

SAS/IML® 15.1 User's Guide

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2018. *SAS/IML® 15.1 User's Guide*. Cary, NC: SAS Institute Inc.

SAS/IML® 15.1 User's Guide

Copyright © 2018, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

November 2018

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

SAS software may be provided with certain third-party software, including but not limited to open-source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to http://support.sas.com/thirdpartylicenses.

15.1-P1:imlug

Contents

Chapter 1. What's New in SAS/IML 15.1 . . . 1
Chapter 2. Introduction to SAS/IML Software . . . 7
Chapter 3. Understanding the SAS/IML Language . . . 13
Chapter 4. Tutorial: A Module for Linear Regression . . . 25
Chapter 5. Working with Matrices . . . 37
Chapter 6. Programming Statements . . . 59
Chapter 7. Working with SAS Data Sets . . . 83
Chapter 8. File Access . . . 111
Chapter 9. Mixed-Type Tables . . . 127
Chapter 10. Lists and Data Structures . . . 147
Chapter 11. Packages . . . 171
Chapter 12. General Statistics Examples . . . 187
Chapter 13. Submitting SAS Statements . . . 235
Chapter 14. Calling Functions in the R Language . . . 243
Chapter 15. Robust Regression Examples . . . 259
Chapter 16. Time Series Analysis and Examples . . . 287
Chapter 17. Nonlinear Optimization Examples . . . 313
Chapter 18. Statistical Graphics . . . 387
Chapter 19. Storage Features . . . 403
Chapter 20. Using SAS/IML Software to Generate SAS/IML Statements . . . 407
Chapter 21. Wavelet Analysis . . . 417
Chapter 22. Genetic Algorithms . . . 437
Chapter 23. Sparse Matrix Algorithms . . . 465
Chapter 24. Further Notes . . . 473
Chapter 25. Language Reference . . . 481
Chapter 26. Module Library . . . 1129

Subject Index **1139**

Syntax Index **1149**

Chapter 1
What's New in SAS/IML 15.1

Contents

Enhancements in SAS/IML 15.1 1
Enhancements in SAS/IML 14.3 2
Enhancements in SAS/IML 14.2 3
Enhancements in SAS/IML 14.1 4

Enhancements in SAS/IML 15.1

SAS/IML 15.1 supports several new statements and functions:

- The CONTINUE statement stops the processing of the current iteration of a DO loop and resumes processing at the next iteration of the DO loop.
- The new digital filtering functions enable you to design digital filters and apply those filters to signals. Many of the digital filtering functions start with the 'DF' prefix. The new function and subroutines are CCEPSTRUM, DFCONV, DFDESIGN, DFFILT, DFFREQZ, DFFREQZZPK, DFMEDFILT, DFORDER, DFSOSFILT, DFSOSFREQZ, DFSOSFREQZZPK, ICCEPSTRUM, and RCEPSTRUM.
- The experimental KPCATRAIN subroutine computes a kernel principal component (kPCA) analysis from training data. The experimental KCPASCORE function uses the kPCA model to score new data.
- The FEVAL function enables you to evaluate a function indirectly by specifying the name of the function and its arguments.
- The LEAVE statement exits the current DO loop and resumes processing at the statement that follows the DO loop.
- The MODULESTACK function returns the names of all modules in the module call stack.
- The SPECTROGRAM subroutine displays a spectrogram of a short-time Fourier transform of a time series signal.
- The TABLESORT subroutine sorts a table by one or more columns.

Several statements, functions, and subroutines have been enhanced:

- If the SAS VALIDVARNAME option is set to ANY, then you can use a name literal as the name of a SAS/IML symbol.

- The INV, ROOT, and SWEEP functions support multithreaded algorithms for large symmetric matrices.
- The CREATE FROM statement and the APPEND FROM statement accept multiple matrices, making it easier to write mixed data types to a SAS data set.
- The PRINT statement accepts numerical vectors for the ROWNAME= and COLNAME= options.
- The RANDGEN subroutine and RANDFUN function support new random number generators. You can use the RANDSEED subroutine to specify the following generators:

MTHYBRID	(Default) Hybrid 1998/2002 32-bit Mersenne twister
MT2002	2002 32-bit Mersenne twister
MT64	64-bit Mersenne twister
PCG	64-bit permuted congruential generator
TF2	Threefry 2x64-bit counter-based RNG
TF4	Threefry 4x64-bit counter-based RNG
RDRAND	Intel hardware-based RdRand instructions

- The VARMASIM function enables you to use the INITIAL= option to specify a matrix that initializes a simulated VARMA time series.

Enhancements in SAS/IML 14.3

SAS/IML 14.3 supports several new features:

- The SAS/IML language supports new syntax for defining and manipulating lists. You can use square brackets to define a list, subscript notation to extract a sublist, a dollar sign ($) to extract an item from a list, and the concatenation operator (||) to concatenate lists. For examples and a description of the syntax, see Chapter 10, "Lists and Data Structures."
- You can transfer data between SAS/IML tables and R data frames by using the ExportTableToR subroutine and the ImportTableFromR function.
- You can analyze complex-valued time series data by using the following new functions for time-frequency analysis:

FFTC function	computes the Fourier transform of a complex-valued signal.
IFFTC function	computes the inverse Fourier transform.
TFHILBERT function	computes a Hilbert transformation.
TFPWV function	computes a pseudo-Wigner-Ville transformation.
TFWINDOW function	generates common windows for windowed Fourier transforms.
TFSTFT function	computes a short-time Fourier transform analysis.

In addition, there are new functions, new graphics, and enhancements to existing functions:

- The DISTANCE function supports an optional argument to compute distances between observations in two different sets.
- The LAMBERTW function evaluates a branch of the Lambert W function.
- The TABLECREATE function supports creating a table from multiple matrices.
- The SAS/IML wavelet functions now support ODS graphics.
- The WideToLong subroutine converts data from wide form to long form. This is useful when you want to use the SERIES subroutine to plot multiple times series.

In this release, the traditional graphics functions and statements in SAS/IML are deprecated and have been removed from the documentation. These include the following subroutines: GBLKVP, GBLKVPD, GCLOSE, GDELETE, GDRAW, GDRAWL, GGRID, GINCLUDE, GOPEN, GPIE, GPIEXY, GPOINT, GPOLY, GPORT, GPORTPOP, GPORTSTK, GSCRIPT, GSET, GSHOW, GSTART, GSTOP, GSTRLEN, GTEXT, GVTEXT, GWINDOW, GXAXIS, and GYAXIS.

Enhancements in SAS/IML 14.2

The fundamental data type in the IML procedure is the matrix. Prior to SAS/IML 14.2, every symbol in a PROC IML program represented a matrix. A new feature in SAS/IML 14.2 is support for new nonmatrix data types: tables and lists.

- A table is a rectangular data structure that is an in-memory version of a data set. A table contains columns, which can contain either numeric and character data. You can create a table from a SAS data set or from a SAS/IML matrix. You can add new columns to a table or extract columns from a table into matrices. You can pass a table to a SAS/IML module and return a table from a module. The SAS/IML functions that manipulate tables begin with the prefix "TABLE." For more information about tables, see Chapter 9, "Mixed-Type Tables."
- A list is a nonrectangular data structure that can contain objects of different sizes and types. A list can contain numeric matrices, character matrices, tables, and other lists. You can insert new items into a list or extract elements from a list. You can pass a list to a SAS/IML module and return a list from a module. The SAS/IML functions that manipulate lists begin with the prefix "LIST." To help you work with lists, there is a new system package, called the ListUtil package. For more information about lists, see Chapter 10, "Lists and Data Structures."

The RANDGEN subroutine supports new distributions and enhancements to existing distributions:

- The F and inverse Gaussian (Wald) distributions use more accurate algorithms to generate random variates when parameter values are small.

- The new 'ConMaxPoi' distribution generates discrete random variates from the Conway-Maxwell-Poisson distribution.
- The new 'ExtremeValue' distribution generates random variates from the generalized extreme value distribution.
- The new 'GenPoisson' distribution generates discrete random variates from the generalized Poisson distribution.
- The new 'Gompertz' distribution generates random variates from the Gompertz distribution.
- The new 'Gumbel' distribution generates random variates from the Gumbel distribution.
- The new 'Integer' distribution generates random integers between two specified integers.
- The new 'ShGompertz' distribution generates random variates from the shifted Gompertz distribution.

At the time of its release, SAS/IML 14.2 interfaces correctly with recent versions of R 3.3.x. You can contact SAS Technical Support for the latest information regarding support for newer versions of R.

SAS/IML 14.2 also introduces several enhancements to the SUBMIT statement. Global SAS statements that are executed inside a SUBMIT block now also affect the SAS/IML program after the SUBMIT block. In particular:

- Macro variables that are created inside a SUBMIT block are defined and accessible in PROC IML after the SUBMIT block.
- Global SAS statements (such as the LIBNAME, FILEREF, and OPTIONS statements) that are executed inside a SUBMIT block affect the environment outside the SUBMIT block.
- ODS statements executed inside a SUBMIT block can affect subsequent output outside the SUBMIT block.

Enhancements in SAS/IML 14.1

SAS/IML supports creating large matrices and using them in many computations. When previous releases of SAS/IML ran on the Windows operating system, an individual matrix was limited to 2GB. Within this limit, you could create square numerical matrices that had about 16,000 rows and columns. In SAS/IML 14.1, you can create matrices that have up to $2^{31} - 1$ elements, provided that your system has enough RAM. (A numerical matrix that has $2^{31} - 1$ elements requires 16 GB of RAM.) This size increase enables you to create square numerical matrices that have approximately 46,000 rows and columns. Equivalently, you can create matrices that have millions of rows and hundreds of columns. You can create these large matrices on all operating systems.

However, keep in mind that many matrix computations scale cubically with the number of elements in the matrix. That is, many computations on an $n \times n$ matrix require on the order of n^3 floating point operations. Consequently, although you might be able to create extremely large matrices, computing with large matrices can be very expensive.

In addition to supporting large matrices on Windows, SAS/IML 14.1 supports the following new statements and functions:

- The PACKAGE statement enables you to install and use packages. A package consists of SAS/IML source code, documentation, data sets, and sample programs. Packages are a convenient way for programmers to download and install functions that extend the functionality of SAS/IML software. Packages are supported only on Linux and Windows operating systems. For more information, see Chapter 11, "Packages."
- The EIGEN subroutine uses vendor-supplied eigenvalue routines such as the Intel Math Kernel Library (MKL), if they are available on your system. The EIGVAL and EIGVEC computations also use vendor-supplied libraries. Because eigenvectors are not unique, the results of eigenvector computations in SAS/IML 14.1 are not necessarily identical to the results from earlier releases. If you want to restore the algorithm that was used before SAS/IML 14.1, you can use the RESET EIGEN93 statement.
- The RANDSEED subroutine uses a different initialization algorithm for certain seeds.

The interface to R in SAS/IML 14.1 supports up through R 3.2.5 on Windows. All recent releases of R are supported on Linux.

Lastly, the SAS/IML File Exchange now has a convenient shortcut: `https://communities.sas.com/sas-iml-file-exchange`. Recall that the SAS/IML File Exchange is a website where you can share SAS/IML programs and download programs that are written by others.

Chapter 2
Introduction to SAS/IML Software

Contents

Overview of SAS/IML Software 7
Highlights of SAS/IML Software 8
An Introductory SAS/IML Program 9
Support for SAS/IML Programmers 9
PROC IML Statement 10
Conventions Used in This Book 10
Typographical Conventions 10
Output of Examples 11
References 11

Overview of SAS/IML Software

SAS/IML software gives you access to a powerful and flexible programming language in a dynamic, interactive environment. The acronym IML stands for "interactive matrix language."

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 submit blocks of statements or an entire program. You can also encapsulate a series of statements by defining a module; you can call the module later to execute all of the statements in the module.

SAS/IML software is powerful. SAS/IML software enables you to concentrate on solving problems because necessary (but distracting) activities such as memory allocation and dimensioning of matrices are performed automatically. You can use built-in operators and call routines to perform complex tasks in numerical linear algebra such as matrix inversion or the computation of eigenvalues. You can define your own functions and subroutines by using SAS/IML modules. You can perform operations on a single value or take advantage of matrix operators to perform operations on an entire data matrix. For example, the following statement adds 1 to every element of the matrix **x**, regardless of the dimensions of **x**:

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

The SAS/IML language contains statements that enable you to manage data. You can read, create, and update SAS data sets in SAS/IML software without using the DATA step. For example, the following statement reads a SAS data set to obtain phone numbers for all individuals whose last name begins with "Smith":

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

The result is **phone**, a vector of phone numbers.

Highlights of SAS/IML Software

SAS/IML provides a high-level programming language.

You can program easily and efficiently with the many features for arithmetic and character expressions in SAS/IML software. You can access a wide variety of built-in functions and 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. See the section "Control Statements" on page 18 for details.

SAS/IML software operates on matrices.

Functions and statements in most programming languages manipulate and compare a single data element. However, the fundamental data element in SAS/IML software is the matrix, a two-dimensional (row × 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 can access many matrix operators, functions, and subroutines.

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

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

SAS/IML software is interactive.

You can execute SAS/IML statements one at a time and see the results immediately, or you can submit blocks of statements or an entire program. You can also define a module that encapsulates a series of statements. You can interact with an executing module by using the PAUSE statement, which enables you to enter additional statements before continuing execution.

SAS/IML software is dynamic.

You do not need to declare, dimension, or 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 observations from a SAS data set. You can create either multiple vectors (one for each variable in the data set) or a single 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.

An Introductory SAS/IML Program

This section presents a simple introductory SAS/IML program that implements a numerical algorithm that estimates the square root of a number, accurate to three decimal places. The following statements define a function module named MySqrt that performs the calculations:

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

start MySqrt(x);             /* begin module */
   y = 1;                    /* initialize y */
   do until(w<1e-3);         /* begin DO loop */
      z = y;                 /* set z=y */
      y = 0.5#(z+x/z);       /* estimate square root */
      w = abs(y-z);          /* compute change in estimate */
   end;                      /* end DO loop */
   return(y);                /* return approximation */
finish;                      /* end module */
```

You can call the MySqrt module to estimate the square root of several numbers given in a matrix literal (enclosed in braces) and print the results:

```
t = MySqrt({3,4,7,9});       /* call function MySqrt  */
s = sqrt({3,4,7,9});         /* compare with true values */
diff = t - s;                /* compute differences */
print t s diff;              /* print matrices */
```

Figure 2.1 Approximate Square Roots

t	s	diff
1.7320508	1.7320508	0
2	2	2.22E-15
2.6457513	2.6457513	4.678E-11
3	3	1.397E-9

Support for SAS/IML Programmers

SAS/IML programmers might be interested in the following resources about the SAS/IML language:

- The book *Statistical Programming with SAS/IML Software* (Wicklin 2010) provides tips and techniques for efficient SAS/IML programming and discusses IMLPlus, the programming language in the SAS/IML Studio application.
- You can ask for programming help at the SAS/IML Support Community: https://communities.sas.com/sas-iml.
- You can search for and download programs that are written by other SAS/IML programmers at the SAS/IML File Exchange: https://communities.sas.com/sas-iml-file-exchange.

- The blog *The DO Loop* (http://blogs.sas.com/content/iml/) provides advice about statistical programming in SAS, with an emphasis on SAS/IML programming.

If you are a SAS customer and are unable to resolve your problem by using the self-help resources, you are welcome to contact SAS Technical Support: http://support.sas.com/techsup/contact/.

If you are a student or faculty member at a college or university, the IML procedure is available free as part of the SAS University Edition software. Students and professors can ask questions and interact with one another at the SAS Analytics U community: https://communities.sas.com/community/sas-analytics-u.

PROC IML Statement

PROC IML <**SYMSIZE=***n1*> <**WORKSIZE=***(n2)*> **;**
  < ***SAS/IML language statements*** > **;**
  QUIT ;

You can specify the following options in the PROC IML statement:

SYMSIZE=*n1*
specifies the size of memory, in kilobytes, that is allocated to the PROC IML symbol space.

WORKSIZE=*n2*
specifies the size of memory, in kilobytes, that is allocated to the PROC IML workspace.

If you do not specify any options, PROC IML uses host-dependent defaults. In general, you do not need to be concerned with the details of memory usage because memory allocation is done automatically. However, see the section "Memory and Workspace" on page 473 for special situations.

Conventions Used in This Book

Typographical Conventions

This book uses several type styles for presenting information. The following list explains the meaning of the typographical conventions used in this book:

text	is the standard type style used for most text.
FUNCTION	is used for the name of SAS/IML functions, subroutines, and statements when they appear in the text. This convention is also used for SAS statements and options. However, you can enter these elements in your own SAS programs in lowercase, uppercase, or a mixture of the two.
SYNTAX	is used in the "Syntax" sections' initial lists of SAS statements and options.
argument	is used for option values that must be supplied by the user in the syntax definitions.

VariableName	is used for the names of variables and data sets when they appear in the text.
LibName	is used for the names of SAS librefs (such as Sasuser) when they appear in the text.
bold	is used to refer to *mathematical* matrices and vectors such as in the equation $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$.
`Code`	is used to refer to SAS/IML matrices, vectors, and expressions in the SAS/IML language such as the expression `y = A*x`. This convention is also used for example code. In most cases, this book uses lowercase type for SAS/IML statements.
italic	is used for terms that are defined in the text, for emphasis, and for references to publications.

Output of Examples

This documentation contains many short examples that illustrate how to use the SAS/IML language. Many examples end with a PRINT statement; the output for these examples appears immediately after the program statements.

References

Wicklin, R. (2010). *Statistical Programming with SAS/IML Software*. Cary, NC: SAS Institute Inc.

Chapter 3
Understanding the SAS/IML Language

Contents

Defining a Matrix **13**
Matrix Names and Literals **14**
Matrix Names 14
Matrix Literals 14
Creating Matrices from Matrix Literals **15**
Scalar Literals 15
Numeric Literals 15
Character Literals 16
Repetition Factors 16
Reassigning Values 16
Assignment Statements 17
Types of Statements **18**
Control Statements 18
Functions 18
CALL Statements and Subroutines 20
Command Statements 21
Missing Values **24**
Summary **24**

Defining a Matrix

A matrix is the fundamental structure in the SAS/IML language. A matrix is a two-dimensional array of numeric or character values. Matrices are useful for working with data and have the following properties:

- Matrices can be either numeric or character. Elements of a numeric matrix are double-precision values. Elements of a character matrix are character strings of equal length.
- The name of a matrix must be a valid SAS name. If the SAS VALIDVARNAME option is set to V7, then names must begin with a letter or the underscore character. Subsequent characters can be letters, numerals, or underscores. If the VALIDVARNAME option is set to ANY, then variable names can be name literals such as 'My Var Name'n or 'Net Products'n.
- Matrices have dimensions defined by the number of rows and columns.
- Matrices can contain elements that have missing values (see the section "Missing Values" on page 24).

The dimensions of a matrix are defined by the number of rows and columns. An $n \times p$ matrix has np elements arranged in n rows and p columns. The following nomenclature is standard in this book:

- 1×1 matrices are called *scalars*.
- $1 \times p$ matrices are called *row vectors*.
- $n \times 1$ matrices are called *column vectors*.
- The *type* of a matrix is "numeric" if its elements are numbers; the type is "character" if its elements are character strings. A matrix that has not been assigned values has an "undefined" type.

Matrix Names and Literals

Matrix Names

The name of a matrix must be a valid SAS name. If the SAS VALIDVARNAME option is set to V7, then a valid name is a character string that contains between 1 and 32 characters, begins with a letter or underscore, and contains only letters, numbers, and underscores. If the VALIDVARNAME option is set to ANY, then variable names can be a valid V7 name or can be a name literal such as 'My Var Name'n.

You associate a name with a matrix when you create or define the matrix. A matrix name exists independently of values. This means that 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 an enumeration of the values of a matrix. For example, `{1,2,3}` is a numeric matrix with three elements. A matrix literal can have a single element (a scalar), or it can be an array of many elements. The matrix can be numeric or character. The dimensions of the matrix are automatically determined by the way you punctuate the values.

Use curly braces ({ }) to enclose the values of a matrix. Within the braces, values must be either all numeric or all character. Use commas to separate the rows. If you specify multiple rows, all rows must have the same number of elements.

You can specify any of the following types of elements:

- a number. You can specify numbers with or without decimal points, and in standard or scientific notation. For example, `5`, `3.14`, or `1E-5`.
- a period (.), which represents a missing numeric value.
- a number in brackets ([ ]), which represents a repetition factor.

- a character string. Character strings can be enclosed in single quotes (') or double quotes ("), but they do not need to have quotes. Quotes are required when there are no enclosing braces or when you want to preserve case, special characters, or blanks in the string. Special characters include the following: ?, =, *, :, (,), {, and }.

  If the string has embedded quotes, you must double them, as shown in the following statements:

```
w1 = "I said, ""Don't fall!""";
w2 = 'I said, "Don''t fall!"';
```

Creating Matrices from Matrix Literals

You can create a matrix by using matrix literals: simply list the element values inside of curly braces. You can also create a matrix by calling a function, a subroutine, or an assignment statement. The following sections present some simple examples of matrix literals. For more information about matrix literals, see Chapter 5, "Working with Matrices."

Scalar Literals

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

```
a = 12;
a = . ;          /* reassign to another numeric value */
a = 'hi there'; /* reassign to a character value */

option validvarname = any;
'my message'n = "Hi there";     /* a name literal */
```

Numeric Literals

To specify a matrix literal with multiple elements, enclose the elements in braces. Use commas to separate the rows of a matrix. For example, the following statements assign and print matrices of various dimensions:

```
x = {1 . 3 4 5 6};                  /* 1 x 6 row vector */
y = {1,2,3,4};                      /* 4 x 1 column vector */
z = 3#y;                            /* 3 times the vector y */
w = {1 2, 3 4, 5 6};                /* 3 x 2 matrix */
print x, y z w;
```

Figure 3.1 Matrices Created from Numeric Literals

x
1 . 3 4 5 6

Figure 3.1 *continued*

y	z	w	
1	3	1	2
2	6	3	4
3	9	5	6
4	12		

Character Literals

You can define a character matrix literal by specifying character strings between braces. If you do not place quotes around the strings, all characters are converted to uppercase. You can use either single or double quotes to preserve case and to specify strings that contain blanks or special characters. For character matrix literals, the length of the elements is determined by the longest element. Shorter strings are padded on the right with blanks. For example, the following statements define and print two 1×2 character matrices with string length 4 (the length of the longer string):

```
a = { abc   defg};                  /* no quotes; uppercase */
b = {'abc' 'DEFG'};                 /* quotes; case preserved */
print a, b;
```

Figure 3.2 Matrices Created from Character Literals

a	
ABC	DEFG

b	
abc	DEFG

Repetition Factors

A repetition factor can be placed in brackets before a literal element to have the element repeated. For example, the following two statements are equivalent:

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

Reassigning Values

You can assign new values to a matrix at any time. The following statements create a 2×3 numeric matrix named **a**, then redefine **a** to be a 1×3 character matrix:

```
a = {1 2 3, 6 5 4};
a = {'Sales' 'Marketing' 'Administration'};
```

Assignment Statements

Assignment statements create matrices by evaluating expressions and assigning the results. The expressions can be composed of operators (for example, matrix multiplication) or functions that operate on matrices (for example, matrix inversion). 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.

Functions as Expressions

You can create matrices as a result of a function call. Scalar functions such as the LOG function or the SQRT function operate on each element of a matrix. Matrix functions such as the INV function or the RANK function operate on the entire matrix. The following statements are examples of function calls:

