

specifies whether additional information is printed from a variety of operations, such as when files are opened and closed. The default is NODETAILS.

FLOW
NOFLOW

specifies whether operations are shown as executed. It is used for debugging only. The default is NOFLOW.

FUZZ <**=***number*>
NOFUZZ

specifies whether very small numbers are printed as zero rather than in scientific notation. If the absolute value of the number is less than the value specified in *number*, it will be printed as 0. The *number* argument is optional, and the default value varies across hosts but is typically around $1E-12$. The default is NOFUZZ.

FW=*number*

sets the field width for printing numeric values. The default field width is 9.

LINESIZE=*n*

specifies the linesize for printing. The default value is usually 78.

LOG
NOLOG

specifies whether output is routed to the log file rather than to the print file. On the log, the results are interleaved with the statements and messages. The NOLOG option routes output to the OUTPUT window in 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 usually 21.

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

resumes execution

> **RESUME;**

The RESUME statement enables you to continue execution from the line in the module where the most recent PAUSE statement was executed. PROC IML issues an automatic pause when an error occurs inside a module. If a module was paused due to an error, the RESUME statement resumes execution immediately after the statement that caused the error. The SHOW *pause* statement displays the current state of all paused modules.

RETURN Statement

returns to caller

> **RETURN** <**(***operand***)**>**;**

where *operand* is the value of the function returned. Use *operand* only in function modules.

The RETURN statement causes IML to return to the calling point in a program. If a LINK statement has been issued, IML returns to the statement following the LINK. If no LINK statement was issued, the RETURN statement exits a module. If not in a module, execution is stopped (as with a STOP statement), and IML looks for more statements to parse.

The RETURN statement with an *operand* is used in function modules that return a value. The *operand* can be a variable name or an expression. It is evaluated, and the value is returned.

See the description of the LINK statement. Also, see Chapter 5, "Programming Statements," for details.

If you use a LINK statement, you need a RETURN statement at the place where you want to go back to the statement after LINK.

If you are writing a function, use a RETURN to return the value of the function. An example is shown below.

```
start sum1(a,b);
    sum=a+b;
    return(sum);
finish;
```

ROOT Function

performs the Cholesky decomposition of a matrix

> **ROOT(** *matrix***)**

where *matrix* is a symmetric positive-definite matrix.

The ROOT function performs the Cholesky decomposition of a matrix (for example, **A**) such that

$$\mathbf{U}'\mathbf{U} = \mathbf{A}$$

where **U** is upper triangular. The matrix **A** must be symmetric and positive definite.

For example, the statements

```
xpx={25 0 5, 0 4 6, 5 6 59};
U=root(xpx);
```

produce the result shown below:

```
                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 below.

```
    b = {5, 2, 53};
    /* want to solve xpx * t = b.
       First solve U' z = b,
       then solve U t = z */
    z = trisolv(2,U,b);
    t = trisolv(1,U,z);
```

The solution is

```
                  T

                  0
                 -1
                  1
```

The ROOT function performs most of its computations in the memory allocated for returning the Cholesky decomposition.

ROWCAT Function

concatenates rows without using blank compression

  ROWCAT(*matrix<, rows<, columns>>***);**

The inputs to the ROWCAT function are as follows:

matrix   is a character matrix or quoted literal.

rows   select the rows of *matrix*.

columns   select the columns of *matrix*.

The ROWCAT function takes a character matrix or submatrix as its argument and creates a new matrix with one column whose elements are the concatenation of all row elements into a single string. If the argument has n rows and m columns, the result willhave n rows and 1 column. The element length of the result will be m times the element length of the argument. The optional rows and columns arguments may be used to select which rows and columns are concatenated.

For example, the statements

```
    b={"ABC"  "D "  "EF ",
       " GH"  " I "  " JK"};
    a=rowcat(b);
```

produce the 2×1 matrix:

```
      A          2 rows        1 col      (character, size 9)

                           ABCD  EF
                            GH I  JK
```

Quotes (") are needed only if you want to embed blanks or special characters or to maintain uppercase and lowercase distinctions.

The form

ROWCAT(*matrix, rows, columns***)**

returns the same result as

ROWCAT(*matrix***[***rows, columns***])**

The form

ROWCAT(*matrix, rows***)**

returns the same result as

ROWCAT(*matrix***[***rows,***])**

ROWCATC Function

concatenates rows using blank compression

ROWCATC(*matrix<, rows<, columns>>***);**

The inputs to the ROWCATC function are as follows:

matrix　　　　　　is a character matrix or quoted literal.

rows　　　　　　select the rows of *matrix*.

columns　　　　　select the columns of *matrix*.

The ROWCATC function works the same way as the ROWCAT function except that blanks in element strings are moved to the end of the concatenation. For example, the statements

```
b={"ABC"  "D  "  "EF ",
   " GH"  " I "  " JK"};
a=rowcatc(b);
```

produce the matrix **A** as shown:

```
    A          2 rows      1 col      (character, size 9)

                        ABCDEF
                        GHIJK
```

Quotes (") are needed only if you want to embed blanks or special characters or to maintain uppercase and lowercase distinctions.

RUN Statement

executes statements in a module

> **RUN** *<name> <***(***arguments***)***>***;**

The inputs to the RUN statement are as follows:

name   is the name of a user-defined module or an IML built-in subroutine.

arguments   are arguments to the subroutine. Arguments can be both local and
                global.

The RUN statement executes a user-defined module or invokes PROC IML's built-in
subroutines.

The resolution order for the RUN statement is

1. A user-defined module
2. An IML built-in function or subroutine

This resolution order need only be considered if you have defined a module that has
the same name as an IML built-in subroutine. If a RUN statement cannot be resolved
at resolution time, a warning is produced. If the RUN statement is still unresolved
when executed and a storage library is open at the time, IML attempts to load a
module from that storage. If no module is found, then the program is interrupted and
an error message is generated. By default, the RUN statement tries to run the module
named MAIN.

You will usually want to supply both a name and arguments, as follows.

```
run myf1(a,b,c);
```

See Chapter 5, "Programming Statements," for further details.

RUPDT Call

update QR and Cholesky decompositions

> **CALL RUPDT(** *rup, bup, sup,* r, z $<$, b, y $<$, *ssq*$>>$**);**

See the entry for the RDODT subroutine for details.

RZLIND Call

**computes rank deficient linear least-squares solutions, complete orthogonal fac-
torization, and Moore-Penrose inverses**

CALL RZLIND(*lindep, rup, bup, r* <, *sing*><, *b*>**);**

The RZLIND subroutine returns the following values:

lindep	is a scalar giving the number of linear dependencies that are recognized in $\mathbf{R}$ (number of zeroed rows in `rup[n,n]`).
rup	is the updated $n \times n$ upper triangular matrix $\mathbf{R}$ containing zero rows corresponding to zero recognized diagonal elements in the original $\mathbf{R}$.
bup	is the $n \times p$ matrix $\mathbf{B}$ of right-hand sides that is updated simultaneously with $\mathbf{R}$. If b is not specified, *bup* is not accessible.

The inputs to the RZLIND subroutine are as follows:

r	specifies the $n \times n$ upper triangular matrix $\mathbf{R}$. Only the upper triangle of r is used; the lower triangle may contain any information.
sing	is an optional scalar specifying a relative singularity criterion for the diagonal elements of $\mathbf{R}$. The diagonal element r_{ii} is considered zero if $r_{ii} \leq sing\|\mathbf{r}_i\|$, where $\|\mathbf{r}_i\|$ is the Euclidean norm of column $\mathbf{r}_i$ of $\mathbf{R}$. If the value provided for *sing* is not positive, the default value *sing*$= 1000\epsilon$ is used, where ϵ is the relative machine precision.
b	specifies the optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with $\mathbf{R}$.

The singularity test used in the RZLIND subroutine is a relative test using the Euclidean norms of the columns $\mathbf{r}_i$ of $\mathbf{R}$. The diagonal element r_{ii} is considered as nearly zero (and the ith row is zeroed out) if the following test is true:

$$r_{ii} \leq sing\|\mathbf{r}_i\|, \quad \text{where } \|\mathbf{r}_i\| = \sqrt{\mathbf{r}_i'\mathbf{r}_i}$$

Providing an argument *sing*≤ 0 is the same as omitting the argument *sing* in the RZLIND call. In this case, the default is *sing*$= 1000\epsilon$, where ϵ is the relative machine precision. If $\mathbf{R}$ is computed by the QR decomposition $\mathbf{A} = \mathbf{QR}$, then the Euclidean norm of column i of $\mathbf{R}$ is the same (except for rounding errors) as the Euclidean norm of column i of $\mathbf{A}$.

Consider the following possible application of the RZLIND subroutine. Assume that you want to compute the upper triangular Cholesky factor $\mathbf{R}$ of the $n \times n$ positive semidefinite matrix $\mathbf{A}'\mathbf{A}$,

$$\mathbf{A}'\mathbf{A} = \mathbf{R}'\mathbf{R} \text{ where } \mathbf{A} \in \mathcal{R}^{m \times n}, \text{ rank}(\mathbf{A}) = r, \ r \leq n \leq m$$

The Cholesky factor $\mathbf{R}$ of a positive definite matrix $\mathbf{A}'\mathbf{A}$ is unique (with the exception of the sign of its rows). However, the Cholesky factor of a positive semidefinite (singular) matrix $\mathbf{A}'\mathbf{A}$ can have many different forms.

In the following example, $\mathbf{A}$ is a 12×8 matrix with linearly dependent columns $\mathbf{a}_1 = \mathbf{a}_2 + \mathbf{a}_3 + \mathbf{a}_4$ and $\mathbf{a}_1 = \mathbf{a}_5 + \mathbf{a}_6 + \mathbf{a}_7$ with $r = 6$, $n = 8$, and $m = 12$.

```
a = {1 1 0 0 1 0 0,
     1 1 0 0 1 0 0,
     1 1 0 0 0 1 0,
     1 1 0 0 0 0 1,
     1 0 1 0 1 0 0,
     1 0 1 0 0 1 0,
     1 0 1 0 0 1 0,
     1 0 1 0 0 0 1,
     1 0 0 1 1 0 0,
     1 0 0 1 0 1 0,
     1 0 0 1 0 0 1,
     1 0 0 1 0 0 1};
a = a || uniform(j(12,1,1));
aa = a` * a;
m = nrow(a); n = ncol(a);
```

Applying the ROOT function to the coefficient matrix $\mathbf{A}'\mathbf{A}$ of the normal equations,

```
r1 = root(aa);
ss1 = ssq(aa - r1` * r1);
print ss1 r1 [format=best6.];
```

generates an upper triangular matrix $\mathbf{R}_1$ where linearly dependent rows are zeroed out, and you can verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_1'\mathbf{R}_1$.

Applying the QR subroutine with column pivoting on the original matrix $\mathbf{A}$ yields a different result, but you can also verify $\mathbf{A}'\mathbf{A} = \mathbf{R}_2'\mathbf{R}_2$ after pivoting the rows and columns of $\mathbf{A}'\mathbf{A}$:

```
ord = j(n,1,0);
call qr(q,r2,pivqr,lindqr,a,ord);
ss2 = ssq(aa[pivqr,pivqr] - r2` * r2);
print ss2 r2 [format=best6.];
```

Using the RUPDT subroutine for stepwise updating of $\mathbf{R}$ by the m rows of $\mathbf{A}$ will finally result in an upper triangular matrix $\mathbf{R}_3$ with $n - r$ nearly zero diagonal elements. However, other elements in rows with nearly zero diagonal elements can have significant values. The following statements verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_3'\mathbf{R}_3$,

```
r3 = shape(0,n,n);
call rupdt(rup,bup,sup,r3,a);
r3 = rup;
ss3 = ssq(aa - r3` * r3);
print ss3 r3 [format=best6.];
```

The result $\mathbf{R}_3$ of the RUPDT subroutine can be transformed into the result $\mathbf{R}_1$ of the ROOT function by left applications of Givens rotations to zero out the remaining significant elements of rows with *small* diagonal elements. Applying the RZLIND subroutine on the upper triangular result $\mathbf{R}_3$ of the RUPDT subroutine will generate a Cholesky factor $\mathbf{R}_4$ with zero rows corresponding to diagonal elements that are small, giving the same result as the ROOT function (except for the sign of rows) if its singularity criterion recognizes the same linear dependencies.

```
call rzlind(lind,r4,bup,r3);
ss4 = ssq(aa - r4' * r4);
print ss4 r4 [format=best6.];
```

Consider the rank-deficient linear least-squares problem:

$$\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|^2 \text{ where } \mathbf{A} \in \mathcal{R}^{m \times n}, \ \text{rank}(\mathbf{A}) = r, \ r \le n \le m$$

For $r = n$, the optimal solution, $\hat{\mathbf{x}}$, is unique; however, for $r < n$, the rank-deficient linear least-squares problem has many optimal solutions, each of which has the same least-squares residual sum of squares:

$$ss = (\mathbf{A}\hat{\mathbf{x}} - \mathbf{b})'(\mathbf{A}\hat{\mathbf{x}} - \mathbf{b})$$

The solution of the full-rank problem, $r = n$, is illustrated in the QR call. The following list shows several solutions to the singular problem. This example uses the 12×8 matrix from the preceding section and generates a new column vector $\mathbf{b}$. The vector $\mathbf{b}$ and the matrix $\mathbf{A}$ are shown in the output.

```
b = uniform(j(12,1,1));
ab = a' * b;
print b a [format=best6.];
```

Each entry in the following list solves the rank-deficient linear least-squares problem. Note that while each method minimizes the residual sum of squares, not all of the given solutions are of minimum Euclidean length.

- Use the singular value decomposition of $\mathbf{A}$, given by $\mathbf{A} = \mathbf{UDV}'$. 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 $\hat{\mathbf{x}}_1$ obtained by singular value decomposition, $\hat{\mathbf{x}}_1 = \mathbf{VD}^-\mathbf{U}'\mathbf{b}/4$, is of minimum Euclidean length.

- Use QR decomposition with column pivoting:

$$\mathbf{A}\mathbf{\Pi} = \mathbf{QR} = \begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \mathbf{Y} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \end{bmatrix}$$

Set the right part $\mathbf{R}_2$ to zero and invert the upper triangular matrix $\mathbf{R}_1$ to obtain a generalized inverse $\mathbf{R}^-$ and an optimal solution $\hat{\mathbf{x}}_2$:

$$\mathbf{R}^- = \begin{bmatrix} \mathbf{R}_1^{-1} \\ \mathbf{0} \end{bmatrix} \qquad \hat{\mathbf{x}}_2 = \mathbf{\Pi}\mathbf{R}^-\mathbf{Y}'\mathbf{b}$$

```
ord = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr = n - lindqr;
r = r2[1:nr,1:nr];
x2 = shape(0,n,1);
x2[pivqr] = trisolv(1,r,qtb[1:nr]) // j(lindqr,1,0.);
len2 = x2` * x2;
ss2 = ssq(a * x2 - b);
x2 = x2`;
print ss2 len2, x2 [format=best6.];
```

Note that the residual sum of squares is minimal, but the solution $\hat{\mathbf{x}}_2$ is not of minimum Euclidean length.

- Use the result $\mathbf{R}_1$ of the ROOT function on page 891 to obtain the vector *piv* indicating the zero rows in the upper triangular matrix $\mathbf{R}_1$:

```
r1 = root(aa);
nr = n - lind;
piv = shape(0,n,1);
j1 = 1; j2 = nr + 1;
do i=1 to n;
   if r1[i,i] ^= 0 then do;
      piv[j1] = i; j1 = j1 + 1;
   end;
   else do;
      piv[j2] = i; j2 = j2 + 1;
   end;
end;
```

Now compute $\hat{\mathbf{x}}_3$ by solving the equation $\hat{\mathbf{x}}_3 = \mathbf{R}^{-1}\mathbf{R}^{-\prime}\mathbf{A}'\mathbf{b}$.

```
r   = r1[piv[1:nr],piv[1:nr]];
x   = trisolv(2,r,ab[piv[1:nr]]);
x   = trisolv(1,r,x);
x3 = shape(0,n,1);
x3[piv] = x // j(lind,1,0.);
len3 = x3` * x3;
ss3   = ssq(a * x3 - b);
x3 = x3`;
print ss3 len3, x3 [format=best6.];
```

Note that the residual sum of squares is minimal, but the solution $\hat{\mathbf{x}}_3$ is not of minimum Euclidean length.

- Use the result $\mathbf{R}_3$ of the RUPDT call on page 891 and the vector *piv* (obtained in the previous solution), which indicates the zero rows of upper triangular matrices $\mathbf{R}_1$ and $\mathbf{R}_3$. After zeroing out the rows of $\mathbf{R}_3$ belonging to small diagonal pivots, solve the system $\hat{\mathbf{x}}_4 = \mathbf{R}^{-1}\mathbf{Y}'\mathbf{b}$.

```
                 r3 = shape(0,n,n);
                 qtb = shape(0,n,1);
                 call rupdt(rup,bup,sup,r3,a,qtb,b);
                 r3 = rup; qtb = bup;
                 call rzlind(lind,r4,bup,r3,,qtb);
                 qtb = bup[piv[1:nr]];
                 x   = trisolv(1,r4[piv[1:nr],piv[1:nr]],qtb);
                 x4 = shape(0,n,1);
                 x4[piv] = x // j(lind,1,0.);
                 len4 = x4` * x4;
                 ss4 = ssq(a * x4 - b);
                 x4 = x4`;
                 print ss4 len4, x4 [format=best6.];
```

Since the matrices $\mathbf{R}_4$ and $\mathbf{R}_1$ are the same (except for the signs of rows), the solution $\hat{\mathbf{x}}_4$ is the same as $\hat{\mathbf{x}}_3$.

- Use the result $\mathbf{R}_4$ of the RZLIND call in the previous solution, which is the result of the first step of *complete QR decomposition*, and perform the second step of complete QR decomposition. The rows of matrix $\mathbf{R}_4$ can be permuted to the upper trapezoidal form

$$\left[\begin{array}{cc} \widehat{\mathbf{R}} & \mathbf{T} \\ \mathbf{0} & \mathbf{0} \end{array} \right],$$

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

$$\left[\begin{array}{c} \widehat{\mathbf{R}}' \\ \mathbf{T}' \end{array} \right] = \bar{\mathbf{Y}} \left[\begin{array}{c} \bar{\mathbf{R}} \\ \mathbf{0} \end{array} \right],$$

which leads to the upper triangular matrix $\bar{\mathbf{R}}$.

```
                 r = r4[piv[1:nr],]`;
                 call qr(q,r5,piv2,lin2,r);
                 y   = trisolv(2,r5,qtb);
                 x5 = q * (y // j(lind,1,0.));
                 len5 = x5` * x5;
                 ss5 = ssq(a * x5 - b);
                 x5 = x5`;
                 print ss5 len5, x5 [format=best6.];
```

The solution $\hat{\mathbf{x}}_5$ obtained by complete QR decomposition has minimum Euclidean length.

- Perform both steps of complete QR decomposition. The first step performs the pivoted QR decomposition of $\mathbf{A}$,

$$\mathbf{A}\mathbf{\Pi} = \mathbf{Q}\mathbf{R} = \mathbf{Y} \left[\begin{array}{c} \mathbf{R} \\ \mathbf{0} \end{array} \right] = \mathbf{Y} \left[\begin{array}{c} \widehat{\mathbf{R}}\mathbf{T} \\ \mathbf{0} \end{array} \right]$$

where $\widehat{\mathbf{R}}$ is nonsingular and upper triangular and $\mathbf{T}$ is rectangular. The second step performs a QR decomposition as described in the previous method. This results in

$$\mathbf{A}\mathbf{\Pi} = \mathbf{Y} \left[\begin{array}{cc} \bar{\mathbf{R}}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{array} \right] \bar{\mathbf{Y}}'$$

where $\bar{\mathbf{R}}'$ is lower triangular.

```
ord  = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr   = n - lindqr;
r    = r2[1:nr,]';
call qr(q,r5,piv2,lin2,r);
y    = trisolv(2,r5,qtb[1:nr]);
x6   = shape(0,n,1);
x6[pivqr] = q * (y // j(lindqr,1,0.));
len6 = x6' * x6;
ss6  = ssq(a * x6 - b);
x6   = x6';
print ss6 len6, x6 [format=best6.];
```

The solution $\hat{\mathbf{x}}_6$ obtained by complete QR decomposition has minimum Euclidean length.

- Perform complete QR decomposition with the QR and LUPDT calls:

```
ord  = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr   = n - lindqr;
r    = r2[1:nr,1:nr]'; z = r2[1:nr,nr+1:n]';
call lupdt(lup,bup,sup,r,z);
rd   = trisolv(3,lup,r2[1:nr,]);
rd   = trisolv(4,lup,rd);
x7   = shape(0,n,1);
x7[pivqr] = rd' * qtb[1:nr,];
len7 = x7' * x7;
ss7  = ssq(a * x7 - b);
x7 = x7';
print ss7 len7, x7 [format=best6.];
```

The solution $\hat{\mathbf{x}}_7$ obtained by complete QR decomposition has minimum Euclidean length.

- Perform complete QR decomposition with the RUPDT, RZLIND, and LUPDT calls:

```
r3 = shape(0,n,n);
qtb = shape(0,n,1);
call rupdt(rup,bup,sup,r3,a,qtb,b);
r3 = rup; qtb = bup;
call rzlind(lind,r4,bup,r3,,qtb);
nr = n - lind; qtb = bup;
r = r4[piv[1:nr],piv[1:nr]]';
z = r4[piv[1:nr],piv[nr+1:n]]';
```

```
call lupdt(lup,bup,sup,r,z);
rd = trisolv(3,lup,r4[piv[1:nr],]);
rd = trisolv(4,lup,rd);
x8 = shape(0,n,1);
x8 = rd` * qtb[piv[1:nr],];
len8 = x8` * x8;
ss8 = ssq(a * x8 - b);
x8 = x8`;
print ss8 len8, x8 [format=best6.];
```

The solution $\hat{x}_8$ obtained by complete QR decomposition has minimum Euclidean length. The same result can be obtained with the APPCORT or COMPORT call.

You can use various methods to compute the Moore-Penrose inverse $\mathbf{A}^-$ of a rectangular matrix $\mathbf{A}$ using orthogonal methods. The entries in the following list find the Moore-Penrose inverse of the matrix $\mathbf{A}$ shown on page 892.

- Use the GINV operator. The GINV operator in IML uses the singular decomposition $\mathbf{A} = \mathbf{UDV'}$. The result $\mathbf{A}^- = \mathbf{VD}^-\mathbf{U'}$ should be identical to the result given by the next solution.