```
a = sqrt(b);                    /* elementwise square root */
y = inv(x);                     /* matrix inversion */
r = rank(x);                    /* ranks (order) of elements */
```

The SQRT function assigns each element of **a** the square root of the corresponding element of **b**. The INV function computes the inverse matrix of **x** and assigns the results to **y**. The RANK function creates a matrix **r** with elements that are the ranks of the corresponding elements of **x**.

Operators within Expressions

Three types of operators can be used in assignment statement expressions. The matrices on which an operator acts must have types and dimensions that 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 (**-A**).
- Binary operators are placed between operands (**A*B**).
- Postfix operators are placed after an operand (**A′**).

All operators can work on scalars, vectors, or matrices, provided that the operation makes sense. For example, you can add a scalar to a matrix or divide a matrix by a scalar. The following statement is an example of using operators in an assignment statement:

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

This assignment statement creates a matrix **y** in which each negative element of the matrix **x** is replaced with zero. The statement actually contains two expressions that are evaluated. The expression **x>0** is an operation that compares each element of **x** to zero and creates a temporary matrix of results; an element of the temporary matrix is 1 when the corresponding element of **x** is positive, and 0 otherwise. The original matrix **x** is then multiplied elementwise by the temporary matrix, resulting in the matrix **y**.

See Chapter 25, "Language Reference," for a complete listing and explanation of operators.

Types of Statements

Statements in the SAS/IML language 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 EIGEN subroutine computes eigenvalues and eigenvectors.

Command statements
: perform special processing, such as setting options, displaying windows, and handling input and output. For example, the MATTRIB statement associates matrix characteristics with matrix names.

Control Statements

The SAS/IML language has statements that control program execution. You can use control statements to direct the execution of your program and to define DO groups and modules. Some control statements are shown in the following table:

Table 3.1 Control Statements

Statement	Description
DO, END	Specifies a group of statements
Iterative DO, END	Defines an iteration loop
GOTO, LINK	Specifies the next program statement to be executed
IF-THEN/ELSE	Conditionally routes execution
PAUSE	Instructs a module to pause during execution
QUIT	Exits from the IML procedure
RESUME	Instructs a module to resume execution
RETURN	Returns from a LINK statement or module
RUN	Executes a module
START, FINISH	Defines a module
STOP, ABORT	Stops the execution of an IML program

See Chapter 6, "Programming Statements," for more information about control statements.

Functions

The general form of a function is *result* = FUNCTION(*arguments*) where *arguments* is a list of matrix names, matrix literals, or expressions. Functions always return a single matrix, whereas subroutines can return

multiple matrices or no matrices at all. If a function returns a character matrix, the matrix to hold the result is allocated with a string length equal to the longest element, and all shorter elements are padded on the right with blanks.

Categories of Functions

Many functions fall into one of the following general categories:

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.

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.

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 reshaping functions
: manipulate the matrix argument and returns a reshaped matrix. For example, the DIAG function returns a diagonal matrix with values and dimensions that are determined by the argument matrix.

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

statistical functions
: perform statistical operations on the matrix argument. For example, the RANK function returns a matrix that contains the ranks of the argument matrix.

The SAS/IML language also provides functions in the following general categories:

- matrix sorting and BY-group processing
- numerical linear algebra
- optimization
- random number generation
- time series analysis
- wavelet analysis

See the section "Statements, Functions, and Subroutines by Category" on page 492 for a complete listing of SAS/IML functions.

Exceptions to the SAS DATA Step

The SAS/IML language supports most functions that are supported in the SAS DATA step. These functions almost always accept matrix arguments and usually act elementwise so that the result has the same dimension as the argument. See the section "Base SAS Functions Accessible from SAS/IML Software" on page 1104 for a list of these functions and also a small list of functions that are not supported by SAS/IML software or that behave differently than their Base SAS counterparts.

The SAS/IML random number functions RANDGEN and RANDFUN are built-in functions that produce the same streams as the RAND function of the DATA step. The seed for the SAS/IML functions is set by using the RANDSEED subroutine. For example, you can use the following statement to create a 10×1 vector of random numbers:

```
call randseed(12345);
x = randfun(10, "uniform");
/* or use RANDGEN: allocate a vector, then fill it */
x = j(10, 1);
call randgen(x, "uniform");
```

SAS/IML software does not support the OF clause of the SAS DATA step. For example, the following statement cannot be interpreted in SAS/IML software:

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

The term **x1-x10** would be interpreted as subtraction of the two matrix arguments rather than its DATA step meaning, the variables X1 through X10.

CALL Statements and Subroutines

Subroutines (also called "CALL statements") perform calculations, operations, or interact with the SAS system. 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 a list of matrix names, matrix literals, or expressions. If you specify several arguments, use commas to separate them. When using output arguments that are computed by a subroutine, always use variable names instead of expressions or literals.

Creating Matrices with CALL Statements

Matrices are created whenever a CALL statement returns one or more result matrices. For example, the following statement returns two matrices (vectors), **val** and **vec**, that contain the eigenvalues and eigenvectors, respectively, of the matrix **A**:

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

You can program your own subroutine by using the START and FINISH statements to define a module. You can then execute the module with a CALL or RUN statement. For example, the following statements define a module named MyMod which returns matrices that contain 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,{1 2 4 9});
```

Execution of the module statements creates matrices **S** and **L** which contain the square roots and natural logs, respectively, of the elements of the third argument.

Interacting with the SAS System

You can use CALL statements to manage SAS data sets or to access the PROC IML graphics system. For example, the following statement deletes the SAS data set named MyData:

```
call delete(MyData);
```

The following statements activate the traditional graphics system and produce a crude scatter plot:

```
x = 0:100;
y = 50 + 50*sin(6.28*x/100);
call gstart;                      /* activate the graphics system */
call gopen;                       /* open a new graphics segment */
call gpoint(x,y);                 /* plot the points */
call gshow;                       /* display the graph */
call gclose;                      /* close the graphics segment */
```

SAS/IML software supports two other kinds of graphics:

- The high-level ODS statistical graphics in PROC IML are discussed in Chapter 18, "Statistical Graphics."
- SAS/IML Studio, which is distributed as part of SAS/IML software, contains dynamically linked graphics. See the *SAS/IML Studio: User's Guide* for a description of the graphs in SAS/IML Studio.

Command Statements

Command statements are used to perform specific system actions, such as storing and loading matrices and modules, or to perform special data processing requests. The following table lists some commands and the actions they perform.

Table 3.2 Command Statements

Statement	Description
FREE	Frees memory associated with a matrix
LOAD	Loads a matrix or module from a 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	Displays system information
STORE	Stores a matrix or module in the storage library

These commands play an important role in SAS/IML software. You can use them to control information displayed about matrices, symbols, or modules.

If a certain computation requires almost all of the memory on your computer, you can use commands to store extraneous matrices in the storage library, free the matrices of their values, and reload them later when you need them again. For example, the following statements define several matrices:

```
proc iml;
a = {1 2 3, 4 5 6, 7 8 9};
b = {2 2 2};
show names;
```

Figure 3.3 List of Symbols in RAM

```
SYMBOL   ROWS   COLS TYPE   SIZE
------ ------ ------ ---- ------
a           3      3 num       8
b           1      3 num       8
 Number of symbols = 2  (includes those without values)
```

Suppose that you want to compute a quantity that does not involve the **a** matrix or the **b** matrix. You can store **a** and **b** in a 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 (or the STORAGE function), as shown in the following statements:

```
store a b;                  /* store the matrices */
show storage;               /* make sure the matrices are saved */
free a b;                   /* free the RAM */
```

The output from the SHOW STORAGE statement (see Figure 3.4) indicates that there are two matrices in storage. (There are no modules in storage for this example.) Catalogs support only V7 names; you cannot store the value of a name literal in a catalog.

Figure 3.4 List of Symbols in Storage

```
Contents of storage library = WORK.IMLSTOR

Matrices:
A                               B

Modules:
```

You can load these matrices from the storage library into RAM with the LOAD command, as shown in the following statement:

```
load a b;
```

See Chapter 19, "Storage Features," for more details about storing modules and matrices.

Data Management Commands

SAS/IML software has many 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 table lists some commands and the actions they perform.

Table 3.3 Data Management Statements

Statement	Description
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 satisfy some condition
LIST	Lists records
PURGE	Purges records marked for deletion
READ	Reads records from a SAS data set into IML matrices
SETIN	Sets a SAS data set to be the input data set
SETOUT	Sets a SAS data set to be the output data set
SORT	Sorts a SAS data set
USE	Opens an existing SAS data set for reading

These commands can be used to perform data management. 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 and append or delete records. If you have a matrix of values, you can output the values to a SAS data set with the APPEND command. See Chapter 7, "Working with SAS Data Sets," and Chapter 8, "File Access," for more information about these commands.

Missing Values

With SAS/IML software, a numeric element can have a special value called a *missing value*, which 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 that involve a SAS data set, you can append or delete missing values. All operations that move values also move missing values.

However, for efficiency reasons, SAS/IML software does not support missing values in most matrix operations and functions. For example, matrix multiplication of a matrix with missing values is not supported. Furthermore, many linear algebraic operations are not mathematically defined for a matrix with missing values. For example, the inverse of a matrix with missing values is meaningless.

See Chapter 5, "Working with Matrices," and Chapter 24, "Further Notes," for more details about missing values.

Summary

This chapter introduced the fundamentals of the SAS/IML language, including the basic data element, the matrix. You learned several ways to create matrices: assignment statements, matrix literals, and CALL statements that return matrix results.

The chapter also introduced various types of programming statements: commands, control statements, iterative statements, module definitions, functions, and subroutines.

Chapter 4, "Tutorial: A Module for Linear Regression," offers an introductory tutorial that demonstrates how to use SAS/IML software for statistical computations.

Chapter 4
Tutorial: A Module for Linear Regression

Contents

Overview of Linear Regression **25**
Example: Solving a System of Linear Equations **26**
A Module for Linear Regression **27**
Orthogonal Regression **30**
Plotting Regression Results **32**
Creating ODS Graphics 32
SAS/IML Studio Graphics 34

Overview of Linear Regression

You can use SAS/IML software to solve mathematical problems or implement new statistical techniques and algorithms. Formulas and matrix equations are easily translated in the SAS/IML language. For example, if $\mathbf{X}$ is a data matrix and $\mathbf{Y}$ is a vector of observed responses, then you might be interested in the solution, $\boldsymbol{b}$, to the matrix equation $\mathbf{X}\boldsymbol{b} = \mathbf{Y}$. In statistics, the data matrices that arise often have more rows than columns and so an exact solution to the linear system is impossible to find. Instead, the statistician often solves a related equation: $\mathbf{X}'\mathbf{X}\boldsymbol{b} = \mathbf{X}'\mathbf{Y}$. The following mathematical formula expresses the solution vector in terms of the data matrix and the observed responses:

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

This mathematical formula can be translated into the following SAS/IML statement:

```
b = inv(X`*X) * X`*Y;       /* least squares estimates */
```

This assignment statement uses a built-in function (INV) and matrix operators (transpose and matrix multiplication). It is mathematically equivalent to (but less efficient than) the following alternative statement:

```
b = solve(X`*X, X`*Y);     /* more efficient computation */
```

If a statistical method has not been implemented directly in a SAS procedure, you can program it by using the SAS/IML language. The most commonly used mathematical and matrix operations are built directly into the language, so programs that require many statements in other languages require only a few SAS/IML statements.

Example: Solving a System of Linear Equations

Because the syntax of the SAS/IML language is similar to the notation used in linear algebra, it is often possible to directly translate mathematical methods from matrix-algebraic expressions into executable SAS/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

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

where $\mathbf{A}$ is the matrix of coefficients for the linear system. Because $\mathbf{A}$ is nonsingular, the system has a solution given by

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

This example solves this linear system of equations.

1 Define the matrices $\mathbf{A}$ and $\mathbf{c}$. Both of these matrices are input as matrix literals; that is, you type the row and column values as discussed in Chapter 3, "Understanding the SAS/IML Language."

```
proc iml;
a = {3  -1  2,
     2  -2  3,
     4   1 -4};
c = {8, 2, 9};
```

2 Solve the equation by using the built-in INV function and the matrix multiplication operator. The INV function returns the inverse of a square matrix and * is the operator for matrix multiplication. Consequently, the solution is computed as follows:

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

Figure 4.1 The Solution of a Linear System of Equations

x
3
5
2

3 Equivalently, you can solve the linear system by using the more efficient SOLVE function, as shown in the following statement:

```
x = solve(a, c);
```

After SAS/IML executes the statements, the rows of the vector **x** contain the x_1, x_2, and x_3 values that solve the linear system.

You can end PROC IML by using the QUIT statement:

```
quit;
```

A Module for Linear Regression

The linear systems that arise naturally in statistics are usually *overconstrained*, meaning that the $\mathbf{X}$ matrix has more rows than columns and that an exact solution to the linear system is impossible to find. Instead, the statistician assumes a linear model of the form

$$\mathbf{y} = \mathbf{X}\mathbf{b} + \mathbf{e}$$

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

The following example illustrates some programming techniques by using SAS/IML statements to perform linear regression. (The example module does not replace regression procedures such as the REG procedure, which are more efficient for regressions and offer a multitude of diagnostic options.)

Suppose you have response data $\mathbf{y}$ measured at five values of the independent variable $\mathbf{X}$ and you want to perform a quadratic regression. In this case, you can define the design matrix $\mathbf{X}$ and the data vector $\mathbf{y}$ as follows:

```
proc iml;
x = {1 1 1,
     1 2 4,
     1 3 9,
     1 4 16,
     1 5 25};
y = {1, 5, 9, 23, 36};
```

You can compute the least squares estimate of **b** by using the following statement:

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

Figure 4.2 Parameter Estimates

b
2.4
-3.2
2

The predicted values are found by multiplying the data matrix and the parameter estimates; the residuals are the differences between actual and predicted responses, as shown in the following statements:

```
yhat = x*b;
r = y-yhat;
print yhat r;
```

Figure 4.3 Predicted and Residual Values

yhat	r
1.2	-0.2
4	1
10.8	-1.8
21.6	1.4
36.4	-0.4

To estimate the variance of the responses, calculate the sum of squared errors (SSE), the error degrees of freedom (DFE), and the mean squared error (MSE) as follows:

```
sse = ssq(r);
dfe = nrow(x)-ncol(x);
mse = sse/dfe;
print sse dfe mse;
```

Figure 4.4 Statistics for a Linear Model

sse	dfe	mse
6.4	2	3.2

Notice that in computing the degrees of freedom, you use the function NCOL to return the number of columns of **X** and the function NROW to return the number of rows.

Now suppose you want to solve the problem repeatedly on new data. To do this, you can define a module. 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(x`*x);                 /* inverse of X'X        */
  beta = xpxi * (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               */
  results = sse || dfe || mse || rsquare;
  print results[c={"SSE" "DFE" "MSE" "RSquare"}
                L="Regression Results"];

  stdb = sqrt(vecdiag(xpxi)*mse); /* std of estimates      */
  t = beta/stdb;                    /* parameter t tests     */
  prob = 1-probf(t#t,1,dfe);        /* p-values              */
  paramest = beta || stdb || t || prob;
  print paramest[c={"Estimate" "StdErr" "t" "Pr>|t|"}
                 L="Parameter Estimates" f=Best6.];
  print y yhat resid;
finish Regress;                     /* end module            */
```

Assuming that the matrices **x** and **y** are defined, you can run the Regress module as follows:

```
run Regress;                      /* run module            */
```

Figure 4.5 The Results of a Regression Module

Regression Results

SSE	DFE	MSE	RSquare
6.4	2	3.2	0.9923518

Parameter Estimates

Estimate	StdErr	t	Pr>|t|
2.4	3.8367	0.6255	0.5955
-3.2	2.9238	-1.094	0.388
2	0.4781	4.1833	0.0527

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

Orthogonal Regression

In the previous section, you ran a module that computes parameter estimates and statistics for a linear regression model. All of the matrices used in the Regress module are global variables because the Regress module does not have any arguments. Consequently, you can use those matrices in additional calculations.

Suppose you want to correlate the parameter estimates. To do this, you can calculate the covariance of the estimates, then scale the covariance into a correlation matrix with values of 1 on the diagonal. The following statements perform these operations:

```
covb = xpxi*mse;              /* covariance of estimates */
s = 1/sqrt(vecdiag(covb));    /* standard errors */
corrb = diag(s)*covb*diag(s); /* correlation of estimates */
print covb, s, corrb;
```

The results are shown in Figure 4.6. The covariance matrix of the estimates is contained in the **covb** matrix. The vector **s** contains the standard errors of the parameter estimates and is used to compute the correlation matrix of the estimates (**corrb**).

Equivalently, you can form the **covb** matrix and then call the COV2CORR function in order to generate the **corrb** matrix: **corrb = cov2corr(covb)**.

Figure 4.6 Covariance and Correlation Matrices for Estimates

Regression Results

SSE	DFE	MSE	RSquare
6.4	2	3.2	0.9923518

Parameter Estimates

Estimate	StdErr	t	Pr>|t|
2.4	3.8367	0.6255	0.5955
-3.2	2.9238	-1.094	0.388
2	0.4781	4.1833	0.0527

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

covb

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

s
0.260643
0.3420214
2.0916501

Figure 4.6 *continued*

corrb		
1	-0.941376	0.8722784
-0.941376	1	-0.981105
0.8722784	-0.981105	1

You can also use the Regress module to carry out an orthogonalized regression version of the previous polynomial regression. In general, the columns of **X** are not 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 you use orthogonal polynomial regression, you can expect the statistics of fit to be the same and expect the estimates to be more stable and uncorrelated.

To perform an orthogonal regression on the data, you must first create a vector that contains 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. The following statements perform these operations:

```
x1 = x[,2];                    /* data = second column of X */
x = orpol(x1, 2);              /* generate orthogonal polynomials */
run Regress;                   /* run Regress module */

covb = xpxi*mse;               /* covariance of estimates */
s = 1 / sqrt(vecdiag(covb));
corrb = diag(s)*covb*diag(s);
print covb, s, corrb;
```

Figure 4.7 Covariance and Correlation Matrices for Estimates

Regression Results			
SSE	DFE	MSE	RSquare
6.4	2	3.2	0.9923518

Parameter Estimates			
Estimate	StdErr	t	Pr>|t|
33.094	1.7889	18.5	0.0029
27.828	1.7889	15.556	0.0041
7.4833	1.7889	4.1833	0.0527

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

covb		
3.2	0	0
0	3.2	0
0	0	3.2

Figure 4.7 *continued*

s
0.559017
0.559017
0.559017

corrb		
1	0	0
0	1	0
0	0	1

For these data, the off-diagonal values of the **corrb** matrix are displayed as zeros. For some analyses you might find that certain matrix elements are very close to zero but not exactly zero because of the computations of floating-point arithmetic. You can use the RESET FUZZ option to control whether small values are printed as zeros.

Plotting Regression Results

You can produce high-resolution ODS graphics by using modules in the IMLMLIB library. See Chapter 18, "Statistical Graphics," for more information about high-resolution graphics.

Alternatively, you can create graphics by using the SAS/IML Studio application, which is a Window application that is distributed as part of SAS/IML software. SAS/IML Studio is an environment for developing SAS/IML programs. SAS/IML Studio includes high-level statistical graphics such as scatter plots, histograms, and bar charts. You can use the SAS/IML Studio graphical user interface (GUI) to create graphs, or you can create and modify graphics by writing programs. The GUI is described in the *SAS/IML Studio: User's Guide*. See *SAS/IML Studio for SAS/STAT Users* for an introduction to programming in SAS/IML Studio.

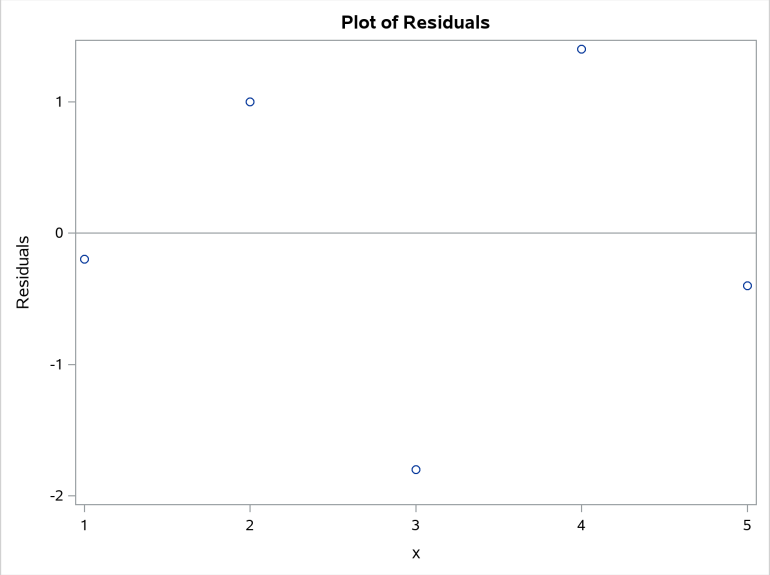
Creating ODS Graphics

You can continue the example of this chapter by using the SCATTER subroutine to create scatter plots of the data, the predicted values, and the residuals.

The following statements plot the residual values versus the explanatory variable. The graph is shown in Figure 4.8.

```
title "Plot of Residuals";
call scatter(x1, resid) label={"x" "Residuals"}
                        other="refline 0 / axis=y";
```

Figure 4.8 Residual Plot



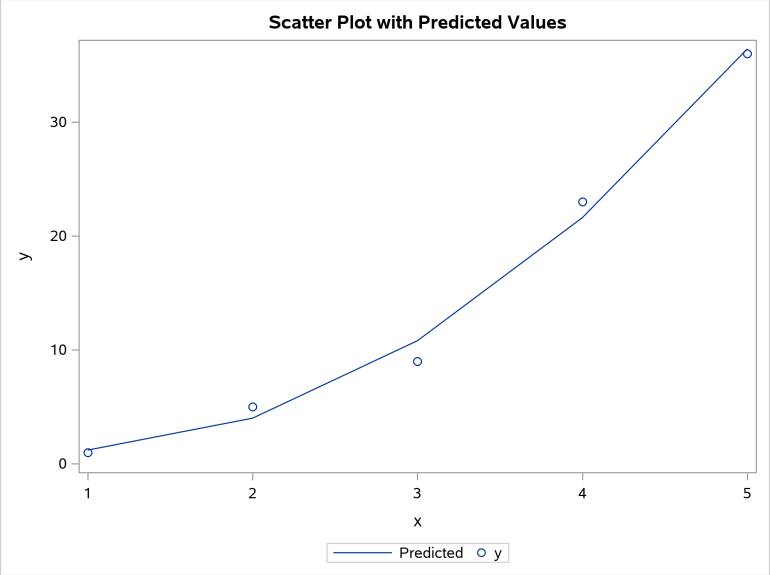
In a similar way, you can use the SCATTER routine to plot the predicted values $\hat{\mathbf{y}}$ against $\mathbf{x}$.

For more complicated graphs, you might choose to call the SGPLOT procedure directly from within your SAS/IML program. You can use the SUBMIT statement and ENDSUBMIT statement to call any SAS procedure from within PROC IML. For this example, you need to first create a SAS data set that contains the data. The following statements write the data to the RegData data set, then call the SGPLOT procedure to create a scatter plot overlaid with a line plot:

```
create RegData var {"x1" "y" "yhat"};
append;
close RegData;

submit;
title "Scatter Plot with Predicted Values";
proc sgplot data=RegData;
   label x1="x" yhat="Predicted";
   series x=x1 y=yhat;
   scatter x=x1 y=y;
run;
endsubmit;
```

The CREATE statement creates a SAS data set named RegData. The APPEND statement writes the data to the data set. The SUBMIT and ENDSUBMIT statements bracket SAS programs statements that generate Figure 4.9.

Figure 4.9 Plot of Predicted and Observed Values



SAS/IML Studio Graphics

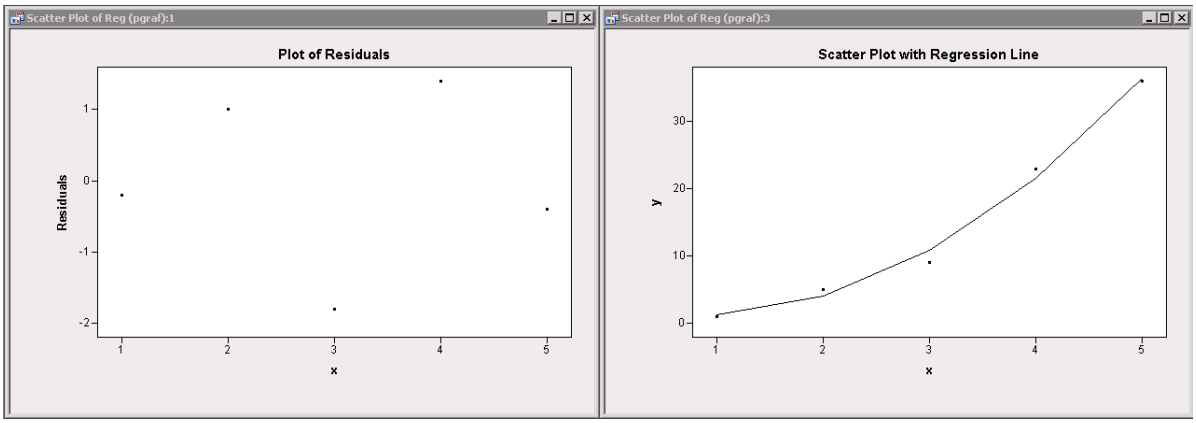
If you develop your SAS/IML programs in SAS/IML Studio, you can use high-level statistical graphics. For example, the following statements create three scatter plots that duplicate the low-resolution plots created in the previous section. Two of the plots are shown in Figure 4.10. The main steps in the program are indicated by numbered comments; these steps are explained in the list that follows the program.

```
x = {1 1 1, 1 2 4, 1 3 9, 1 4 16, 1 5 25};        /* 1 */
y = {1, 5, 9, 23, 36};
x1 = x[,2];                  /* data = second column of X */
x = orpol(x1,2);             /* generates orthogonal polynomials */
run Regress;                 /* runs the Regress module */

declare DataObject dobj;                           /* 2 */
dobj = DataObject.Create("Reg",                    /* 3 */
         {"x" "y" "Residuals" "Predicted"},
          x1 || y || resid || yhat);

declare ScatterPlot p1, p2, p3;
p1 = ScatterPlot.Create(dobj, "x", "Residuals");   /* 4 */
p1.SetTitleText("Plot of Residuals", true);

p2 = ScatterPlot.Create(dobj, "x", "Predicted");   /* 5 */
p2.SetTitleText("Plot of Predicted Values", true);

p3 = ScatterPlot.Create(dobj, "x", "y");           /* 6 */
p3.SetTitleText("Scatter Plot with Regression Line", true);
p3.DrawUseDataCoordinates();
p3.DrawLine(x1,yhat);                              /* 7 */
```

To completely understand this program, you should read *SAS/IML Studio for SAS/STAT Users*. The following list describes the main steps of the program:

1. Use SAS/IML to create the data and run the Regress module.
2. Specify that the **`dobj`** variable is an object of the DataObject class. SAS/IML Studio extends the SAS/IML language by adding object-oriented programming techniques.
3. Create an object of the DataObject class from SAS/IML vectors.
4. Create a scatter plot of the residuals versus the values of the explanatory variable.
5. Create a scatter plot of the predicted values versus the values of the explanatory variable.
6. Create a scatter plot of the observed responses versus the values of the explanatory variable.
7. Overlay a line for the predicted values.

Figure 4.10 Graphs Created by SAS/IML Studio



Chapter 5
Working with Matrices

Contents

Overview of Working with Matrices **37**
Entering Data as Matrix Literals **38**
Scalars 38
Matrices with Multiple Elements 38
Using Assignment Statements **40**
Simple Assignment Statements 40
Functions That Generate Matrices 41
Index Vectors 45
Using Matrix Expressions **46**
Operators 46
Compound Expressions 46
Elementwise Binary Operators 47
Subscripts 49
Subscript Reduction Operators **55**
Displaying Matrices with Row and Column Headings **56**
The AUTONAME Option in the RESET Statement 57
The ROWNAME= and COLNAME= Options in the PRINT Statement 57
The MATTRIB Statement 57
More about Missing Values **58**

Overview of Working with Matrices

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

- entering data as a matrix literal
- using assignment statements
- using functions that generate matrices
- creating submatrices from existing matrices with subscripts
- using SAS data sets (see Chapter 7, "Working with SAS Data Sets," for more information)

Chapter 3, "Understanding the SAS/IML Language," describes some of these techniques.

After you define matrices, you have access to many operators and functions for forming matrix expressions. These operators and functions facilitate programming and enable you to refer to submatrices. This chapter describes how to work with matrices in the SAS/IML language.

Entering Data as Matrix Literals

The simplest way to create a matrix is to define a matrix literal by entering the matrix elements. A matrix literal can contain numeric or character data. 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 $n \times p$ matrix has n rows and p columns.

Scalars

Scalars are matrices that have only one element. You can define a scalar by typing the matrix name on the left side of an assignment statement and its value on the right side. The following statements create and display several examples of scalar literals:

```
option validvarname = any;
proc iml;
x = 12;
y = 12.34;
z = .;
a = 'Hello';
'my msg'n = "Hi there";
print x y z a 'my msg'n;
```

The output is displayed in Figure 5.1. Notice that you need to use either single quotes (') or double quotes (") when defining a character literal. Using quotes preserves the case and embedded blanks of the literal. It is also always correct to enclose data values within braces ({ }).

Figure 5.1 Examples of Scalar Quantities

x	y	z	a	my msg
12	12.34	.	Hello	Hi there

Matrices with Multiple Elements

To enter a matrix having multiple elements, use braces ({ }) to enclose the data values. If the matrix has multiple rows, use commas to separate them. Inside 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 you have one week of data on daily coffee consumption (cups per day) for four people in your office. Create a 4×5 matrix called **coffee** with each person's consumption represented by a row of the matrix and each day represented by a column. The following statements use the RESET PRINT command so that the result of each assignment statement is displayed automatically:

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

Figure 5.2 A 4×5 Matrix

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

Next, you can create a character matrix called **names** with rows that contains the names of the coffee drinkers in your office. Notice in Figure 5.3 that if you do not use quotes, characters are converted to uppercase.

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

Figure 5.3 A Column Vector of Names

names	4 rows	1 col	(character, size 6)

JENNY
LINDA
JIM
SAMUEL

Notice that RESET PRINT statement produces output that includes the name of the matrix, its dimensions, its type, and (when the type is character) the element size of the matrix. The element size represents the length of each string, and it is determined by the length of the longest string.

Next display the **coffee** matrix using the elements of **names** as row names by specifying the ROWNAME= option in the PRINT statement:

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

Figure 5.4 Rows of a Matrix Labeled by a Vector

	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 about writing expressions are described in the section "Using Matrix Expressions" on page 46.

Simple Assignment Statements

Simple assignment statements involve an equation that has a matrix name on the left side and either an expression or a function that generates a matrix on the right side.

Suppose that 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**. You can turn off the automatic printing so that you can customize the output with the ROWNAME=, FORMAT=, and LABEL= options in the PRINT statement, as shown in the following statements:

```
reset noprint;
dayCost = 0.30 # coffee;   /* elementwise multiplication */
print dayCost[rowname=names format=8.2 label="Daily totals"];
```

Figure 5.5 Daily Cost for Each Employee

	Daily totals				
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 by using the matrix multiplication operator (*). First create a 5×1 vector of ones. This vector sums the daily costs for each person when multiplied with the **coffee** matrix. (A more efficient way to do this is by using subscript reduction operators, which are discussed in the section "Using Matrix Expressions" on page 46.) The following statements perform the multiplication:

```
ones = {1,1,1,1,1};
weektot = dayCost * ones;  /* matrix-vector multiplication */
print weektot[rowname=names format=8.2 label="Weekly totals"];
```

Figure 5.6 Weekly Total for Each Employee

	Weekly totals
JENNY	3.90
LINDA	3.00
JIM	1.80
SAMUEL	6.00

You might want to calculate the average number of cups consumed per day in the office. You can use the SUM function, which returns the sum of all elements of a matrix, to find the total number of cups consumed in the office. Then divide the total by 5, the number of days. The number of days is also the number of columns in the **coffee** matrix, which you can determine by using the NCOL function. The following statements perform this calculation:

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

Figure 5.7 Total and Average Number of Cups for the Office

Total number of cups
49

Daily average
9.8

Functions That Generate Matrices

SAS/IML software has many useful built-in functions that generate matrices. For example, the J function creates a matrix with a given dimension and specified element value. You can use this function to initialize a matrix to a predetermined size. Here are several functions that generate matrices:

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
REPEAT	creates a new matrix by repeating elements of the argument matrix
SHAPE	shapes a new matrix from the argument

The sections that follow illustrate the functions that generate matrices. The output of each example is generated automatically by using the RESET PRINT statement:

```
reset print;
```

The BLOCK Function

The BLOCK function has the following general form:

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

The BLOCK function creates a block-diagonal matrix from the argument matrices. For example, the following statements form a block-diagonal matrix:

```
a = {1 1, 1 1};
b = {2 2, 2 2};
c = block(a,b);
```

Figure 5.8 A Block-Diagonal 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 following general form:

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

It creates a matrix that has *nrow* rows, *ncol* columns, and all elements equal to *value*. The *ncol* and *value* arguments are optional; if they are not specified, default values are used. In many statistical applications, it is helpful to be able to create a row (or column) vector of ones. (You did so to calculate coffee totals in the previous section.) You can do this with the J function. For example, the following statement creates a 5×1 column vector of ones:

```
ones = j(5,1,1);
```

Figure 5.9 A Vector of Ones

ones	5 rows	1 col	(numeric)

1
1
1
1
1

The I Function

The I function creates an identity matrix of a given size. It has the following 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);
```

Figure 5.10 An Identity Matrix

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, which is useful in calculating ANOVA tables. It has the following 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});
```

Figure 5.11 A Design Matrix

d 7 rows 2 cols (numeric)

1	0
1	0
1	0
0	1
0	1
-1	-1
-1	-1

The REPEAT Function

The REPEAT function creates a new matrix by repeating elements of the argument matrix. It has the following syntax:

REPEAT *(matrix, nrow, ncol)* **;**

The function repeats *matrix* a total of *nrow*$\times$ *ncol* times. The argument is repeated *nrow* times in the vertical direction and *ncol* times in the horizontal direction. For example, the following statement creates a 4×6 matrix:

```
x = {1 2, 3 4};
r = repeat(x, 2, 3);
```

Figure 5.12 A Matrix of Repeated Values

r	4 rows	6 cols	(numeric)

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

The SHAPE Function

The SHAPE function creates a new matrix by reshaping an argument matrix. It has the following general form:

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

The *ncol* and *pad-value* arguments are optional; if they are not specified, default values are used. The following statement uses the SHAPE function to create a 3 × 3 matrix that contains the values 99 and 33. The function cycles back and repeats values to fill in the matrix when no *pad-value* is given.

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

Figure 5.13 A Matrix of Repeated Values

aa	3 rows	3 cols	(numeric)

99	33	33
99	99	33
33	99	99

Alternatively, you can specify a value for *pad-value* that is used for filling in the matrix:

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

Figure 5.14 A Matrix Padded with Zeroes

bb	3 rows	3 cols	(numeric)

99	33	33
99	0	0
0	0	0

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

Index Vectors

You can create a row vector by using the index operator (:). The following statements show that you can use the index operator to count up, count down, or to create a vector of character values with numerical suffixes:

```
r = 1:5;
s = 10:6;
t = 'abc1':'abc5';
```

Figure 5.15 Row Vectors Created with the Index Operator

r	1 row	5 cols	(numeric)

1	2	3	4	5

s	1 row	5 cols	(numeric)

10	9	8	7	6

t	1 row	5 cols	(character, size 4)

abc1	abc2	abc3	abc4	abc5

To create a vector based on an increment other than 1, use the DO function. For example, if you want a vector that ranges from –1 to 1 by 0.5, use the following statement:

```
u = do(-1,1,.5);
```

Figure 5.16 Row Vector Created with the DO Function

u	1 row	5 cols	(numeric)

-1	-0.5	0	0.5	1

Using Matrix Expressions

A matrix expression is a sequence of names, literals, operators, and functions that perform some calculation, evaluate some condition, or manipulate values. Matrix 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, **-A** uses the sign reversal prefix operator (−) in front of the matrix **A** to reverse the sign of each element of **A**.

Binary operators — are placed between operands. For example, **A + B** uses the addition binary operator (+) between matrices **A** and **B** to add corresponding elements of the matrices.

Postfix operators — are placed after an operand. For example, **A`** uses the transpose postfix operator (`) after the matrix **A** to transpose the matrix.

Matrix operators are described in detail in Chapter 25, "Language Reference."

Table 5.1 shows the precedence of matrix operators in the SAS/IML language.

Table 5.1 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 that involve several matrix operators and operands. For example, the following statements are valid matrix assignment statements:

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

The rules for evaluating compound expressions are as follows:

- Evaluation follows the order of operator precedence, as described in Table 5.1. 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. Consider the following statement:

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

This statement 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 I operators. Consider the following statement:

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

This statement 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 I, described in Table 5.1, are evaluated from right to left. For example, the following expression is evaluated as $-(\mathbf{X}^2)$:

```
-x**2
```

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

For example, the following expression is evaluated as $(\mathbf{A}')_{ij}$:

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

- All expressions enclosed in parentheses are evaluated first, using the two preceding rules. Consider the following statement:

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

This statement is evaluated by first dividing elements of **y** by the elements of **z**, then dividing this result into **x**.

Elementwise Binary Operators

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

Table 5.2 lists the elementwise binary operators.

Table 5.2 Elementwise Binary Operators

Operator	Description
+	Addition; string 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
^=	Not equal to
=	Equal to

For example, consider the following two matrices:

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

The addition operator (+) adds corresponding matrix elements, as follows:

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

The elementwise multiplication operator (#) multiplies corresponding elements, as follows:

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

The elementwise power operator (##) raises elements to powers, as follows:

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

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

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

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

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

All operators can work on scalars, vectors, or matrices, provided that the operation makes sense. For example, you can add a scalar to a matrix or divide a matrix by a scalar. For example, the following statement replaces each negative element of the matrix **x** with 0:

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

The expression **x>0** is an operation that compares each element of **x** to (scalar) zero and creates a temporary matrix of results; an element of the temporary matrix is 1 when the corresponding element of **x** is positive, and 0 otherwise. The original matrix **x** is then multiplied elementwise by the temporary matrix, resulting in the matrix **y**. To fully understand the intermediate calculations, you can use the RESET statement with the PRINTALL option 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 a scalar or vector expression that selects one or more rows from the operand.
column	refers to a scalar or vector expression that selects one or more columns from the operand.

You can use subscripts to do any of the following:

- 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. A reduction is a statistical operation (often a sum or mean) applied to the rows or to the columns of a matrix.

In expressions, subscripts have the same (high) precedence as the transpose postfix operator (`). When both *row* and *column* subscripts are used, they are separated by a comma. If a matrix has row or column names associated with it from a MATTRIB or READ statement, then the corresponding row or column subscript can also be a character matrix whose elements match the names 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 index the elements in row-major order.

For example, for the coffee example used previously in this chapter, there are several ways to find the element that corresponds to the number of cups that Samuel drank on Monday.

First, you can refer to the element by row and column location. In this case, you want the fourth row and first column. The following statements extract the datum and place it in the matrix **c41**:

```
coffee={4 2 2 3 2, 3 3 1 2 1, 2 1 0 2 1, 5 4 4 3 4};
names={Jenny, Linda, Jim, Samuel};
print coffee[rowname=names];
c41 = coffee[4,1];
print c41;
```

Figure 5.17 Datum Extracted from a Matrix

	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

c41
5

You can also use row and column names, which can be assigned with an MATTRIB statement as follows:

```
mattrib coffee rowname=names
   colname={'MON' 'TUE' 'WED' 'THU' 'FRI'};
cSamMon = coffee['SAMUEL','MON'];
print cSamMon;
```

Figure 5.18 Datum Extracted from a Matrix with Assigned Attributes

cSamMon
5

You can also look for the element by enumerating the elements of the matrix in row-major order. In this case, you refer to this element as the sixteenth element of **coffee**:

```
c16 = coffee[16];
print c16;
```

Figure 5.19 Datum Extracted from a Matrix by Specifying the Element Number

c16
5

Selecting a Row or Column

To refer to an entire row of a matrix, specify the subscript for the row but omit the subscript for the column. For example, to refer to the row of the **coffee** matrix that corresponds to Jim, you can specify the submatrix that consists of the third row and all columns. The following statements extract and print this submatrix:

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

Alternately, you can use the row names assigned by the MATTRIB statement. Both results are shown in Figure 5.20.

```
jim2 = coffee['JIM',];
print jim2;
```

Figure 5.20 Row Extracted from a Matrix

jim
2 1 0 2 1

jim2
2 1 0 2 1

If you want to extract the data for Friday, you can specify the subscript for the fifth column. You omit the row subscript to indicate that the operation applies to all rows. The following statements extract and print this submatrix:

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

Figure 5.21 Column Extracted from a Matrix

friday
2
1
1
4

Alternatively, you could also index by the column name as follows:

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

Submatrices

You refer to a submatrix by specifying the rows and columns that determine the submatrix. For example, to create the submatrix of **coffee** that consists of the first and third rows and the second, third, and fifth columns, use the following statements:

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

Figure 5.22 Submatrix Extracted from a Matrix

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 of indices and use them to extract the submatrix, as shown in following statements:

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

You can also use the row and column names:

```
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, the following statements extract the first three rows and last three columns of **coffee**:

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

Figure 5.23 Submatrix of Contiguous Rows and Columns

submat2		
2	3	2
1	2	1
0	2	1

Selecting Multiple Elements

All SAS/IML matrices are stored in row-major order. This means that you can index multiple elements of a matrix by listing the position of the elements in an $n \times p$ matrix. The elements in the first row have positions 1 through p, the elements in the second row have positions $p + 1$ through $2p$, and the elements in the last row have positions $(n - 1)p + 1$ through np.

For example, in the coffee data discussed previously, you might be interested in finding occurrences for which some person (on some day) drank more than two cups of coffee. The LOC function is useful for creating an index vector for a matrix that satisfies some condition. The following statement uses the LOC function to find the data that satisfy the desired criterion:

```
h = loc(coffee > 2);
print h;
```

Figure 5.24 Indices That Correspond to a Criterion

	h								
	COL1	COL2	COL3	COL4	COL5	COL6	COL7	COL8	COL9
ROW1	1	4	6	7	16	17	18	19	20

The row vector **h** contains indices of the **coffee** matrix that satisfy the criterion. If you want to find the number of cups of coffee consumed on these occasions, you need to subscript the **coffee** matrix with the indices, as shown in the following statements:

```
cups = coffee[h];
print cups;
```

Figure 5.25 Values That Correspond to a Criterion

cups
4
3
3
3
5
4
4
3
4

Notice that SAS/IML software returns a column vector when a matrix is subscripted by a single array of indices. This might surprise you, but clearly the **cups** matrix cannot be the same shape as the **coffee** matrix since it contains a different number of elements. Therefore, the only reasonable alternative is to return either a row vector or a column vector. Either would be a valid choice; SAS/IML software returns a column vector.

Even if the original matrix is a row vector, the subscripted matrix will be a column vector, as the following example shows:

```
v = {-1 2 5 -2 7};  /* v is a row vector */
v2 = v[{1 3 5}];    /* v2 is a column vector */
print v2;
```

Figure 5.26 Column Vector of Extracted Values

v2
-1
5
7

If you want to index into a row vector and you want the resulting variable also to be a row vector, then use the following technique:

```
v3 = v[ ,{1 3 5}];  /* Select columns. Note the comma. */
print v3;
```

Figure 5.27 Row Vector of Extracted Values

v3
-1 5 7

Subscripted Assignment

You can assign values into a matrix by using subscripts to refer to the element or submatrix. In this type of assignment, the subscripts appear on the left side of the equal sign. For example, to assign the value 4 in the first row, second column of **coffee**, use subscripts to refer to the appropriate element in an assignment statement, as shown in the following statements and in Figure 5.27:

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

To change the values in the last column of **coffee** to zeros, use the following statements:

```
coffee[,5] = {0,0,0,0};   /* alternatively: coffee[,5] = 0; */
print coffee;
```

Figure 5.28 Matrices after Assigning Values to Elements

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

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

In the next example, you locate the negative elements of a matrix and set these elements to zero. (This can be useful in situations where negative elements might indicate errors.) The LOC function is useful for creating an index vector for a matrix that satisfies some criterion. The following statements use the LOC function to find and replace the negative elements of the matrix **T**:

```
t = {3  2 -1,
     6 -4  3,
     2  2  2 };
i = loc(t<0);
print i;
t[i] = 0;
print t;
```

Figure 5.29 Results of Finding and Replacing Negative Values

i	
3	5

t		
3	2	0
6	0	3
2	2	2

Subscripts can also contain expressions. For example, the previous example could have been written as follows:

```
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

A reduction operator is a statistical operation (for example, a sum or a mean) that returns a matrix of a smaller dimension. Reduction operators are often encountered in frequency tables: the marginal frequencies represent the sum of the frequencies across rows or down columns.

In SAS/IML software, you can use reduction operators in place of values for subscripts to get reductions across all rows or columns. Table 5.3 lists operators for subscript reduction.

Table 5.3 Subscript Reduction Operators

Operator	Description
+	Addition
#	Multiplication
<>	Maximum
><	Minimum
<:>	Index of maximum
>:<	Index of minimum
:	Mean
##	Sum of squares

For example, to get row sums of a matrix **X**, you can sum across the columns with the syntax **X[,+]**. Omitting the first subscript specifies that the operator apply to all rows. The second subscript (+) specifies that summation reduction take place across the columns. The elements in each row are added, and the new matrix consists of one column that contains the row sums.

To give a specific example, consider the coffee data from earlier in the chapter. The following statements use the summation reduction operator to compute the sums for each row:

```
coffee={4 2 2 3 2, 3 3 1 2 1, 2 1 0 2 1, 5 4 4 3 4};
names={Jenny, Linda, Jim, Samuel};
mattrib coffee rowname=names colname={'MON' 'TUE' 'WED' 'THU' 'FRI'};
Total = coffee[,+];
print coffee Total;
```

Figure 5.30 Summation across Columns to Find the Row Sums

coffee	MON	TUE	WED	THU	FRI	Total
JENNY	4	2	2	3	2	13
LINDA	3	3	1	2	1	10
JIM	2	1	0	2	1	6
SAMUEL	5	4	4	3	4	20

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

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

You can repeat reduction operators. To get the sum of the row maxima, use the expression $\mathbf{A}[, <>][+,]$, or, equivalently, $\mathbf{A}[, <>][+]$.

A subscript such as $\mathbf{A}[\{2\ 3\}, +]$ first selects the second and third rows of $\mathbf{A}$ and then finds the row sums of that submatrix.

The following examples demonstrate how to use the operators for subscript reduction. Consider the following matrix:

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

The following statements are true:

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

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

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

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

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

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

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

Displaying Matrices with Row and Column Headings

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

The AUTONAME Option in the RESET Statement

You can use the RESET statement with the AUTONAME option to automatically display row and column headings. If your matrix has *n* rows and *p* columns, the row headings are ROW1 to ROW*n* and the column headings are COL1 to COL*p*. (You can use the RESET WIDE option to suppress row and column headings.)

For example, the following statements produce the subsequent matrix:

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

Figure 5.31 Result of the AUTONAME Option

	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

The ROWNAME= and COLNAME= Options in the PRINT Statement

You can specify your own row and column headings. The easiest way is to create vectors that contain the headings and then display the matrix by using the ROWNAME= and COLNAME= options in the PRINT statement. For example, the following statements display row names and column names for a matrix:

```
names={Jenny, Linda, Jim, Samuel};
days={Mon Tue Wed Thu Fri};
print coffee[rowname=names colname=days];
```

Figure 5.32 Result of the ROWNAME= and COLNAME= Options

	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

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 customize your displayed output:

```
mattrib coffee rowname=names
        colname=days
        label='Weekly Coffee'
        format=2.0;
print coffee;
```

Figure 5.33 Result of the MATTRIB Statement

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 about Missing Values

Missing values in matrices are discussed in Chapter 3, "Understanding the SAS/IML Language." You should carefully read that chapter and Chapter 24, "Further Notes," so that you are aware of the way SAS/IML software handles missing values. The following examples show how missing values are handled for elementwise operations and for subscript reduction operators.

Consider the following two matrices $\mathbf{X}$ and $\mathbf{Y}$:

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

The following operations handle missing values in matrices:

Matrix addition: $\mathbf{X} + \mathbf{Y}$ is $\begin{bmatrix} 5 & . & . \\ . & 6 & 9 \\ 13 & . & 14 \end{bmatrix}$

Elementwise multiplication: $\mathbf{X}\#\mathbf{Y}$ is $\begin{bmatrix} 4 & . & . \\ . & 5 & 18 \\ 42 & . & 45 \end{bmatrix}$

Subscript reduction: $\mathbf{X}[+,]$ is $\begin{bmatrix} 8 & 7 & 15 \end{bmatrix}$

Chapter 6
Programming Statements

Contents

Overview of Programming Statements **59**
Selection Statements **60**
Compound Statements **61**
Iteration Statements **62**
Jump Statements **64**
Statements That Define and Execute Modules **65**
Defining and Executing a Module 66
Understanding Symbol Tables 66
Modules with No Arguments 67
Modules with Arguments 67
Passing Arguments by Using Keyword-Value Pairs 69
Defining Function Modules 70
Using the GLOBAL Clause 71
Modules with Optional and Default Arguments 72
Nesting Module Definitions 75
Calling a Module from Another Module 76
Understanding Argument Passing 77
Storing and Loading Modules 79
Termination Statements **79**
PAUSE Statement 79
STOP Statement 81
ABORT Statement 81
QUIT Statement 81

Overview of Programming Statements

The SAS/IML programming language has control statements that enable you to control the path of execution in a program. The control statements in the SAS/IML language are similar to the corresponding statements in the SAS DATA step. This chapter describes the following concepts:

- selection statements
- compound statements

- iteration statements
- jump statements
- statements that define and execute modules
- termination statements

Selection Statements

Selection statements choose one of several control paths in a program. The SAS/IML language supports the IF-THEN and the IF-THEN/ELSE statements. You can use an IF-THEN statement to test an expression and to conditionally perform an operation. You can also optionally specify an ELSE statement. The general form of the IF-THEN/ELSE statement is as follows:

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

ELSE *statement2* **;**

The expression is evaluated first. If the value of expression is true (which means nonzero and nonmissing), the THEN statement is executed. If the value of expression is false (which means zero or missing), the ELSE statement (if present) is executed. If an ELSE statement is not present, control passes to the next statement in the program.

The expression in an IF-THEN statement is often a comparison, as in the following example:

```
a = {17 22, 13 10};
if max(a)<20 then
   p = 1;
else p = 0;
```

The IF clause evaluates the expression `max(a)<20`. If all values of the matrix `a` are less than 20, `p` is set to 1. Otherwise, `p` is set to 0. For the given values of `a`, the IF condition is false, since `a[1,2]` is not less than 20.

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

```
w = 0;
if n>0 then
   if x>y then w = x;
   else w = y;
```

There is an ambiguity associated with the previous statements. Is the ELSE statement associated with the first IF-THEN statement or the second? As the indenting indicates, an ELSE statement is associated with the closest previous IF-THEN statement. (If you want the ELSE statement to be associated with the first IF-THEN statement, you need to use a DO group, as described in the next section.)

When the condition to be evaluated is a matrix expression, the result of the evaluation is a temporary matrix of zeros, ones, 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 zero or missing, the condition is false. This evaluation is equivalent to using the ALL function. For example, the following two statements produce the same result:

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

If you are testing whether at least one element in **x** is less than the corresponding element in **y**, use the ANY function, as shown in the following statement:

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

Compound Statements

Several statements can be grouped together into a compound statement (also called a *block* or a DO group). You use a DO statement to define the beginning of a DO group and an END statement to define the end. DO groups have two principal uses:

- 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 ;
   statements ;
END ;
```

As with IF-THEN/ELSE statements, you can nest DO groups to any number of levels. The following is an example of nested DO groups:

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

It is a good programming convention to indent the statements in DO groups as shown, so that each statement's position indicates its level of nesting.