  ```
  ga = ginv(a);
  t1 = a * ga; t2 = t1`;
  t3 = ga * a; t4 = t3`;
  ss1 = ssq(t1 - t2) + ssq(t3 - t4) +
        ssq(t1 * a - a) + ssq(t3 * ga - ga);
  print ss1, ga [format=best6.];
  ```

- Use singular value decomposition. The singular decomposition $\mathbf{A} = \mathbf{UDV'}$ with $\mathbf{U'U} = \mathbf{I}_m$, $\mathbf{D} = \mathrm{diag}(d_i)$, and $\mathbf{V'V} = \mathbf{VV'} = \mathbf{I}_n$, can be used to compute $\mathbf{A}^- = \mathbf{VD}^\dagger\mathbf{U'}$, with $\mathbf{D}^\dagger = \mathrm{diag}(d_i^\dagger)$ and

$$d_i^\dagger = \left\{ \begin{array}{ll} 0 & \text{where } d_i \leq \epsilon \\ 1/d_i & \text{otherwise} \end{array} \right.$$

  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

$$\mathbf{A} = \mathbf{Y} \left[\begin{array}{cc} \bar{\mathbf{R}}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{array} \right] \bar{\mathbf{Y}}' \mathbf{\Pi}'$$

where $\bar{\mathbf{R}}'$ is lower triangular, yields the Moore-Penrose inverse

$$\mathbf{A}^- = \mathbf{\Pi} \bar{\mathbf{Y}} \left[\begin{array}{cc} \bar{\mathbf{R}}^{-\prime} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{array} \right] \mathbf{Y}'$$

```
ord = j(n,1,0);
call qr(q1,r2,pivqr,lindqr,a,ord);
nr = n - lindqr;
q1 = q1[,1:nr]; r = r2[1:nr,]`;
call qr(q2,r5,piv2,lin2,r);
tt = trisolv(4,r5`,q1`);
ga = shape(0,n,m);
ga[pivqr,] = q2 * (tt // shape(0,n-nr,m));
t1 = a * ga; t2 = t1`;
t3 = ga * a; t4 = t3`;
ss3 = ssq(t1 - t2) + ssq(t3 - t4) +
        ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss3;
```

- Use complete QR decomposition with QR and LUPDT:

```
ord = j(n,1,0);
call qr(q,r2,pivqr,lindqr,a,ord);
nr = n - lindqr;
r = r2[1:nr,1:nr]`; z = r2[1:nr,nr+1:n]`;
call lupdt(lup,bup,sup,r,z);
rd = trisolv(3,lup,r2[1:nr,]);
rd = trisolv(4,lup,rd);
ga = shape(0,n,m);
ga[pivqr,] = rd` * q[,1:nr]`;
t1 = a * ga; t2 = t1`;
t3 = ga * a; t4 = t3`;
ss4 = ssq(t1 - t2) + ssq(t3 - t4) +
        ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss4;
```

- Use complete QR decomposition with RUPDT and LUPDT:

```
r3 = shape(0,n,n);
y = i(m); qtb = shape(0,n,m);
call rupdt(rup,bup,sup,r3,a,qtb,y);
r3 = rup; qtb = bup;
call rzlind(lind,r4,bup,r3,,qtb);
nr = n - lind; qtb = bup;
r = r4[piv[1:nr],piv[1:nr]]`;
```

```
            z = r4[piv[1:nr],piv[nr+1:n]]';
            call lupdt(lup,bup,sup,r,z);
            rd = trisolv(3,lup,r4[piv[1:nr],]);
            rd = trisolv(4,lup,rd);
            ga = shape(0,n,m);
            ga = rd' * qtb[piv[1:nr],];
            t1 = a * ga; t2 = t1';
            t3 = ga * a; t4 = t3';
            ss5 = ssq(t1 - t2) + ssq(t3 - t4) +
                  ssq(t1 * a - a) + ssq(t3 * ga - ga);
            print ss5;
```

SAVE Statement

saves data

> **SAVE;**

The SAVE statement forces out any data residing in output buffers for all active output data sets and files to ensure that the data are written to disk. This is equivalent to closing and then reopening the files.

SEQ, SEQSCALE, and SEQSHIFT Calls

perform discrete sequential tests

> **CALL SEQ(** *prob, domain* $<$, $<$**TSCALE=***tscale*$><$, $<$**EPS=***eps*$>$
>
> $<$, $<$**DEN=***den*$>>>>$**);**
>
> **CALL SEQSCALE(** *prob, gscale, domain, level*$<$, $<$**IGUESS=***iguess*$>$
>
> $<$, $<$**TSCALE=***tscale*$><$, $<$**EPS=***eps*$><$, $<$**DEN=***den*$>>>>>$**);**
>
> **CALL SEQSHIFT(** *prob, shift, domain, plevel*$<$, $<$**IGUESS=***iguess*$>$
>
> $<$, $<$**TSCALE=***tscale*$><$, $<$**EPS=***eps*$><$, $<$**DEN=***den*$>>>>>$**);**

The SEQSHIFT subroutine returns the following values:

prob   is an $(m+1) \times n$ matrix. The $[i, j]$ entry in the array contains the probability at the $[i, j]$ entry of the argument *domain*. Also, the probability at infinity at every level j is returned in the last entry ($[m + 1, j]$) of column j. Upon a successful completion of any routine, this variable is always returned.

gscale   is a numeric variable that returns from the routine SEQSCALE and contains the scaling of the current geometry defined by *domain* that would yield a given significance level *level*.

shift   is a numeric variable that returns from the routine SEQSHIFT and contains the shift of current geometry defined by *domain* that would yield a given power level *plevel*.

The inputs to the SEQSHIFT subroutine are as follows:

domain   specifies an $m \times n$ matrix containing the boundary points separating the intervals of continuation/stopping of the sequential test. Each column k contains the boundary points at level k sorted in an ascending order, with .M and .P representing $-\infty$ and $+\infty$, respectively. They must start on the first row, and any remaining entries must be filled with a missing value. Elements that follow the missing value in any column will be ignored. The number of columns n is equal to the number of stages present in the sequential test. The row dimension m must be even, and it is equal to the maximum number of boundary points in a level. In fact, *domain* is the tabular form of the finite boundary points. Entries in *domain* with absolute values that exceed a standardized value of 8 at any level will be internally reset to a standardized value of 8 or -8, depending on the sign of the entry. This is reflected in the results returned for the probabilities and the densities.

tscale   specifies an optional $n - 1$ vector that describes the time intervals between two consecutive stages. In the absence of *tscale*, these time intervals will be internally set to 1. The IML keyword for *tscale* is TSCALE.

eps   specifies an optional numeric parameter for controlling the absolute precision of the computation. In the absence of *eps*, the precision will be internally set to 1E-7. The IML keyword for *eps* is EPS.

den   specifies an optional character string to describe the name of an $m \times n$ matrix. The $[i, j]$ entry in the matrix returns the density of the distribution at the $[i, j]$ entry of the matrix specified by the *domain* argument. The IML keyword for *den* is DEN.

iguess   specifies an optional numeric parameter that contains an initial guess for the variable *gscale* in the SEQSCALE subroutine or for the variable *mean* in the SEQSHIFT subroutine. In general, very good estimates for these initial guesses can be provided by an iterative process, and these estimates become extremely valuable near convergence. The IML keyword for *iguess* is IGUESS.

level   specifies a numeric parameter in the SEQSCALE subroutine that contains the required significance level to be achieved through scaling the *domain* (see the description of SEQSCALE).

plevel   specifies a numeric parameter in the SEQSHIFT subroutine that provides the required power level to be achieved through shifting the *domain* (see the description of SEQSHIFT).

SEQ Call

To compute the probability from a sequential test, you must specify a matrix containing the boundaries. With the optional additional information concerning the time intervals and the target accuracy, or their default values, the SEQ subroutine returns the matrix that contains the probability and optionally returns the density from a

sequential test evaluated at each given point of the boundary. Let C_j denote the continuation set at each level j. C_j is defined to be the union at the 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 call computes, with $\mu = 0$, the densities

$$f_j(s, \mu) = \int_{C_{j-1}} \phi(s - y, \mu, t_{j-1}) f_{j-1}(y, \mu) \, dy, \text{ for } j = 2, 3, \ldots$$

with

$$f_1(s, \mu) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{(s - \mu)^2}{2}\right]$$

and

$$\phi(s, \mu, t) = \frac{1}{\sqrt{2\pi t}} \exp\left[-\frac{(s - \mu)^2}{2t}\right]$$

with the associated probability at any point a at level j to be

$$P_j(a, \mu) = \int_{C_{j-1}} \Phi(a - y, \mu, t_j) f_{j-1}(y, \mu) \, dy, \text{ for } j = 2, 3, \ldots$$

with

$$\Phi(b, \mu, t) = \int_{-\infty}^{b} \phi(s, \mu, t) \, ds$$

The notation τ denotes the vector of time intervals $t_1, \ldots, t_{n-1}$, and $P_j(g, \mu, \tau)$ denotes the probability of continuation at the jth level for a given domain g, a given mean μ, and a given time vector τ. The variance at the jth level can be calculated from τ.

$$\begin{aligned} \sigma_1^2 &= 1 \\ \sigma_{j+1}^2 &= \sigma_j^2 + \tau_j, \text{ for } j = 1, 2, \ldots \end{aligned}$$

It is important to understand the limitations that are imposed internally on the domain by the numerical method. Any element g_{ij} will always be limited within a symmetric interval with standardized values not to exceed 8. That is,

$$g_{ij} = \max[\min(g_{ij}, 8\sigma_j), -8\sigma_j]$$

SEQSCALE Call

Given a domain g, an optional time vector τ, and a probability level p_s, the SEQSCALE subroutine finds the amount of scaling s that would solve the problem

$$P_n(gs, 0) = p_s$$

The result for the amount of scaling s is returned as the second argument of the SEQSCALE subroutine, *scale*. Note that because of the complexity of the problem, the SEQSCALE subroutine will not attempt to scale a domain with multiple intervals of continuation.

For a significance level of α, set $p_s = 1 - \alpha$.

SEQSHIFT Call

Given a geometry g, an optional time vector τ, and a power level $1 - \beta$, the SEQSHIFT subroutine finds the mean μ that solves $\mu \geq 0$ such that $P_n(g, \mu) = \beta$.

Actually, a simple transformation of the variables in the sequential problem yields the following result:

$$P_j(g^\mu, 0) = P_j(g, \mu), \text{ for } j = 1, 2, \ldots, n$$

where g^μ is given by $g_{ij}^\mu = g_{ij} - \mu j$.

Many options are available with the NLP family of optimization routines, which are described in Chapter 4, "Nonlinear Optimization Subroutines."

Consider the following continuation intervals:

$$
\begin{aligned}
C_1 &= \{-6, 2\} \\
C_2 &= \{-6, 3\} \\
C_3 &= \{-6, 4, 5, 6\} \\
C_4 &= \{-6, 4\}
\end{aligned}
$$

The following IML program computes the probability from the sequential test at each boundary point specified in the geometry.

```
/* function to insert in m the geometry column a at level k*/
start table(m,a,k);
   if ncol(m) = 0 & nrow(m) = 0  then    m = j(nrow(a),k,.);
   if nrow(m) < nrow(a) then  m = m// j(nrow(a)-nrow(m),ncol(m),.);
   if ncol(m) < k  then  m = m || j(nrow(m),k-ncol(m),.);
   m[1:nrow(a),k] = a;
finish;

call table(m,{-6,2},1);
call table(m,{-6,3},2);
call table(m,{-6,4,5,6},3);
call table(m,{-6,4},4);
```

```
call seq(prob,m) eps = 1.e-8 den="density";
print m;
print prob;
print density;
```

The following output displays the values returned for m, *prob* and *den*, respectively.

The probability at the level $k = 3$ at the point $x = 6$ is $prob[4, 3] = 0.96651$, while the density at the same point is $density[4, 3] = 0.0000524$.

Consider the continuation intervals

$$
\begin{aligned}
C_1 &= \{-20, 2\} \\
C_2 &= \{-20, 20\} \\
C_3 &= \{-3, 3\}
\end{aligned}
$$

Note that the continuation at level 2 can be effectively considered infinite, and it does not numerically affect the results of the computation at level 3. The following IML program verifies this by using the *tscale* parameter to compute this problem.

```
reset nocenter;
/* function to insert in m the geometry column a at level k*/
start table(m,a,k);
   if ncol(m) = 0 & nrow(m) = 0  then    m = j(nrow(a),k,.);
   if nrow(m) < nrow(a) then  m = m// j(nrow(a)-nrow(m),ncol(m),.);
   if ncol(m) < k  then  m = m || j(nrow(m),k-ncol(m),.);
   m[1:nrow(a),k] = a;
finish;

call table(m,{-20,2},1);
call table(m,{-20,20},2);
call table(m,{-3,3},3);

/***********************************/
/* TSCALE has the default value of 1  */
/***********************************/
call seq(prob1,m) eps = 1.e-8 den="density";
print m[format=f5.]  prob1[format=e12.5];

call table(mm,{-20,2},1);
call table(mm,{-3,3},2);
   /* We can show a 2-step separation between the levels */
   /* while dropping the intermediate level at 2           */
tscale = { 2 };
call seq(prob2,mm) eps = 1.e-8 den="density" TSCALE=tscale;
print mm[format=f5.]  prob2[format=e12.5];
```

The values returned for the variables m and *prob1* as well as mm and *prob2* are shown in the output.

Some internal limitations are imposed on the geometry. Consider the three-level case with geometry m in the preceding code. Since the *tscale* variable is not specified, it is

set to its default value, $(1, 1)$. The variance at the 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 IML program verifies for the given domain that the significance level is 0.05 and that the power is $1 - \beta$ under the alternative hypothesis:

```
/***************************************/
/* 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

$$-\Gamma j^{\delta} \le C_j \le \Gamma j^{\delta}$$

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

$$-\Gamma\sqrt{j} \leq C_j \leq \Gamma\sqrt{j}$$

for five groups. The overall significance level is α (the probability $palpha = 1 - \alpha$), and the power is $1 - \beta$.

```
%let nl = 5;
start func(plevel) global(level,scale,mean,palpha,beta,tn,asn);
    m        = sqrt((1: &nl));
    mm       =  -m //m;
    /*****************************/
    /* meet the significance level */
    /* by scaling                */
    /*****************************/

    call seqscale(prob,scale,mm,level);
    palpha  = (prob[2,]-prob[1,])[&nl];
    mm      = mm *scale;

    /*****************************/
    /* meet the power condition    */
    /*****************************/

    call seqshift(prob,mean,mm,plevel);
    return(mean);
finish;

/***************/
/* alpha = 0.95 */
/***************/

level   = 9.50000E-01;
bplevel = { 0.5 .25 .1 0.05 0.01};
free summary;
do i = 1 to 5;
```

```
        summary = summary || func(bplevel[i]);
    end;
    print summary[format=10.5];
```

The value returned for SUMMARY are shown in the following table, and the entries agree with Table 2 of Pocock (1977).

This example illustrates how to find the optimal boundary of the δ-class of Wang and Tsiatis (1987). The δ-class boundary has the form

$$-\Gamma j^{\delta} \leq C_{j} \leq \Gamma j^{\delta}$$

The δ-class boundary is optimal if it minimizes the average sample number while satisfying the required significance level α and the required power $1 - \beta$. You can use the following program to verify some of the results published in Tables 2 and 3 of Wang and Tsiatis (1987):

```
    %let nl=5;
    start func(delta) global(level,plevel,mean,
                             scale,alpha,beta,tn,asn);

    m        = ((1: &nl))##delta;
    mm       = (-m//m);

    /*****************************/
    /* meet the significance level */
    /*****************************/

    call seqscale(prob,scale,mm,level);
    alpha    = (prob[2,]-prob[1,])[&nl];
    mm       = mm *scale;

    /*****************************/
    /* meet the power condition     */
    /*****************************/

    call seqshift(prob,mean,mm,plevel);
    beta     = (prob[2,]-prob[1,])[&nl];

    /***********************************/
    /* compute the average sample number */
    /***********************************/

    p        = prob[3,]-prob[2,]+prob[1];
    tn       = 4*mean##2; /* number per group */
    asn      = tn *( &nl - p *(%eval(&nl-1):0)`);
    return(asn);
    finish;


    /*********************************************/
    /* set up the global variables needed by func */
```

```
          /*********************************************/

          level   = 0.95;
          plevel  = 0.01;

          /**************************************/
          /* set up the controlling options of the */
          /* optimization routine                */
          /**************************************/

          opt     = {0 2 0 1 6};
          tc      = repeat(.,1,12);
          tc[1]   = 100;
          tc[7]   = 1.e-4;
          par     = { 1.e-13 . 1.e-10 . . .} || . || epsd;


          /***************************/
          /* provide the initial guess */
          /* and let nlpdd do the work */
          /***************************/

          delta   = 0.5;
          call nlpdd(rc,rx,"func",delta) opt=opt tc=tc par=par;
```

The following output displays the results.

```
                        Optimization Start
                       Parameter Estimates

                                              Gradient
                                             Objective
            N Parameter         Estimate      Function

            1 X1               -1.500000      -8.09752

                 Value of Objective Function = 35.232023082




                  Double Dogleg Optimization
     Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)
                  Without Parameter Scaling
               Gradient Computed by Finite Differences
            Number of Parameter Estimates 1

               Parameter Estimates                  2
               Functions (Observations)             2

                        Optimization Start

     Active Constraints            0  Criterion = 35.232
     Max Abs Gradient Element   8.098  Radius = 1.000
```

| | | Function | Active | Objective |
Iter	Restart	Calls	Constraints	Function
1	0	3	0	34.8914
2*	0	4	0	34.8774
3*	0	5	0	34.8774

Iter	difcrit	maxgrad	lambda	slope
1	0.3406	1.644	49.273	-0.830
2*	0.0140	0.0440	0	-0.0144
3*	0.00001	0.00013	0	-1E-5

                    Optimization Results

Iterations	3	Function Calls	6
Gradient Calls	5	Active Constraints	0
Criterion	34.877417	Max Grad Element	0.000126832
Slope	-0.0000100034	Radius	1

   NOTE: FCONV convergence criterion satisfied.

            Optimization Results
            Parameter Estimates

N Parameter	Estimate	Gradient
1 X1	0.586554	-0.0001268

   Value of Objective Function = 34.877416815

The optimal function value of 34.88 agrees with the entry in Table 2 of Wang and Tsiatis (1987) for five groups, $\alpha = 0.05$, and $1 - \beta = 0.99$. Note that the variables *eps* and *tscale* are internally set to their default values. For more information on the NLPDD subroutine, see the section "NLPDD Call" on page 786. For details on the *opt*, *tc*, and *par* arguments in the NLPDD call, see the "Options Vector" section on page 343, the "Termination Criteria" section on page 349, and the section "Control Parameters Vector" on page 356, respectively.

You can replicate other values in Table 2 of Wang and Tsiatis (1987) by changing the values of the variables NL and PLEVEL. You can obtain values from Table 3 by changing the value of the variable LEVEL to 0.99 and specifying NL and PLEVEL accordingly.

This example illustrates how to find the boundaries that minimize ASN given the required significance level and the required power. It replicates some of the results published in Table 3 of Pocock (1982). The IML program computes the domain that

- minimizes the ASN
- yields a given significance level of 0.05
- yields a given power $1 - \beta$ under the alternative hypothesis

The last two nonlinear conditions on the optimization process can be incorporated as a penalty applied on the error in these nonlinear conditions. The following IML program does the computations for a power of 0.9.

```
%let nl=5;
start func(m) global(level,plevel,sigma,epss,
              geometry,stgeom,gscale,mean,alpha,beta,tn,asn);
   m       =  abs(m);
   mm      = ( -m  // m)*sigma;
   /*****************************/
   /* meet the significance level */
   /*****************************/

   call seqscale(prob,gscale,mm,level) iguess=gscale eps=epss;
   stgeom  = gscale*m;
   geometry= mm*gscale;

   alpha   = (prob[2,]-prob[1,])[&nl];

   /*****************************/
   /* meet the power condition    */
   /*****************************/

   call seqshift(prob,mean,geometry,plevel) iguess=mean eps=epss;
   beta    = (prob[2,]-prob[1,])[&nl];
   p       = prob[3,] - prob[2,]+prob[1,];

   /***********************************/
   /* compute the average sample number */
   /***********************************/

   tn      = 4*mean##2; /* number per group */
   asn     = tn *( &nl - p *(%eval(&nl-1):0)`);
   return(asn);
finish;

/********************************************/
/* set up the global variables needed by func */
/********************************************/
epss    = 1.e-8;
epso    = 1.e-5;
level   = 9.50000E-01;
plevel  = 0.05;
sigma   = diag(sqrt(1:5));

/****************************************/
/* set up the controlling options of the */
/* optimization routine                 */
```

```
/****************************************/

opt     = {0 2 0 1 6};
tc      = repeat(.,1,12);
tc[1]   = 100;
tc[7]   = 1.e-4;
par     = { 1.e-13 . 1.e-10 . . .} || . || epso;

/**********************************/
/* provide the constraint matrix    */
/* we need monotonically increasing */
/* significance levels              */
/**********************************/

con     = { .  .   .   .   .    .  . ,
            .  .   .   .   .    .  . ,
            1 -1   .   .   .    1  0 ,
            .  1  -1   .   .    1  0 ,
            .  .   1  -1   .    1  0 ,
            .  .   .   1  -1    1  0 };

/***************************/
/* provide the initial guess */
/* and let nlp do the work   */
/***************************/

m       = { 1  1  1  1  1  };
call nlpdd(rc,rx,"func",m) opt=opt blc = con tc=tc par=par;
print stgeom;
```

Note that while *eps* has been set to *eps*=10^{-8}, *tscale* has been internally set to its default value. You may choose to run the IML program with and without the specification of the keyword IGUESS to see the effect on the execution time.

Note the following about the optimization process:

- Different levels of precision are imposed on different modules. In this example, *epss*, which is used as the precision for the sequential tests, is 1E−8. The absolute and relative function criteria for the objective function are set to *par*[7]=1E−5 and *tc*[7]=1E−4, respectively. Since finite differences are used to compute the first and second derivatives, the sequential test should be more precise than the optimization routine. Otherwise, the finite difference estimation is worthless. Optimally, if the precision of the function evaluation is $O(\epsilon)$, the first- and second-order derivatives should be estimated with perturbations $O(\epsilon^{\frac{1}{2}})$ and $O(\epsilon^{\frac{1}{3}})$, respectively. For example, if all three precision levels are set to 1E−5, the optimization process does not work properly.

- Line search techniques that do not depend on the computation of the derivative are preferable.

- The amount of printed information from the optimization routines is controlled by *opt*[2] and can be set to any value between 0 and 3, with larger numbers representing more printed output.

SEQSCALE Call

perform discrete sequential tests

> **CALL SEQSCALE(** *prob, gscale, domain, level*<, <**IGUESS=***iguess*>
>
> <, <**TSCALE=***tscale*><, <**EPS=***eps*><, <**DEN=***den*>>>>>**);**

See the entry for the SEQ subroutine for details.

SEQSHIFT Call

perform discrete sequential tests

> **CALL SEQSHIFT(** *prob, shift, domain, plevel*<, <**IGUESS=***iguess*>
>
> <, <**TSCALE=***tscale*><, <**EPS=***eps*><, <**DEN=***den*>>>>>**);**

See the entry for the SEQ subroutine for details.

SETDIF Function

compares elements of two matrices

> **SETDIF(** *matrix1, matrix2***)**

The inputs to the SETDIF function are as follows:

matrix1	is a reference matrix. Elements of *matrix1* not found in *matrix2* are returned in a vector. It can be either numeric or character.
matrix2	is the comparison matrix. Elements of *matrix1* not found in *matrix2* are returned in a vector. It can be either numeric or character, depending on the type of *matrix1*.

The SETDIF function returns as a row vector the sorted set (without duplicates) of all element values present in *matrix1* but not in *matrix2*. If the resulting set is empty, the SETDIF function returns a null matrix (with zero rows and zero columns). The argument matrices and result can be either both character or both numeric. For character matrices, the element length of the result is the same as the element length of the *matrix1*. Shorter elements in the second argument are padded on the right with blanks for comparison purposes.

For example, the statements

```
a={1 2 4 5};
b={3 4};
c=setdif(a,b);
```

produce the result

```
            C              1 row      3 cols     (numeric)

                           1          2          5
```

SETIN Statement

makes a data set current for input

> **SETIN** *SAS-data-set* <**NOBS** *name*> <**POINT** *operand*>**;**

The inputs to the SETIN statement are as follows:

SAS-data-set   can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on data sets in *SAS Language Reference: Concepts*.

name   is the name of a variable to contain the number of observations in the data set.

operand   specifies the current observation.

The SETIN statement chooses the specified data set from among the data sets already opened for input by the EDIT or USE statement. This data set becomes the current input data set for subsequent data management statements. The NOBS option is not required. If specified, the NOBS option returns the number of observations in the data set in the scalar variable *name*. The POINT option makes the specified observation the current one. It positions the data set to a particular observation. The SHOW *datasets* command lists data sets already opened for input.

In the example that follows, if the data set WORK.A has 20 observations, the variable SIZE is set to 20. Also, the current observation is set to 10.

```
setin work.a nobs size point 10;
list;                  /* lists observation 10 */
```

SETOUT Statement

makes a data set current for output

> **SETOUT** *SAS-data-set* <**NOBS** *name*> <**POINT** *operand*>**;**

The inputs to the SETOUT statement are as follows:

SAS-data-set   can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on SAS data sets in *SAS Language Reference: Concepts*.

name   is the name of a variable to contain the number of observations in the data set.

operand   specifies the observation to be made the current observation.

The SETOUT statement chooses the specified data set from among those data sets already opened for output by the EDIT or CREATE statement. This data set becomes the current output data set for subsequent data management statements. If specified,

the NOBS option returns the number of observations currently in the data set in the scalar variable *name*. The POINT option makes the specified observation the current one.

In the example that follows, the data set WORK.A is made the current output data set and the fifth observation is made the current observation. The number of observations in WORK.A is returned in the variable SIZE.

```
setout work.a nobs size point 5;
```

SHAPE Function

reshapes and repeats values

SHAPE(*matrix*<, *nrow*<, *ncol*<, *pad-value*>>>**)**

The inputs to the SHAPE function are as follows:

matrix	is a numeric or character matrix or literal.
nrow	gives the number of rows of the new matrix.
ncol	gives the number of columns of the new matrix.
pad-value	is a fill value.

The SHAPE function shapes a new matrix from a matrix with different dimensions; *nrow* specifies the number of rows, and *ncol* specifies the number of columns in the new matrix. The operator works for both numeric and character operands. The three ways of using the function are outlined below:

- If only *nrow* is specified, the number of columns is determined as the number of elements in the object matrix divided by *nrow*. The number of elements must be exactly divisible; otherwise, a conformability error is diagnosed.

- If both *nrow* and *ncol* are specified, but not *pad-value*, the result is obtained moving along the rows until the desired number of elements is obtained. The operation cycles back to the beginning of the object matrix to get more elements, if needed.

- If *pad-value* is specified, the operation moves the elements of the object matrix first and then fills in any extra positions in the result with the *pad-value*.

If *nrow* or *ncol* is specified as 0, the number of rows or columns, respectively, becomes the number of values divided by *ncol* or *nrow*.

For example, the statement

```
r=shape(12,3,4);
```

produces the result shown:

```
                    R              3 rows      4 cols     (numeric)

                         12           12           12          12
                         12           12           12          12
                         12           12           12          12
```

The next statement

```
    r=shape(77,1,5);
```

produces the result matrix by moving along the rows until the desired number of elements is obtained, cycling back as necessary:

```
                R           1 row        5 cols     (numeric)

            77           77           77           77           77
```

The statement below

```
    r=shape({1 2, 3 4, 5 6},2);
```

has *nrow* specified and converts the 3×2 matrix into a 2×3 matrix.

```
                    R              2 rows      3 cols     (numeric)

                          1            2            3
                          4            5            6
```

The statement

```
    r=shape({99 31},3,3);
```

demonstrates the cycling back and repetition of elements in row-major order until the number of elements desired is obtained.

```
                    R              3 rows      3 cols     (numeric)

                         99           31           99
                         31           99           31
                         99           31           99
```

SHOW Statement

prints system information

SHOW *operands***;**

where *operands* are any of the valid operands to the SHOW statement. These are given below.

The SHOW statement prints system information. The following *operands* are available:

ALL	shows all the information included by OPTIONS, SPACE, DATASETS, FILES, and MODULES.
ALLNAMES	behaves like NAMES, but also shows names without values.
CONTENTS	shows the names and attributes of the variables in the current SAS data set.
DATASETS	shows all open SAS data sets.
FILES	shows all open files.
MEMORY	returns the size of the largest chunk of main memory available.
MODULES	shows all modules that exist in the current IML environment. A module already referenced but not yet defined is listed as undefined.
name	shows attributes of the specified matrix. If the name of a matrix is one of the SHOW keywords, then both the information for the keyword and the matrix are shown.
NAMES	shows attributes of all matrices having values. Attributes include number of rows, number of columns, data type, and size.
OPTIONS	shows current settings of all IML options (see the RESET statement).
PAUSE	shows the status of all paused modules that are pending resume.
SPACE	shows the workspace and symbolspace size and their current usage.
STORAGE	shows the modules and matrices in the current IML library storage.
WINDOWS	shows all active windows opened by WINDOW statements.

An example of a valid statement follows:

```
show all;
```

SOLVE Function

solves a system of linear equations

SOLVE(A, B**)**

The inputs to the SOLVE function are as follows:

A is an $n \times n$ nonsingular matrix.

B is an $n \times p$ matrix.

The SOLVE function solves the set of linear equations $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$. $\mathbf{A}$ must be square and nonsingular.

$\mathbf{X} = \text{SOLVE}(\mathbf{A}, \mathbf{B})$ is equivalent to using the INV function as $\mathbf{X} = \text{INV}(\mathbf{A}) * \mathbf{B}$. However, the SOLVE function is recommended over the INV function because it is more efficient and more accurate. An example follows:

```
x=solve(a,b);
```

The solution method used is discussed in Forsythe, Malcolm, and Moler (1967).

The SOLVE function (as well as the DET and INV functions) uses the following criterion to decide whether the input matrix, $\mathbf{A} = [a_{ij}]_{i,j=1,\ldots,n}$, is singular:

$$sing = 100 \times \textit{MACHEPS} \times \max_{1 \leq i,j \leq n} |a_{ij}|$$

where *MACHEPS* is the relative machine precision.

All matrix elements less than or equal to *sing* are now considered rounding errors of the largest matrix elements, so they are taken to be zero. For example, if a diagonal or triangular coefficient matrix has a diagonal value less than or equal to *sing*, the matrix is considered singular by the DET, INV, and SOLVE functions.

Previously, a much smaller singularity criterion was used, which caused algebraic operations to be performed on values that were essentially floating point error. This occasionally yielded numerically unstable results. The new criterion is much more conservative, and it generates far fewer erroneous results. In some cases, you may need to scale the data to avoid singular matrices. If you think the new criterion is too strong,

- try the GINV function to compute the generalized inverse.
- examine the size of the singular values returned by the SVD function. The SVD function can be used to compute a generalized inverse with a user-specified singularity criterion.

If 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

solves a sparse symmetric linear system by direct decomposition

CALL SOLVELIN(*x, status, A, b, method***);**

The SOLVELIN call returns the following values:

x   is the solution to $Ax = b$.

status   is the final status of the solution.

The inputs to the SOLVELIN call are as follows:

A   is the sparse coefficient matrix in the equation $Ax = b$.

b   is the right side of the equation $Ax = b$.

method   is the name of the decomposition to be used.

The SOLVELIN call uses direct decomposition to solve sparse symmetric linear systems. 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 call is unable to solve your system, you can try one of the iterative method calls ITCGRAD, ITBICG, or ITMINRES.

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 ITBICG call or ITMINRES call.

The following is an example of using 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:

```
              STATUS           X

                 0          2.68
                           -6.4
                            0.74
                            7.16
```

SORT Call

sort a matrix by specified columns

> **CALL SORT(** *matrix, by*<, *descend* >**)**

The inputs to the SORT call are as follows:

matrix is the input matrix, which is sorted in place by the call.

by is either a numeric matrix of column numbers, or a character matrix containing the names of columns corresponding to column labels assigned to *matrix* by a MATTRIB statement or READ statement.

descend is an optional matrix, indicating which columns, if any, should be 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 call is used to sort a matrix, rearranging its rows according to the columns and order determined by the *by* and *descend* inputs. Because the sort is done in place, very little additional memory space is required. The SORT call will not be as fast as the SORTNDX call for matrices with large 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,

```
    m = { 1 1 0,
          2 2 0,
          1 1 1,
          2 2 2};
    call sort( m, {1 3}, {3} );
```

results in

```
                        M

                1       1       1
                1       1       0
                2       2       2
                2       2       0
```

SORT Statement

sorts a SAS data set

> **SORT** <**DATA=**>*SAS-data-set* <**OUT=**\ *SAS-data-set*>
>
> > **BY** <**DESCENDING**> *variables***;**

where you can use the following clauses with the SORT statement:

DATA=*SAS-data-set*	names the SAS data set to be sorted. It can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on SAS data sets in *SAS Language Reference: Concepts*. Note that the DATA= portion of the specification is optional.
OUT=*SAS-data-set*	specifies a name for the output data set. If this clause is omitted, the DATA= data set is sorted and the sorted version replaces the original data set.
BY *variables*	specifies the variables to be sorted. A BY clause *must* be used with the SORT statement.
DESCENDING	specifies the variables are to be sorted in descending order.

The SORT statement sorts the observations in a SAS data set by one or more variables, stores the resulting sorted observations in a new SAS data set, or replaces the original. As opposed to all other IML data processing statements, it is *mandatory* that the data set to be sorted be closed prior to the execution of the SORT statement.

The SORT statement first arranges the observations in the order of the first variable in the BY clause; then it sorts the observations with a given value of the first variable by the second variable, and so forth. Every variable in the BY clause can be preceded by the keyword DESCENDING to denote that the variable that follows is to be sorted in descending order. Note that the SORT statement in IML always retains the same relative positions of the observations with identical BY variable values.

For example, the IML statement

```
sort class out=sclass by descending age height;
```

sorts the SAS data set CLASS by the variables AGE and HEIGHT, where AGE is sorted in descending order, and all observations with the same AGE value are sorted by HEIGHT in ascending order. The output data set SCLASS contains the sorted observations. When a data set is sorted in place (without the OUT= clause) any indexes associated with the data set become invalid and are automatically deleted.

Note that all the clauses of the SORT statement must be specified in the order given above.

SORTNDX Call

creates an index to re-order a matrix by specifed columns

> **CALL SORTNDX(** *index, matrix, by*<, *descend* >**)**

The SORTNDX call 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 inputs to the SORTNDX call are as follows:

matrix   is the input matrix, which is not modified by the call.

by   is either a numeric matrix of column numbers, or a character matrix containing the names of columns corresponding to column labels assigned to *matrix* by a MATTRIB statement or READ statement.

descend   is an optional matrix, indicating which columns, if any, should be 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 call can be used to process the rows of a matrix in different sorted order, without having to actually modify it.

For example,

```
m = { 1 1 0,
      2 0 0,
      1 3 1,
      2 2 2 };
call SORTNDX( ndx, m, {1 3}, {3} );
```

results in

```
                              NDX


                               3
                               1
                               4
                               2
```

The matrix can be physically sorted with (see also the SORT call):

```
    call  SORTNDX( ndx, m, by );
    m = m[ndx,];
```

The SORTNDX call can be used with the UNIQUEBY function to extract the unique combinations of values in the *by* columns.

SOUND Call

produces a tone

> **CALL SOUND(** *freq*<, *dur*>**);**

The inputs to the SOUND subroutine are as follows:

freq   is a numeric matrix or literal giving the frequency in hertz.

dur   is a numeric matrix or literal giving the duration in seconds. Note that the *dur* argument differs from that in the DATA step.

The SOUND subroutine generates a tone using *freq* for frequency (in hertz) and *dur* for duration (in seconds). Matrices may be specified for frequency and duration to produce multiple tones, but if both arguments are nonscalar, then the number of elements must match. The duration argument is optional and defaults to 0.25 (one quarter second).

For example, the following statements produce tones from an ascending musical scale, all with a duration of 0.2 seconds:

```
    notes=400#(2##do(0, 1, 1/12));
    call sound(notes,0.2);
```

SPLINE and SPLINEC Calls

provide cubic spline fits

> **CALL SPLINE(** *fitted, data*<, *smooth*><, *delta*><, *nout*>
>
> > <, *type*><, *slope*>**);**

CALL SPLINEC(*fitted, coeff, endSlopes, data*<, *smooth*><, *delta*>

<, *nout*><, *type*><, *slope*>**);**

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 call returns *coeff*.

endSlopes   is a 1×2 matrix containing the slopes of the two ends of the curve expressed as angles in degrees. The SPLINEC call returns the *endSlopes* argument.

The inputs to the SPLINEC subroutine are as follows:

data   specifies a $n \times 2$ (or $n \times 3$) matrix of (x, y) points on which the spline is to be fit. The optional third column is used to specify a weight for each data point. If *smooth* > 0, the weight column is used in calculations. A weight ≤ 0 causes the data point to be ignored in calculations.

smooth   is an optional scalar specifying the degree of smoothing to be used. If *smooth* is omitted or set equal to 0, then a cubic interpolating spline is fit to the data. If *smooth* > 0, then a cubic spline is used. Larger values of *smooth* generate more smoothing.

delta   is an optional scalar specifying the resolution constant. If *delta* is specified, the fitted points are spaced by the amount *delta* on the scale of the first column of *data* if a regular spline is used or on the scale of the curve length if a parametric spline is used. If both *nout* and *delta* are specified, *nout* is used and *delta* is ignored.

nout   is an optional scalar specifying the number of fitted points to be computed. The default is *nout*=200. If *nout* is specified, then *nout* equally spaced points are returned. The *nout* argument overrides the *delta* argument.

type   is an optional 1×1 (or 1×2) character matrix or quoted literal giving the type of spline to be used. The first element of *type* should be one of the following:

- `periodic`, which requests periodic endpoints
- `zero`, which sets second derivatives at endpoints to 0

The *type* argument controls the endpoint constraints unless the *slope* argument is specified. If `periodic` is specified, the response values at the beginning and end of column 2 of *data* must be the same unless the smoothing spline is being used. If the values are not the same, an error message is printed and no spline is fit. The default value is `zero`. The second element of *type* should be one of the following.

- `nonparametric`, which requests a nonparametric spline
- `parametric`, which requests a parametric spline

If `parametric` is specified, a parameter sequence $\{t_i\}$ is formed as follows: $t_1 = 0$ and

$$t_i = t_{i-1} + \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Splines are then fit to both the first and second columns of *data*. The resulting splined values are paired to form the output. Changing the relative scaling of the first two columns of *data* changes the output because the sequence $\{t_i\}$ assumes Euclidean distance.

Note that if the points are not arranged in 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).

Refer to Stoer and Bulirsch (1980), Reinsch (1967), and Pizer (1975) for descriptions of the methods used to fit the spline. For simplicity, the explanation below 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 $\{a = x_1 < x_2 < \cdots < x_n = b\}$ 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 using `type='zero'`.

- Periodic conditions. If your data is 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 code gives 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 corresponding to that X value. As the *smooth* parameter gets very large, the spline will approximate a linear regression.

The following code computes two smoothing splines for the same data as above. 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 corresponding 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 (or *score*) an interpolating or smoothing spline at an arbitrary set of points. The following program shows how to construct the parameters corresponding to the original data using the formula specified in the documentation for the *type* argument. These parameters are used to construct the evenly-spaced parameters corresponding 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

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 code defines 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 will get back a vector that represents the evaluation of the spline along evenly spaced points. To get around this, the function evaluates the spline at the vector x `//` x and then takes 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 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

provides cubic spline evaluations

> **SPLINEV(** *coeff*<, *delta*<, *nout*>>**)**

The SPLINEV function returns a two-column matrix containing the points of evaluation in the first column and the corresponding fitted values of the spline in the second column.

The inputs to the SPLINEV function are as follows:

coeff   is an $n \times 5$ (or $n \times 9$) matrix of spline coefficients, as returned by the SPLINEC Call. The *coeff* argument should not contain missing values.

delta   is an optional vector specifying evaluation points. If *delta* is a scalar, the spline will be evaluated at equally spaced points *delta* apart. If *delta* is a vector arranged in ascending order, the spline will be evaluated at each of these values. Evaluation at a point outside the support of the spline results in a missing value in the output. If you specify the *delta* argument, you cannot specify the *nout* argument.

nout   is an optional scalar specifying the number of fitted points desired. The default is *nout*=200. If you specify the *nout* argument, you cannot specify the *delta* argument.

See the section "SPLINE and SPLINEC Calls" on page 921 for details and examples.

SPOT Function

calculates a column vector of spot rates given vectors of forward rates and times

> **SPOT(** *times,forward_rates***)**

The SPOT function returns an $n \times 1$ vector of (per-period) spot rates.

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 code

```
time=T(do(1,5,1));
forward=T(do(0.05,0.09,0.01));
spot=spot(time,forward);
print spot;
```

produces the following output:

```
      SPOT

        0.05
   0.0549882
   0.0599686
   0.0649413
   0.0699065
```

SQRSYM Function

converts a symmetric matrix to a square matrix

SQRSYM(*matrix*)

where *matrix* is a symmetric numeric matrix.

The SQRSYM function takes a matrix such as those generated by the SYMSQR function and transforms it back into a square matrix. The elements of the argument are unpacked into the lower triangle of the result and reflected across the diagonal into the upper triangle.