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

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

```
      z1 = abs(x-y);
      z2 = abs(x+y);
   end;
```

An alternative formulation that requires less indented space is to write the DO statement on the same line as the THEN or ELSE clause, as shown in following statements:

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

For some programming applications, you must use either a DO group or a module. For example, LINK and GOTO statements must be programmed inside a DO group or a module.

Iteration Statements

The DO statement supports clauses that iterate over compound statements. With an iterative DO statement, you can repeatedly execute a set of statements until some condition stops the execution. The following table lists the different kinds of iteration statements in the SAS/IML language:

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 the clauses must be specified in the order listed in the preceding table.

DO DATA Statements

The general form of the DO DATA statement is as follows:

DO DATA ;

The DATA keyword specifies that iteration stops when an end-of-file condition occurs. Other DO specifications exit after tests are performed at the top or bottom of a loop.

See Chapter 7, "Working with SAS Data Sets," and Chapter 8, "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 a file is reached.

The following example reads data from an external file named *MyData.txt* that contains the following data:

Rick	2.40	3.30
Robert	3.90	4.00
Simon	3.85	4.25

The data values are read one at a time into the scalar variables Name, x, and y.

```
filename MyFile 'MyData.txt';
infile MyFile;                  /* infile statement       */
do data;                        /* begin read loop        */
   input Name $6. x y;          /* read a data value      */
   /* do something with each value */
end;
closefile MyFile;
```

Iterative DO Statements

The general form of the iterative DO statement is as follows:

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

The value of the *variable* matrix is initialized to the value of the *start* matrix. 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 initializes `i` to the value 1 and increments `i` by 2 after each loop. The loop ends when the value of `i` is greater than 10.

```
y = 0;
do i = 1 to 10 by 2;
   y = y + i;
end;
```

DO WHILE Statements

The general form of the DO WHILE statement is as follows:

DO WHILE *expression* **;**

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

For example, the following statements initialize `count` to 1 and then increment `count` four times:

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

DO UNTIL Statements

The general form of the DO UNTIL statement is as follows:

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, the following statements initialize **count** to 1 and then increment **count** five times:

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

Jump Statements

During normal execution, each statement in a program is executed in sequence, one after another. The GOTO and LINK statements cause a SAS/IML program to jump from one statement in a program to another statement without executing intervening statements. 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 LINK statements instruct SAS/IML software to jump immediately to a labeled statement. However, if you use a LINK statement, then the program returns to the statement following the LINK statement when the program executes a RETURN statement. 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 statements that define the subroutine begin with the label and end with a RETURN statement. LINK statements can be nested within other LINK statements; any number of nesting levels is allowed.

NOTE: The GOTO and LINK statements must be 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 DO groups and modules. The following statements shows an example of using the GOTO statement, followed by an equivalent set of statements that do not use the GOTO statement:

```
x = -2;
do;
   if x<0 then goto negative;
   y = sqrt(x);
   print y;
   goto TheEnd;
negative:
   print "Sorry, value is negative";
TheEnd:
end;

/* same logic, but without using a GOTO statement */
if x<0 then print "Sorry, value is negative";
```

```
else do;
   y = sqrt(x);
   print y;
end;
```

The output of each section of the program is identical. It is shown in Figure 6.1.

Figure 6.1 Output That Demonstrates the GOTO Statement

Sorry, value is negative

Sorry, value is negative

The following statements show an example of using the LINK statements. You can also rewrite the statements in a way that avoids using the LINK statement.

```
x = -2;
do;
   if x<0 then link negative;
   y = sqrt(x);
   print y;
   goto TheEnd;
negative:
   print "Using absolute value of x";
   x = abs(x);
   return;
TheEnd:
end;
```

The output of the program is shown in Figure 6.2.

Figure 6.2 Output That Demonstrates the LINK Statement

Using absolute value of x

y
1.4142136

Statements That Define and Execute Modules

Modules are used for two purposes:

- to create a user-defined subroutine or function. That is, you can define a group of statements that can be called from anywhere in the program.
- to define variables that are local to the module. That is, you can create a separate environment with its own symbol table (see "Understanding Symbol Tables" on page 66).

A module always begins with a START statement and ends with a FINISH statement. A module is either a function or a subroutine. When a module returns a single parameter, it is called a function. A function is

invoked by its name in an assignment statement. Otherwise, a module is a subroutine. You can execute a subroutine by using either the RUN statement or the CALL statement.

Defining and Executing a Module

A module definition begins with a START statement, which has the following general form:

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

A module definition ends with a FINISH statement, which has the following general form:

FINISH < *name* > **;**

If no name appears in the START statement, the name of the module defaults to MAIN. If no name appears on the FINISH statement, the name of the most recently defined module is used.

There are two ways you can execute a module: you can use either a RUN statement or a CALL statement. The general forms of these statements are as follows:

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

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

The main difference between the RUN and CALL statements is the order of resolution. If you define a module that has the same name as a SAS/IML subroutine, you can use the RUN statement to call the user-defined module and the CALL statement to call the built-in subroutine. For more on the order of resolution, see "Order of Resolution for Functions and Subroutines" on page 1132.

The RUN and CALL statements must have arguments that correspond to the ones defined in the START statement. A module can call other modules but cannot recursively call itself.

After the last statement in a module is executed, control returns to the statement that initially called the module. You can also force a return from a module by using the RETURN statement.

Understanding Symbol Tables

The *scope* of a variable is the set of locations in a program where a variable can be referenced. A variable defined outside of any module is said to exist at the program's *main scope*. For a variable defined inside a module, the scope of the variable is the body of the module.

A *symbol* is the name of a SAS/IML matrix. For example, if **x** and **y** are matrices, then the names 'x' and 'y' are the symbols. Whenever a matrix is defined, its symbol is stored in a *symbol table*. There are two kinds of symbol tables. When a matrix is defined at the main scope, its name is stored in the *global symbol table*. In contrast, each module with arguments is given its own *local symbol table* that contains all symbols used inside the module.

There can be many local symbol tables, one for each module with arguments. (Modules without arguments are described in the next section.) You can have a symbol 'x' in the global table and the same symbol in a local table, but these correspond to separate matrices. By default, the value of the matrix at global scope is independent from the value of a local matrix of the same name that is defined inside a module. Similarly, you can have two modules that each use the matrix **x**, and these matrices are not related. You can force a module

to use a variable at main scope by using a GLOBAL clause as described in "Using the GLOBAL Clause" on page 71.

Values of symbols in a local table are temporary; that is, they exist only while the module is executing. You can no longer access the value of a local variable after the module exits.

Modules with No Arguments

The previous section emphasized that modules with arguments are given a local symbol table. In contrast, a module that has no arguments shares the global symbol table. All variables in such a module are global, which implies that if you modify the value of a matrix inside the module, that change persists when the module exits.

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      */
   c = 40;                              /* c already global */
finish;                                 /* end module       */

run Mod1;                               /* run the module   */
print a b c p;
```

Figure 6.3 Output from Module with Global Variables

a	b	c	p
10	20	40	30

Notice that after the module exits, the following conditions exist:

- **a** is still 10.
- **b** is still 20.
- **c** has been changed to 40.
- **p** is created, added to the global symbol table, and set to 30.

Modules with Arguments

Most modules contain one or more arguments, and therefore contain a separate local symbol table. The following statements apply to modules with arguments:

- You can specify arguments as variable names, expressions, or literal values.

- If you specify several arguments, use commas to separate them.
- Arguments are passed by reference, not by value. This means that a module can change the value of an argument. An argument that is modified by a module is called an *output argument*.
- If you have both output arguments and input arguments, the SAS/IML convention is to list the output arguments first.
- When a module is run, the input arguments can be a matrix name, expression, or literal. However, you should specify only matrix names for output arguments.

When a module is run, 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:

```
proc iml;
a = 10;
b = 20;
c = 30;
start Mod2(x,y);                    /* begin module */
   p = x+y;
   x = 100;                         /* change the value of an argument */
   c = 25;
finish Mod2;                        /* end module   */

run Mod2(a,b);
print a b c;
```

The first three statements are submitted in the main scope and define the variables **a**, **b**, and **c**. The values of these variables are stored in the global symbol table. The START statement begins the 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**, and **c**) are in the local symbol table. There is also a 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** and **b** exist only in the global symbol table, whereas **x**, **y**, and **p** exist only in the local symbol table. The symbol **c** exists in both symbol tables, but the values are completely independent.

When Mod2 is executed with the RUN statement, the local variable **x** becomes the "owner" of the data in the global matrix **a**. Similarly, the local variable **y** becomes the owner of the data in **b**. 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 variables **a** and **b** at main scope regain ownership of the data in **x** and **y**, respectively. The local symbol table that contains **x** and **y** is deleted. If the data were modified within the module, the values of **a** and **b** reflect the change, as shown in Figure 6.4.

Figure 6.4 Output from Module with Arguments

a	b	c
100	20	30

Notice that after the module is executed, the following are true:

- **a** is changed to 100 since the corresponding argument, **x**, was changed to 100 inside the module.
- **b** is still 20.

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

Also note that, inside the module, the symbols **a** and **b** do not exist. Outside the module, the symbols **p**, **x**, and **y** do not exist.

Passing Arguments by Using Keyword-Value Pairs

The standard way to pass arguments to a module is to supply a comma-separated list of parameter values that is enclosed in parentheses. In this syntax, the arguments are supplied as *positional parameters*. You can call module subroutines by using an alternative syntax, called the *keyword-value pairs* syntax (also known as the *named arguments* syntax and the *keyword parameters* syntax).

When you define a module, the names of the parameters become keywords for calling that module. You can call the module by specifying some of the positional parameters and then specifying the remaining parameters as keyword-value pairs outside parentheses. For example, the Mod2 module that is defined in the previous section has two parameters, named **x** and **y**. The following calls are equivalent:

```
run Mod2(a,b);          /* two positional parameters              */
run Mod2(a)    y=b;     /* one positional parameter; one keyword  */
run Mod2()     x=a y=b; /* no positional parameters; two keywords */
run Mod2()     y=b x=a; /* order of keywords does not matter      */
```

Although the order of positional parameters is important, you can specify keyword-value pairs in any order, as shown in the last statement.

Keyword-value pairs are especially useful for modules that have a large number of optional arguments (see the section "Modules with Optional and Default Arguments" on page 72). For example, if a module accepts five optional parameters and you want to supply a value for the fifth, it is easier to use keyword-value pairs than to use positional parameters, as shown in the following example:

```
start Sum5(x, a=1, b=2, c=3, d=4, e=5);
   x = a + b + c + d + e;
finish;

run Sum5(x,,,,,10);  /* skip some positional parameters    */
run Sum5(x) e=10;    /* equivalent, but simpler           */
print x;
```

Figure 6.5 Specifying a Keyword-Value Pair

x
20

You can also use keyword-value pairs to specify the optional parameters for many of the SAS/IML built-in subroutines. For an example, see the QUAD subroutine.

The ability to pass arguments to modules by using keyword-value pairs is supported only for subroutines. Function modules, which are described in the next section, do not support keyword-value pairs.

Defining Function Modules

Functions are special modules that return a single value. To write a function module, include a RETURN statement that specifies the value to return. The RETURN statement is necessary for a module to be a function. You can use a function module in an assignment statement, as you would a built-in function.

The symbol-table logic described in the preceding section also applies to function modules. In the following function module, the value of **c** in the local symbol table is assigned to the symbol **z** at main scope:

```
proc iml;
a = 10;
b = 20;
c = 30;
start Mod3(x,y);
   c = 2#x + y;
   return (c);            /* return function value */
finish Mod3;

z = Mod3(a,b);            /* call function         */
print a b c z;
```

Figure 6.6 Output from a Function Module

a	b	c	z
10	20	30	40

Notice the following about this example:

- **a** is still 10 and **b** is still 20.
- **c** is still 30. The symbol **c** in the global table has no connection with the symbol **c** in the local table.
- **z** assigned the value 40, which is the value returned by the module.

Again notice that, inside the module, the symbols **a**, **b**, and **z** do not exist. Outside the module, the symbols **x** and **y** do not exist.

In the next example, you define your own function module, Add, which adds its two arguments:

```
proc iml;
start Add(x,y);
   sum = x+y;
   return(sum);
finish;

a = {9 2, 5 7};
b = {1 6, 8 10};
c = Add(a,b);
print c;
```

Figure 6.7 Output from a Function Module

c	
10	8
13	17

Function modules can also be called as arguments to other modules or to built-in functions. For example, in the following statements, the Add module is called twice, and the results from those calls are used as arguments to call the Add module a third time:

```
d = Add(Add({1 2},{3 4}), Add({5 6}, {7 8}));
print d;
```

Figure 6.8 Output from a Nested Module Call

d	
16	20

Functions are resolved in the following order:

1. SAS/IML built-in functions
2. user-defined function modules
3. SAS DATA step functions

Because of this order of resolution, it is an error to try to define a function module that has the same name as a SAS/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 that exist 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. The GLOBAL clause specifies variables that you want to share between local and global symbol tables. The following is an example of a module that uses 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 symbol table and the value of **c** in the local symbol table.

```
proc iml;
a = 10;
b = 20;
c = 30;
start Mod4(x,y) global (c);
   x = 100;
   c = 40;
   b = 500;
finish Mod4;

run Mod4(a,b);
```

```
print a b c;
```

The output is shown in Figure 6.9.

Figure 6.9 Output from a Module with a GLOBAL Clause

a	b	c
100	20	40

After the module is called, the following facts are true:

- **a** is changed to 100.
- **b** is still 20 and not 500, since **b** exists independently in the global and local symbol tables.
- **c** is changed to 40 because it was declared to be a global variable. The matrix **c** inside the module is the same matrix as the one outside the module.

Because every module with arguments has its own local table, it is possible to have many local tables. You can use the GLOBAL clause with many (or all) modules to share a single global variable among many local symbol tables.

Modules with Optional and Default Arguments

You can define a module that has optional arguments. You can also assign default values to optional arguments. Optional arguments can be skipped when the module is called. Optional arguments are supported for all user-defined modules, both functions and subroutines.

By convention, optional arguments appear at the end of a module argument list.

Optional Arguments without Default Values

To designate an argument as optional, type an equal sign (=) after the argument when defining the module. Arguments that are not followed by an equal sign are required arguments. Optional arguments can be skipped when you call the module. If you skip an optional argument, the local variable is set to the empty matrix, as shown in the following example:

```
proc iml;
start MyAdd(x, y=);
   if ncol(y)=0 then return(x); /* y is empty matrix */
   return(x+y);
finish;

z = MyAdd(5);    /* z = 5 */
w = MyAdd(5, 3);/* w = 8 */
```

When the MyAdd module is called the first time, the second argument is skipped. Inside the MyAdd function, the local variable **y** is set to the empty matrix: it has no rows and no columns. Consequently, the IF-THEN condition is true and the function returns the value of the local variable **x**.

When the MyAdd module is called the second time, the second argument is provided. Inside the MyAdd function, the local variable **y** is not empty. Consequently, the IF-THEN condition is false and the function returns the sum of the two arguments.

Because the optional argument appears last in the module argument list, you can call the argument with a single argument. You can also skip optional arguments by not providing an argument when the module is called. For example, the following statement is valid syntax:

```
z = MyAdd(5, );   /* skip second argument */
```

In previous versions of SAS/IML software, you could skip any parameter in a user-defined subroutine. With the new syntax, you can skip only those arguments that are explicitly designated as optional.

Detecting Skipped Arguments

Arguments can be skipped in the call by using white space and a comma, or by simply not supplying the maximum number of arguments declared in the START statement.

The ISSKIPPED function enables you to determine at run time whether a module is being called with a skipped argument. The ISSKIPPED function returns 1 if the argument to the function is skipped, and 0 otherwise. For example, the following function returns the inner product (dot product) of two column vectors. If the function is called with a single argument, the function returns the inner product of the first argument with itself. If the function is called with two arguments, the function returns their inner product.

```
start MyDot(x, y=);
   if isskipped(y) then return(x`*x);
   return(x`*y);
finish;
z = MyDot({1,2,3});           /* z = 14 */
w = MyDot({1,2,3}, {-1,0,1});/* w = 2 */
```

Optional Arguments with Default Values

You can assign a default value for an optional argument by specifying the value after the equal sign in the module argument list. For example, you might define the following module:

```
start MySum(x=1, y=2);
   return(x+y);
finish;
```

In this module, if the first argument is skipped, the local variable **x** is assigned the value 1. Similarly, if the second argument is skipped, the local variable **y** is assigned the value 2. Consequently, the syntax that assigns the default values is logically equivalent to the following statements:

```
   if IsSkipped(x) then x=1;
   if IsSkipped(y) then y=2;
```

Optional Arguments with Constant Default Values

As indicated in the previous section, you can assign a default value for an optional argument by specifying the value after the equal sign in the module argument list. For example, the following module returns the vector sum $ax + y$ for vectors x and y and constant a. If the a parameter is not specified, a default value of 1 is used. The default value is specified as follows:

```
start axpy(a=1, x, y=); /* compute ax + y */
   if isskipped(y) then return(a#x);
   else                 return(a#x + y);
finish;

p = {1 2 3};
q = {1 1 1};
z1 = axpy( , p);    /* a and y skipped; a has default value */
z2 = axpy(2, p);    /* y skipped */
z3 = axpy(2, p, q); /* no arguments are skipped */
print z1, z2, z3;
```

Figure 6.10 Default Values and Skipped Arguments

z1
1 2 3

z2
2 4 6

z3
3 5 7

The module is called three times. Each call uses a different combination of skipped arguments. The results are shown in Figure 6.10. During the first call, the local variable **a** is set to 1 inside the module, whereas **y** is an empty matrix. During the second call, **a** is set to 2 and **y** is an empty matrix. During the third call, no arguments are skipped.

Optional Arguments with Data-Dependent Default Values

In the previous section, a constant value was used as the default value of an argument. You can also provide an expression for a default value. If the argument is skipped, the expression is evaluated and assigned to the local variable for the skipped argument. The expression can refer to other arguments, so the default values are *data dependent*.

For example, the following module standardizes columns of a matrix:

```
start stdize(x, loc=mean(x), scale=std(x));
   return ( (x-loc)/scale );
finish;

x = {1, 1, 0, -1, -1};
z = stdize(x);              /* use default values */

center = 1;  s  = 2;
z1 = stdize(x,  center);    /* skip 3rd argument */
z2 = stdize(x,        , s); /* skip 2nd argument */
z3 = stdize(x,  center, s); /* no arguments are skipped */
print z z1 z2 z3;
```

Figure 6.11 Data-Dependent Default Values

z	z1	z2	z3
1	0	0.5	0
1	0	0.5	0
0	-1	0	-0.5
-1	-2	-0.5	-1
-1	-2	-0.5	-1

The module is called four times. The results are shown in Figure 6.11. As discussed previously, the syntax that defines the default arguments is logically equivalent to beginning the module with the following two statements:

```
if IsSkipped(loc)   then loc=mean(x);
if IsSkipped(scale) then scale=std(x);
```

During the first call, the second and third arguments are skipped. The local variable **loc** is set to the mean values of the columns of the required argument, **x**, and the local variable **scale** is set to the standard deviation of the columns of **x**. The MEAN and STD functions are evaluated only when the second and third arguments, respectively, are skipped.

During the second call, the third argument is skipped. The local variable **loc** is set to the value 1, and the local variable **scale** is set to the standard deviation of the columns of **x**.

During the third call, the local variable **loc** is set to the mean value of the columns of **x**, and the local variable **scale** is set to 2. During the fourth call, no arguments are skipped.

The argument list is parsed from left to right. Consequently, a good programming practice is to use data-dependent expressions that depend only on arguments that appear earlier in the argument list. The syntax does not forbid referring to variables that appear later in the argument list, but it is often an error to evaluate an expression that involves unassigned (empty) matrices.

Data-dependent expressions can also use global variables that are specified in the GLOBAL statement. For example, the following statements use global variables to form a data-dependent default value:

```
start MyFunc(x, a=max(1,gMax)) global(gmax);
   return(a#x);
finish;

gMax = 2;
y = MyFunc(5);
```

Default values for skipped arguments apply only to local variables in modules. The GLOBAL statement does not support default arguments.

Nesting Module Definitions

You can define one module while in the midst of defining another. Each module definition must be completely contained inside the parent module definition, as shown in the following example:

```
start ModA;
   start ModB;
      x = 1;
   finish ModB;
   run ModB;
finish ModA;

run ModA;
```

In this example, SAS/IML software starts parsing statements for a module called ModA. In the middle of this module definition, it recognizes the start of a new module called ModB. It parses ModB until it encounters the first FINISH statement. It then finishes parsing ModA.

Although it looks like ModB might be "local" to ModA, that is not the case. The previous statements are equivalent to the following:

```
start ModB;
   x = 1;
finish ModB;

start ModA;
   run ModB;
finish ModA;

run ModA;
```

In particular, you can call the ModB module from the program's main scope or from other modules. The SAS/IML language does not support local modules. All modules are defined at global scope.

Calling a Module from Another Module

Consider the following example of calling one module from another module:

```
proc iml;
start Mod5(a,b);
   c = a+b;
   d = a-b;
   run Mod6(c,d);
   print "In Mod5:" c d;
finish;

start Mod6(x,y);
   x = x#y;
finish;

run Mod5({1 2}, {3 4});
```

When one module calls another, you can pass in any symbol defined in the scope of the calling module. In the previous example, the Mod5 module calls the Mod6 module and passes in the local variables `c` and `d`. The Mod6 module multiplies its arguments and overwrites the first argument, as shown in Figure 6.12.

Figure 6.12 Output from Nested Modules

	c		d	
In Mod5:	-8	-12	-2	-2

The variables in the local symbol table of Mod5 are available to pass into Mod6. If Mod6 changes the values of an argument, those values are also changed in the environment from which Mod6 was called. For the previous example, this means that the local variable **c** is modified by Mod6.

If a module has no arguments, it can access variables in the environment from which it is called. For example, consider the following modules:

```
x = 123;

start Mod7;
   print "In Mod7:" x;
finish;

start Mod8(p);
   print "In Mod8:" p;
   run Mod7;
finish;

run Mod8(x);
```

In this example, module Mod7 is called from module Mod8. Therefore, the variables available to Mod7 are those defined in the scope of Mod8. There is no variable named **x** in the environment of Mod8. Therefore an error occurs on the PRINT statement in Mod7, as shown in Figure 6.13. An error would not occur if you call Mod7 from the main scope, because **x** is defined at main scope.

Figure 6.13 Error Message When a Variable Is Not Defined in a Module

```
NOTE: IML Ready
NOTE: Module MOD7 defined.
NOTE: Module MOD8 defined.
ERROR: Matrix x has not been set to a value.

 statement : PRINT at line 1877 column 4
 traceback : module MOD7 at line 1877 column 4
             module MOD8 at line 1882 column 4

NOTE: Paused in module MOD7.
```

Understanding Argument Passing

You can use expressions and subscripted matrices as arguments to a module, but it is important to understand the way the SAS/IML software passes the 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. In the following example, notice that the matrix **x** does not change; you might expect **x** to contain the squared values of **y**.