For example, the following statement

```
sqr=sqrsym(symsqr({1 2, 3 4}));
```

which is the same as

```
sqr=sqrsym({ 1, 3, 4} );
```

produces the result

```
                SQR           2 rows       2 cols    (numeric)

                                 1            3
                                 3            4
```

SQRT Function

calculates the square root

SQRT(*matrix***)**

where *matrix* is a numeric matrix or literal.

The SQRT function is the scalar function returning the positive square roots of each element of the argument. An example of a valid statement follows.

```
a = { 1 2 3 4 };
c=sqrt(a);
print c;
```

```
                              C

                1 1.4142136 1.7320508          2
```

SSQ Function

calculates the sum of squares of all elements

SSQ(*matrix1<, matrix2,..., matrix15>***)**

where *matrix* is a numeric matrix or literal.

The SSQ function returns as a single numeric value the (uncorrected) sum of squares for all the elements of all arguments. You can specify as many as 15 numeric argument matrices.

The SSQ function checks for missing arguments and does not include them in the accumulation. If all arguments are missing, the result is 0.

An example of a valid statement follows:

```
a={1 2 3, 4 5 6};
x=ssq(a);
```

START and FINISH Statements

define a module

START *<name> <(arguments)> <***GLOBAL(***arguments)>***;**

    *module statements***;**

FINISH *<name>***;**

The inputs to the START and FINISH statements are as follows:

name	is the name of a user-defined module.
arguments	are names of variable arguments to the module. Arguments can be either input variables or output (returned) variables. Arguments listed in the GLOBAL clause are treated as global variables. Otherwise, the arguments are local.
module statements	are statements making up the body of the module.

The START statement instructs IML to enter a module-collect mode to collect the statements of a module rather than execute them immediately. The FINISH statement signals the end of a module. Optionally, the FINISH statement can take the module name as its argument. When no *name* argument is given in the START statement, the module name MAIN is used by default. If an error occurs during module compilation, the module is not defined. See Chapter 5, "Programming Statements," for details.

The example below defines a module named MYMOD that has two local variables (A and B) and two global variables (X and Y). The module creates the variable Y from the arguments A, B, and X.

```
start mymod(a,b) global(x,y);
   y=a*x+b;
finish;
```

STOP Statement

stops execution of statements

   STOP;

The STOP statement stops the IML program, and no further matrix statements are executed. However, IML continues to execute if more statements are entered. See also the descriptions of the RETURN and ABORT statements.

If IML execution was interrupted by a PAUSE statement or by a break, the STOP statement clears all the paused states and returns to immediate mode.

IML supports STOP processing of both regular and function modules.

STORAGE Function

lists names of matrices and modules in storage

   STORAGE();

The STORAGE function returns a matrix of the names of all of the matrices and modules in the current storage library. The result is a character vector with each matrix or module name occupying a row. Matrices are listed before modules. The SHOW *storage* command separately lists all of the modules and matrices in storage.

For example, the following statements reset the current library storage to MYLIB and then print a list of the modules and matrices in storage:

```
reset storage="MYLIB";
```

Then issue the command below to get the resulting matrix:

```
a=storage();
print a;
```

STORE Statement

stores matrices and modules in library storage

> **STORE** <**MODULE=(***module-list***)**> <*matrix-list*>**;**

The inputs to the STORE statement are as follows.

module-list   is a list of module names.

matrix-list   is a list of matrix names.

The STORE statement stores matrices or modules in the storage library. For example, the following statement stores the modules A, B, and C and the matrix X:

```
store module=(A B C) X;
```

The special operand _ALL_ can be used to store all matrices or all modules. For example, the following statement stores all matrices and modules:

```
store _all_ module=_all_;
```

The storage library can be specified using the RESET *storage* command and defaults to WORK.IMLSTOR. The SHOW *storage* command lists the current contents of the storage library. The following statement stores all matrices:

```
store;
```

See Chapter 14, "Storage Features," and also the descriptions of the LOAD, REMOVE, RESET, and SHOW statements for related information.

SUBSTR Function

takes substrings of matrix elements

> **SUBSTR(** *matrix, position*<*, length*>**)**

The inputs to the SUBSTR function are as follows:

matrix   is a character matrix or quoted literal.

position   is a numeric matrix or scalar giving the starting position.

length   is a numeric matrix or scalar giving the length of the substring.

The SUBSTR function takes a character matrix as an argument along with starting positions and lengths and produces a character matrix with the same dimensions as the argument. Elements of the result matrix are substrings of the corresponding argument elements. Each substring is constructed using the starting *position* supplied. If a *length* is supplied, this length is the length of the substring. If no *length* is supplied, the remainder of the argument string is the substring.

The *position* and *length* arguments can be scalars or numeric matrices. If *position* or *length* is a matrix, its dimensions must be the same as the dimensions of the argument matrix or submatrix. If either one is a matrix, its values are applied to the substringing of the corresponding elements of the *matrix*. If *length* is supplied, the element length of the result is MAX(*length*); otherwise, the element length of the result is

$$\text{NLENG}(\textit{matrix}) - \text{MIN}(\textit{position}) + 1 .$$

The statements

```
B={abc def ghi, jkl mno pqr};
a=substr(b,3,2);
```

return the matrix

```
        A        2 rows      3 cols    (character, size 2)

                           C  F  I
                           L  O  R
```

The element size of the result is 2; the elements are padded with blanks.

SUM Function

sums all elements

> **SUM(** *matrix1<, matrix2,..., matrix15>***)**

where *matrix* is a numeric matrix or literal.

The SUM function returns as a single numeric value the sum of all the elements in all arguments. There can be as many as 15 argument matrices. The SUM function checks for missing values and does not include them in the accumulation. It returns 0 if all values are missing.

For example, the statements

```
a={2 1, 0 -1};
b=sum(a);
```

return the scalar

```
            B              1 row      1 col      (numeric)

                                       2
```

SUMMARY Statement

computes summary statistics for SAS data sets

$$\textbf{SUMMARY} \ <\textbf{CLASS} \ operand> \ <\textbf{VAR} \ operand> \ <\textbf{WEIGHT} \ operand>$$

$$<\textbf{STAT} \ operand> \ <\textbf{OPT} \ operand> \ <\textbf{WHERE(}expression\textbf{)}>;$$

where the *operands* used by most clauses take either a matrix name, a matrix literal, or an expression yielding a matrix name or value. A discussion of the clauses and *operands* follows.

The SUMMARY statement computes statistics for numeric variables for an entire data set or a subset of observations in the data set. The statistics can be stratified by the use of class variables. The computed statistics are displayed in tabular form and optionally can be saved in matrices. Like most other IML data processing statements, the SUMMARY statement works on the current data set.

The following options are available with the SUMMARY statement:

CLASS *operand*

specifies the variables in the current input SAS data set to be used to group the summaries. The *operand* is a character matrix containing the names of the variables, for example,

```
summary class { age sex} ;
```

Both numeric and character variables can be used as class variables.

VAR *operand*

calculates statistics for a set of numeric variables from the current input data set. The *operand* is a character matrix containing the names of the variables. Also, the special keyword _NUM_ can be used as a VAR operand to specify all numeric variables. If the VAR clause is missing, the SUMMARY statement produces only the number of observations in each class group.

WEIGHT *operand*

specifies a character value containing the name of a numeric variable in the current data set whose values are to be used to weight each observation. Only one variable can be specified.

STAT *operand*

computes the statistics specified. The *operand* is a character matrix containing the names of statistics. For example, to get the mean and standard deviation, specify

```
summary stat{mean std};
```

Below is a list of the keywords that can be specified as the STAT *operand*:

CSS   computes the corrected sum of squares.

MAX   computes the maximum value.

MEAN   computes the mean.

MIN   computes the minimum value.

N   computes the number of observations in the subgroup used in the computation of the various statistics for the corresponding analysis variable.

NMISS   computes the number of observations in the subgroup having missing values for the analysis variable.

STD   computes the standard deviation.

SUM   computes the sum.

SUMWGT   computes the sum of the WEIGHT variable values if WEIGHT is specified; otherwise, IML computes the number of observations used in the computation of statistics.

USS   computes the uncorrected sum of squares.

VAR   computes the variance.

When the STAT clause is omitted, the SUMMARY statement computes these statistics for each variable in the VAR clause:

- MAX
- MEAN
- MIN
- STD.

Note that NOBS, the number of observations in each CLASS group, is always given.

OPT *operand*

sets the PRINT or NOPRINT and SAVE or NOSAVE options. The NOPRINT option suppresses the printing of the results from the SUMMARY statement. The SAVE option requests that the SUMMARY statement save the resultant statistics in matrices. The *operand* is a character matrix containing one or more of the options.

When the SAVE option is set, the SUMMARY statement creates a class vector for each class variable, a statistic matrix for each analysis variable, and a column vector named _NOBS_. The class vectors are named by the corresponding class variable and have an equal number of rows. There are as many rows as there are subgroups defined by the interaction of all class variables. The statistic matrices are named by the corresponding analysis variable. Each column of the statistic matrix corresponds to a statistic requested, and each row corresponds to the statistics of the subgroup defined by the class variables. If no class variable has been specified, each statistic matrix has one row, containing the statistics of the entire population. The _NOBS_ vector contains the number of observations for each subgroup.

The default is PRINT NOSAVE.

WHERE *expression*

conditionally selects observations, within the *range* specification, according to conditions given in *expression*. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable   is a variable in the SAS data set.

comparison-op   is one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
ˆ=	not equal to
?	contains a given string
ˆ?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand   is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

   ˆ= ˆ? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

   = ? =: =*

Logical expressions can be specified within the WHERE clause, using the AND (&) and OR (|) operators. The general form is

   clause&clause   (for an AND clause)
   clause|clause   (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

See Chapter 6, "Working with SAS Data Sets," for an example using the SUMMARY statement.

SVD Call

computes the singular value decomposition

> **CALL SVD(** u, q, v, a **);**

In the SVD subroutine:

a is the input matrix that is decomposed as described below.

u, q, and v are the returned decomposition matrices.

The SVD subroutine decomposes a real $m \times n$ matrix $\mathbf{A}$ (where m is greater than or equal to n) into the form

$$\mathbf{A} = \mathbf{U} * \mathrm{diag}(\mathbf{Q}) * \mathbf{V}'$$

where

$$\mathbf{U}'\mathbf{U} = \mathbf{V}'\mathbf{V} = \mathbf{V}\mathbf{V}' = \mathbf{I}_n$$

and $\mathbf{Q}$ contains the singular values of $\mathbf{A}$. $\mathbf{U}$ is $m \times n$, $\mathbf{Q}$ is $n \times 1$, and $\mathbf{V}$ is $n \times n$.

When m is greater than or equal to n, $\mathbf{U}$ consists of the orthonormal eigenvectors of $\mathbf{A}\mathbf{A}'$, and $\mathbf{V}$ consists of the orthonormal eigenvectors of $\mathbf{A}'\mathbf{A}$. $\mathbf{Q}$ contains the square roots of the eigenvalues of $\mathbf{A}'\mathbf{A}$ and $\mathbf{A}\mathbf{A}'$, except for some zeros.

If m is less than n, a corresponding decomposition is done where $\mathbf{U}$ and $\mathbf{V}$ switch roles:

$$\mathbf{A} = \mathbf{U} * \mathrm{diag}(\mathbf{Q}) * \mathbf{V}'$$

but

$$\mathbf{U}'\mathbf{U} = \mathbf{U}\mathbf{U}' = \mathbf{V}'\mathbf{V} = \mathbf{I}_w \ .$$

The singular values are sorted in descending order.

For information about the method used in the SVD subroutine, refer to Wilkinson and Reinsch (1971). Consider the following example (Wilkinson and Reinsch 1971, p. 149):

```
a={22  10   2   3   7,
   14   7  10   0   8,
   -1  13  -1 -11   3,
   -3  -2  13  -2   4,
    9   8   1  -2   4,
    9   1  -7   5  -1,
    2  -6   6   5   1,
    4   5   0  -2   2};
call svd(u,q,v,a);
reset fuzz; /* print small numbers as zero */
zero = ssq(a - u*diag(q)*v`);
```

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

```
    U                8 rows       5 cols     (numeric)

  0.7071068 0.1581139 -0.176777 -0.212785 -0.560643
  0.5303301 0.1581139 0.3535534 0.0801354 0.3127085
  0.1767767 -0.790569 0.1767767  0.486821 -0.155628
          0 0.1581139 0.7071068 0.1118328 -0.175184
  0.3535534 -0.158114         0 -0.082888  0.348706
  0.1767767 0.1581139  -0.53033 0.5984857 0.1586763
          0 0.4743416 0.1767767 0.4882498 0.1463314
  0.1767767 -0.158114         0 -0.308798 0.6039844


    Q                5 rows       1 col      (numeric)

                     35.327043
                            20
                     19.595918
                     1.113E-15
                     5.079E-16


    V                5 rows       5 cols     (numeric)

  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
```

The SVD routine performs most of its computations in the memory allocated for returning the singular value decomposition.

SWEEP Function

sweeps a matrix

SWEEP(*matrix, index-vector***)**

The inputs to the SWEEP function are as follows:

matrix　　　　is a numeric matrix or literal.

index-vector　　is a numeric vector indicating the pivots.

The SWEEP function sweeps *matrix* on the pivots indicated in *index-vector* to produce a new matrix. The values of the index vector must be less than or equal to the number of rows or the number of columns in *matrix*, whichever is smaller.

For example, suppose that $\mathbf{A}$ is partitioned into

$$\begin{bmatrix} \mathbf{R} & \mathbf{S} \\ \mathbf{T} & \mathbf{U} \end{bmatrix}$$

such that $\mathbf{R}$ is $q \times q$ and $\mathbf{U}$ is $(m - q) \times (n - q)$. Let

```
I = [1 2 3 . . . q]
```

Then, the statement

```
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($\mathbf{A}$,1),2)=SWEEP($\mathbf{A}$,{ 1 2 })
- SWEEP(SWEEP($\mathbf{A}$,$\mathbf{I}$),$\mathbf{I}$)=$\mathbf{A}$

See Beaton (1964) for more information about these properties.

To use the SWEEP function for regression, suppose the matrix $\mathbf{A}$ contains

$$\begin{bmatrix} \mathbf{X}'\mathbf{X} & \mathbf{X}'\mathbf{Y} \\ \mathbf{Y}'\mathbf{X} & \mathbf{Y}'\mathbf{Y} \end{bmatrix}$$

where $\mathbf{X}'\mathbf{X}$ is $k \times k$.

Then $\mathbf{B} = \text{SWEEP}(\mathbf{A}, 1 \ldots k)$ contains

$$\begin{bmatrix} (\mathbf{X}'\mathbf{X})^{-1} & (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y} \\ -\mathbf{Y}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} & \mathbf{Y}'(\mathbf{I} - \mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}')\mathbf{Y} \end{bmatrix}$$

The partitions of $\mathbf{B}$ form the beta values, SSE, and a matrix proportional to the co-variance of the beta values for the least-squares estimates of $\mathbf{B}$ in the linear model

$$\mathbf{Y} = \mathbf{X}\mathbf{B} + \epsilon .$$

If any pivot becomes very close to zero (less than or equal to $1\text{E}-12$), the row and column for that pivot are zeroed. See Goodnight (1979) for more information.

An example of using 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

converts a square matrix to a symmetric matrix

> **SYMSQR(** *matrix***)**

where *matrix* is a square numeric matrix.

The SYMSQR function takes a square numeric matrix (size $n \times n$) and compacts the elements from the lower triangle into a column vector ($n(n+1)/2$ rows). The matrix is not checked for actual symmetry.

Therefore, the statement

```
sym=symsqr({1 2, 3 4});
```

sets

```
                SYM            3 rows      1 col      (numeric)

                                            1
                                            3
                                            4
```

Note that the 2 is lost since it is only present in the upper triangle.

T Function

transposes a matrix

> **T(** *matrix***)**

where *matrix* is a numeric or character matrix or literal.

The T (transpose) function returns the transpose of its argument. It is equivalent to the transpose operator as written with a transpose postfix operator ('), but since some keyboards do not support the backquote character, this function is provided as an alternate.

For example, the statements

```
x={1 2, 3 4};
y=t(x);
```

result in the matrix

```
                Y              2 rows      2 cols     (numeric)

                                     1          3
                                     2          4
```

TEIGEN Call

computes the eigenvalues and eigenvectors of square matrices

The TEIGEN subroutine is an alias for the EIGEN subroutine.

TEIGVAL Function

compute eigenvalues of square matrices

The TEIGVAL function is an alias for the EIGVAL function.

TEIGVEC Function

compute eigenvectors of square matrices

The TEIGVEC function is an alias for the EIGVEC function.

TOEPLITZ Function

generates a Toeplitz or block-Toeplitz matrix

> **TOEPLITZ(** a **)**

where a is either a vector or a numeric matrix.

The TOEPLITZ function generates a Toeplitz matrix from a vector, or a block Toeplitz matrix from a matrix. A block Toeplitz matrix has the property that all matrices on the diagonals are the same. The argument a is an $(np) \times p$ or $p \times (np)$ matrix; the value returned is the $(np) \times (np)$ result.

The TOEPLITZ function uses the first $p \times p$ submatrix, $\mathbf{A}_1$, of the argument matrix as the blocks of the main diagonal. The second $p \times p$ submatrix, $\mathbf{A}_2$, of the argument matrix forms one secondary diagonal, with the transpose $\mathbf{A}_2'$ forming the other. The remaining diagonals are formed accordingly. If the first $p \times p$ submatrix of the argument matrix is symmetric, the result is also symmetric. If $\mathbf{A}$ is $(np) \times p$, the first p columns of the returned matrix, $\mathbf{R}$, will be the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, the first p rows of $\mathbf{R}$ will be the same as $\mathbf{A}$. The TOEPLITZ function is especially useful in time-series applications, where the covariance matrix of a set of variables with its lagged set of variables is often assumed to be a block Toeplitz matrix.

If

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2|\mathbf{A}_3|\cdots|\mathbf{A}_n]$$

and if $\mathbf{R}$ is the matrix formed by the TOEPLITZ function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & | & \mathbf{A}_2 & | & \mathbf{A}_3 & | & \cdots & | & \mathbf{A}_n \\ \mathbf{A}_2' & | & \mathbf{A}_1 & | & \mathbf{A}_2 & | & \cdots & | & \mathbf{A}_{n-1} \\ \mathbf{A}_3' & | & \mathbf{A}_2' & | & \mathbf{A}_1 & | & \cdots & | & \mathbf{A}_{n-2} \\ \vdots & & & & & & & & \\ \mathbf{A}_n' & | & \mathbf{A}_{n-1}' & | & \mathbf{A}_{n-2}' & | & \cdots & | & \mathbf{A}_1 \end{bmatrix}$$

If

$$
\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}
$$

and if **R** is the matrix formed by the TOEPLITZ function, then

$$
\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & | & \mathbf{A}_2' & | & \mathbf{A}_3' & | & \cdots & | & \mathbf{A}_n' \\ \mathbf{A}_2 & | & \mathbf{A}_1 & | & \mathbf{A}_2' & | & \cdots & | & \mathbf{A}_{n-1}' \\ \vdots & & & & & & & & \\ \mathbf{A}_n & | & \mathbf{A}_{n-1} & | & \mathbf{A}_{n-2} & | & \cdots & | & \mathbf{A}_1 \end{bmatrix}
$$

Three examples follow.

```
r=toeplitz(1:5);
```

```
          R               5 rows       5 cols     (numeric)

                1           2           3           4           5
                2           1           2           3           4
                3           2           1           2           3
                4           3           2           1           2
                5           4           3           2           1
```

```
r=toeplitz({1 2 ,
            3 4 ,
            5 6 ,
            7 8});
```

```
          R               4 rows       4 cols     (numeric)

                1           2           5           7
                3           4           6           8
                5           6           1           2
                7           8           3           4
```

```
r=toeplitz({1 2 3 4 ,
            5 6 7 8});
```

```
          R               4 rows       4 cols     (numeric)

                1           2           3           4
                5           6           7           8
                3           7           1           2
                4           8           5           6
```

TPSPLINE Call

computes thin-plate smoothing splines

CALL TPSPLINE(*fitted, coeff, adiag, gcv,* x, y <, *lambda*> **);**

The TSPLINE subroutine computes thin-plate smoothing spline (TPSS) fits to approximate smooth multivariate functions that are observed with noise. The generalized cross validation (GCV) function is used to select the smoothing parameter.

The TPSPLINE subroutine returns the following values:

fitted	is an $n \times 1$ vector of fitted values of the TPSS fit evaluated at the design points $\mathbf{x}$. The n is the number of observations. The final TPSS fit depends on the optional *lambda*.
coeff	is a vector of spline coefficients. The vector contains the coefficients for basis functions in the null space and the representer of evaluation functions at unique design points. (Refer to Wahba 1990 for more detail on reproducing kernel Hilbert space and representer of evaluation functions.) The length of *coeff* vector depends on the number of unique design points and the number of variables in the spline model. In general, let *nuobs* and k be the number of unique rows and the number of columns of $\mathbf{x}$ respectively. The length of *coeff* equals to $k + nuobs + 1$. The *coeff* vector can be used as an input of TPSPLNEV to evaluate the resulting TPSS fit at new data points.
adiag	is an $n \times 1$ vector of diagonal elements of the "hat" matrix. See the "Details" section.
gcv	If *lambda* is not specified, then *gcv* is the minimum value of the GCV function. If *lambda* is specified, then *gcv* is a vector (or scalar if *lambda* is a scalar) of GCV values evaluated at the *lambda* points. It provides users both with the ability to study the GCV curves by plotting *gcv* against *lambda*, and with the chance to identify a possible local minimum.

The inputs to the TPSPLINE subroutine are as follows:

x	is an $n \times k$ matrix of design points on which the TPSS is to be fit. The k is the number of variables in the spline model. The columns of x need to be linearly independent and contain no constant column.
y	is the $n \times 1$ vector of observations.
lambda	is a optional $q \times 1$ vector containing λ values in $log_{10}(n\lambda)$ scale. This option gives users the power to control how they want the TPSPLINE subroutine to function. If *lambda* is not specified (or *lambda* is specified and $q > 1$) the GCV function is used to choose the "best" λ and the returning *fitted* values are based on the λ that

minimizes the GCV function. If *lambda* is specified and $q = 1$, no minimization of the GCV function is involved and the *fitted*, *coeff* and *adiag* values are all based on the TPSS fit using this particular *lambda*. This gives users the freedom to choose the λ that they think appropriate.

Aside from the values returned, the TPSPLINE subroutine also prints other useful information such as the number of unique observations, the dimensions of the null space, the number of parameters in the model, a GCV estimate of λ, the smoothing penalty, the residual sum of square, the trace of $(I - A(\lambda))$, an estimate of σ^2, and the sum of squares for replication.

Note: No missing values are allowed within the input arguments. Also, you should use caution if you want to specify small *lambda* values. Since the true $\lambda = (10^{\log_{10} lambda})/n$, a very small value for *lambda* can cause λ to be smaller than the magnitude of machine error and usually the returned *gcv* values from such a λ cannot be trusted. Finally, when using TPSPLINE be aware that TPSS is a computationally intensive method. Therefore a large data set (that is, a large number of unique design points) will take a lot of computer memory and time.

For convenience, we illustrate the TPSS method with a two-dimensional independent variable $\mathbf{X} = (\mathbf{x}^1, \mathbf{x}^2)$. More details can be found in Wahba (1990), or in Bates, *et al.* (1987).

Assume that the data is from the model

$$y_i = f(\mathbf{x}_i) + \epsilon_i,$$

where $(\mathbf{x}_i, y_i), i = 1, \ldots, n$ are the observations. The function f is unknown and you assume that it is reasonably smooth. The error terms $\epsilon_i, i = 1, \ldots, n$ are independent zero-mean random variables.

You will measure the smoothness of f by the integral over the entire plane of the square of the partial derivatives of f of total order 2, that is

$$J_2(f) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \left[\frac{\partial^2 f}{\partial x_1^2}\right]^2 + 2\left[\frac{\partial^2 f}{\partial x_1 \partial x_2}\right]^2 + \left[\frac{\partial^2 f}{\partial x_2^2}\right]^2 \, dx_1 dx_2.$$

Using this as a smoothness penalty, the thin-plate smoothing spline estimate f_λ of f is the minimizer of

$$S_\lambda(f) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(\mathbf{x}_i))^2 + \lambda J_2(f).$$

Duchon (1976) derived that the minimizer f_λ can be represented as

$$f_\lambda(\mathbf{x}) = \sum_{i=1}^{3} \beta_i \phi_i(\mathbf{x}) + \sum_{i=1}^{n} \delta_i E_2(\mathbf{x} - \mathbf{x}_i),$$

where $(\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \phi_3(\mathbf{x})) = (1, \mathbf{x}^1, \mathbf{x}^2)$ and $E_2(\mathbf{s}) = \frac{1}{2^3\pi}\|\mathbf{s}\|^2 ln(\|\mathbf{s}\|)$.

Let matrix $\mathbf{K}$ have entries $(\mathbf{K})_{ij} = E_2(\mathbf{x}_i - \mathbf{x}_j)$ and matrix $\mathbf{T}$ have entries $(\mathbf{T})_{ij} = \phi_j(\mathbf{x}_i)$. Then the minimization problem can be rewritten as finding coefficients β and δ to minimize

$$S_\lambda(\beta, \delta) = \frac{1}{n}\|\mathbf{y} - \mathbf{T}\beta - \mathbf{K}\delta\|^2 + \lambda\delta^T\mathbf{K}\delta.$$

The final TPSS fits can be viewed as a type of generalized ridge regression estimator. The λ is called the smoothing parameter, which controls the balance between the goodness of fit and the smoothness of the final estimate. The smoothing parameter can be chosen by minimizing the Generalized Cross Validation function (GCV). If you write

$$\hat{\mathbf{y}} = \mathbf{A}(\lambda)\mathbf{y},$$

and call the $\mathbf{A}(\lambda)$ as the "*hat*" matrix, the GCV function $V(\lambda)$ is defined as

$$V(\lambda) = \frac{(1/n)\|(\mathbf{I} - \mathbf{A}(\lambda)\mathbf{y}\|^2}{[(1/n)\text{tr}(\mathbf{I} - \mathbf{A}(\lambda))]^2}.$$

The returned values from this function call will provide the $\hat{\mathbf{y}}$ as *fitted*, the (β, δ) as *coeff*, and $diag(A(\lambda))$ as *adiag*.

To evaluate the TPSS fit $f_\lambda(\mathbf{x})$ at new data points, you can use the TPSPLNEV call.

Suppose $\mathbf{X}^{\text{new}}$, a $m \times k$ matrix, contains the m new data points at which you want to evaluate f_λ. Let $(\mathbf{T}_{ij}^{\text{new}}) = \phi_j(\mathbf{x}_i^{\text{new}})$ and $(\mathbf{K}_{ij}^{\text{new}}) = E_2(\mathbf{x}_i^{\text{new}} - \mathbf{x}_j)$ be the ij^{th} elements of $\mathbf{T}^{\text{new}}$ and $\mathbf{K}^{\text{new}}$ respectively. The prediction at new data points $\mathbf{X}^{\text{new}}$ is

$$\mathbf{y}_{\text{pred}} = T^{\text{new}}\beta + K^{\text{new}}\delta.$$

Therefore, using the coefficient (β, δ) obtained from TPSPLINE call, the $\mathbf{y}_{\text{pred}}$ can be easily evaluated.

An example is given in the documentation for the TPSPLNEV call.

TPSPLNEV Call

evaluates the thin-plate smoothing spline at new data points

It can be used only after the TPSPLINE call.

    CALL TPSPLNEV(*pred, xpred, x, coeff***);**

The TPSPLNEV subroutine returns the following value:

pred   is an $m \times 1$ vector of the predicated values of the TPSS fit evaluated at m new data points.

The inputs to the TPSPLNEV subroutine are as follows:

xpred   is an $m \times k$ matrix of data points at which the f_λ is evaluated, where m is the number of new data points and k is the number of variables in the spline model.

x   is an $n \times k$ matrix of design points 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 20.1.

```
x={ -1.0 -1.0,     -1.0 -1.0,      -.5 -1.0,       -.5 -1.0,
     .0 -1.0,       .0 -1.0,       .5 -1.0,        .5 -1.0,
    1.0 -1.0,      1.0 -1.0,      -1.0 -.5,       -1.0 -.5,
     -.5 -.5,      -.5 -.5,        .0 -.5,         .0 -.5,
      .5 -.5,       .5 -.5,       1.0 -.5,        1.0 -.5,
    -1.0 .0,       -1.0 .0,        -.5 .0,         -.5 .0,
      .0 .0,         .0 .0,         .5 .0,          .5 .0,
     1.0 .0,        1.0 .0,       -1.0 .5,        -1.0 .5,
     -.5 .5,        -.5 .5,         .0 .5,          .0 .5,
      .5 .5,         .5 .5,        1.0 .5,         1.0 .5,
    -1.0 1.0,      -1.0 1.0,       -.5 1.0,        -.5 1.0,
      .0 1.0,        .0 1.0,        .5 1.0,         .5 1.0,
     1.0 1.0,       1.0 1.0 };

y={15.54483570,  15.76312613,   18.67397826,    18.49722167,
   19.66086310,  19.80231311,   18.59838649,    18.51904737,
   15.86842815,  16.03913832,   10.92383867,    11.14066546,
   14.81392847,  14.82830425,   16.56449698,    16.44307297,
   14.90792284,  15.05653924,   10.91956264,    10.94227538,
   9.614920104,  9.646480938,   14.03133439,    14.03122345,
   15.77400253,  16.00412514,   13.99627680,    14.02826553,
   9.557001644,  9.584670472,   11.20625177,    11.08651907,
   14.83723493,  14.99369172,   16.55494349,    16.51294369,
   14.98448603,  14.71816070,   11.14575565,    11.17168689,
   15.82595514,  15.96022497,   18.64014953,    18.56095997,
   19.54375504,  19.80902641,   18.56884576,    18.61010439,
   15.86586951,  15.90136745 };
```

Now generate a sequence of λ from -3.8 to -3.3 so that we can study the GCV function within this range.

```
lambda=T(do(-3.8,-3.3,0.1));
```

Use the following IML statement to do the thin-plate smoothing spline fit and returning the fitted values on those design points.

```
call tpspline(fit,coef,adiag,gcv, x, y,lambda);
```

The output from this call follows.

```
              SUMMARY OF TPSPLINE CALL

    Number of observations                  50
    Number of unique design points          25
    Dimension of polynomial Space            3
    Number of Parameters                    28



    GCV Estimate of Lambda          0.00000668
    Smoothing Penalty               2558.14323
    Residual Sum of Squares            0.24611
    Trace of (I-A)                    25.40680
    Sigma^2 estimate                   0.00969
    Sum of Squares for Replication     0.24223
```

After this TPSPLINE call, you obtained the fitted value. The fitted surface is plotted in the second panel of Figure 20.1. Also in Figure 20.1, panel 4, you plot the GCV function values against *lambda*. From panel 2, you see that because of the spare design points, the fitted surface is a little bit rough. In order to study the TPSS fit $f_\lambda(\mathbf{x})$ more closely, you use the following IML statements to generate a more dense grid on $[-1, 1] \times [-1, 1]$.

```
do i1=-1 to 1 by 0.1;
   do i2=-1 to 1 by 0.1;
      x1=x1||i1;
      x2=x2||i2;
   end;
end;
x1=t(x1);
x2=t(x2);
xpred=x1||x2;
```

Now you can use the function TPSPLNEV to evaluate $f_\lambda(\mathbf{x})$ on this dense grid.

```
call tpsplnev(pred,  xpred, x, coef);
```

The final fitted surface is plotted in Figure 20.1, panel 3.

Figure 20.1. Plots of Fitted Surface

TRACE Function

sums diagonal elements

TRACE(*matrix***)**

where *matrix* is a numeric matrix or literal.

The TRACE function produces a single numeric value that is the sum of the diagonal elements of *matrix*. For example, the statement

```
a=trace({5 2, 1 3});
```

produces the result

```
          A              1 row        1 col      (numeric)
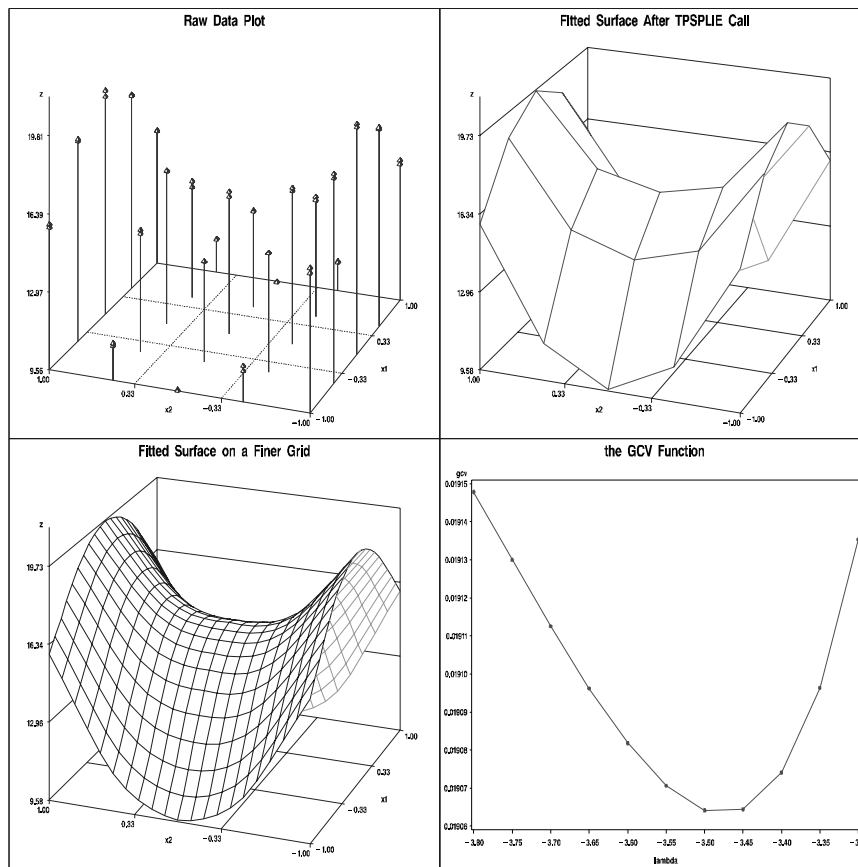
                                    8
```

TRISOLV Function

solves linear systems with triangular matrices

> **TRISOLV(** *code, r, b* <, *piv*>**)**

The TRISOLV function returns the following value:

x is the $n \times p$ matrix $\mathbf{X}$ containing p solutions of the p linear systems specified by *code*, r, and b.

The inputs to the TRISOLV function are as follows:

code   specifies which of the following forms of triangular linear system has to be solved:

code=1	solve $\mathbf{Rx} = \mathbf{b}$, $\mathbf{R}$ upper triangular
code=2	solve $\mathbf{R}'\mathbf{x} = \mathbf{b}$, $\mathbf{R}$ upper triangular
code=3	solve $\mathbf{R}'\mathbf{x} = \mathbf{b}$, $\mathbf{R}$ lower triangular
code=4	solve $\mathbf{Rx} = \mathbf{b}$, $\mathbf{R}$ lower triangular

r   specifies the $n \times n$ nonsingular upper (*code*=1,2) or lower (*code*=3,4) triangular coefficient matrix $\mathbf{R}$. Only the upper or lower triangle of argument matrix r is used; the other triangle can contain any information.

b   specifies the $n \times p$ matrix, $\mathbf{B}$, of p right-hand sides $\mathbf{b}_k$.

piv   specifies an optional n vector that relates the order of the columns of matrix $\mathbf{R}$ to the order of the columns of an original coefficient matrix $\mathbf{A}$ for which matrix $\mathbf{R}$ has been computed as a factor. For example, the vector *piv* can be the result of the QR decomposition of a matrix $\mathbf{A}$ whose columns were permuted in the order $\mathbf{A}_{piv[1]}, \ldots, \mathbf{A}_{piv[n]}$.

For *code*=1 and *code*=3, the solution is obtained by backward elimination. For *code*=2 and *code*=4, the solution is obtained by forward substitution.

If TRISOLV recognizes the upper or lower triangular matrix $\mathbf{R}$ as a singular matrix (that is, one that contains at least one zero diagonal element), it exits with an error message.

```
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;
```

$$
\begin{array}{c}
\text{X} \\[4pt]
1 \\
-1 \\
0 \\
1
\end{array}
$$

Also see the example in the QR call section.

TSBAYSEA Call

performs Bayesian seasonal adjustment modeling

CALL TSBAYSEA(*trend, season, series, adjust, abic, data*

*<,order, sorder, rigid, npred, opt, cntl, print>***);**

The inputs to the TSBAYSEA subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector.

order specifies the order of trend differencing. The default is *order*=2.

sorder specifies the order of seasonal differencing. The default is *sorder*=1.

rigid specifies the rigidity of the seasonal pattern. The default is *rigid*=1.

npred specifies the length of the forecast beyond the available observations. The default is *npred*=0.

opt specifies the options vector.

opt[1] specifies the number of seasonal periods (*speriod*). By default, *opt*[1]=12.

opt[2] specifies the year when the series starts (*year*). If *opt*[2]=0, there will be no trading day adjustment. By default, *opt*[2]=0.

opt[3] specifies the month when the series starts (*month*). If *opt*[2]=0, this option is ignored. By default, *opt*[3]=1.

opt[4] specifies the upper limit value for outlier determination (*rlim*). Outliers are considered as missing values. If this value is less than or equal to 0, TSBAYSEA assumes that the input data does not contain outliers. The default is *rlim*=0. See the "Missing Values" section on page 288.

opt[5] refers to the number of time periods processed at one time (*span*). The default is *opt*[5]=4.

opt[6] specifies the number of time periods to be shifted (*shift*). By default, *opt*[6]=1.

opt[7] controls the transformation of the original series (*logt*). If *opt*[7]=1, log transformation is requested. No transformation (*opt*[7]=0) is the default.

cntl specifies control values for the TSBAYSEA subroutine. These values will be automatically set. Be careful if you change these values.

     cntl[1] controls the adaptivity of the trading day adjustment component (*wtrd*). The default is *cntl*[1]=1.0.

     cntl[2] controls the sum of seasonal components within a period (*zersum*). The larger the value of *cntl*[2], the closer to zero this sum is. By default, *cntl*[2]=1.0.

     cntl[3] controls the leap year effect (*delta*). The default is *cntl*[3]=7.0.

     cntl[4] specifies the prior variance of the initial trend (*alpha*). The default is *cntl*[4]=0.01.

     cntl[5] specifies the prior variance of the initial seasonal component (*beta*). The default is *cntl*[5]=0.01. [.03in]

     cntl[6] specifies the prior variance of the initial sum of seasonal components (*gamma*). The default is *cntl*[6]=0.01.

print requests the power spectrum and the estimated and forecast values of time series components. If *print*=2, the spectra of irregular, differenced trend and seasonal series are printed, together with estimates and forecast values. If *print*=1, only the estimates and forecast values of time series components are printed.

    If *print*=0, printed output is suppressed. The default is *print*=0.

The TSBAYSEA subroutine returns the following values:

trend refers to the estimate and forecast of the trend component.

season refers to the estimate and forecast of the seasonal component.

series refers to the smoothed and forecast values of the time series.

adjust refers to the seasonally adjusted series.

abic refers to the value of ABIC from the final estimates.

Bayesian seasonal adjustments are performed with the TSBAYSEA subroutine. The smoothness of the trend and seasonal components is controlled by the prior distribution. The Akaike Bayesian Information Criterion (ABIC) is defined to compare with alternative models. The basic TSBAYSEA procedure processes the block of data in which the length is SPAN*SPERIOD, while the first block of data consists of length (2*SPAN-1)*SPERIOD. The block of data is shifted successively by SHIFT*SPERIOD.

The TSBAYSEA call decomposes the series y_t into the following form:

$$y_t = T_t + S_t + \epsilon_t$$

where T_t is a trend component, S_t denotes a seasonal component, and ϵ_t is an irregular component. To estimate the seasonal and trend components, some constraints are imposed such that the sum of squares of $\nabla^k T_t$, $\nabla_L^l S_t$, and $\sum_{i=0}^{L-1} S_{t-i}$ is small, where ∇ and ∇_L are difference operators. Then the solution can be obtained by minimizing

$$\sum_{t=1}^{N} \left\{ (y_t - T_t - S_t)^2 + d^2 \left[s^2 (\nabla^k T_t)^2 + (\nabla_L^l S_t)^2 + z^2 (S_t + \cdots + S_{t-L+1})^2 \right] \right\}$$

where d measures the smoothness of the trend and seasonality, s measures the smoothness of the trend, and z is a smoothness constant for the sum of the seasonal variability. The value of d is estimated while the constants, s and z, are chosen *a priori*. The value of s is equal to $\frac{1}{RIGID}$, and the constant z is determined as ZERSUM*RIGID/SPERIOD$^{1/2}$. The larger the constant RIGID, the more rigid the seasonal pattern is. See the section, Bayesian Constrained Least Squares, for more information.

To analyze the monthly data with rigidity 0.5, you can specify

```
call tsbaysea(trend,season,series,adj,abic) data=z order=2
        sorder=1 rigid=0.5 npred=10 print=2;
```

or