```
start Square(a,b);
   a = b##2;
finish;

x = {. .};                  /* initialize with missing values     */
y = {3 4};
reset printall;             /* print all intermediate results     */
do i = 1 to 2;              /* pass elements of matrix to modules */
   run Square(x[i],y[i]);   /* WRONG: x[i] is not changed         */
end;
print x;                    /*  show that x is unchanged          */
```

The output is shown in Figure 6.14. The names of the temporary matrices created by the subscript operators are **_TEM1001** and **_TEM1002**. These are the matrices passed into the square module. The module assigns the value 9 to the local matrix **a**, and this value is returned to main scope in the temporary matrix **_TEM1001**, which promptly vanishes! The same sequence of operations repeats for the next call to the Square module.

Figure 6.14 Temporary and Local Matrices in a Module

```
i    1 row    1 col    (numeric)

  1

_TEM1002    1 row    1 col    (numeric)

  3

_TEM1001    1 row    1 col    (numeric)

  .

a    1 row    1 col    (numeric)

  9

_TEM1002    1 row    1 col    (numeric)

  4

_TEM1001    1 row    1 col    (numeric)

  .

a    1 row    1 col    (numeric)

 16
```

Consequently, the values of **x** remain unchanged by the previous calls, as shown in Figure 6.15. The lesson to learn from this example is this: do not pass in an expression or literal as an output argument to a module. Use only matrix names for output arguments. For example, the correct way to call the Square module is to eliminate the loop and simply use the statement **run Square(x, y);**.

Figure 6.15 An Unchanged Matrix



Storing and Loading Modules

You can store and reload modules by using the STORE statement. The STORE statement saves the module in a storage library. The stored module persists even when you exit PROC IML or exit the SAS System. After a module is stored, you can use the module in other SAS/IML programs by using the LOAD statement prior to calling the module. The syntax of the STORE and LOAD statements are as follows:

STORE MODULE= *name* **;**

LOAD MODULE= *name* **;**

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

```
show storage;
```

See Chapter 19, "Storage Features," for details about using the library storage facilities.

Termination Statements

You can stop execution with a PAUSE, STOP, or ABORT statement. The QUIT statement is also a termination statement, but it causes the IML procedure to immediately exit. The other termination statements do not cause PROC IML to exit until the statements are executed. The following sections describe the PAUSE, STOP, ABORT, and QUIT statements.

PAUSE Statement

The general form of the PAUSE statement is as follows:

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

The PAUSE statement does the following:

- stops execution of a module
- remembers where it stopped
- prints a message that you can specify
- sets the current program environment and symbol table to be that of the module that contains the PAUSE statement. This means that you can type statements that reference local variables in the module. For example, you might want to use a PAUSE statement while debugging a module so that you can print the value of local variables.

A RESUME statement enables you to continue execution at the location of the most recent PAUSE statement.

You can use a STOP statement as an alternative to the RESUME statement to remove the paused state and to return to the main scope outside the module. You can specify a message in the PAUSE statement. This message is displayed in the output window when the PAUSE statement is executed. For example, the following PAUSE statements each display a message:

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

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

The PAUSE statement also writes a note to the SAS log. To suppress the note, use the * option, as shown in the following statement:

```
pause *;
```

When you use a PAUSE, RESUME, STOP, or ABORT statement, keep in mind the following details:

- The PAUSE statement must be used from inside a module.
- It is an error to execute a RESUME statement without any outstanding pauses.
- You can define and execute modules while paused within another module.
- If a run-time error occurs inside a module, a PAUSE statement is automatically executed. 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 statement to exit from the module environment, or an ABORT statement to exit PROC IML.
- You cannot reenter or redefine an active (paused) module.
- When paused, you can run another module that also 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.
- You can use the PAUSE and RESUME statements in both subroutine and function modules.
- If you pause in a subroutine module that has its own symbol table, then the statements executed while paused use this symbol table. 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 20, "Using SAS/IML Software to Generate SAS/IML Statements," to execute SAS/IML statements that you generate within a module.

STOP Statement

The general form of the STOP statement is as follows:

STOP < *error-message* > **;**

The STOP statement clears all pauses and returns to the main scope.

ABORT Statement

The general form of the ABORT statement is as follows:

ABORT < *error-message* > **;**

The ABORT statement stops execution and exits from PROC IML, much like a QUIT statement. The difference is that the ABORT statement is an executable statement that can be used in IF-THEN statements and in modules. For example, you might want to exit PROC IML if a certain error occurs. You can check for the error in a module and execute the ABORT statement if the error occurs.

QUIT Statement

The syntax of the QUIT statement is as follows:

QUIT ;

The QUIT statement stops execution and exits from PROC IML. The QUIT statement is executed as soon as the statement is parsed. Consequently, you cannot use QUIT in a module or as part of an IF-THEN/ELSE statement.

Chapter 7
Working with SAS Data Sets

Contents

Overview **84**
Open a SAS Data Set **85**
Syntax for Specifying a SAS Data Set **86**
Make a SAS Data Set Current **87**
Display SAS Data Set Information **88**
List Observations **89**
Specify a Range of Observations 90
Select a Set of Variables 91
Select a Set of Observations 91
Read Observations from a SAS Data Set **92**
Use the READ Statement with the VAR Clause 93
Use the READ Statement with the INTO Clause 93
Use the READ Statement with the WHERE Clause 94
Edit a SAS Data Set **95**
Update Observations 95
Delete Observations 96
Create a SAS Data Set from a Matrix **97**
Use the CREATE Statement with the FROM Option 97
Use the CREATE Statement with the VAR Clause 98
Understand the End-of-File Condition **99**
Produce Summary Statistics **100**
Sort a SAS Data Set **101**
Index a SAS Data Set **101**
Data Set Maintenance Functions **103**
Summary of Commands **103**
Shared Concepts for Processing Data **104**
Process a Range of Observations 104
Select Variables with the VAR Clause 105
Process Data by Using the WHERE Clause 106
Using Data Set Options 109
Comparison with the SAS DATA Step **109**

Overview

SAS/IML software has statements for creating a matrix from a SAS data set and for creating a SAS data set from a matrix.

You can create a matrix from a SAS data set in several ways. For example:

- You can create a column vector for each data set variable.
- You can create a matrix whose columns correspond to data set variables.
- You can use all the observations in a data set or use a subset of them.

You can read observations from a SAS data set into a matrix. You can read them sequentially (by record number) or conditionally (by using a WHERE clause).

You can also create a SAS data set from a matrix. Each column of the matrix becomes a variable in the data set, and each row becomes an observation.

There are SAS/IML statements that enable you to edit, append, index, rename, and delete SAS data sets from within the SAS/IML environment.

You can dynamically specify which observations and variables are read. For example, the READ statement can do the following:

- read all records
- read the next record
- read any number of specified records
- read records that satisfy one or more conditions
- read specified variables, all numeric variables, or all character variables

This chapter demonstrates how to use the SAS/IML language to do the following:

- open a SAS data set
- examine the contents of a SAS data set
- display data values by using the LIST statement
- read observations from a SAS data set into matrices or vectors
- edit a SAS data set
- create a SAS data set from a matrix or from vectors
- produce summary statistics for variables in a data set

- sort a SAS data set
- index a SAS data set

The chapter also discusses similarities and differences between the data set and the SAS DATA step. All examples use the Sashelp.Class data set, which is distributed as part of SAS software. The Sashelp.Class data set contains data about 19 students. The variables are Name, Sex, Age, Height, and Weight.

Open a SAS Data Set

You must open a SAS data set before you can access the data. There are three ways to open a SAS data set:

- To read from an existing data set, submit the USE statement, which opens a data set for input. The general form of the USE statement is as follows:

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

  You can use the FIND, INDEX, LIST, and READ statements after the data set is open.

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

  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, which opens a new data set for both output and input. The general form of the CREATE statement is as follows:

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

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

  Use the APPEND statement to place data into the newly created data set. If you do not use the APPEND statement, the new data set will not contain any observations.

See the section "Process Data by Using the WHERE Clause" on page 106 for details about using the WHERE clause; see the section "Select Variables with the VAR Clause" on page 105 for details about using the VAR clause.

Specify a data set name as the first operand to the USE, EDIT, and CREATE statements. 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, and the second level refers to the data set name. If the libref is Work, the data set is stored in a temporary directory. All data in the Work library are deleted at the end of the SAS session.

You can use the LIBNAME statement to assign a libref that refers to a permanent directory, as described in *SAS Language Reference: Concepts*. If you specify only a single name, then a default libref is used. The default libref is Sasuser if Sasuser is defined or Work otherwise. You can reset the default libref by using the RESET DEFLIB statement, as shown in the following statements:

```
libname mydir "C:\Users\userid\Documents\My SAS Files";
reset deflib=mydir;
```

If you run these statements, one-level names are read from and written to the mydir library.

Syntax for Specifying a SAS Data Set

You can specify a SAS data set by using a literal value, such as "Sashelp.Class," or by specifying an expression that resolves to the name of a SAS data set. Most of the examples in this chapter use a literal value, such as the following statements:

```
proc iml;
use Sashelp.Class;
read all var _NUM_ into X;
close Sashelp.Class;
```

The statements open the Sashelp.Class data set, read all the numerical variables into a matrix named **X**, and close the Sashelp.Class data set. The previous statements are equivalent to the following statements, which use an expression (which must be enclosed in parentheses) to specify the data set:

```
dsname = "Sashelp.Class";
use (dsname);
read all var _NUM_ into X;
close (dsname);
```

This alternate syntax is available for specifying a data set name in the CLOSE, CREATE, EDIT, SETIN, SETOUT, SORT, and USE statements.

You can use expressions to interact with a data set whose name is not known until run time. For example, the following statements read several data sets and perform an analysis on each:

```
lib = "Sashelp";
dsnames = {"Class" "Enso" "Iris"};
do i = 1 to ncol(dsnames);
   dsname = concat(lib,".",dsnames[i]);
   use (dsname); /* Sashelp.Class, Sashelp.Enso, etc. */
   read all var _NUM_ into X[c=varNames];
   /* do something with the data in X */
   print dsname varNames;
   close (dsname);
end;
```

Figure 7.1 Looping over Data Sets

dsname	varNames			
Sashelp.Class	Age	Height	Weight	

dsname	varNames		
Sashelp.Enso	Month	Year	Pressure

dsname	varNames			
Sashelp.Iris	SepalLength	SepalWidth	PetalLength	PetalWidth

Make a SAS Data Set Current

The SAS/IML statements that process data operate on the current data set. It is therefore unnecessary to specify the data set as an operand to most statements. There are two current data sets, one for input and one for output. When you open a data set, it is set to be "current." You can also make a data set current by using the SETIN statement or the SETOUT statement. The following list summarizes the statements that change the current data set:

- 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.

The SHOW DATASETS statement displays which data sets are open and which are current for input and output.

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 changed to 1. You can make the Sashelp.Class data set current for input and position the pointer at the tenth observation by using the following statements:

```
use Sashelp.Class;
setin Sashelp.Class point 10;
```

You can then read the tenth observation by using the READ statement, as follows:

```
read current var _NUM_ into x[colname=numVars];
read current var _CHAR_ into c[colname=charVars];
print x[colname=numVars], c[colname=charVars];
```

Figure 7.2 A Single Observation

x		
Age	**Height**	**Weight**
12	59	99.5

c	
Name	**Sex**
John	M

Display SAS Data Set Information

You can use the SHOW statement 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, the following statements display information about the Sashelp.Class data set:

```
use Sashelp.Class;
show datasets;
```

Figure 7.3 Open Data Sets

LIBNAME	MEMNAME	OPEN MODE	STATUS
SASHELP	CLASS	Input	Current Input

As shown in Figure 7.3, Sashelp.Class is the only data set that is open. The USE statement opens the data set for input and makes it the current input data set.

You can see the names of variables, their lengths, and whether they are numeric or character by using the SHOW CONTENTS statement, as follows:

```
show contents;
```

Figure 7.4 Variable Names and Types

```
DATASET : SASHELP.CLASS.DATA
LABEL   : Student Data

 VARIABLE                        TYPE  SIZE
 -------------------------------  ----  ----
 Name                            char     8
 Sex                             char     1
 Age                             num      8
 Height                          num      8
 Weight                          num      8

Number of Variables   : 5
Number of Observations: 19
```

The five variables are shown in Figure 7.4. Name and Sex are character variables; Age, Height, and Weight are numeric variables. The variable Sex has length 1, which means that each observation contains a single character.

List Observations

You can list variables and observations in a SAS data set by using the LIST statement. The general form of the LIST statement is as follows:

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

where

range specifies a range of observations. For details, see the section "Process a Range of Observations" on page 104.

operand selects a set of variables. For details about the VAR clause, see the section "Select Variables with the VAR Clause" on page 105.

expression is an expression that is evaluated as being true or false. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

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

Specify a Range of Observations

You can specify a range of observations with a keyword or by record number by using the POINT option. For example, if you want to list all observations in the Sashelp.Class data set, use the ALL keyword to indicate that the range is all observations, as shown in the following example:

```
use Sashelp.Class;
list all;
```

Figure 7.5 All Observations

OBS	Name	Sex	Age	Height	Weight
1	Alfred	M	14.0000	69.0000	112.5000
2	Alice	F	13.0000	56.5000	84.0000
3	Barbara	F	13.0000	65.3000	98.0000
4	Carol	F	14.0000	62.8000	102.5000
5	Henry	M	14.0000	63.5000	102.5000
6	James	M	12.0000	57.3000	83.0000
7	Jane	F	12.0000	59.8000	84.5000
8	Janet	F	15.0000	62.5000	112.5000
9	Jeffrey	M	13.0000	62.5000	84.0000
10	John	M	12.0000	59.0000	99.5000
11	Joyce	F	11.0000	51.3000	50.5000
12	Judy	F	14.0000	64.3000	90.0000
13	Louise	F	12.0000	56.3000	77.0000
14	Mary	F	15.0000	66.5000	112.0000
15	Philip	M	16.0000	72.0000	150.0000
16	Robert	M	12.0000	64.8000	128.0000
17	Ronald	M	15.0000	67.0000	133.0000
18	Thomas	M	11.0000	57.5000	85.0000
19	William	M	15.0000	66.5000	112.0000

If you do not explicitly specify a range of observations, the LIST statement displays the current observation. Because of the previous LIST statement, the current observation for the Sashelp.Class data is the last observation, as shown in Figure 7.6:

```
list;
```

Figure 7.6 Current Observation

OBS	Name	Sex	Age	Height	Weight
19	William	M	15.0000	66.5000	112.0000

To display a specific set of observations, use the POINT keyword and specify a vector of observation numbers, as shown in the following statement:

```
p = {3 6 9};
list point p;
```

Figure 7.7 Other Observations

OBS	Name	Sex	Age	Height	Weight
3	Barbara	F	13.0000	65.3000	98.0000
6	James	M	12.0000	57.3000	83.0000
9	Jeffrey	M	13.0000	62.5000	84.0000

Select a Set of Variables

You can use the VAR clause to select a set of variables. For example, the following statements list students' names from the Sashelp.Class data set:

```
varNames = {Name Sex Age};
p = {3 6 9};
use Sashelp.Class;
list point p var varNames;
```

Figure 7.8 Listing Specific Variables and Observations

OBS	Name	Sex	Age
3	Barbara	F	13.0000
6	James	M	12.0000
9	Jeffrey	M	13.0000

Select a Set of Observations

The WHERE clause conditionally selects observations, within the *range* specification, according to conditions given in the *expression*. For example, to list the names of all teenage males in the Sashelp.Class data set, use the following statements:

```
varNames = {Name Sex Age};
use Sashelp.Class;
list all var varNames where(Sex='M' & Age>12);
```

Figure 7.9 A Subset of Observations

OBS	Name	Sex	Age
1	Alfred	M	14.0000
5	Henry	M	14.0000
9	Jeffrey	M	13.0000
15	Philip	M	16.0000
17	Ronald	M	15.0000
19	William	M	15.0000

You can use matrices on the right-hand side of the comparison operator. The following example uses the =* operator to find a string that sounds like or is spelled like certain strings:

```
list all var varNames where(name=*{"JON","CAROL","JUDI"});
```

Figure 7.10 Names That Are Close to Specified Strings

OBS	Name	Sex	Age
4	Carol	F	14.0000
7	Jane	F	12.0000
10	John	M	12.0000
12	Judy	F	14.0000

Read Observations from a SAS Data Set

You can use the READ statement to create a SAS/IML matrix from data in a SAS data set. You must first open a SAS data set by using the USE or EDIT statement. If you have several data sets open, you can use the SETIN statement to make one the current input data set.

The general form of the READ statement is as follows:

READ <*range*> <**VAR** *operand*> <**WHERE**(*expression*)> <**INTO** *name*> **;**

where

range specifies a range of observations. For details, see the section "Process a Range of Observations" on page 104.

operand selects a set of variables. For details about the VAR clause, see the section "Select Variables with the VAR Clause" on page 105.

expression is an expression that is evaluated as being true or false. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

name names a target matrix for the data.

Use 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. 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 that are 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 Sashelp.Class data set, use the following statements:

```
proc iml;
use Sashelp.Class;
read all var {Age Height Weight};
close Sashelp.Class;
```

Now use the SHOW NAMES statement to display all the matrices in the current SAS/IML session:

```
show names;
```

Figure 7.11 Matrices Created from Data

```
SYMBOL   ROWS   COLS TYPE   SIZE
------ ------ ------ ---- ------
Age        19      1 num       8
Height     19      1 num       8
Weight     19      1 num       8
 Number of symbols = 5  (includes those without values)
```

Figure 7.11 shows that the READ statement created three numeric vectors: **Age**, **Height**, and **Weight**.

Notice, however, that Figure 7.11 tells you that there are five symbols. The USE statement creates SAS/IML symbols for Name and Sex, but these variables were never read and so the symbols were not assigned values. (You can use the SHOW ALLNAMES statement to see the unassigned symbols.) If the data set contains many variables, it can be more efficient to subset the data set by using the VAR clause in the USE statement. For example, the following statements create the same vectors but do not create symbols for the unread variables:

```
use Sashelp.Class var {Age Height Weight};
read all;
close Sashelp.Class;
```

Use the READ Statement with the INTO Clause

Sometimes it is convenient to read all the numeric variables into columns of a matrix. To do this, use the READ statement with the INTO clause and specify the name of a matrix to create. Each variable that is specified 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 is an $n \times p$ matrix.

The following statement creates a matrix **X** that contains the first five observations of the numeric variables of the Sashelp.Class data set. The keyword _NUM_ in the VAR clause specifies that all numeric variables be read.

```
proc iml;
use Sashelp.Class;
read point (1:5) var _NUM_ into X;
/* Equivalent: range=1:5; read point range var _NUM_ into X; */
print X;
```

Figure 7.12 All Numeric Variables

X		
14	69	112.5
13	56.5	84
13	65.3	98
14	62.8	102.5
14	63.5	102.5

Every SAS/IML matrix is of either character or numeric type. Therefore, when you read data by using the INTO clause, you should use the _NUM_ or _CHAR_ keyword to specify the types of variables that you want to read. If you use the _ALL_ keyword with the INTO statement, all numeric variables are read.

Use the READ Statement with the WHERE Clause

Use the WHERE clause to conditionally select observations from within the specified range. The following statements create a matrix to contain the variables Age, Height, and Weight for females in the Sashelp.Class data set:

```
use Sashelp.Class;
read all var _num_ into Female where(sex="F");
print Female;
```

Figure 7.13 Female Students

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

The section "Process Data by Using the WHERE Clause" on page 106 describes other features of the WHERE clause. For example, you can create a matrix to contain the student names that begin with the letter "J," by using the following statements:

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

Figure 7.14 Names That Begin with the Letter J

J
James
Jane
Janet
Jeffrey
John
Joyce
Judy

Edit a SAS Data Set

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

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

where

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

operand selects a set of variables. For details about the VAR clause, see the section "Select Variables with the VAR Clause" on page 105.

expression is an expression that is evaluated as being true or false. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

This section edits and deletes observations, so make a copy of the Sashelp.Class data set by running the following DATA step:

```
data Class;
set Sashelp.Class;
run;
```

Update Observations

Suppose you have updated data and want to change some values in the Class data set. For instance, suppose the student named Henry recently had a birthday. You can do the following:

- use the EDIT statement to open the Class data set for input and output
- read the data

- change the appropriate data value
- replace the changed data in the data set

The following statements open the Class data set and use the FIND statement to find the observation number that corresponds to Henry. The observation number is stored in the matrix **Obs**, as shown in Figure 7.15.

```
proc iml;
edit Class;
find all where(name={'Henry'}) into Obs;
list point Obs;
```

Figure 7.15 Selected Observation

OBS	Name	Sex	Age	Height	Weight
5	Henry	M	14.0000	63.5000	102.5000

You can read in the Age variable, increment its value, and use the REPLACE statement to overwrite the value in the Class data set, as follows:

```
read point Obs var {Age};
Age = Age + 1;
replace point Obs var {Age};
list point Obs;
close Class;
```

Figure 7.16 New Value

OBS	Name	Sex	Age	Height	Weight
5	Henry	M	15.0000	63.5000	102.5000

Figure 7.16 shows that the value for Henry's age has been updated.

Delete Observations

Use the DELETE statement to mark an observation for subsequent deletion. The general form of the DELETE statement is as follows:

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

where

range specifies a range of observations. For details, see the section "Process a Range of Observations" on page 104.

expression is an expression that is evaluated as being true or false. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

The DELETE statement marks observations for deletion. To actually delete the marked observations and renumber the remaining observations, use the PURGE statement.

Suppose the student named John has moved and you want to delete the corresponding observation from the Class data set, which was created in the section "Edit a SAS Data Set" on page 95. The following statements find the observation for John and mark it for deletion:

```
edit Class;
find all where(name={'John'}) into Obs;
delete point Obs;
```

To update the data set and renumber the observations, use the PURGE statement. Be aware that the PURGE statement deletes any indexes associated with a data set.