```
call tsbaysea(trend,season,series,adj,abic,z,2,1,0.5,10,,,2);
```

The TREND, SEASON, and SERIES components contain 10-period-ahead forecast values as well as the smoothed estimates. The detailed result is also printed since the PRINT=2 option is specified.

TSDECOMP Call

analyzes nonstationary time series by using smoothness priors modeling

> **CALL TSDECOMP(** *comp, est, aic, data,* $<$*,xdata, order, sorder,*
>
> *nar, npred, init, opt, icmp, print*$>$**);**

The inputs to the TSDECOMP subroutine are as follows:

data   specifies a $T \times 1$ (or $1 \times T$) data vector.

xdata   specifies a $T \times K$ explanatory data matrix.

order   specifies the order of trend differencing (0, 1, 2, or 3). The default is 2.

sorder   specifies the order of seasonal differencing (0, 1, or 2). The default is 1.

nar   specifies the order of the AR process. The default is 0.

npred   specifies the length of the forecast beyond the available observations. The default is 0.

init   specifies the initial values of parameters. The initial values are specified as variances for trend difference equation, AR process, seasonal difference equation, regression equation, and partial AR coefficients. The corresponding default variance values are 0.005, 0.8, 1E−5, and 1E−5. The default partial AR coefficient values are determined as

$$\psi_i = 0.88 \times (-0.6)^{i-1} \qquad i = 1, 2, \ldots, nar$$

opt   specifies the options vector.

opt[1] specifies the mean deletion option. The mean of the original se-
ries is subtracted from the series if *opt*[1]=−1. By default, the
original series is processed (*opt*[1]=0). When regressors are spec-
ified, only the *opt*[1]=0 option is allowed.

opt[2] specifies the trading day adjustment. The default is *opt*[2]=0.

opt[3] specifies the year (≥ 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 transforma-
tion (*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 opti-
mization method. The default is *opt*[8] = 2.

    opt[8]=1 specifies a line search method that requires the same
number of objective function and gradient calls for
cubic interpolation and extrapolation.

    opt[8]=2 specifies a line search method that requires more ob-
jective function calls than gradient calls for cubic in-
terpolation and extrapolation.

    opt[8]=3 specifies a line search method that requires the same
number of objective function and gradient calls for
cubic interpolation and extrapolation.

    opt[8]=4 specifies a line search method that requires the same
number of objective function and gradient calls for
cubic interpolation and stepwise extrapolation.

    opt[8]=5 specifies a line search method that is a modified ver-
sion of *opt*[8]=4.

    opt[8]=6 specifies the golden section line search method that
uses only function values for linear approximation.

    opt[8]=7 specifies the bisection line search method that uses
only function values for linear approximation.

    opt[8]=8 specifies the Armijo line search method that uses only
function values for linear approximation.

opt[9] specifies the upper bound of the variance estimates. If you specify
opt[9]=*value*, the variances are estimated with the constraint that
$\sigma \leq value$. When you specify the *opt*[9]=0 option, the upper
bound is not imposed. The default is *opt*[9]=0.

opt[10] specifies the length of data used in backward filtering for the
Kalman filter initialization. The default value of *opt*[10] is 100

if the number of observations is greater than 100; otherwise, the default value is the number of observations.

icmp specifies which component is calculated.

icmp=1 requests the estimate and forecast of trend component.
icmp=2 requests the estimate and forecast of seasonal component.
icmp=3 requests the estimate and forecast of AR component.
icmp=4 requests the trading day adjustment component.
icmp=5 requests the regression component.
icmp=6 requests the time-varying regression coefficients.

You can calculate multiple components by specifying a vector. For example, you can specify *icmp*={1 2 3 5}.

print specifies the print option. By default, printed output is suppressed (*print*=0). If you specify *print*=1, the subroutine prints the final estimates. The iteration history is printed if you specify *print*=2.

The TSDECOMP subroutine returns the following values:

comp refers to the estimate and forecast of the trend component.

est refers to the parameter estimates including coefficients of the AR process.

aic refers to the AIC statistic obtained from the final estimates.

The TSDECOMP subroutine analyzes nonstationary time series by using smoothness priors modeling (see the "Smoothness Priors Modeling" section on page 270 for more details). The likelihood function is maximized with respect to hyperparameters. The Kalman filter algorithm is used for filtering, smoothing, and forecasting. The TSDECOMP call decomposes the time series y_t as follows:

$$y_t = T_t + S_t + TD_t + u_t + R_t + \epsilon_t$$

where T_t represents the trend component, S_t denotes the seasonal component, TD_t represents the trading day adjustment component, u_t denotes the autoregressive process component, R_t denotes regression effect components, and ϵ_t represents the irregular term with zero mean and constant variance.

The trend components are constrained as follows:

$$\nabla^k T_t = w_{1t}, \qquad w_{1t} \sim N(0, \tau_1^2)$$

When you specify the ORDER=0 option, the trend component is not estimated. The maximum order of differencing is 3 ($k = 0, \ldots, 3$).

The seasonal components are denoted as a stochastically perturbed equation:

$$\left(1 + \sum_{i=1}^{L-1} \mathbf{B}^i\right)^l S_t = w_{2t}, \qquad w_{2t} \sim N(0, \tau_2^2)$$

When you specify SORDER=0, the seasonal component is not estimated. The maximum value of l is 2 ($l = 0, 1,$ or 2).

The stationary autoregressive (AR) process is denoted as a stochastically perturbed equation:

$$u_t = \sum_{i=1}^{p} \alpha_i u_{t-i} + w_{3t}, \qquad w_{3t} \sim N(0, \tau_3^2)$$

where p is the order of AR process. When NAR=0 is specified, the AR process component is not estimated.

The time-varying regression coefficients are estimated if you include exogenous variables:

$$R_t = \mathbf{X}_t \beta_t$$

where $\mathbf{X}_t$ contains m regressors except the constant term and $\beta_t' = (\beta_{1t}, \ldots, \beta_{mt})$. The time-varying coefficients β_t follow the random walk process:

$$\beta_{jt} = \beta_{jt-1} + v_{jt}, \qquad v_{jt} \sim N(0, \sigma_j^2)$$

where β_{jt} is an element of the coefficient vector β_t.

The trading day adjustment component TD_t is deterministically restricted. See the section, State Space and Kalman Filter Method, for more information.

You can estimate the time-varying coefficient model as follows:

```
call tsdecomp COMP=beta ORDER=0 SORDER=0 NAR=0
              DATA=y XDATA=x ICMP=6;
```

The output matrix BETA contains time-varying regression coefficients.

TSMLOCAR Call

analyzes nonstationary or locally stationary time series by using the minimum AIC procedure

CALL TSMLOCAR(*arcoef, ev, nar, aic, start, finish, data*

  <*,maxlag, opt, missing, print*>**);**

The inputs to the TSMLOCAR subroutine are as follows:

data   specifies a $T \times 1$ (or $1 \times T$) data vector.

maxlag   specifies the maximum lag of the AR process. This value should be less than half the length of locally stationary spans. The default is *maxlag*=10.

opt   specifies an options vector.

  opt[1]   specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=−1. An intercept coefficient is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean value of the input series is 0. The default is *opt*[1]=0.

  opt[2]   specifies the number (J) of basic spans. By default, *opt*[2]=1.

  opt[3]   specifies the minimum AIC option. If *opt*[3]=0, the *maximum lag* AR process is estimated. If *opt*[3]=1, the minimum AIC procedure is performed. The default is *opt*[3]=1.

missing   specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean. *print*] specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the AR estimation result, while the *print*=2 option plots the power spectral density as well as the AR estimates.

The TSMLOCAR subroutine returns the following values:

arcoef   refers to an *nar* $\times 1$ AR coefficient vector of the final model if the intercept estimate is not included. If *opt*[1]=1, the first element of the *arcoef* vector is an intercept estimate.

ev   refers to the error variance.

nar   is the selected AR order of the final model. If *opt*[3]=0, *nar*=*maxlag*.

aic   refers to the minimum AIC value of the final model.

start   refers to the starting position of the input series, which corresponds to the first observation of the final model.

finish   refers to the ending position of the input series, which corresponds to the last observation of the final model.

The TSMLOCAR subroutine analyzes nonstationary (or locally stationary) time series by using the minimum AIC procedure. The data of length T is divided into J locally stationary subseries, which consist of $\frac{T}{J}$ observations. See the "Nonstationary Time Series" section on page 274 for details.

TSMLOMAR Call

analyzes nonstationary or locally stationary multivariate time series by using the minimum AIC procedure

CALL TSMLOMAR(*arcoef, ev, nar, aic, start, finish, data*

$<$,*maxlag, opt, missing, print*$>$**);**

The inputs to the TSMLOMAR subroutine are as follows:

data specifies a $T \times M$ data matrix, where T is the number of observations and M is the number of variables to be analyzed.

maxlag specifies the maximum lag of the vector AR (VAR) process. This value should be less than $\frac{1}{2M}$ of the length of locally stationary spans. The default is *maxlag*=10.

opt specifies an options vector.

opt[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=-1. An intercept coefficient is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean values of input series are zeroes. The default is *opt*[1]=0.

opt[2] specifies the number (J) of basic spans. By default, *opt*[2]=1.

opt[3] specifies the minimum AIC option. If *opt*[3]=0, the *maximum lag* VAR process is estimated. If *opt*[3]=1, a minimum AIC procedure is used. The default is *opt*[3]=1.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the AR estimates, minimum AIC, minimum AIC order, and innovation variance matrix.

The TSMLOMAR subroutine returns the following values.

arcoef refers to an $M \times (M * nar)$ VAR coefficient vector of the final model if the intercept vector is not included. If *opt*[1]=1, the first column of the *arcoef* matrix is an intercept estimate vector.

ev refers to the error variance matrix.

nar is the selected VAR order of the final model. If *opt*[3]=0, *nar*=*maxlag*.

aic refers to the minimum AIC value of the final model.

start refers to the starting position of the input series *data*, which corresponds to the first observation of the final model.

finish   refers to the ending position of the input series *data*, which corresponds to the last observation of the final model.

The TSMLOMAR subroutine analyzes nonstationary (or locally stationary) multivariate time series by using the minimum AIC procedure. The data of length T is divided into J locally stationary subseries. See "Nonstationary Time Series" in the "Nonstationary Time Series" section on page 274 for details.

TSMULMAR Call

estimates VAR processes by using the minimum AIC procedure

CALL TSMULMAR(*arcoef, ev, nar, aic, data*

$<$*,maxlag, opt, missing, print*$>$**);**

The inputs to the TSMULMAR subroutine are as follows:

data   specifies a $T \times M$ data matrix, where T is the number of observations and M is the number of variables to be analyzed.

maxlag specifies the maximum lag of the VAR process. This value should be less than $\frac{1}{2M}$ of the length of input data. The default is *maxlag*=10.

opt   specifies an options vector.

> *opt*[1]   specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=−1. An $M \times 1$ intercept vector is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean value of the input data is 0. The default is *opt*[1]=0.

> *opt*[2]   specifies the minimum AIC option. If *opt*[2]=0, the *maximum lag* AR process is estimated. If *opt*[2]=1, the minimum AIC procedure is used, while the *opt*[2]=2 option specifies the VAR order selection method based on the AIC. The default is *opt*[2]=1.

> *opt*[3]   specifies instantaneous response modeling if *opt*[3]=1. The default is *opt*[3]=0. See the section "Multivariate Time Series Analysis" on page 277 for more information.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print   specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the final estimation result, while the *print*=2 option prints intermediate and final results.

The   TSMULMAR   subroutine   returns   the   following   values:

arcoef   refers to an $M \times (M * nar)$ AR coefficient matrix if the intercept is not included. If *opt*[1]=1, the first column of the *arcoef* matrix is an intercept vector estimate.

ev   refers to the error variance matrix.

nar   is the selected VAR order of the minimum AIC procedure. If *opt*[2]=0, *nar=maxlag. aic*] refers to the minimum AIC value.

The TSMULMAR subroutine estimates the VAR process by using the minimum AIC method. The widely used VAR order selection method is added to the original TIMSAC program, which considers only the possibilities of zero coefficients at the beginning and end of the model. The TSMULMAR subroutine can also estimate the instantaneous response model. See the "Multivariate Time Series Analysis" section on page 277 for details.

TSPEARS Call

analyzes periodic AR models with the minimum AIC procedure

> **CALL TSPEARS(** *arcoef, ev, nar, aic, data*
>
>   <*,maxlag, opt, missing, print*>**);**

The inputs to the TSPEARS subroutine are as follows:

data   specifies a $T \times 1$ (or $1 \times T$) data matrix.

maxlag   specifies the maximum lag of the periodic AR process. This value should be less than $\frac{1}{2J}$ of the input series. The default is *maxlag*=10.

opt   specifies an options vector.

> *opt*[1]   specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=−1. An intercept coefficient is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean values of input series are zeroes. The default is *opt*[1]=0.
>
> *opt*[2]   specifies the number of instants per period. By default, *opt*[2]=1.
>
> *opt*[3]   specifies the minimum AIC option. If *opt*[3]=0, the *maximum lag* AR process is estimated. If *opt*[3]=1, the minimum AIC procedure is used. The default is *opt*[3]=1.

missing   specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print   specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the periodic AR estimates and intermediate process.

The   TSPEARS   subroutine   returns   the   following   values:

arcoef   refers to a periodic AR coefficient matrix of the periodic AR model. If *opt*[1]=1, the first column of the *arcoef* matrix is an intercept estimate vector.

ev   refers to the error variance.

nar   refers to the selected AR order vector of the periodic AR model.

aic   refers to the minimum AIC values of the periodic AR model.

The TSPEARS subroutine analyzes the periodic AR model by using the minimum AIC procedure. The data of length T are divided into d periods. There are J instants in one period. See the "Multivariate Time Series Analysis" section on page 277 for details.

TSPRED Call

provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are input

> **CALL TSPRED(** *forecast, impulse, mse, data, coef, nar, nma*
>
> $<$,*ev, npred, start, constant*$>$**);**

The inputs to the TSPRED subroutine are as follows:

data   specifies a $T \times M$ data matrix if the intercept is not included, where T denotes the length of the time series and M is the number of variables to be analyzed. If the univariate time series is analyzed, the input data should be a column vector.

coef   refers to the $M(P+Q) \times M$ ARMA coefficient matrix, where P is an AR order and Q is an MA order. If the intercept term is included (*constant*=1), the first row of the coefficient matrix is considered as the intercept term and the coefficient matrix is an $M(P+Q+1) \times M$ matrix. If there are missing values in the *coef* matrix, these are converted to zero.

nar   specifies the order of the AR process. If the subset AR process is requested, *nar* should be a row or column vector. The default is *nar*=0.

nma   specifies the order of the MA process. If the subset MA process is requested, *nma* should be a vector. The default is *nma*=0.

ev   specifies the error variance matrix. If the *ev* matrix is not provided, the prediction error covariance will not be computed.

npred   specifies the maximum length of multistep forecasting. The default is *npred*=0.

start   specifies the position where the multistep forecast starts. The default is *start*=T.

constant specifies the intercept option. No intercept estimate is included if *constant*=0; otherwise, the intercept estimate is included in the first row of the coefficient matrix. If *constant*=−1, the coefficient matrix is estimated by using mean deleted series. By default, *constant*=0.

The TSPRED subroutine returns the following values:

forecast refers to predicted values.

impulse refers to the impulse response function.

mse refers to the mean square error of s-step-ahead forecast. A scalar missing value is returned if the error variance (*ev*) is not provided.

TSROOT Call

calculates AR and MA coefficients from the characteristic roots of the model or calculates the characteristic roots of the model from the AR and MA coefficients

> **CALL TSROOT(** *matout, matin, nar, nma,* <*,qcoef, print*>**);**

The inputs to the TSROOT subroutine are as follows:

matin refers to the $(nar + nma) \times 2$ characteristic root matrix if the polynomial (ARMA) coefficients are requested (*qcoef*=1), where the first column of the *matin* matrix contains the real part of the root and the second column of the *matin* matrix contains the imaginary part of the root. When the characteristic roots are requested (*qcoef*=0), the first *nar* rows are complex AR coefficients and the last *nma* rows are complex MA coefficients. The default is *qcoef*=0.

nar specifies the order of the AR process. If you specify the subset AR model, the input *nar* should be a row or column vector.

nma specifies the order of the MA process. If you specify the subset MA model, the input *nma* should be a row or column vector.

qcoef requests the ARMA coefficients when the characteristic roots are provided (*qcoef*=1). By default, the characteristic roots of the polynomial are calculated (*qcoef*=0).

print specifies the print option if *print*=1. By default, printed output is suppressed (*print*=0).

The TSROOT subroutine returns the following values

matout refers to the characteristic root matrix if *qcoef*=0; otherwise, the *matout* matrix contains the AR and MA coefficients.

TSTVCAR Call

analyzes time series that are nonstationary in the covariance function

> **CALL TSTVCAR(** *arcoef, variance, est, aic, data*
>
> < *,nar, init, opt, outlier, print* >**);**

The inputs to the TSTVCAR subroutine are as follows:

data   specifies a $T \times 1$ (or $1 \times T$) data vector.

nar   specifies the order of the AR process. The default is *nar*=8.

init   specifies the initial values of the parameter estimates. The default is (1E−4, 0.3, 1E−5, 0).

opt   specifies an options vector.

> *opt*[1]   specifies the mean deletion option. The mean of the original se-
> ries 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 his-
tory is printed if *print*=2.

The TSTVCAR subroutine returns the following values:

arcoef   refers to the time-varying AR coefficients.

variance   refers to the time-varying error variances.   See the "Smoothness Priors
Modeling" section on page 270 for details.

est   refers to the parameter estimates.

aic   refers to the value of AIC from the final estimates.

Nonstationary time series modeling usually deals with nonstationarity in the mean.
The TSTVCAR subroutine analyzes the model that is nonstationary in the covariance.
Smoothness priors are imposed on each time-varying AR coefficient and frequency
response function. See the "Nonstationary Time Series" section on page 274 for
details.

TSUNIMAR Call

determines the order of an AR process with the minimum AIC procedure and estimates the AR coefficients

CALL TSUNIMAR(*arcoef, ev, nar, aic, data*

<,*maxlag, opt, missing, print*>**);**

The inputs to the TSUNIMAR subroutine are as follows:

data specifies a $T \times 1$ (or $1 \times T$) data vector, where T is the number of observations.

maxlag specifies the maximum lag of the AR process. This value should be less than half the number of observations. The default is *maxlag*=10.

opt specifies an options vector.

opt[1] specifies the mean deletion option. The mean of the original data is deleted if *opt*[1]=−1. An intercept term is estimated if *opt*[1]=1. If *opt*[1]=0, the original input data is processed assuming that the mean value of the input data is 0. The default is *opt*[1]=0.

opt[2] specifies the minimum AIC option. If *opt*[2]=0, the *maximum lag* AR process is estimated. The minimum AIC option, *opt*[2]=1, is the default.

missing specifies the missing value option. By default, only the first contiguous observations with no missing values are used (*missing*=0). The *missing*=1 option ignores observations with missing values. If you specify the *missing*=2 option, the missing values are replaced with the sample mean.

print specifies the print option. By default, printed output is suppressed (*print*=0). The *print*=1 option prints the final estimation result, while the *print*=2 option prints intermediate and final results.

The TSUNIMAR subroutine returns the following values.

arcoef refers to an *nar* $\times 1$ AR coefficient vector if the intercept is not included. If *opt*[1]=1, the first element of the *arcoef* vector is an intercept estimate.

ev refers to the error variance.

nar refers to the selected AR order by minimum AIC procedure. If *opt*[2]=0, *nar = maximum lag*.

aic refers to the minimum AIC value.

The TSUNIMAR subroutine determines the order of the AR process by using the minimum AIC procedure and estimates the AR coefficients. All AR coefficient estimates up to maximum lag are printed if you specify the print option. See the section, Least Squares and Householder Transformation, for more information.

TYPE Function

determines the type of a matrix

TYPE(*matrix***)**

where *matrix* is a numeric or character matrix or literal.

The TYPE function returns a single character value; it is **N** if the type of the matrix is numeric; it is **C** if the type of the matrix is character; it is **U** if the matrix does not have a value. Examples of valid statements follow.

The statements

```
a={tom};
r=type(a);
```

set R to **C**. The statements

```
free a;
r=type(a);
```

set R to **U**. The statements

```
a={1 2 3};
r=type(a);
```

set R to **N**.

UNIFORM Function

generates pseudo-random uniform deviates

UNIFORM(*seed***)**

where *seed* is a numeric matrix or literal. The *seed* can be any integer value up to $2^{31} - 1$.

The UNIFORM function returns one or more pseudo-random numbers with a uniform distribution over the interval 0 to 1. The UNIFORM function returns a matrix with the same dimensions as the argument. The first argument on the first call is used for the seed, or if that argument is 0, the system clock is used for the seed. The function is equivalent to the DATA step function RANUNI. An example of a valid statement follows:

```
seed = 123456;
c = j(5,1,seed);
b = uniform(c);
print b;
```

```
                        0.73902
                      0.2724794
                      0.7095326
                      0.3191636
                       0.367853
```

UNION Function

performs unions of sets

> **UNION(** *matrix1<, matrix2,. . ., matrix15>***)**

where *matrix* is a numeric or character matrix or quoted literal.

The UNION function returns as a row vector the sorted set (without duplicates) which is the union of the element values present in its arguments. There can be up to 15 arguments, which can be either all character or all numeric. For character arguments, the element length of the result is the longest element length of the arguments. Shorter character elements are padded on the right with blanks. This function is identical to the UNIQUE function. For example, the statements

```
a={1 2 4 5};
b={3 4};
c=union(a,b);
```

set

```
    C               1 row       5 cols    (numeric)

        1           2       3           4           5
```

The UNION function can be used to sort elements of a matrix when there are no duplicates by calling UNION with a single argument.

UNIQUE Function

sorts and removes duplicates

> **UNIQUE(** *matrix1<, matrix2,. . ., matrix15>***)**

where *matrix* is a numeric or character matrix or quoted literal.

The UNIQUE function returns as a row vector the sorted set (without duplicates) of all the element values present in its arguments. The arguments can be either all numeric or all character, and there can be up to 15 arguments specified. This function is identical to the UNION function, the description of which includes an example.

UNIQUEBY Function

returns the locations of the unique by-group combinations for a sorted or indexed matrix

> **UNIQUEBY(** *matrix, by, index***)**

The inputs 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 containing the names of columns corresponding to column labels assigned to *matrix* by a MATTRIB statement or READ statement.

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 just 1:nrow(*matrix*)

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. This means that if

```
unique_rows = uniqueby( matrix, by, index);
```

is executed, then

```
unique = matrix[ index[ unique_rows ], by ];
```

gives the values of the unique *by* combinations. In addition,

```
n = nrow( unique );
```

gives the number of unique values, and

```
size = unique_rows[i+1] - unique_rows[i];
```

gives the number of rows in the *i*th *by* combination, except for the last combination, which is given by

```
size_last = nrow(matrix) - unique_rows[nrow(unique_rows)] + 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:

```
unique_loc = uniqueby( matrix, by, 1:nrow(matrix) );
```

USE Statement

opens a SAS data set for reading

> **USE** *SAS-data-set* <**VAR** *operand*> <**WHERE(***expression***)**>
>
> <**NOBS** *name*>**;**

The inputs to the USE statement are as follows:

SAS-data-set	can be specified with a one-word name (for example, A) or a two-word name (for example, SASUSER.A). For more information on specifying SAS data sets, see the chapter on SAS data sets in *SAS Language Reference: Concepts*.
operand	selects a set of variables.
expression	is evaluated for being true or false.
name	is the name of a variable to contain the number of observations.