```
purge;
show contents;
```

Figure 7.17 Updated Data Set

```
DATASET : WORK.CLASS.DATA

 VARIABLE                        TYPE  SIZE
 -------------------------------  ----  ----
 Name                            char     8
 Sex                             char     1
 Age                             num      8
 Height                          num      8
 Weight                          num      8

Number of Variables   : 5
Number of Observations: 18
```

The data set now has 18 observations, whereas it used to have 19.

Create a SAS Data Set from a Matrix

SAS/IML software provides the capability to create a new SAS data set from matrices. You can use the CREATE and APPEND statements to create a SAS data set from one or more matrices. An $n \times p$ matrix creates a SAS data set with p variables and n observations. That is, the columns of the matrix become the data set variables, and the rows of the matrix become the observations. The CREATE statement creates the new SAS data set, and the APPEND statement writes the observations.

Use the CREATE Statement with the FROM Option

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

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

where

SAS-data-set	specifies the name of the new data set.
matrix-names	specifies one or more matrices that contain the data.
column-name	specifies names for the data set variables.
row-name	adds a character variable that identifies each row in the data set.

Suppose you want to create a SAS data set to contain a variable with the height-to-weight ratio for each student. The following statements read variables from the Sashelp.Class data set, form the ratio, and use the CREATE and APPEND statements to write a new SAS data set called Ratio:

```
proc iml;
use Sashelp.Class;
read all var {Name Height Weight};

HWRatio = Height/Weight;
create Ratio from HWRatio[colname="HtWt"];
append from HWRatio;
show contents;
close Ratio;
```

Figure 7.18 New Data Set from a Matrix

```
DATASET : WORK.RATIO.DATA

 VARIABLE                         TYPE  SIZE
 -------------------------------  ----  ----
 HtWt                             num      8

Number of Variables   : 1
Number of Observations: 19
```

The variable in the Ratio data set is called HtWt. If you do not specify the COLNAME= option, the variables in the new data set are named COL1, COL2, and so forth.

Use the CREATE Statement with the VAR Clause

You can use a VAR clause with the CREATE statement to select the variables that you want to include in the new data set.

The syntax is as follows:

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

In the previous example, the new data set Ratio had one variable. You can use the VAR clause to create a similar data set to contain both HWRatio and Name. Notice that the variable HWRatio is numeric and the variable Name is character. Consequently, these two variables cannot coexist in a single matrix. The following statements create a new data set, Ratio2, to contain the variables Name and HWRatio:

```
create Ratio2 var {"Name" "HWRatio"};
append;
show contents;
close Ratio2;
```

Figure 7.19 New Data Set from Variables

```
DATASET : WORK.RATIO2.DATA

 VARIABLE                        TYPE  SIZE
 ------------------------------  ----  ----
 Name                            char     8
 HWRatio                         num      8

Number of Variables   : 2
Number of Observations: 19
```

Understand the End-of-File Condition

An *end-of-file condition* occurs when you try to read past the end of a data set or when you point to an observation that exceeds the number of observations in a data set. If an end-of-file condition occurs inside a DO DATA iteration loop, control passes to the first statement after the DO DATA loop.

The following example uses a DO DATA loop to read observations from the Sashelp.Class data set. The loop reads the data one observation at a time and accumulates the weights of the students in the SAS/IML matrix named **sum**. After the data are read, the variable **sum** contains the sum of the weights for the class.

```
proc iml;
use Sashelp.Class;
/* if data set already open, use
   setin class point 0; */
sum=0;
do data;
   read next var{weight};
   sum = sum + weight;
end;
print sum;
```

Figure 7.20 Sum of Data Values

sum
1900.5

The example shows how to read data one observation at a time within a DO loop. However, there are more efficient ways to read data in the SAS/IML language. If all the data can fit into memory, it is more efficient to read all the data into a vector and use vector operations to compute statistical quantities such as a sum. For example, the following statements compute the same quantity (the sum of the Weight variable) but are more efficient:

```
read all var{weight};
sum = sum(weight);
```

Produce Summary Statistics

You can use the SUMMARY statement to compute summary statistics of numeric variables in a SAS data set. The statistics can be computed for subgroups of the data by using the CLASS clause. The SAVE option in the OPT clause enables you to save the computed statistics in matrices. For example, consider the following statements:

```
proc iml;
use Sashelp.class;
summary class {sex} var {height weight};
```

Figure 7.21 Summary Statistics

Sex	Nobs	Variable	MIN	MAX	MEAN	STD
F	9	Height	51.30000	66.50000	60.58889	5.01833
		Weight	50.50000	112.50000	90.11111	19.38391
M	10	Height	57.30000	72.00000	63.91000	4.93794
		Weight	83.00000	150.00000	108.95000	22.72719
All	19	Height	51.30000	72.00000	62.33684	5.12708
		Weight	50.50000	150.00000	100.02632	22.77393

As shown in Figure 7.21, the default statistics are the minimum, maximum, mean, and standard deviation of each variable specified in the VAR clause. The default behavior is to display the summary statistics in a table.

You can also store the summary statistics in SAS/IML matrices. For example, the following statement creates four matrices: Sex, _OBS_, Height, and Weight:

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

Because the SAVE option was specified, the statistics of the variables are stored in matrices under the name of the corresponding variables: each column corresponds to a statistic, and each row corresponds to a subgroup. Two other vectors, **Sex** and **_NOBS_**, are also created. The vector **Sex** contains the two distinct values of the CLASS variable. The vector **_NOBS_** contains the number of observations in each subgroup. The following statements display the SAS/IML matrices that are defined and print the **height** and **weight** matrices:

```
show names;
/* print matrices that show the stats */
print height[r=sex c={"Mean" "Std" "Var"}],
      weight[r=sex c={"Mean" "Std" "Var"}];
```

Figure 7.22 Summary Statistics

```
SYMBOL   ROWS   COLS TYPE   SIZE
------ ------ ------ ---- ------
Height      2      3 num       8
Sex         2      1 char      1
Weight      2      3 num       8
_NOBS_      2      1 num       8
 Number of symbols = 6  (includes those without values)
```

	Height		
	Mean	**Std**	**Var**
F	60.588889	5.0183275	25.183611
M	63.91	4.937937	24.383222

	Weight		
	Mean	**Std**	**Var**
F	90.111111	19.383914	375.73611
M	108.95	22.727186	516.525

You can specify more than one CLASS variable, in which case subgroups are defined by the joint combinations of the values of the CLASS variables.

Sort a SAS Data Set

The observations in a SAS data set can be ordered (sorted) by specific variables. To sort a SAS data set, first close the data set if it is open. Then submit a SORT statement and specify the ordering variables. You can also specify an output data set name if you do not want to overwrite the original data set. For example, the following statements create a new SAS data set named Sorted:

```
proc iml;
sort Sashelp.Class out=Sorted by name;
```

The new data set contains the observations from the data set Sashelp.Class, ordered by the variable Name. If you omit the OUT= option, the original data set is replaced by the sorted data set.

You can specify multiple sort variables. Optionally, each variable can be preceded by the keyword DESCENDING, which denotes that the subsequent variable is to be sorted in descending order.

Index a SAS Data Set

Searching through a large data set for observations that satisfy some complex criteria can take a long time. You can reduce this search time by indexing the data set. The INDEX statement builds a special companion file that contains the values and record numbers of the indexed variables. After the index is built, queries that use a WHERE clauses might use the index to make the processing more efficient. Any number of variables can be indexed, but only one index is in use at a given time.

If you sort a data set in place or use the PURGE statement to delete observations, indices for the data set are deleted.

After you index a data set, the SAS/IML language has the option to use the index when 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. In other words, you cannot use the POINT clause when an index is in effect.

To see how the INDEX statement works, make a copy of the Sashelp.Class data set, as follows:

```
data Class;
set Sashelp.Class;
run;
```

If you want a list of all female students in the Class data set, you can first index the data by the Sex variable:

```
proc iml;
use Class;
index Sex;
```

If you subsequently submit a WHERE clause that uses the Sex variable, the index is used. Of course, the Class data set is small, so you will not notice any performance improvement for these data. However, for large data sets indexing can improve performance.

Now list all students by using the following statement:

```
list all;
```

Figure 7.23 Indexed Observations

OBS	Name	Sex	Age	Height	Weight
2	Alice	F	13.0000	56.5000	84.0000
3	Barbara	F	13.0000	65.3000	98.0000
4	Carol	F	14.0000	62.8000	102.5000
7	Jane	F	12.0000	59.8000	84.5000
8	Janet	F	15.0000	62.5000	112.5000
11	Joyce	F	11.0000	51.3000	50.5000
12	Judy	F	14.0000	64.3000	90.0000
13	Louise	F	12.0000	56.3000	77.0000
14	Mary	F	15.0000	66.5000	112.0000
1	Alfred	M	14.0000	69.0000	112.5000
5	Henry	M	14.0000	63.5000	102.5000
6	James	M	12.0000	57.3000	83.0000
9	Jeffrey	M	13.0000	62.5000	84.0000
10	John	M	12.0000	59.0000	99.5000
15	Philip	M	16.0000	72.0000	150.0000
16	Robert	M	12.0000	64.8000	128.0000
17	Ronald	M	15.0000	67.0000	133.0000
18	Thomas	M	11.0000	57.5000	85.0000
19	William	M	15.0000	66.5000	112.0000

Notice that the indexed observations are sorted by Sex rather than by the OBS number. Retrievals that use the Sex variable are quicker than retrievals of data that are not indexed.

Data Set Maintenance Functions

The following functions and subroutines perform data set maintenance tasks:

DATASETS function	obtains members in a data library. This function returns a character matrix that contains the names of the SAS data sets in a library.
CONTENTS function	obtains variables in a SAS data set. This function returns a character matrix that contains the variable names for the data set. The variable list is returned in alphabetical 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 25 for details and examples of these functions and routines.

Summary of Commands

You can use the functions, subroutines, and statements in this chapter to interact with SAS data sets. Table 7.1 summarizes the statements that you can use to perform management tasks from within the SAS/IML language.

Table 7.1 Data Management Commands

Command	Description
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 SAS/IML variables
REPLACE	Writes observations back into a SAS data set
RESET DEFLIB	Names default libref
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

Shared Concepts for Processing Data

This section describes concepts that are common to two or more SAS/IML statements and that are related to reading or writing data.

Process a Range of Observations

The following SAS/IML statements enable you to specify a range of observations to process:

- DELETE statement
- FIND statement
- LIST statement
- READ statement
- REPLACE statement

You can specify a range of observations by using one of the following keywords:

ALL specifies all observations.

CURRENT specifies the current observation.

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

AFTER specifies all observations after the current one.

POINT *value* specifies observations by number.

Usually the **ALL** keyword is used, but the default value for the range is **CURRENT**. The NEXT and POINT keywords support values. The values can be literals, expressions, and numeric matrices, as shown in Table 7.2.

Table 7.2 Values Supported by the POINT and NEXT Keywords

Value	Example
A single record number	`read point 5`
A literal that contains several record numbers	`read point {2 5 10}`
The name of a matrix that contains record numbers	`p=1:5; read point p;`
An expression in parentheses	`read point (p+1);`

If the current data set has an index in use (see the INDEX statement), the **POINT** keyword is invalid.

The following statements specify ranges of observations to the LIST statement. The output is not shown.

```
proc iml;
use Sashelp.class;

list all;                                /* lists whole data set */
list;                                  /* lists current observation */
list var{name age};          /* lists NAME and AGE in current obs */
list all where(age<=13); /* lists all obs where condition holds */
list next;                               /* lists next observation */
list point 18;                             /* lists observation 18 */
range = 10:15;
list point range;             /* lists observations 10 through 15 */

close Sashelp.class;
```

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

Statement	Default Range
LIST	Current
READ	Current
FIND	All
REPLACE	Current
DELETE	Current

The APPEND statement does not support a *range* operand; new observations are always appended to the end of a data set.

Select Variables with the VAR Clause

Several SAS/IML statements support a VAR clause that specifies variables to use for subsequent processing. The VAR clause is supported by the following statements:

- APPEND statement
- CREATE statement
- EDIT statement
- LIST statement
- READ statement
- REPLACE statement
- SUMMARY statement
- USE statement

The general form of the VAR clause is

VAR *vars* **;**

The argument *vars* is one of the following:

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

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

The following examples demonstrate ways to use the VAR clause:

```
proc iml;
use Sashelp.Class;
read all var {age sex};         /* a literal matrix of names        */
varNames = {"weight" "height"};
read all var varNames;          /* a matrix that contains the names */
read all var _NUM_ into X;      /* a keyword                        */
close Sashelp.Class;

x1 = X[,1]; x2 = X[,2]; x3 = X[,3];
create Test var ("x1":"x3");    /* an expression                    */
append;
close Test;
```

Process Data by Using the WHERE Clause

Several SAS/IML statements support a WHERE clause that selects observations that satisfy specified criteria. The WHERE clause is supported by the following statements:

- DELETE statement
- EDIT statement
- FIND statement
- LIST statement
- READ statement
- REPLACE statement
- SUMMARY statement

- USE statement

The WHERE clause conditionally selects observations that satisfy some criterion. The general form of the WHERE clause is

WHERE *variable comparison-op operand* **;**

The arguments to the WHERE clause are as follows:

variable is a variable in the SAS data set.

comparison-op is one of the following comparison operators:

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

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

For example, a typical use of the WHERE clause is to subset data:

```
proc iml;
use Sashelp.Class where(age>14);
read all var {Age Weight} into X;
close Sashelp.Class;
print X[colname={"Age" "Weight"}];
```

Figure 7.24 Observations That Satisfy a Criterion

X	
Age	**Weight**
15	112.5
15	112
16	150
15	133
15	112

You can also use a WHERE clause in the READ statement. For example, to conduct BY-group processing of all the students in the Sashelp.Class data set, first call the FREQ procedure to find the unique BY groups, and then use a WHERE clause in a DO loop to read observations from each BY group, as shown in the following example:

```
/* find unique BY combinations of Age and Sex */
proc freq data=Sashelp.Class;
tables Age*Sex / out=freqout
              nocum norow nocol nopercent;
run;

proc iml;
/* read unique BY groups */
use freqout;
read all var {Age Sex};
close freqout;

use Sashelp.Class;            /* open data set for reading */
MeanHeight = j(nrow(Sex), 1); /* allocate vector for results */
do i = 1 to nrow(Age);        /* for each BY group */
   /* read data for the i_th group */
   read all var {Height} where(Sex=(sex[i]) & Age=(age[i]));
   MeanHeight[i] = mean(Height); /* analyze this BY group */
end;
close Sashelp.Class;

print Age Sex MeanHeight[format=4.1];
```

Figure 7.25 BY-Group Processing

The FREQ Procedure

Frequency	Table of Age by Sex		
		Sex	
Age	F	M	Total
11	1	1	2
12	2	3	5
13	2	1	3
14	2	2	4
15	2	2	4
16	0	1	1
Total	9	10	19

Age	Sex	MeanHeight
11	F	51.3
11	M	57.5
12	F	58.1
12	M	60.4
13	F	60.9
13	M	62.5
14	F	63.6
14	M	66.3
15	F	64.5
15	M	66.8
16	M	72.0

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

=, ?, =:, =*

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

^=, ^?, <, <=, >, >=

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

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

In the WHERE clause, the expression on the left side of a comparison operator refers to values of the data set variables, whereas the expression on the right side is a constant or SAS/IML matrix. Expressions that involve more than one data set variable in a single clause are not supported. For example, you cannot use either of the following expressions:

```
list all where(weight>height);  /* not supported */
list all where(weight-height>0);/* not supported */
```

Using Data Set Options

The SAS/IML USE, EDIT, and CREATE statements support most of the standard SAS data set options, as documented in *SAS Data Set Options: Reference*. For example, the following statements use the OBS=, RENAME=, and DROP= data set options to read data from the Sashelp.Class data set:

```
proc iml;
use sashelp.class(obs=5
      rename=(sex=Gender)
      drop=Age);
read all var _NUM_ into X[colname=nNames];
read all var _CHAR_ into C[colname=cNames];
close sashelp.class;
print X[c=NNames] C[c=cNames];
```

Comparison with the SAS DATA Step

The SAS/IML environment enables you to perform basic manipulation of data. However, there are some differences between the SAS/IML language and the SAS DATA step:

- With SAS/IML software, you open a file for output by using the CREATE statement. You must explicitly set up all your variables with the correct attributes before you create a data set. This means that you must define character variables to have the desired length. 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 each observation 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. (However, PROC IML automatically closes all open data sets when the procedure exits.) The DATA step closes the data set automatically at the end of the step.
- When reading or writing data, the DATA step usually executes faster than the equivalent operation in the SAS/IML language.

In short, the DATA step treats the problem with greater simplicity, allowing shorter programs. However, the SAS/IML language is more flexible and interactive, and it has powerful matrix-handling capabilities.

Chapter 8
File Access

Contents

Overview of File Access **111**
Referring to an External File **112**
Types of External Files 113
Reading from an External File **113**
Using the INFILE Statement 113
Using the INPUT Statement 114
Writing to an External File **121**
Using the FILE Statement 122
Using the PUT Statement 122
Examples 123
Listing Your External Files **126**
Closing an External File **126**
Summary **126**

Overview of File Access

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 by using the INFILE and INPUT statements and how to write data to an external file by 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 SAS/IML statements used to access files are very similar to the corresponding statements in the SAS DATA step. The following table summarizes the SAS/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 these data into SAS/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 *path name* and a *file name*. A path name is the name of the file as it is known to the operating system. A file name is an indirect SAS reference to the file made by using the FILENAME statement. You can identify a file in either way by using the FILE and INFILE statements.

For example, you can refer to the input file where the class data are stored by using a literal path name—that is, a quoted string. The following statement opens the file USER.TEXT.CLASS for input:

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

Similarly, if you want to output data to the file USER.TEXT.NEWCLASS, you need to reference the output file with the following statement:

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

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

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

The following statements achieve the same result as the previous INFILE statement with the quoted path name:

```
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 file names have special meaning in the SAS/IML language: CARDS, LOG, and PRINT. These refer to the standard input and output streams for all SAS sessions, as follows:

CARDS is a special file name for in-stream input data.

LOG is a special file name for log output.

PRINT is a special file name for standard print output.

When the path name 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 can 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 by using the byte operand (>) in the INPUT or PUT statement.

You write data to an external file by 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 files and text files in output is that with binary files, the PUT statement does not put the record-separator characters at the end of each record written.

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 specify 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 that contains 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 that contains 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.

The following example uses 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 path name with a quoted literal also. The preceding statements could be written as follows:

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

Using the INPUT Statement

Once you have referenced the data file that contains your data with an INFILE statement, you need specify the following information about 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 specify 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 SAS/IML variables.

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

- list (or scanning) input
- formatted input

Following 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 specify 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 specify the following:

- NAME is a character variable; its value begins in column 1 (indicated by @1) and occupies eight columns ($CHAR8.).
- SEX is a character variable; its value is found in column 10 ($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 the SAS/IML language 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, SAS/IML looks for two blanks, a comma, or the end of the record.

The general form of the INPUT statement for list input is as follows:

INPUT *variable* < *$* > < *&* > < . . . *variable* < *$* > > < *&* > > **;**

where

variable names the variable to be read by the INPUT statement.

$ indicates that the preceding variable is character.

& indicates that a character value can 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, SAS/IML scans the input lines for values. Consider using list input in the following cases:

- when blanks or commas separate input values
- when periods rather than blanks represent missing values

List input is the default in several situations. Descriptions of these situations and the behavior of SAS/IML follow:

- If no input format is specified for a variable, SAS/IML scans for a number.
- If a single dollar sign or ampersand format modifier is specified, SAS/IML scans for a character value. The ampersand format modifier enables single embedded blanks to occur.
- If a format is given with width unspecified or zero, SAS/IML scans for the first blank or comma.

If the end of a record is encountered before a value is found, then the behavior is as described by the record overflow options in the INFILE statement discussed in the section "Using the INFILE Statement" on page 113.

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 these 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 can be used with pointer controls and format modifiers. However, 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 (+). A complete list follows:

@*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 of 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 that used formatted input, you used several pointer controls. Here are the statements:

```
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 follows:

```
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 SAS/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 SAS/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, 10 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 specify how to read the values by 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 (@) with a character operand. SAS/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 following statement searches for the pattern **NAME=** and then uses list input to read the value after the found pattern:

```
input @ 'NAME=' name $;
```

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" on page 113. 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 in the same record, even if some of them are not found. For example, the following statements are able to read in the ADDR variable even if **NAME=** is not found (in which case, NAME is unvalued):

```
infile in1 missover;
input @1 @ "NAME=" name $
      @1 @ "ADDR=" addr &
      @1 @ "PHONE=" phone $;
```

The pattern operand can use any characters except for the following:

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

Record Directives

Each INPUT statement goes to a new record except in 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 BZ*w*. format turns blanks into zeros. See *SAS Language Reference: Dictionary* for more information about 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 inside a DO DATA loop, execution is passed to the statement after the END statement in the loop.

For text files, 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, end of file can result in the input mechanism returning a record that is shorter than the requested length. In this case SAS/IML still attempts to process the record, using the rules described in the section "Using the INFILE Statement" on page 113.

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 that specifies 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 statements should 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;
```

Notice that the APPEND statement is not executed if the INPUT statement reads past the end of file because SAS/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 SAS/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 SAS/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 these data to an external file, you need to reference, or point to, the file (as discussed in the section "Referring to an External File" on page 112. 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 are 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 path name. Here is the statement:

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

Otherwise, you can first establish a fileref and then refer to the file by its fileref, as follows:

```
> 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 SAS/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. SAS/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" on page 114.

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 you have a matrix **X** that contains data that you want written to the file USER.MATRIX. Suppose also that **X** contains ones and zeros so that the format for output can be one column. You need to do the following:

1. Establish a fileref, such as OUT.
2. Use a FILE statement to open the file for output.
3. Specify a DO loop for the rows of the matrix.
4. Specify a 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 statements 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;
```

Here is the resulting output:

```
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 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 by using 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 path name. 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 by using the FILE and PUT statements. Finally, you learned how to close your files.

Chapter 9
Mixed-Type Tables

Contents

Introduction to Tables **127**
Create Tables from SAS Data Sets **128**
Create Tables from Matrices **129**
Create Tables from Other Sources **129**
Query Properties of Tables **129**
Set Properties of Tables **131**
Extract Data from a Table to Matrices **131**
Modify Tables **131**
Save Tables to SAS Data Sets **132**
Copy Tables **133**
Release Table Memory **133**
Compute Statistics for Columns of Tables **134**
Print Tables **135**
Advanced Printing of Tables **136**
Using Custom Templates 137
Using Dynamic Variables in Custom Templates 138
Using Column Templates 140
Using Dynamic Variables in Column Templates 141
Using Existing SAS Templates 143
References **145**

Introduction to Tables

The matrix is the fundamental data type for SAS/IML computations. A matrix must contain either all numeric or all character values. You cannot create a matrix in which one column has numeric values and another column has character values.

In contrast, SAS data sets can contain mixed-type data. Beginning with SAS/IML 14.2, you can create *tables*, which are in-memory versions of data sets. You can create a table from a SAS data set or from a SAS/IML matrix. You can add new columns to the table or extract values from the table into matrices. You can pass the table to a SAS/IML module, and modules can create new tables or modify existing tables.

Tables are a convenient way to store mixed-type data and to pass data to modules. You cannot perform linear algebraic operations on a table. For example, you cannot add two tables together or compute the eigenvalues

of a table, even if all the columns of the table are numeric. However, SAS/IML includes functions for creating matrices from the columns of tables, so you can extract the data and then compute with the matrices.

In this chapter, the word "variable" can have several meanings. When there is the potential for confusion, this chapter uses the following conventions:

- A variable is a column in a data set.
- A *column* is a column in a SAS/IML table.
- A *symbol* is the name of a SAS/IML matrix, vector, or table.
- A *dynamic variable* is a symbol that is named in the DYNAMIC statement in the TEMPLATE procedure. The value of a dynamic variable is specified at run time.

For example, you might read a data set that has three variables into a SAS/IML table that has three columns. The name of the table might be the symbol tbl.

Create Tables from SAS Data Sets

The easiest way to create an in-memory table is to use the TableCreateFromDataSet function to create a table directly from a SAS data set. The following statements read the Sashelp.Class data set into a table:

```
proc iml;
tClass = TableCreateFromDataSet("Sashelp", "Class");
```

The tClass symbol is a table. It can be passed as an argument to any SAS/IML function that supports tables. For example, you can use the NROW and NCOL functions to obtain the number of rows and columns, respectively, for the tClass table:

```
nrow = nrow(tClass);
ncol = ncol(tClass);
print nrow ncol;
```

Figure 9.1 Number of Rows and Columns for a Table

nrow	ncol
19	5

You can also use data set options to filter data. For example, you can use the DROP=, KEEP=, OBS=, and WHERE= options to restrict the data that are read into a table, as shown in the following statements:

```
dsOpt = "drop=Name rename=(Weight=Mass) where=(Sex='M')";
tblBoys = TableCreateFromDataSet("Sashelp", "Class", dsOpt);
```

Create Tables from Matrices

You can use the TableCreate function to create a table from a SAS/IML matrix. You can also use the TableAddVar subroutine to add columns to an existing table.

The following statements define two character vectors that contain data about historical hurricanes in the Atlantic basin. The TableCreate function is used to create a table from the data:

```
proc iml;
Hurr  = {"Katrina", "Ike",       "Andrew", "Wilma"};
Month = {"August",  "September", "August", "October"};
tbl = TableCreate({"Name" "Month"}, Hurr||Month);
```

Because a matrix cannot contain mixed types, you might need to call the TableAddVar subroutine to add columns to the table. The following statements add two new columns to the existing table:

```
Yr   = {2005, 2008, 1992, 2005};
Wind = { 175,  145,  175,  185};   /* max wind, mph */
call TableAddVar(tbl, {"Year" "MaxWind"}, Yr||Wind);  /* add numeric cols */
```

The order of the columns in the table is the order in which the columns were added. For the current table, the character columns appear first, followed by the numeric columns. You can use the TableGetVarName function to obtain the column names of a table:

```
colNames = TableGetVarName(tbl);
print colNames;
```

Figure 9.2 Names of Table Columns

colNames			
Name	Month	Year	MaxWind

The TABLECREATE function supports an alternative syntax that enables you to create a matrix from multiple matrices, both numeric and character. For example, you can also create the previous table as follows:

```
tbl = TableCreate({"Name" "Month"}, Hurr||Month,  /* character data */
                  {"Year" "MaxWind"}, Yr||Wind);  /* numeric data */
```

Create Tables from Other Sources

You can use the ImportTableFromR function to create a table from an R data frame or matrix. You can use the ExportTableToR subroutine to create an R data frame from a SAS/IML table.

Query Properties of Tables

The following list describes functions that enable you to retrieve properties of a table:

- The TYPE function returns the type of the input symbol:
  - If a symbol is undefined, the TYPE function returns U.
  - If a symbol is a character matrix, the TYPE function returns C.
  - If a symbol is a numeric matrix, the TYPE function returns N.
  - If a symbol is a table, the TYPE function returns T.
- The following functions return the dimensions of a table:
  - The NROW function returns the number of rows.
  - The NCOL function returns the number of columns.
  - The DIMENSION function returns both quantities in a row vector.
- The following functions return a row vector that contains information about the columns, such as names, labels, and types (character or numeric):
  - The TableGetVarFormat function returns the formats that are associated with columns. If a column does not have a format, a blank is returned.
  - The TableGetVarIndex function returns the column numbers when you specify the column names.
  - The TableGetVarInformat function returns the informats that are associated with columns. If a column does not have an informat, a blank is returned.
  - The TableGetVarLabel function returns the labels that are associated with columns. If a column does not have a label, a blank is returned.
  - The TableGetVarName function returns the names of columns in a table when you specify the column numbers (indices).
  - The TableGetVarType function returns a vector that contains values C and N to represent character and numeric columns, respectively.
  - The TableIsExistingVar function determines whether a table contains a specified column name. The function returns a binary indicator vector.
  - The TableIsVarNumeric function returns a vector that contains values 0 and 1 to indicate character and numeric columns, respectively.

The following statements create a table from the Sashelp.Class data set and return information about the columns. The information is shown in Figure 9.3.

```
proc iml;
tClass = TableCreateFromDataSet("Sashelp", "Class");
colnames = TableGetVarName(tClass);
type = TableGetVarType(tClass);
isNum = TableIsVarNumeric(tClass);
print (type // char(isNum))[c=colnames r={"Type" "IsNum"}];
```

Figure 9.3 Information about a Table

	Name	Sex	Age	Height	Weight
Type	C	C	N	N	N
IsNum	0	0	1	1	1

Set Properties of Tables

The columns of tables have properties such as names, formats, informats, and labels. The following functions enable you to set the properties for columns of a table:

- The TableRenameVar call sets the names of columns.
- The TableSetVarFormat call sets the formats that are associated columns.
- The TableSetVarInformat call sets the informats that are associated with columns.
- The TableSetVarLabel call sets the labels that are associated with columns.

Extract Data from a Table to Matrices

You can use the TableGetVarData function to extract values from the table into a matrix. You can then perform standard matrix computations on the data. For example, the following statements create a table from the Sashelp.Class data set. The TableGetVarData function extracts the numeric variables Height and Weight into a matrix. The COV function then computes the covariance matrix for those two variables. The covariance matrix is shown in Figure 9.4.

```
proc iml;
tClass = TableCreateFromDataSet("Sashelp", "Class");
colNames = {"Height" "Weight"};
X = tableGetVarData(tClass, colNames);
cov = cov(X);
print cov[c=colNames r=colNames];
```

Figure 9.4 Covariance Matrix for Numeric Variables

cov		
	Height	**Weight**
Height	26.286901	102.49342
Weight	102.49342	518.65205

Because the TableGetVarData function returns a matrix, the variable names that you specify must be all numeric or all character.

Modify Tables

Tables enable you to use a single parameter to send mixed-type data into a SAS/IML module. The module can use the TableGetVarData function to extract values from the table into a matrix. It can also use the TableAddVar function to add new columns to the table.

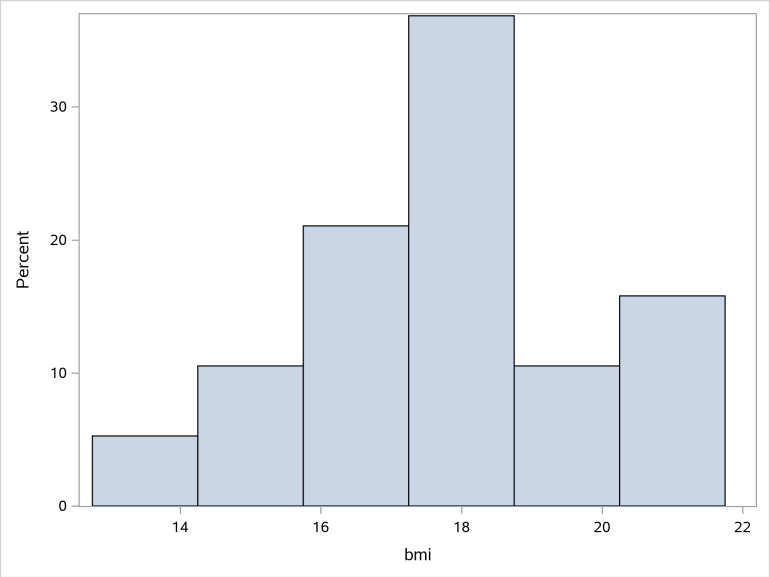
The following module takes a table as an argument. It assumes that the table contains columns named Height and Weight, where height is measured in inches and weight is measured in pounds. (It could use the TableIsExistingVar function to verify that columns with these names exist.) The module extracts the data from these columns into vectors, and uses a standard formula to compute the body mass index (BMI) for each person in the data. The BMI values are then added to the table.

```
proc iml;
/* Compute BMI from the Height and Weight vars   */
start AddBMI(tbl);
   weight = TableGetVarData(tbl, "Weight");
   height = TableGetVarData(tbl, "Height");
   BMI = weight / height##2 * 703;    /* standard formula */
   call TableAddVar(tbl, "BMI", BMI); /* add numeric col */
   return;
finish;
```

The following statements create a table from the Sashelp.Class data set. The table is passed into the AddBMI module, which modifies the table. When the module returns, the main program extracts the BMI values from the table and displays a histogram of the BMI values. The histogram is shown in Figure 9.5.

```
tClass = TableCreateFromDataSet("Sashelp", "Class");
run AddBMI(tClass);          /* call a module that modifies the table */
bmi = TableGetVarData(tClass, "BMI");
call histogram(bmi);         /* graph distribution of derived column */
```

Figure 9.5 Histogram of Computed Column



Save Tables to SAS Data Sets

Just as the TableCreateFromDataSet function enables you to read a data set into a SAS/IML table, the TableWriteToDataSet call enables you to create a SAS data set from a table. The following statements create

a table from two vectors and then write the data to a SAS data set named Letters in the Work library. The CONTENTS function is used to retrieve the names of the variables in the data set. The names are displayed in Figure 9.6.

```
proc iml;
tbl = TableCreate("Letters", T("A":"Z"));  /* create a character column   */
call TableAddvar(tbl, "ID", T(1:26));      /* create a numeric column     */
call TableWriteToDataSet(tbl, "work", "Letters");  /* create SAS data set */

vars = contents("Letters");                /* variables in work.Letters   */
print vars;
```

Figure 9.6 Creating a SAS Data Set from a Table

vars
Letters
ID

You can include data set options in order to write a subset of the table to a data set. The following statement uses the WHERE= option to write half of the data to a data set. The Letters2 data set contains all data for which the ID column is a multiple of 2.

```
call TableWriteToDataSet(tbl, "work", "Letters2",
                         "where=(mod(ID,2)=0)");  /* write subset */
```

In a similar way, you can use the DROP= and KEEP= data set options to create a SAS data set that contains only certain columns in a table.

Copy Tables

You can copy a table from one symbol into another by using the assignment operator (=) as shown in the following statements:

```
proc iml;
tClass = TableCreateFromDataSet("Sashelp", "Class");
tNew = tClass;        /* make a copy of the table */
quit;
```

Tables are copied by value. After you make a copy of a table, changes to the original table do not affect the copy. Similarly, you can modify the copy without affecting the original table.

The VALUE function and VALSET call also enable you to copy a table by specifying the name of the table.

Release Table Memory

Like matrices, tables are stored in memory. Consequently, large tables might consume a lot of the RAM on your computer. When you are finished using a table, you can use the FREE statement to free the symbol and release the memory. You can then reuse the symbol.

For example, the following statements create a table and then use the FREE statement to delete the table from memory and free the symbol. The SHOW NAMES statement displays the symbols that have values. Figure 9.7 shows that the **tbl** symbol is no longer defined.

```
proc iml;
x = 1:10;
y = {1 2, 3 4};
tbl = TableCreateFromDataSet("Sashelp", "BWeight");   /* load a big table */
free tbl;                                             /* delete the big table */
show names;
quit;
```

Figure 9.7 Symbols after Deleting Table

```
SYMBOL   ROWS   COLS TYPE   SIZE
------ ------ ------ ---- ------
x           1     10 num       8
y           2      2 num       8
 Number of symbols = 3  (includes those without values)
```

Compute Statistics for Columns of Tables

The computational functions and operators in the SAS/IML language are designed to operate on matrices, not tables. You cannot add two tables or pass a table to the MEAN function to compute the mean of each column. However, you can write a SAS/IML module that accepts a table as a parameter and computes statistics for one or more columns. To do this, use the TableGetVarData function to extract the data from a table into a matrix. For example, the following function prints a table of descriptive statistics for each numeric column in a table. The statistics are shown in Figure 9.8.

```
proc iml;
start PrintDescStats( tbl );
   cols = loc( TableIsVarNumeric(tbl) ); /* get column numbers */
   if ncol(cols)=0 then do;
      print "The table does not contain any numeric columns.";
      return;
   end;
   stats = j(5, ncol(cols));            /* allocate matrix for results */
   m = TableGetVarData(tbl, cols);      /* extract data into matrix */
   stats[1,] = countn(m);               /* N       for each column  */
   stats[2,] = mean(m);                 /* Mean    for each column  */
   stats[3,] = std(m);                  /* Std Dev for each column  */
   stats[4,] = m[><, ];                 /* Minimum for each column  */
   stats[5,] = m[<>, ];                 /* Maximum for each column  */
   varNames = TableGetVarName(tbl, cols);
   rowNames = {"N", "Mean", "Std Dev", "Minimum", "Maximum"};
   print stats[L="Descriptive Statistics" r=rowNames c=varNames];
finish;

table = TableCreateFromDataSet("Sashelp", "Class");
run PrintDescStats(table);
```

Figure 9.8 Descriptive Statistics for Numeric Columns

Descriptive Statistics			
	Age	**Height**	**Weight**
N	57	57	57
Mean	13.315789	62.336842	100.02632
Std Dev	1.4926722	5.1270752	22.773933
Minimum	11	51.3	50.5
Maximum	16	72	150

Print Tables

The SAS Output Delivery System (ODS) displays tables by using a template. A template determines the layout of the table and the styles that are used to display the individual headers, columns, and cells. You can print a table in SAS/IML software by using the TablePrint subroutine. By default, the TablePrint subroutine uses the IML.Table template. The following PROC TEMPLATE statements display the IML.Table template in the SAS log:

```
proc template;
   source IML.Table;
run;
```

The IML.Table template is not complicated. It defines columns that can be used to display numeric or character variables. The IML.Table template supports standard tables that contain a table header, column headers, and row headers.

The following basic options for the TablePrint subroutine control the appearance of the displayed table:

- The VAR= option specifies the columns to print. By default, all columns are printed.
- The ID= option specifies a column to use as row headers. By default, row numbers are used as row headers.
- The LABEL= option specifies a header for the entire table. By default, the subroutine prints the name of the SAS/IML symbol as the table header.
- The FIRSTOBS= option specifies the initial row to print. By default, the subroutine prints the first row in the table as the initial row.
- The NUMOBS= option specifies the number of rows to print. By default, the initial row and all subsequent rows are printed.
- The COLHEADER= option specifies the column headers. Valid values are “Names”, “Labels”, and “None”. By default, COLHEADER=“Labels”.
- The JUSTIFY= option specifies the horizontal alignment (left, center, or right) for each column. By default, character columns are left-justified and numeric columns are right-justified.

The following program creates a table from the Sashelp.Class data set. The TablePrint subroutine options request that only a portion of the table be printed. The output is shown in Figure 9.9.

```
proc iml;
tClass = TableCreateFromDataSet("Sashelp", "Class");
call TablePrint(tClass) label="Subset of Students"
                        var={"Name" "Sex" "Age" "Weight"}
                        justify={'R' 'C' 'C' 'R'}
                        ID="Name"
                        firstObs=3 numobs=5;
```

Figure 9.9 A Printed Table

Subset of Students			
Name	**Sex**	**Age**	**Weight**
Barbara	F	13	98
Carol	F	14	102.5
Henry	M	14	102.5
James	M	12	83
Jane	F	12	84.5

The TablePrint subroutine supports advanced options that enable you to use custom templates to display tables. These advanced options are discussed in the next section.

Advanced Printing of Tables

You can use the TEMPLATE= option in the TablePrint subroutine to specify a custom template that specifies the layout for a SAS/IML table. This section describes how to use the TEMPLATE procedure to define a custom template. For an introduction to table templates, see Smith (2007, 2013).

When you use the TEMPLATE= option to specify a custom template, you can still use the VAR=, FIRSTOBS=, and NUMOBS= options. You can use the LABEL= option if the template includes the HEADER _LABEL_ statement. You cannot use the ID= or JUSTIFY= options with custom templates, nor can you use the COLHEADER="None" option.

The following options to the TablePrint subroutine enable you to specify templates and to pass values at run time to dynamic variables in the templates.

- The TEMPLATE= option specifies the name of a template that ODS should use to display the table.
- The DYNAMIC= option specifies the names of SAS/IML matrices whose values are passed to the template and used as the values of dynamic variables.
- The COLTEMPLATE= option specifies the names of column templates that are defined by using the DEFINE COLUMN statement in PROC TEMPLATE. The names should be known to the template that is specified by the TEMPLATE= option.
- The COLDYNAMIC= option specifies the names of SAS/IML matrices. The values of these matrices are passed to dynamic variables in column templates.

You must specify the TEMPLATE= option in order to specify the DYNAMIC= or COLTEMPLATE= options. You must specify the COLTEMPLATE= option in order to specify the COLDYNAMIC= option.

Using Custom Templates

This example defines a custom template and uses it to print a SAS/IML table. The following call to PROC TEMPLATE defines a table template called "Custom1." The table has three columns that have specified names. The table contains a main header and a secondary header that spans the second and third column. For more information about how to use PROC TEMPLATE to define a template, see *SAS Output Delivery System: Procedures Guide*.

```
proc template;
define table Custom1;
   column Estimate LowerCL UpperCL;     /* names and order of columns */
   define header topHeader;             /* header at top of table */
      text "Parameter Estimates";
   end;
   define header SpanHeader;            /* define spanning header */
      text "95% Confidence Limits";     /* title of spanning header */
      start = LowerCL;                  /* span starts at second column */
      end   = UpperCL;                  /* span ends at third column */
   end;
end;
run;
```

You can use the TEMPLATE= option to request that the "Custom1" template be used to print a SAS/IML table. The following SAS/IML statements create a table that has three columns. The names of the columns agree with the names that were specified in the template. The TablePrint subroutine displays the table. The output is shown in Figure 9.10.

```
proc iml;
x = {1   0.5 1.5,
     1.8 1.6 2.0,
     2.8 2.1 3.5};
ParamEst = TableCreate({"Estimate" "LowerCL" "UpperCL"}, x);
call TablePrint(ParamEst) template="Custom1";
quit;
```

Figure 9.10 Output That Uses a Custom Template

Parameter Estimates		
	95% Confidence Limits	
Estimate	LowerCL	UpperCL
1	0.5	1.5
1.8	1.6	2
2.8	2.1	3.5

Figure 9.10 shows that the table has a header that displays the text "Parameter Estimates." The table also has a spanning header that displays the text "95% Confidence Limits."

A second example of a custom template is a table that colors certain cells in a table according to the values in those cells.

For example, the Sashelp.Class data includes variables that indicate gender and weight. You might want to highlight males by using a light blue background and females by using a pink background, and you might want to highlight any child that weighs more than 100 pounds by using a light orange background. The following call to PROC TEMPLATE uses the CELLSTYLE statement to define a table with these attributes.

```
proc template;
define table CustomColor1;
   cellstyle _COL_ = 2 && _VAL_="M" as {backgroundcolor=LightBlue},
             _COL_ = 2 && _VAL_="F" as {backgroundcolor=Pink},
             _COL_ = 5 && _VAL_>100 as {backgroundcolor=LightOrange};
   end;
run;
```

The following call to the TablePrint subroutine uses this template to display the table. The output is displayed in Figure 9.11.

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class");
call TablePrint(tbl) numobs=6
                     template="CustomColor1";
quit;
```

Figure 9.11 Output That Colors Cells According to Their Values

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83

In Figure 9.11, the cells in the Sex column are colored pink or blue. Cells in the Weight column are colored orange if the student weighs more than 100 pounds. The table does not have a main header because the template does not define one.

Using Dynamic Variables in Custom Templates

Dynamic variables enable you to specify attributes of a table when the table is rendered, rather than when the template is stored.

For example, the previous section defined the CustomColor1 template, which used the colors pink, blue, and orange to highlight certain cells in a table. But suppose you want the flexibility to change the background colors in the CustomColor1 template. Although the DYNAMIC statement in PROC TEMPLATE supports default values, the following template defines three dynamic variables that are unspecified until run time:

```
proc template;
define table CustomColor2;
   dynamic MaleColor FemaleColor OverweightColor;
   cellstyle _COL_ = 2 && _VAL_="M" as {backgroundcolor=MaleColor},
             _COL_ = 2 && _VAL_="F" as {backgroundcolor=FemaleColor},
```

```
                _COL_ = 5 && _VAL_>100 as {backgroundcolor=OverweightColor};
end;
run;
```

You can use the DYNAMIC= option in the TablePrint subroutine to specify (at run time) values for the MaleColor, FemaleColor, and OverweightColor dynamic variables. The argument to the DYNAMIC= option in the TablePrint subroutine is a vector of keyword-value pairs. Each element of the vector specifies the name of a dynamic variable in a template, an equal sign, and the name of a SAS/IML symbol whose value will be used for the corresponding dynamic variable.

For example, the following SAS/IML program contains three symbols: M, F, and Wt. Each symbol contains a string that specifies a valid color. The dynamicVar variable is a vector that specifies the keyword-value pairs. The first element indicates that the value of the M symbol should be used for the MaleColor dynamic variable in the CustomColor2 template, the second element indicates that the value of the F symbol should be used for the FemaleColor dynamic variable, and so on. The output is shown in Figure 9.12.

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class");
M = "LightGreen";
F = "LightRed";
Wt = "LightGrey";
/* Syntax: DynamicVar1=Symbol1, DynamicVar2=Symbol2, ...  */
dynamicVar = {"MaleColor=M" "FemaleColor=F" "OverweightColor=Wt"};
call TablePrint(tbl) numobs=6
                     template="CustomColor2"
                     dynamic=dynamicVar;
```

Figure 9.12 Output from a Custom Template That Contains Dynamic Variables

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83

The DYNAMIC= option in the TablePrint subroutine also supports an alternate syntax. If the name of the SAS/IML symbol is the same as the name of a template dynamic variable, then you can omit the value part of a keyword-value pair. For example, the following program creates the same output as Figure 9.12.

```
MaleColor = "LightGreen";
FemaleColor = "LightRed";
OverweightColor = "LightGrey";
/* names of SAS/IML symbols = names of dynamic variables in template */
dynamicVar = {"MaleColor" "FemaleColor" "OverweightColor"};
call TablePrint(tbl) numobs=6
                     template="CustomColor2"
                     dynamic=dynamicVar;
```

Using Column Templates

The DEFINE COLUMN statement in PROC TEMPLATE enables you to specify the attributes of a column within a table. This column definition is known as the column template. You can use the COLTEMPLATE= option in the TablePrint subroutine to specify which column templates should be used to render particular columns of a table.

For example, the following statements define three column templates. Each template specifies the GENERIC=ON attribute, which means that you can use the column template for multiple columns.

```
proc template;
define table ColTempl1;
   header _LABEL_;              /* main header */
   define column Generic;
      generic=ON;               /* multiple variables can use */
   end;
   /* column for numeric variables */
   define column Float;
      generic=ON;               /* multiple variables can use */
      format=7.2;               /* display in 7.2 format */
      style=DataStrong;         /* boldface */
      cellstyle _COL_ as {color=Blue};  /* blue text */
   end;
   /* column for numeric integer variables */
   define column Integer;
      generic=ON;               /* multiple variables can use */
      format=6.;                /* display in 6. format */
      cellstyle _COL_ as {color=Red backgroundcolor=GrayEE};
      justify=ON; just=C;       /* center text */
   end;
end;
run;
```

The HEADER _LABEL_ statement enables you to set the table label by using the LABEL= option in the TablePrint subroutine.

The ColTempl1 template defines the following three columns:

- The Generic column is defined as a column that contains only one attribute. Character and numeric columns will be given default values for formats and justification.
- The Float column template uses the 7.2 format to displays values of numeric variables. The STYLE= statement causes text to be displayed in a bold typeface, and the CELLSTYLE= statement causes it to be displayed in a blue color.
- The Integer column template displays values for integer values. The values will be centered in the column, and the text will be displayed in a red color on a light gray background.

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class");
call TablePrint(tbl)
     label="Custom Column Templates"
```

```
numobs=6
template="ColTempl1"
var        ={"Name" "Height" "Weight" "Age"}
coltemplate={Generic Generic Float Integer};
```

Figure 9.13 Output from a Custom Template That Defines Column Templates

Custom Column Templates

Name	Height	Weight	Age
Alfred	69	112.50	14
Alice	56.5	84.00	13
Barbara	65.3	98.00	13
Carol	62.8	102.50	14
Henry	63.5	102.50	14
James	57.3	83.00	12

Using Dynamic Variables in Column Templates

You can define a column template that uses dynamic variables. Dynamic variables enable you to specify attributes of a table when the table is rendered, rather than when the template is defined.

For example, the previous section describes the ColTempl1 template, which defines column templates that use hardcoded formats and colors. In the following table template, dynamic variables are used for formats, styles, and colors:

```
proc template;
define table ColTempl2;
   header _LABEL_;
   /* column for character variables */
   define column Generic;
      generic=ON;          /* multiple variables can use */
   end;
   /* column for numeric variables */
   define column Float;
      dynamic ColFormat="7.2"
              ColStyle="DataStrong"
              ColTextColor="Blue";
      generic=ON;          /* multiple variables can use */
      format=ColFormat;
      style=ColStyle;
      cellstyle _COL_ as {color=ColTextColor};
   end;
   define column Integer;
      dynamic ColTextColor="Red"
              ColBGColor="GrayEE";
      generic=ON;          /* multiple variables can use */
      format=6.;
      cellstyle _COL_ as {color=ColTextColor backgroundcolor=ColBGColor};
      justify=ON; just=C; /* center text */
   end;
end;
run;
```

The default values for the dynamic variables are the same as the hardcoded values in the ColTempl1 template. Consequently, if you use the COLTEMPLATE=ColTempl2 option in the TablePrint subroutine, you will obtain the same table as shown in Figure 9.13. However, you can override the default formats, styles, and colors by using the COLDYNAMIC= option to specify values at run time.

For example, suppose you want to print columns of the Sashelp.Class data that contain the Name, Height, Weight, and Age variables. You want to use the Generic column template to display the first two variables, the Float column template for the third variable, and the Integer column template for the fourth variable. Furthermore, suppose you want the values in the Weight column to be displayed by using the 8.3 format and you want to change the colors for the Age column so that the table displays black text on a light pink background.

The following SAS/IML program defines SAS/IML symbols that contain the format and color values:

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class");
varNames     = {"Name" "Height" "Weight" "Age"};
colTemplates = {Generic Generic Float Integer};

Fmt="8.3";              /* specify one value for Float column template */
ColBGColor="LightPink"; /* specify two values for Integer column template */
ColTextColor="Black";
```

You can use the COLDYNAMIC= option to pass these values into the ColTempl2 template. Because you are going to print four columns, you must specify a four-element matrix to the COLDYNAMIC= option, as follows:

```
DynValues = {" "  " "             /* no dynamic vals for Generic columns */
             "ColFormat=Fmt"                          /* vals for Float   */
             "ColBGColor ColTextColor"};              /* vals for Integer */
call TablePrint(tbl)
     label="Column Templates with Dynamic Variables"
     numobs=6
     template="ColTempl2"
     var=varNames
     coltemplate=ColTemplates
     coldynamic=DynValues;
```

Figure 9.14 Output from Column Templates That Define Dynamic Variables

Column Templates with Dynamic Variables

Name	Height	Weight	Age
Alfred	69	112.500	14
Alice	56.5	84.000	13
Barbara	65.3	98.000	13
Carol	62.8	102.500	14
Henry	63.5	102.500	14
James	57.3	83.000	12

The *i*th element of the DynValues matrix specifies the values for the dynamic variables in the *i*th column template. For this example, the first two columns do not have any dynamic variables; therefore the first two elements of DynValues are blank strings. For the third column, the ColFormat dynamic variable in the Float

template is assigned the value of the Fmt variable. Consequently, the third column in Figure 9.14 displays three digits after the decimal point. For the fourth column, the ColBGColor and ColTextColor dynamic variables are assigned values from the SAS/IML variables of the same name. Consequently, the fourth column in Figure 9.14 contains black text on a light pink background.