If the data set has not already been opened, the USE statement opens the data set for read access. The USE statement also makes the data set the current input data set so that subsequent statements act on it. The USE statement optionally can define selection criteria that are used to control access.

The VAR clause specifies a set of variables to use, where *operand* can be any of the following:

- a literal containing variable names
- the name of a matrix containing variable names
- an expression in parentheses yielding variable names
- one of the following keywords:

ALL	for all variables
CHAR	for all character variables
NUM	for all numeric variables

The following examples show each possible way you can use the VAR clause:

```
var {time1 time5 time9}; /* a literal giving the variables */
var time;                /* a matrix containing the names  */
var('time1':'time9');    /* an expression                  */
var _all_;               /* a keyword                      */
```

The WHERE clause conditionally selects observations, within the *range* specification, according to conditions given in the clause. The general form of the WHERE clause is

WHERE(*variable comparison-op operand***)**

In the statement above,

variable   is a variable in the SAS data set.

comparison-op   is one of the following comparison operators:

<	less than
<=	less than or equal to
=	equal to
>	greater than
>=	greater than or equal to
ˆ =	not equal to
?	contains a given string
ˆ ?	does not contain a given string
=:	begins with a given string
=*	sounds like or is spelled similar to a given string

operand   is a literal value, a matrix name, or an expression in parentheses.

WHERE comparison arguments can be matrices. For the following operators, the WHERE clause succeeds if *all* the elements in the matrix satisfy the condition:

   ˆ = ˆ ? < <= > >=

For the following operators, the WHERE clause succeeds if *any* of the elements in the matrix satisfy the condition:

   = ? =: =*

Logical expressions can be specified within the WHERE clause using the AND (&) and OR (|) operators. The general form is

   clause&*clause*   (for an AND clause)
   clause|*clause*   (for an OR clause)

where *clause* can be a comparison, a parenthesized clause, or a logical expression clause that is evaluated using operator precedence.

Note: The expression on the left-hand side refers to values of the data set variables, and the expression on the right-hand side refers to matrix values.

The VAR and WHERE clauses are optional, and you can specify them in any order. If a data set is already open, all the options that the data set was first opened with are still in effect. To override any old options, the new USE statement must explicitly specify the new options. Examples of valid statements follow.

```
use class;
use class var{name sex age};
use class var{name sex age} where(age>10);
```

VALSET Call

performs indirect assignment

CALL VALSET(*char-scalar, argument***);**

The inputs to the VALSET subroutine are as follows:

char-scalar   is a character scalar containing the name of a matrix.

argument   is a value to which the matrix is set.

The VALSET subroutine expects a single character string argument containing the name of a matrix. It looks up the matrix and moves the value of the second argument to this matrix. For example, the following statements find that the value of the argument **B** is **A** and then assign the value 99 to **A**, the indirect result:

```
b="A";
call valset(b,99);
```

The previous value of the indirect result is freed. The following statement sets **B** to 99, but the value of **A** is unaffected by this statement:

```
b=99;
```

VALUE Function

assigns values by indirect reference

VALUE(*char-scalar***)**

where *char-scalar* is a character scalar containing the name of a matrix.

The VALUE function expects a single character string argument containing the name of a matrix. It looks up the matrix and moves its value to the result. For example, the statements

```
a={1 2 3};
b="A";
c=value(b);
```

find that the value of the argument **B** is **A** and then look up **A** and copy the value 1 2 3 to **C**.

```
                    C             1 row        3 cols      (numeric)

                                  1            2            3
```

VARMACOV Call

computes the theoretical cross-covariance matrices for a stationary VARMA(p, q) model

CALL VARMACOV(*cov, phi, theta, sigma <, p, q, lag>* **);**

The inputs to the VARMACOV subroutine are as follows:

phi specifies a $km_p \times k$ matrix, Φ, containing 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 containing 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.

To compute the cross-covariance matrices of a bivariate ($k = 2$) VARMA(1,1) model

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t - \Theta \boldsymbol{\epsilon}_{t-1}$$

where

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

you can specify

```
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

computes the log-likelihood function for a VARMA(p, q) model

CALL VARMALIK(*lnl, series, phi, theta, sigma <, p, q, opt>* **);**

The inputs to the VARMALIK subroutine are as follows:

series specifies an $n \times k$ matrix containing 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 containing 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 containing 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 wis 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 containing 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.

To compute the log-likelihood function of a bivariate ($k = 2$) VARMA(1,1) model

$$\mathbf{y}_t = \Phi\mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t - \Theta\boldsymbol{\epsilon}_{t-1}$$

where

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

you can specify

```
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

generates a VARMA(p,q) time series

CALL VARMASIM(*series, phi, theta, mu, sigma, n <, p, q, initial, seed>***);**

The inputs to the VARMASIM subroutine are as follows:

phi   specifies a $km_p \times k$ matrix containing 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 containing 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   specifies the random number seed. See the VNORMAL subroutine.

The VARMASIM subroutine returns the following value:

series is an $n \times k$ matrix containing 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*.

To generate a bivariate $(k = 2)$ stationary VARMA(1,1) time series

$$\mathbf{y}_t - \boldsymbol{\mu} = \Phi(\mathbf{y}_{t-1} - \boldsymbol{\mu}) + \boldsymbol{\epsilon}_t - \Theta\boldsymbol{\epsilon}_{t-1}$$

where

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \boldsymbol{\mu} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

you can specify

```
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);
```

To generate a bivariate ($k = 2$) nonstationary VARMA(1,1) time series with the same μ, Σ, and Θ in the previous example and the AR coefficient

$$\Phi = \begin{bmatrix} 1.0 & 0 \\ 0 & 0.3 \end{bmatrix}$$

you can specify

```
phi  = { 1.0 0.0, 0.0 0.3 };
call varmasim(yt, phi, theta, mu, sigma, 100) initial=3;
```

VECDIAG Function

creates a vector from a diagonal

> **VECDIAG(** *square-matrix***)**

where *square-matrix* is a square numeric matrix.

The VECDIAG function creates a column vector whose elements are the main diagonal elements of *square-matrix*. For example, the statements

```
a={2 1, 0 -1};
c=vecdiag(a);
```

produce the column vector

```
            C             2 rows        1 col      (numeric)

                                          2
                                         -1
```

VNORMAL Call

generates a multivariate normal random series

> **CALL VNORMAL(** *series, mu, sigma, n <, seed>***);**

The inputs to the VNORMAL subroutine are as follows:

mu   specifies a $k \times 1$ (or $1 \times k$) mean vector, 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   specifies the random number seed. If it is not supplied, the system clock is used to generate the seed. If it is negative, then the absolute value is used as the starting seed; otherwise, subsequent calls ignore the value of *seed* and use the last seed generated internally.

The VNORMAL subroutine returns the following value:

series   is an $n \times k$ matrix that contains the generated normal random series.

To generate a bivariate ($k = 2$) normal random series with mean $\boldsymbol{\mu}$ and covariance matrix Σ, where

$$\boldsymbol{\mu} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \text{ and } \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

you can specify

```
mu   = { 10, 20 };
sigma= { 1.0  0.5, 0.5 1.25};
call vnormal(et, mu, sigma, 100);
```

VTSROOT Call

calculates the characteristic roots of the model from AR and MA characteristic functions

> **CALL VTSROOT(** *root, phi, theta<, p, q>***);**

The inputs to the VTSROOT subroutine are as follows:

phi   specifies a $km_p \times k$ matrix containing 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 containing 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.

To compute 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

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix}$$

you can specify

```
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

computes fast wavelet transform

    CALL WAVFT(*decomp, data, opt* $<$*, levels*$>$ **);**

The Fast Wavelet Transform (WAVFT) subroutine computes a specified discrete wavelet transform of the input data, 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, may be involved. These boundary coefficients are obtained by extending the sequence of interior scaling coefficients using a specified method.

Details of the discrete wavelet transform and the fast wavelet transformation algorithm are available in many references, including Mallat (1989), Daubechies (1997), and Ogden (1997).

The inputs to the WAVFT subroutine are as follows:

data   specifies the data to transform. This data must be 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:

            opt[1]=0 specifies extension by zero.
            opt[1]=1 specifies periodic extension.

opt[1]=2   specifies polynomial extension.
opt[1]=3   specifies extension by reflection.
opt[1]=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.

opt[2]=0   specifies constant extension.
opt[2]=1   specifies linear extension.
opt[2]=2   specifies quadratic extension.

opt[3]   specifies the wavelet family.

opt[3]=1   specifies the Daubechies Extremal phase family (Daubechies, 1992).
opt[3]=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

opt[4]=1 through 10, if *opt*[3]=1
opt[4]=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 will 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 of using 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);

optn = { 2, /* polynominal extension at boundry */
         1, /* using linear polynominal */
         1, /* Daubechies Extremal phase */
         3  /* family member 3 */
         };

call wavft(decomp, y, optn);
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 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

extracts wavelet information

> **CALL WAVGET(** *result, decomp, request* <, *options*> **);**

The WAVGET subroutine is used to return information that is encoded in a wavelet decomposition.

The required inputs are

decomp specifies a wavelet decomposition that has been computed using a call to the WAVFT subroutine.

request specifies a scalar indicating 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 containing this number.
>
> *request*=2 requests the detail coefficients at a specified level. Valid syntax is
>
> > **CALL WAVGET(** *result, decomp, 2, level* <, *opt*> **);**
> >
> > where the argument
> >
> > *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 call for details. If you omit this argument, no thresholding is applied.
> >
> > *result*   returns as a column vector containing the specified detail coefficients.
>
> *request*=3 requests the scaling coefficients at a specified level. Valid syntax is

CALL WAVGET(*result, decomp, 3, level <, opt>* **);**

where the argument

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 call 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 containing 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 call to compute *decomp.*

     result   returns as a column vector with 4 elements containing the specified options vector. See the WAVFT subroutine call for the interpretation of the vector entries.

request=6   requests the index of the top level in *decomp.*

     result   returns as a scalar containing this number.

request=7   requests the index of the lowest level in *decomp.*

     result   returns as a scalar containing 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>* **);**

where the optional argument

power   is the exponent of 2 determining the number of grid points used. *power* defaults to 8 if you do not specify this argument.

result   returns as a column vector containing the specified
evaluation of the father wavelet.

An example is available in the documentation for the WAVFT subroutine.

WAVIFT Call

computes inverse fast wavelet transform

CALL WAVIFT(*result, decomp* <, *opt* <, *level*>> **);**

The Inverse Fast Wavelet Transform (WAVIFT) subroutine computes the inverse
wavelet transform of a wavelet decomposition computed using the WAVFT subrou-
tine. 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, pro-
vided that no smoothing is specified. Alternatively, a smooth reconstruction of the in-
put 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 shrink-
age function and T_i is the threshold value used at level i. The SAS/IML 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:

$$
\begin{aligned}
\delta_T^{\text{hard}}(x) &= \begin{cases} 0 & |x| \le T \\ x & |x| > T \end{cases} \\
\delta_T^{\text{soft}}(x) &= \begin{cases} 0 & |x| \le T \\ x - T & x > T \\ x + T & x < -T \end{cases} \\
\delta_T^{\text{garrote}}(x) &= \begin{cases} 0 & |x| \le T \\ x - T^2/x & |x| > T \end{cases}
\end{aligned}
$$

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 inputs to the WAVIFT subroutine are as follows:

decomp   specifies a wavelet decomposition that has been computed 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.

opt[1]=0   specifies that no thresholding be done. If *opt*[1]=0 then all other entries in the options vector are ignored.

opt[1]=1   specifies hard thresholding.

opt[1]=2   specifies soft thresholding.

opt[1]=3   specifies garrote thresholding.

opt[2]   specifies the method for selecting the threshold.

opt[2]=0   specifies a global user supplied threshold.

opt[2]=1   specifies a global threshold chosen using the minimax criterion of Donoho and Johnstone (1994).

opt[2]=2   specifies a global threshold defined using the universal criterion of Donoho and Johnstone (1994).

opt[2]=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 using the reconstruction with thresholded coefficients (Donoho and Johnstone, 1995).

opt[2]=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 is to be returned. If this argument is not specified then the reconstructed data is 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 call. 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

displays wavelet information

> **CALL WAVPRINT(** *decomp, request* <*, options*> **);**

The WAVPRINT subroutine is used to display the information that is encoded in a wavelet decomposition.

The required inputs are

decomp   specifies a wavelet decomposition that has been computed using a call to the WAVFT subroutine.

request   specifies a scalar indicating 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*>> **);**

where the argument

lower   is optional and specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.

upper   is optional and 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*>> **);**

where the argument

lower   is optional and specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.

> *upper* is optional and 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*>> **);**

where the argument

> *opt* is a required options vector that specifies the thresholding algorithm used. See the WAVIFT subroutine call for a description of this options vector.
>
> *lower* is optional and specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.
>
> *upper* is optional and 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

thresholds wavelet detail coefficients

CALL WAVTHRSH(*decomp, opt* **);**

The Wavelet Threshold (WAVTHRSH) subroutine thresholds the detail coefficients in a wavelet decomposition.

The required inputs are

decomp specifies a wavelet decomposition that has been computed using a call to the WAVFT subroutine.

opt refers to an options vector that specifies the thresholding algorithm used. See the WAVIFT subroutine call 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 subroutine calls.

An example is available in the documentation for the WAVFT subroutine.

WINDOW Statement

opens a display window

> **WINDOW** <**CLOSE=**>*window-name* <*window-options*>
>
> > <**GROUP=***group-name field-specs*>
> >
> > < ...**GROUP=***group-name field-specs*>**;**

where the arguments and options are described below.

The WINDOW statement defines a window on the display and can include a number of fields. The DISPLAY statement actually writes values to the window. The following fields can be specified in the WINDOW statement:

window-name

specifies a name 1 to 8 characters long for the window. This name is displayed in the upper-left border of the window.

CLOSE=*window-name*

closes the window.

window-options

control the size, position, and other attributes of the window. The attributes can also be changed interactively with window commands such as WGROW, WDEF, WSHRINK, and COLOR. A description of the window options follows.

GROUP=*group-name*

starts a repeating sequence of groups of fields defined for the window. The *group-name* specification is a name 1 to 8 characters long used to identify a group of fields in a later DISPLAY statement.

field-specs

are a sequence of field specifications made up of positionals, field operands, formats, and options. These are described in the next section.

The following window options can be specified in the WINDOW statement:

CMNDLINE=*name*

specifies the name of a variable in which the command line entered by the user will be stored.

COLOR=*operand*

specifies the background color for the window. The *operand* is either a quoted character literal, a name, or an operand. The valid values are "WHITE", "BLACK", "GREEN", "MAGENTA", "RED", "YELLOW", "CYAN", "GRAY", and "BLUE". The default value is BLACK.

COLUMNS=*operand*

specifies the starting number of columns for the window. The *operand* is either a literal number, a variable name, or an expression in parentheses. The default value is 78 columns.

ICOLUMN=*operand*

specifies the initial starting column position of the window on the display. The *operand* is either a literal number or a variable name. The default value is column 1.

IROW=*operand*

specifies the initial starting row position of the window on the display. The *operand* is either a literal number or a variable name. The default value is row 1.

MSGLINE=*operand*

specifies the message to be displayed on the standard message line when the window is made active. The *operand* is almost always the name of a variable, but a character literal can be used.

ROWS=*operand*

determines the starting number of rows of the window. The *operand* is either a literal number, the name of a variable containing the number, or an expression in parentheses yielding the number. The default value is 23 rows.

Both the WINDOW and DISPLAY statements allow field specifications, which have the general form:

        <positionals> field-operand <format> <field-options>

In the preceding statement,

positionals	are directives determining the position on the screen to begin the field. There are four kinds of positionals; any number of positionals are allowed for each field operand.
# operand	specifies the row position; that is, it moves the current position to column 1 of the specified line. The *operand* is either a number, a name, or an expression in parentheses.
/	specifies that the current position move to column 1 of the next row.
@ operand	specifies the column position. The *operand* is either a number, a name, or an expression in parentheses. The @ directive should come after the # position if # is specified.
+ operand	specifies a skip of columns. The *operand* is either a number, a name, or an expression in parentheses.
field-operand	is a character literal in quotes or the name of a variable that specifies what is to go in the field.
format	is the format used for display, the value, and the informat applied to entered values. If no format is specified, then the standard numeric or character format is used.
field-options	specify the attributes of the field as follows:

        PROTECT=YES
        P=YES

            specifies that the field is protected; that is, you cannot enter values in the field. If the field operand is a literal, it is already protected.

COLOR=*operand*

> specifies the color of the field. The *operand* is a literal character value in quotes, a variable name, or an expression in parentheses. The colors available are "WHITE", "BLACK", "GREEN", "MAGENTA", "RED", "YELLOW", "CYAN", "GRAY", and "BLUE". Note that the color specification is different from that of the corresponding DATA step value because it is an operand rather than a name without quotes. The default value is "BLUE".

XMULT Function

performs accurate matrix multiplication

> **XMULT(** *matrix1, matrix2***)**

where *matrix1* and *matrix2* are numeric matrices.

The XMULT function computes the matrix product like the matrix multiplication operator (*) except XMULT uses extended precision to accumulate sums of products. You should use the XMULT function only when you need great accuracy.

The following program gives an example of using 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

intersects sets

XSECT(*matrix1<, matrix2,..., matrix15>***)**

where *matrix* is a numeric or character matrix or quoted literal.

The XSECT function returns as a row vector the sorted set (without duplicates) of
the element values that are present in all of its arguments. This set is the intersection
of the sets of values in its argument matrices. When the intersection is empty, the
XSECT function returns a null matrix (zero rows and zero columns). There can be up
to 15 arguments, which must all be either character or numeric. For characters, the
element length of the result is the same as the shortest of the element lengths of the
arguments. For comparison purposes, shorter elements are padded on the right with
blanks.

For example, the statements

```
a={1 2 4 5};
b={3 4};
c=xsect(a,b);
```

return the result shown:

```
        C               1 row        1 col       (numeric)

                                       4
```

YIELD Function

calculates yield-to-maturity of a cash-flow stream and returns a scalar

YIELD(*times,flows,freq,value***)**

The YIELD function returns a scalar containing yield-to-maturity of a cash-flow
stream based on frequency and value specified.

times   is an n-dimensional column vector of times. Elements should be non-
             negative.

flows   is an n-dimensional column vector of cash-flows.

freq   is a scalar that represents the base of the rates to be used for discount-
             ing the cash-flows. If positive, it represents discrete compounding as
             the reciprocal of the number of compoundings. If zero, it represents
             continuous compounding. No negative values are allowed.

value   is a scalar 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 k-th 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 code

```
timesn=T(do(1,100,1));
flows=repeat(10,100);
freq=50;
value=682.31027;
yield=yield(timesn,flows,freq,value);
print yield;
```

produces the following output:

```
        YIELD
      0.0100001
```

Base SAS Functions Accessible from SAS/IML

The following base SAS functions are either not available from IML, or behave differently from the base function of the same name.

Function	Comment
CALL CATS	return variable must be preinitialized
MAD	conflicts with built-in function of the same name
MEDIAN	conflicts with IMLMLIB function of the same name
MOD	base function performs "fuzzing"; IML does not
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 some base functions that are not supported by SAS/IML. For example, the DATA step permits N-literals (strings that end with 'N') to be interpreted as the name of a variable, but IML 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 may 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 the RANGE function.

The functions displayed in italics are documented elsewhere in this user's guide. These functions operate on matrices as well as on 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 hexidecimal 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.
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
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
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
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 hexidecimal 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
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
SCANQ	returns the *n*th word from a character expression, ignoring delimiters that are inside quotation marks

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 a length you specify, using blank padding 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
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

CSS	returns the corrected sum of squares
CV	returns the coefficient of variation
IQR	returns the interquartile range
KURTOSIS	returns the kurtosis
LARGEST	returns the kth largest nonmissing value
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
NMISS	returns the number of missing values
ORDINAL	returns any specified order statistic
PCTL	computes percentiles
RANGE	returns the range of values
SKEWNESS	returns the skewness

SMALLEST	returns the *k*th 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
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
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 cashflow
CONVXP	returns the convexity for a periodic cashflow 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 cashflow
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 cashflow stream
SAVING	returns the future value of a periodic saving
YIELDP	returns the yield-to-maturity for a periodic cashflow stream

Macro Functions and Subroutines

CALL RESOLVE	resolves the value of a text expression at execution time
CALL SYMGET	returns the character value of a macro variable
CALL SYMGETN	returns the numeric value of a macro variable
CALL SYMPUT	sets the character value of a macro variable
CALL SYMPUTN	sets the numeric 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

CALL ALLPERM	generates all permutations of the values of several variables.
ABS	returns the absolute value
AIRY	returns the Airy function
BETA	returns the value of the beta function.
CNONCT	returns the noncentrality parameter from a chi-squared distribution
COMB	returns the number of combinations of n items taken r at a time
CONSTANT	returns some machine and mathematical constants
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
MOD	returns the remainder value
CALL RANPERK	randomly permutes the values of the arguments, and returns a permutation of k out of n values
CALL RANPERM	randomly permutes the values of the arguments
PERM	returns the number of permutations of n items taken r at a time
SIGN	returns the sign of a value
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
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
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
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.
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
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
COS   returns the cosine
COSH   returns the hyperbolic cosine
SIN   returns the sine
SINH   returns the hyperbolic sine
TAN   returns the tangent
TANH   returns the hyperbolic tangent

Truncation Functions

CEIL   returns the smallest integer $\geq$ the argument
CEILZ   returns the smallest integer that is greater than or equal to the argument, using zero fuzzing
FLOOR   returns the largest integer $\leq$ 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 containing HTML numeric character references or HTML character entity references and returns the decoded string
HTMLENCODE   encodes characters using HTML character entity references and returns the encoded string
URLDECODE   returns a string that was decoded using the URL escape syntax
URLENCODE   returns a string that was encoded using the URL escape syntax

References

Abramowitz, M. and Stegun, I.A. (1972), *Handbook of Mathematical Functions*, New York: Dover Publications, Inc.

Aiken, R.C. (1985), *Stiff Computation*, New York: Oxford University Press, Inc.

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 Application*, 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, Inc.

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 *Numerical Methods for Nonlinear Optimization*, ed. F. A. Lootsma (ed.), London: Academic Press.

Bickart, T.A. and Picel, Z. (1973), "High Order Stiffly Stable Composit Multistep Methods for Numerical Integration of Stiff Differential Equations," in *BIT*, 13, 272–286.

Bishop, Y.M., Fienberg, S.E., and Holland, P.W. (1975), *Discrete Multivariate Analysis: Theory and Practice*, Cambridge, MA: MIT Press.

Box, G.E.P. and Jenkins, G.M. (1976), *Time Series Analysis: Forecasting and Control*, Oakland, CA: Holden-Day.

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, Inc.

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, Ronald (1997). *Log-Linear Models and Logistic Regression*, 2nd ed., New York: Springer-Verlag.

Cox, D.R. and Hinkley, D.V. (1974), *Theoretical Statistics*, London: Chapman and 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.

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 Guassiens," *Ann. Sci. Univ. Clermont Ferrand II Math*, 14, 19–27.

Emerson, Phillip 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, England: John Wiley & Sons, Ltd.

Fletcher, R. and Xu, C. (1987), "Hybrid Methods for Nonlinear Least Squares," *Journal of Numerical Analysis*, 7, 371–389.

Forsythe, G.E., Malcom, M.A., and Moler, C.B. (1967), *Computer Solution of Linear Algebraic Systems*, Chapter 17, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Furnival, G.M. and Wilson, R.W. (1974), "Regression by Leaps and Bounds," *Technometrics*, 16, 499–511.

Gaffney, P.W. (1984), "A Performanace 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.

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 *Statistical Computation*, ed. R.C. Milton and J.A. Nelder, 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: M. Dekker, Inc.

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, Inc.

Grizzle, J.E., Starmer, C.F., and Koch, G.G. (1969), "Analysis of Categorical Data by Linear Models," *Biometrics*, 25, 489–504.

Hadley, G. (1963), *Linear Programming*, Reading, MA: Addison-Wesley Publishing Company, Inc.

Harvey, A.C. (1989), *Forecasting, Structural Time Series Models and the Kalman Filter*, Cambridge: Cambridge University Press.

Jenkins, M.A. and Traub, J.F. (1970), " A Three-Stage Algorithm for Real Polynomials Using Quadratic Iteration," *SIAM Journal of Numerical Analysis*, 7, 545–566.

Jenrich, R.I. and Moor,e R.H. (1975), "Maximum Likelihood Estimation by Means of Nonlinear Least Squares," *American Statistical Association, 1975 Proceedings of the Statistical Computing Section*, 57–65.

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.

Kruskal, J.B. (1964), "Nonmetric Multidimensional Scaling," *Psychometrika*, 29, 1-27, 115–129.

Nelder, J.A. and Wedderburn, R.W.M. (1972), "Generalized Linear Models," *Journal of the Royal Statistical Society*, A.3, 370.

Lee, W. and Gentle, J.E. (1986), "The LAV Procedure," *SUGI Supplememtal Library User's Guide*, Cary, NC: SAS Institute, Chapter 21, 257–260.

Lindström, P. and Wedin, P.A. (1984), "A New Linesearch Algorithm for Nonlinear Least-Squares Problems," *Mathematical Programming*, 29, 268–296.

Madsen, K. and Nielsen, H.B. (1993), "A Finite Smoothing Algorithm for Linear 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*, ed. Y. Dodge, Amsterdam: North Holland, 297–305.

McLeod, I. (1975), "Derivation of the Theoretical Autocovariance Function of Autoregressive-Moving Average Time Series," *Applied Statistics*, 24, 255–256.

Mittnik, S. (1990), "Computation of Theoretical Autocovariance Matrices of Multivariate Autoregressive Moving Average Time Series," *Journal of Royal Statistical Society*, B, 52, 151–155.

Monro, D.M. and Branch, J.L. (1976), "Algorithm AS 117. The Chirp Discrete Fourier Transform and General Length," *Applied Statistics*, 26, 351–361.

Moré, J.J. (1978), "The Levenberg-Marquardt Algorithm: Implementation and Theory," *Lecture Notes in Mathematics 630*, ed. G.A. Watson, New York: Springer-Verlag, 105–116.

Moré, J.J. and Sorensen, D.C. (1983), "Computing a Trust-Region Step," *SIAM Journal on Scientific and Statistical Computing*, 4, 553–572.

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.

Pizer, S.M. (1975), *Numerical Computing and Mathematical Analysis*, Chicago, IL: Science Research Associates, Inc.

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, J.M.D. (1977), "Restart Procedures for the Conjugate Gradient Method," *Mathematical Programming*, 12, 241–254.

Powell, J.M.D. (1978a), "A Fast Algorithm for Nonlinearly Constraint Optimization Calculations," *Numerical Analysis, Dundee 1977, Lecture Notes in Mathematics 630*, ed. G.A. Watson, New York: Springer-Verlag, 144–175.

Powell, J.M.D. (1982b), "VMCWD: A Fortran Subroutine for Constrained Optimization," *DAMTP 1982/NA4*, Cambridge, England.

Ralston, A. and Rabinowitz, P. (1978), *A First Course in Numerical Analysis*, New York: McGraw-Hill, Inc.

Rao, C.R. and Mitra, S.K. (1971), *Generalized Inverse of Matrices and Its Applications*, New York: John Wiley & Sons, Inc.

Reinsch, Christian H. (1967), "Smoothing by Spline Functions," *Numerische Mahematik*, 10, 177–183.

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*, 52:5, 702–722.

Rousseeuw, P.J. (1984), "Least Median of Squares Regression," *Journal of the American Statistical Association*, 79, 871–880.

Rousseeuw, P.J. (1985), "Multivariate Estimation with High Breakdown Point," in *Mathematical Statistics and Applications*, ed. by W. Grossmann, G. Pflug, I. Vincze, and W. Wertz, Dordrecht: Reidel Publishing Company, 283–297.

Rousseeuw, P.J. and Croux, C. (1993), "Alternatives to the Median Absolute Deviation," *Journal of the American Statistical Association*, 88, 1273–1283.

Rousseeuw, P.J. and Hubert, M. (1997), "Recent Developments in PROGRESS," L_1-*Statistical Procedures and Related Topics*, ed. by Y. Dodge, IMS Lecture Notes - Monograph Series, No. 31, 201–214.

Rousseeuw, P.J. and Leroy, A.M. (1987), *Robust Regression and Outlier Detection*, New York: John Wiley & Sons, Inc.

Rousseeuw, P.J. and Van Driessen, K. (1998), "Computing LTS Regression for Large Data Sets," Technical Report, University of Antwerp, submitted.

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, 4.

Shampine, L. (1978), "Stability Properties of Adams Codes," *Association for Computing Machinery, Transactions on Mathematical Software*, 4, 323–329.

Singleton, R.C. (1969), "An Algorithm for Computing the Mixed Radix Fast Fourier Transform," *IEEE Transactions on Audio and Electoacoustics*, AU-17, 93–103.

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.

Squire, W. (1987), "Comparison of Gauss-Hermite and Midpoint Quadrature with the Application of the Voigt Function," *Numerical Integration: Recent Developments*, eds. P. Keast and G. Fairweather, Boston: Reidel, 111–112.

Stenger, F. (1973a), "Integration Formulas Based on the Trapezoidal Formula," *Jornal 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, Ronald A. (1988), *Elements of Statistical Computing: Numerical Computation*, London: Chapman and Hall.

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., eds., (1971), *Handbook for Automatic Computation: Linear Algebra*, Volume 2, New York: Springer-Verlag.

Woodfield, Terry J. (1988), "Simulating Stationary Gaussian ARMA Time Series," *Computer Science and Statistics: Proceedings of the 20th Symposium on the Interface*, 612–617.

Young, F.W. (1981), "Quantitative Analysis of Qualitative Data," *Psychometrika*, 46, 357–388.

Chapter 21
Module Library

Chapter Contents

Chapter 21
Module Library

Overview

IMLMLIB is a library of modules written in the IML language. These modules can be used as though they were built-in functions of IML.

The library contains both functions and subroutines. You can invoke functions in assignment statements or expressions. You can invoke subroutines using CALL or RUN statements. IML automatically loads, resolves, and executes a module when you use it.

Contents of IMLMLIB

The library contains the following modules. Many of them are derived from the examples in the IML sample library. Each module is described in detail at the end of this chapter.

COLVEC	converts a matrix into a column vector
CORR	computes correlation coefficients
EXPMATRIX	computes the exponential of a matrix
GBXWHSKR	draws box and whisker diagram
GPROBCNT	draws probability contours for x-y data
GXYPLOT	draws scatter plots of x-y data
MEDIAN	returns the median of numeric data
QUADREG	performs quadratic regression
QUARTILE	computes quartiles
REGRESS	performs regression analysis
ROWVEC	converts a matrix into a row vector
RSUBSTR	replaces substrings
STANDARD	standardizes numeric data
TABPRT	prints matrices in tabular format

IMLMLIB and the STORAGE library

IML enables you to store and load matrices and modules in your own STORAGE library (refer to the chapter on storage features in *SAS/IML Software: Usage and Reference, Version 6, First Edition*). The IMLMLIB library is different from this

STORAGE library. IMLMLIB contains predefined modules that can be loaded only by IML.

The STORAGE library, on the other hand, is under the control of the user. You can store and load both matrices and modules. The STORE, LOAD, REMOVE, and RESET STORAGE commands apply only to the STORAGE library. You cannot store additional modules in IMLMLIB.

You can use the SHOW command to obtain information about the IMLMLIB and STORAGE libraries.

- SHOW OPTIONS displays the current settings of both STORAGE and IMLMLIB libraries and their open status.
- SHOW STORAGE displays the contents of the STORAGE library.
- SHOW IMLMLIB displays the contents of the IMLMLIB library.
- SHOW MODULES displays the names of the modules existing in the current environment. These include modules loaded from the STORAGE library or the IMLMLIB library and modules defined in the current session.

Accessing the IML Source Code

The IMLMLIB library is a catalog residing in the SASHELP directory. The catalog contains one entry of type IMOD for each module. Each entry is a module stored in its compiled form.

The IML source code defining the modules is available in the catalog SASHELP.IML. There is an entry of type SOURCE for each module. You can view the source code in the program editor window under DMS using the COPY command and specifying the four-level name:

SASHELP.IML.*modulename***.SOURCE**

The source code is generally followed by examples of its use.

The source code can be edited for customization or enhancements, and can be included in other IML applications. The modules also illustrate a variety of IML features that can be used to solve more complex problems.

Order of Resolution

SAS/IML resolves functions and subroutines in the following order:

- Functions

  1. IML's built-in functions
  2. User-defined IML modules existing in the current environment
  3. STORAGE library, if open
  4. SAS DATA step functions
  5. IMLMLIB library

- CALL Statement

    1. IML's built-in calls
    2. User-defined IML modules existing in the current enviornment
    3. STORAGE library, if open
    4. SAS DATA step call
    5. IMLMLIB library

- RUN Statement

    1. User-defined IML modules existing in the current environment
    2. STORAGE library, if open
    3. IML's built-in calls
    4. SAS DATA step call
    5. IMLMLIB library

Error Diagnostics

When an error occurs in any IML module, IML pauses in the module and prints error diagnostics with a full traceback that can help in locating the problem. In the case of loaded modules, however, the traceback includes line offsets instead of the absolute SAS LOG line numbers. The offsets can be used to track the problem into the actual source code that originally defined the module. The START statement at the beginning of the module definition is always at offset=1.

Note that offsets apply only to loaded modules. For modules explicitly defined in any given session, absolute line numbers are printed in the traceback.

Modules Reference

COLVEC Function

converts a matrix into a column vector

COLVEC(*matrix*)

where *matrix* is any $n \times m$ matrix.

The COLVEC function returns an $nm \times 1$ vector. It converts a matrix into a column vector in row-major order. The returned vector has 1 column and nm rows. The first n elements in the vector correspond to the first row of the input matrix, the next n elements correspond to the second row, and so on.

CORR Function

computes correlation coefficients

CORR(*matrix*)

where *matrix* is any $n \times m$ matrix, m is the number of variables, and n is the number of data points.

The CORR function returns an $m \times m$ matrix of correlation coefficients. It computes the correlation between variables for any multivariate numeric data.

EXPMATRIX Function

computes the exponential of a matrix

EXPMATRIX(*matrix*)

where *matrix* is any $n \times n$ matrix.

Given a matrix A, the EXPMATRIX function returns an $n \times n$ matrix approximating $e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}$. The function uses a Padé approximation algorithm as presented in Golub and Van Loan (1989), p. 558.

Note that this module does not exponentiate each entry of a matrix; for that, use the EXP function.

The following example demonstrates the EXPMATRIX function. For the matrix used in the example, e^{tA} is the matrix $\begin{pmatrix} e^t & te^t \\ 0 & e^t \end{pmatrix}$.

```
A = { 1 1, 0 1 };
t = 3;
X = ExpMatrix( t*A );
ExactAnswer = ( exp(t) || t*exp(t) ) //
              ( 0       ||   exp(t) );
print X, ExactAnswer;
```

The output from this code is

```
        X

20.085537 60.256611
        0 20.085537


    EXACTANSWER

20.085537 60.256611
        0 20.085537
```

GBXWHSKR Call

draws box and whisker diagrams

RUN GBXWHSKR(*matrix***);**

where *matrix* is any $n \times m$ matrix.

The GBXWHSKR module draws a box-and-whisker plot for univariate numeric data contained in the specified *matrix*. The box outlines the quartile range, and the minimum, median, and maximum points are labeled on the plot. You cannot produce graphics until you invoke the CALL GSTART statement. The plot created by the GBXWHSKR module remains open for further additions until you specify the CALL GCLOSE statement, which terminates the current graphics segment. The module source code can be edited for changes, such as adding viewports, text, or colors.

GPROBCNT Call

draws probability contours

RUN GPROBCNT($x, y <, p >$ **);**

The inputs to the GPROBCNT subroutine are as follows:

x is any $n \times m$ matrix of x-values.

y is a corresponding $n \times m$ matrix of y-values.

p is an optional probability value matrix.

The GPROBCNT module draws one contour curve corresponding to each value in the matrix p, which must contain entries between zero and one. If you do not specify the matrix p, contours for the probability values of 0.5, 0.8 and 0.9 are drawn. You cannot produce graphics until you invoke the CALL GSTART statement. The contour plot remains open for further additions until you specify the CALL GCLOSE statement, which terminates the current graphics segment. Note that this module cannot be used for general contour plots of three-dimensional data.

GXYPLOT Call

draws a scatter plot of any x-y data

RUN GXYPLOT(x, y **);**

The inputs to the GXYPLOT subroutine are as follows:

x is any $n \times m$ matrix of x-values.

y is a corresponding $n \times m$ matrix of y-values.

The GXYPLOT module draws a simple scatter plot of any numeric x-y data. Axes with labeled tickmarks are drawn as well. You cannot produce graphics until you invoke the CALL GSTART statement. The plot remains open for further additions

(such as a title and axis labels) until you specify the CALL GCLOSE staement, which terminates the current graphics segment. The module uses the GPOINT, GXAXIS, and GYAXIS calls to plot the points. The module source code can be edited to specify many of the options available for these calls.

MEDIAN Function

returns the median of numeric data

> **MEDIAN(** *matrix***)**

where *matrix* is any $n \times m$ matrix.

The MEDIAN function returns the median value for each column in the *matrix*. It computes the median of univariate numeric data contained in the specified *matrix*. When the number of data points is odd, it returns the middle element from the sorted order. When the number of data points is even, it returns the mean of the middle two elements. Missing values are ignored. If all values in a column are missing, the return value for that column is missing.

QUADREG Call

performs quadratic response surface regression

> **RUN QUADREG(** *xopt, yopt, type, parms,* x*,* y **);**

The inputs to the GPROBCNT subroutine are as follows:

xopt	is a returned value containing $m \times 1$ critical factor values.
yopt	is a returned value containing the critical response value.
type	is a returned character string containing the solution type (maximum or minimum).
parms	is a returned value containing the parameter estimates for the quadratic model.
x	is an $n \times m$ factor matrix, where m is the number of factor variables and n is the number of data points.
y	is an $n \times 1$ response vector.

The QUADREG module fits a regression model with a complete quadratic set of regressions across several factors. The estimated model parameters are divided into a vector of linear coefficients and a matrix of quadratic coefficients to obtain critical factor values that optimize the response. It further determines the type of the optima (maximum, minimum, or saddlepoint) by computing the eigenvalues of the estimated parameters.

QUARTILE Function

computes quartiles for any univariate numeric data

QUARTILE(*matrix*)

where *matrix* is any $n \times m$ matrix.

The QUARTILE function returns a 5×1 column vector for each column in the *matrix*. The column vector contains the minimum, lower quartile, median, upper quartile, and maximum values for the numeric data in the specified *matrix*. Missing values are ignored. If all values in a column are missing, the return values for that column are missing.

REGRESS Call

performs regression analysis

RUN REGRESS(x, y, *name,* <*tval*>, <*l1*>, <*l2*>, <*l3*>**);**

The inputs to the REGRESS subroutine are as follows:

x	is an $n \times m$ numeric matrix, where m is the number of variables and n is the number of data points.
y	is an $n \times 1$ response vector.
name	is an $m \times 1$ matrix of variable names.
tval	is an optional t-value.
l1, l2, l3	are optional $1 \times m$ vectors that specify linear combinations of coefficients for hypothesis testing.

The REGRESS module does regression analysis and prints results. The design matrix is given by x, and y is the response vector. The *name* vector identifies each of the variables. If you specify a t-value, the module prints a table of observed and predicted values, residuals, hat diagonal, and confidence limits for the mean and predicted values. If you also specify linear combinations with *l1*, *l2*, and *l3*, the module performs the hypothesis test $\mathbf{H} : l'b = 0$, where b is the vector of parameter estimates.

ROWVEC Function

converts a matrix into a row vector

ROWVEC(*matrix*)

where *matrix* is any $n \times m$ matrix.

The ROWVEC function returns a $1 \times nm$ vector. The specified *matrix* is converted into a row vector in row-major order. The returned vector has 1 row and nm columns. The first n elements in the vector correpond to the first row of the input matrix, the next n elements correspond to the second row, and so on.

RSUBSTR Function

replaces substrings in each entry of a given matrix

> **RSUBSTR(** x, p, l, r**)**

The inputs to the RSUBSTR subroutine are as follows:

x is any $m \times n$ character matrix.

p is an $m \times n$ matrix or a scalar that determines the starting positions for substrings to be replaced.

l is an $m \times n$ matrix or a scalar that determines the lengths of substrings to be replaced.

r is an $m \times n$ matrix or a scalar that specifies the replacement strings.

The RSUBSTR function returns an $m \times n$ matrix with substrings replaced. It replaces or substitutes substrings of the input matrix with new strings. If l is zero, the replacement string in r is simply inserted into the input matrix x at the position indicated by p.

For example, the following statements replace the first two characters of each entry in the matrix X with the the corresponding entry in the matrix R:

```
proc iml;
   x = {abc def ghi,jkl mno pqr};
   r = {z y x, w v u};
   p = 1;
   l = 2;
   c=rsubstr(x,p,l,r);
   print x;
   print c;
```

STANDARD Function

standardizes numeric data

> **STANDARD(** *matrix***)**

where *matrix* is any $n \times m$ matrix, n is the number of data points and m is the number of variables.

The STANDARD function returns a standardized $n \times m$ matrix. It standardizes each column of the input matrix, so that the mean of each column is zero and the standard deviation for each column is one.

TABPRT Call

prints matrices in tabular format

> **RUN TABPRT(** *matrix***);**

where *matrix* is any $n \times m$ matrix.

The TABPRT module prints any numeric or character matrix in table format. The regular PRINT command output is often difficult to read, especially for large matrices, where individual rows may wrap around. The module source code can be edited for further cosmetic changes, such as alternative format or field width, or for assigning specific row and column labels.

Subject Index

Syntax Index

Your Turn

If you have comments or suggestions about *SAS/IML® 9.1 User's Guide,* please send them to us on a photocopy of this page or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
E-mail: **yourturn@sas.com**

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
E-mail: **suggest@sas.com**