Notice that the third element of the DynValues vector uses the "keyword=value" syntax, whereas the fourth element uses the alternate syntax. You can use whichever syntax is most convenient.

Using Existing SAS Templates

SAS software is shipped with hundreds of table templates. Every time you run a SAS procedure, it uses an ODS template to format and display the output. Previous sections show how you can define your own table templates. This section describes how to specify an existing template as input to the TablePrint subroutine.

For example, suppose you run the following statements to call the REG procedure:

```
ods trace on;
proc reg data=Sashelp.Class plots=none;
   model height=weight age / CLB;
   ods output ParameterEstimates=Parms;
   ods select ParameterEstimates;
quit;
ods trace off;
```

PROC REG produces the ParameterEstimates table, which is shown in Figure 9.15. The first column is a row header. The last two columns share a common spanning header that displays the text "95% Confidence Limits."

Figure 9.15 Parameter Estimates Table

The REG Procedure
Model: MODEL1
Dependent Variable: Height

Parameter Estimates

Variable	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|	95% Confidence Limits	
Intercept	1	32.19431	5.08227	6.33	<.0001	21.42039	42.96824
Weight	1	0.13805	0.03475	3.97	0.0011	0.06438	0.21171
Age	1	1.22667	0.53019	2.31	0.0343	0.10272	2.35062

Because of the ODS TRACE ON statement, ODS echoes information about SAS templates to the SAS log. The log shows that the template that PROC REG uses to display the parameter estimates table is **`Stat.REG.ParameterEstimates`**. You can view the template by using the SOURCE statement in PROC TEMPLATE, as follows:

```
proc template;
   source Stat.Reg.ParameterEstimates;
run;
```

The source of the template is written to the log. The template definition is long and complex. However, a careful inspection reveals the column names in the templates and also reveals that a dynamic variable named Confidence is used to form the spanning header.

The REG procedure saved the data in the ParameterEstimates table to a data set called Parms. The following statements read the data into a SAS/IML table and print the table by using the default template. The output is shown in Figure 9.16.

```
proc iml;
tbl = TableCreateFromDataSet("Parms");
call TablePrint(tbl) colheader="Names";
```

Figure 9.16 Output When Using the Standard Template

tbl

Obs	Model	Dependent	Variable	DF	Estimate	StdErr	tValue	Probt	LowerCL	UpperCL
1	MODEL1	Height	Intercept	1	32.19431	5.08227	6.33	<.0001	21.42039	42.96824
2	MODEL1	Height	Weight	1	0.13805	0.03475	3.97	0.0011	0.06438	0.21171
3	MODEL1	Height	Age	1	1.22667	0.53019	2.31	0.0343	0.10272	2.35062

Although the table displays information that is consistent with Figure 9.15, the SAS/IML table does not look as nice. You can use the TEMPLATE= option to request that the TablePrint subroutine use the **Stat.REG.ParameterEstimates** template to display the table.

The following statement uses the **Stat.REG.ParameterEstimates** template to display the table. In order to format the spanning header correctly, you must pass a value for the Confidence dynamic variable into the template. The result is shown in Figure 9.17.

```
Confidence=95;
call TablePrint(tbl) template="Stat.Reg.ParameterEstimates"
                     dynamic={Confidence};
```

Figure 9.17 Output When Using a SAS/STAT Template

Parameter Estimates

Variable	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|	95% Confidence Limits	
Intercept	1	32.19431	5.08227	6.33	<.0001	21.42039	42.96824
Weight	1	0.13805	0.03475	3.97	0.0011	0.06438	0.21171
Age	1	1.22667	0.53019	2.31	0.0343	0.10272	2.35062

Notice that Figure 9.15 and Figure 9.17 are identical. This shows that you can use templates for SAS/STAT procedures to display data from within SAS/IML programs.

References

Smith, K. D. (2007). "PROC TEMPLATE Tables from Scratch." In *Proceedings of the SAS Global Forum 2007 Conference*. Cary, NC: SAS Institute Inc. http://www2.sas.com/proceedings/forum2007/221-2007.pdf.

Smith, K. D. (2013). *PROC TEMPLATE Made Easy: A Guide for SAS Users*. Cary, NC: SAS Institute Inc.

Chapter 10
Lists and Data Structures

Contents

Overview **147**
Getting Started **149**
Create a Growable List of Matrices 149
Create a Growable List of Matrices by Using Syntax 151
Create a List of Items of Different Types 152
Create an Associative Array 153
Syntax **155**
Create and Delete Lists 155
Add, Set, or Delete Items in a List 155
Get Items from a List 156
Use Named Items in a List 156
Details **157**
List Syntax 157
Add Items to a List 159
Delete Items from a List 159
Extract Items from a List 159
Release List Memory 159
List Utilities: The ListUtil Package 159
Examples **160**
Create a List of Lists 160
Construct a Stack 161
Construct a Binary Search Tree 165

Overview

The matrix is the fundamental data type for SAS/IML computations. A matrix must contain either all numeric or all character values. You cannot create a matrix in which one column has numeric values and another column has character values.

However, beginning with SAS/IML 14.2, you can create lists. The objects in a list can be of different sizes and types. A list can contain numeric matrices, character matrices, tables, and other lists.

A SAS/IML list is similar to a *dynamic array*. A dynamic array (also called a *growable array* or a *mutable array*) is a random-access array that can grow and shrink. Elements in a dynamic array are directly accessed

by specifying their position (index). Because matrices are sometimes called arrays, the word "list" is used to describe the SAS/IML data structure that can contain other data structures.

Lists are a convenient way to store related data and to pass that data to modules, but they do not support arithmetic operations. For example, you cannot add or multiply two lists.

SAS/IML provides a rich set of functions for creating and working with lists. You can also use syntax for some list operations. For example, you can use square brackets to create a list (`L = ["Ron", 3, {100 94}];`), or you can use the ListCreate and ListAddItem functions to construct the same list. You can use the *list item operator* (`$`) to modify an existing item (`L$3 = {100 95}`) or you can use the ListSetItem subroutine.

You can use the ListAddItem and ListInsertItem subroutines to insert new items into a list. You can use the ListDeleteItem subroutine to remove items from a list.

You can use the list structure and the associated SAS/IML functions to emulate many different data structures, including the following:

- Associative arrays. An *associative array* (also called a *map* or a *dictionary*) is a set of key-value pairs. Elements in an associative array are accessed by specifying the key. In the SAS/IML language, you can use the ListSetName subroutine to assign names to some or all elements. You can then access the elements by name.
- Structs. A *struct* is a collection of named elements called *members*. The members can be *inhomogeneous*, which means they do not have to be the same type or size. For example, you could create a list that contains named fields for a person's name, address, telephone number, and salary. You can use the ListSetName subroutine to assign names to items.
- Stacks. A *stack* is a linear array in which objects can be inserted and removed only at the beginning of the array. A push operation adds an item to the front of the list; a pop operation removes the item at the front of the list. A stack obeys the last-in first-out principle (LIFO). You can access only the first element of a stack. In the SAS/IML language, you can use the ListGetItem function to implement the pop operation and use the ListInsertItem operation to implement the push operation.
- Queues. A *queue* is a linear array in which objects can be inserted at the end of the array and removed from the beginning of the array. A queue obeys the first-in first-out principle (FIFO) but is otherwise similar to a stack.
- Trees. A *tree* contains nodes and directed edges. A tree starts with a root node. The root node is connected via branches to other nodes, called child nodes. Every node except the root node has exactly one parent node. In SAS/IML, lists can contains sublists. For an example, see the section "Construct a Binary Search Tree" on page 165.

Getting Started

This section provides several examples of using lists.

Create a Growable List of Matrices

Suppose you want to store three matrices in a list. One way is to allocate a list that contains three items and use the ListSetItem subroutine to assign each item from a matrix. The following statements store matrices of different sizes in a list. The ListLen function is used to iterate over the items in the list.

```
proc iml;
/* create a list of matrices; use ListSetItem to fill */
L = ListCreate(3);             /* allocate list of 3 items */
do n = 1 to ListLen(L);        /* for each item in list */
   A = j(n, n, n-1);           /* define n x n matrix */
   call ListSetItem(L, n, A); /* assign n_th item of L */
end;
```

Notice the order of the arguments to the list functions. For most functions, the first argument is the name of the list object, and the second argument specifies positions or names. This is the same order that you would use to specify the element of a matrix: M[i]. In particular, the syntax to assign a matrix element is reminiscent of the syntax M[i]=b.

The list L contains three items. You can use the ListGetItem function to extract each item into a SAS/IML matrix, as shown by the following statements:

```
sum = j(ListLen(L), 1);
do n = 1 to ListLen(L);        /* for each item in list */
   B = ListGetItem(L, n);      /* get n_th matrix of L */
   sum[n] = sum(B);            /* compute sum of elements */
end;
```

The vector sum contains the sums of the values in the three matrices.

You can use the ListAddItem subroutine to add more items to the list. The list will automatically grow to accommodate the new items. The new items are added to the end of the list, as shown in the following statements:

```
C = 1:3;
call ListAddItem(L, C);     /* add C as 4th item to L */
D = {4 3, 2 1};
call ListAddItem(L, D);     /* add 5th item to L */
```

If you want to view the contents of the list, you can use the ListPrint or the Struct subroutines in the ListUtil package, as described in the section "List Utilities: The ListUtil Package" on page 159. For example, the following statements load the package and display the structure of the list, which has five items:

```
package load ListUtil;
run struct(L);
```

Figure 10.1 Structure of a List

					L				
Name	**Level**	**NRow**	**NCol**	**Type**	**Value1**	**Value2**	**Value3**	**Value4**	**More**
L	0	.	5	List	L[1]	L[2]	L[3]	L[4]	...
=> L[1]	1	1	1	Num	0				
=> L[2]	1	2	2	Num	1	1	1	1	
=> L[3]	1	3	3	Num	2	2	2	2	...
=> L[4]	1	1	3	Num	1	2	3		
=> L[5]	1	2	2	Num	4	3	2	1	

Figure 10.1 shows the structure of the list. The first row provides information about the list `L`, including that the list contains five unnamed items. Subsequent rows provide information about each item, including the dimensions of the matrix, the type of the matrix, and the first few matrix values.

The meaning of each column is as follows:

- The **Name** column provides the name of an object or a subobject. Unnamed items are named by using subscripts. For example, `L[1]` is the first element of a list `L`. By default, the string "==>" is used to indicate subobjects that are contained in a higher-level object.
- The **Level** column contains the value 0, 1, or 2. The value 0 means that the row is a top-level object that you can directly reference by using a SAS/IML symbol. The value 1 indicates a subobject, such as an element of a list or a variable in a table. The value 2 indicates a subobject of a subobject.
- The **NRow** column indicates the number of rows for a matrix or a table. For a list, this column has a missing value.
- The **NCol** column indicates the number of columns for a matrix or a table. For a list, this value is the number of items.
- The **Type** column indicates the type of each object. Valid values are List, Table, Num (for a numeric matrix), Char (for a character matrix), or Empty (for an undefined matrix).
- The **Value1** through **Value4** columns display the values of the first few elements of a matrix. For a table, they display the names of the first few variables. For a list, they display the names of the first few items.
- The **More** column is an indicator variable. The column has two possible values: blank indicates that an object has fewer than four elements, and the symbol "..." indicates that there are more than four elements in a matrix or list, or more than four columns in a table.

Not only can you add and modify items, but you can also remove items from a list. The following statements use the ListDeleteItem subroutine to remove the first, third, and fifth items, thus leaving only two items:

```
call ListDeleteItem(L, {1 3 5}); /* remove three items */
```

Create a Growable List of Matrices by Using Syntax

This section repeats the example in the previous section, but it uses list syntax instead of calls to functions and subroutines.

Suppose you want to store three matrices in a list. One way is to use the list creation operator (`[]`) to specify the items in the list. The following statement creates a list that contains three matrices of different sizes:

```
proc iml;
ones = {1 1, 1 1};
L = [0,  ones,  j(3,3,2)];    /* create a list of matrices */
```

The items in the list are separated by commas. You can use matrix literals, variable names, expressions (such as `1:3`), and function calls to initialize the items in a list.

You can create the same list by allocating a three-item list and then using the list item operator (`$`) on the left side of the assignment operator. The expression `L$1` indicates the item in the first position of a list. Similarly, `L$2` and `L$3` indicate the items in the second and third positions, respectively. In addition to literal position numbers, you can use a scalar expression as shown in the following example, which demonstrates how to use a loop to assign the items in a list:

```
L = ListCreate(3);            /* allocate list of 3 items */
do n = 1 to ListLen(L);       /* for each item in list */
   L$n = j(n, n, n-1);        /* assign n x n matrix */
end;
```

The list `L` contains three items. You can use the list item operator on the right side of an assignment operator to extract each item, as shown by the following statements:

```
sum = j(ListLen(L), 1);
do n = 1 to ListLen(L);       /* for each item in list */
   sum[n] = sum(L$n);         /* compute sum of elements of n_th item */
end;
```

To add multiple items to a list, you can use the list concatenation operator (||). The concatenation operator acts on lists, so you have to create a list that contains the items that you want to add, as follows:

```
C = 1:3;
D = {4 3, 2 1};
L = L || [C, D];              /* concatenate lists */
```

Not only can you add and modify items, but you can also remove items from a list by using the sublist operator (`[]`). The sublist operator (which looks like the subscript operator for matrices) always returns a sublist. The following statements use the sublist operator to keep only the second and fourth items of the list:

```
L = L[ {2 4} ];               /* extract sublist */
```

You can also use the sublist operator on the left side of an assignment operator if the right side of the assignment is a list that has the same number of items.

Create a List of Items of Different Types

The items of a list can be matrices, tables, or other lists. A list can store inhomogeneous types. The following SAS/IML statements create four different types of variables: a numeric matrix, a character vector, a table, and a list that contains three items.

```
proc iml;
M = {1 2, 3 4};
C = "A":"G";
sublist = ListCreate(3);
do i = 1 to ListLen(sublist);
   call ListSetItem(sublist, i, j(i, i, i##2));
end;
```

You can create a second list that contains a copy of the previous list as a sublist. The following statements create the new list:

```
list = ListCreate();
call ListAddItem(list, M);       /* add numeric matrix   */
call ListAddItem(list, C);       /* add character vector */
call ListAddItem(list, sublist); /* add sublist          */
```

If you want to print the items in the list, you can use the ListPrint subroutine. The ListPrint subroutine is included in the ListUtil package, which is described in the section "List Utilities: The ListUtil Package" on page 159. The results of the ListPrint call are shown in Figure 10.2.

```
package load ListUtil;           /* load ListPrint module */
run ListPrint(list);
```

Figure 10.2 Printing a List

--------- List = list---------

Item 1
1 2
3 4

Item 2
A B C D E F G

Item 3
[LIST]

Item 3[1]
1

Item 3[2]
4 4
4 4

Figure 10.2 *continued*

Item 3[3]
9 9 9
9 9 9
9 9 9

The first output is a label that provides the name of the symbol being printed. Then each item in the list is printed. The first item (labeled "Item 1") is a numeric matrix. The second item (labeled "Item 2") is a character vector. The third item (labeled "Item 3") is a list that contains three items, each of which is a numeric matrix. The matrices are printed with the labels "Item 3[1]," "Item 3[2]," and "Item 3[2]."

As an alternative to using the ListCreate and ListAddItem functions, you can also create a three-item list by using the list creation operator (`[]`):

```
list = [M, C, sublist];
```

Create an Associative Array

This example shows how to create a list of named items, which is sometimes called an *associative array*. Some of the items are numeric vectors; others are character vectors. The items can have different lengths. Associative arrays are useful because they enable you to create a single object that contains related data elements.

As an example, suppose you want to create a list that contains information about a linear regression. The following call to the REG procedure creates two output data sets:

- The PE data set contains information about parameter estimates, standard errors, and *p*-values.
- The Out data set contains original data and the predicted values and residual values for each observation.

```
proc reg data=Sashelp.Class plots=none;
   where sex="M";
   model weight = height;
   output out=Out p=Pred r=Res;
   ods output ParameterEstimates=PE;
quit;
```

The PE data set has two observations, whereas the Out data set has 10 observations. The following SAS/IML statements create vectors from variables in these data sets:

```
proc iml;
use PE; read all var {"Variable" "Estimate"}; close;
use Out; read all var {"Weight" "Pred" "Res"}; close;
```

The following statements create a new list that contains five named items. Data from the PE data set are copied into two items. Data from the Out data set are copied into three items.

```
StructNames = {"Variable" "Estimate" "Dep Var" "Predicted" "Residual"};
RegModel = ListCreate( StructNames );
call ListSetItem(RegModel, "Variable", Variable);
call ListSetItem(RegModel, "Estimate", Estimate);
call ListSetItem(RegModel, "Dep Var", Weight);
```

```
call ListSetItem(RegModel, "Predicted", Pred);
call ListSetItem(RegModel, "Residual", Res);
```

Equivalently, you can use the list creation operator (`[]`) to create a named list. Specify the name of an item by prefixing it with the '#' symbol. You can specify the name as a character literal or as a quoted string. For example, the following statement is an alternative way to specify a named list that contains five items. Notice that quotes are optional except for names that contain blanks or special characters:

```
RegModel = [#"Variable"=Variable, #Estimate=Estimate,
            #"Dep Var"=Weight, #Predicted=Pred, #Residual=Res];
```

If you want to print the items in the RegModel list, you can use the ListPrint subroutine. The ListPrint subroutine is included in the ListUtil package, which is described in the section "List Utilities: The ListUtil Package" on page 159. The results are shown in Figure 10.3.

```
package load ListUtil;                          /* load ListPrint module */
run ListPrint(RegModel);
```

Figure 10.3 Printing an Associative Array

-------- List = RegModel--------

Item 1 : "Variable"
Intercept Height

Item 2 : "Estimate"
-141.101 3.9125492

Item 3 : "Dep Var"									
112.5	102.5	83	84	99.5	150	128	133	85	112

Item 4 : "Predicted"									
128.86488	107.34585	83.08805	103.43331	89.739384	140.60252	112.43217	121.03978	83.87056	119.0835

Item 5 : "Residual"									
-16.36488	-4.845855	-0.08805	-19.43331	9.7606164	9.3974773	15.567831	11.960223	1.1294401	-7.083502

The output shows the five items in the list. Two items are 1×2 matrices, and three items are 1×10 matrices.

You can use a list to pass multiple objects to a module. For example, the following module creates a plot of the observed response versus the predicted response. Provided that you pass in a list object that contains fields named "Dep Var" and "Predicted", this module can handle a wide variety of regression models. The output of this module is not shown.

```
/* Module that creates a plot of observed vs predicted response.
   Pass in a list that contains items named "Dep Var"and "Predicted" */
start PredPlot(L);
   Observed = ListGetItem(L, "Dep Var");      /* = L$"Dep Var"  */
   Predicted = ListGetItem(L, "Predicted"); /* = L$"Predicted" */
   call Scatter(Observed, Predicted) procopt="noautolegend"
                    other="lineparm x=0 y=0 slope=1 / clip";
finish;
```

```
run PredPlot(RegModel);
```

As shown in the comments in the module, you can also use the sublist operator (`$`) to get the items that are named “Dep Var” and “Predicted.”

Syntax

Chapter 25, “Language Reference,” describes the syntax of the functions and subroutines that create and manipulate lists. You can group the functions into the following categories:

- Functions that create or destroy a list
- Functions that add, set, or delete items in a list
- Functions that get items from a list
- Functions related to named items

Create and Delete Lists

- The ListCreate subroutine creates a new list. You can create an empty list and use the ListAddItem subroutine to add items to the list. Alternatively, you can create a list that contains a specified number of initially empty items, then use the ListSetItem subroutines to set the value for each item.
- The FREE statement frees the symbol and memory associated with a list.
- The ListLen function returns the number of items in a list. Uninitialized items and items that have an empty value are included in the count.
- The list creation operator (`[]`) creates and initializes a list.

Add, Set, or Delete Items in a List

- The ListAddItem subroutine adds a new item to the end of a list. The length of the list increases by one.
- The ListSetItem subroutine sets or replaces the value of an existing list item. The length of the list stays the same.
- The ListDeleteItem subroutine deletes an item from a list. The length of the list decreases by one.
- The ListInsertItem subroutine inserts an item at a specified position in the list. The length of the list increases by one.

- The ListSetSubItem sets the value of an item in a nested sublist. The length of the list stays the same.
- The list concatenation operator (||) concatenates two lists.
- The list item operator (`$`) can be used on the left side of an assignment operator to set a particular item. For example, `L$1 = 2:4`.
- The sublist operator (`[]`) returns a list that contains a set of specified items. For example, `L = L[2:4]`.
- When used on the left side of an assignment operator, the sublist operator (`[]`) sets multiple items of a list from another list. For example, `L[{2 3 5}]=[10, "Q", 1:3]` sets the second item to 10, the third item to "A", and the fifth item to a numerical matrix.

Get Items from a List

- The ListGetItem function returns a list item.
- The ListGetSubItem function returns an item from a nested sublist.
- The list item operator (`$`) can be used on the right side of an assignment operator to get a particular item. For example, `A = L$1`. You can use the operator several times to get an item from a nested sublist. For example, `B = L$1$2` gets the second item in the first item, which must be a sublist.

Use Named Items in a List

- The ListDeleteName subroutine removes a name from an item, but it does not remove the item itself.
- The ListGetAllNames function returns the complete set of names for items in a list.
- The ListGetName function returns the names that are associated with specific items in a list.
- The ListIndex function returns the positions of named items in the list.
- The ListSetName subroutine sets the name of an item.
- The list item operator (`$`) can get or set an item. When used on the left side of an assignment operator, it sets the item: `L$"A" = 1`. When used on the right side of an assignment operator, it gets the item: `X = L$"A"`.
- The sublist operator (`[]`) gets or sets multiple items of a list from another list. For example, the statement `L2 = L[{"A" "B" "C"}]` creates a list that contains a copy of three named items in `L`. The statement `L[{"A" "B" "C"}]=[10, "Q", 1:3]` sets the values of three named items.

Details

This section provides additional information about creating and modifying lists.

List Syntax

This section provides an overview of the syntax that creates and manipulates lists. There are three special list operators:

- The list creation operator (`[]`) creates a list. When you create a list, you can optionally use the '#' symbol to name items in the list.
- The list item operator (`$`) refers to an item in a list or sublist.
- The list sublist operator (`[]`) refers to a sublist of a list.

In addition, you can use the horizontal concatenation operator (||) to concatenate lists. If `L1` and `L2` are two lists, then `L3 = L1 || L2` creates the list that contains all items of `L1` followed by all items of `L2`.

Create a List

You can use the list creation operator (`[]`) to create a list. The following syntax creates a three-item list:

```
proc iml;
L = [ 1:4,                  /* 1st item is a numeric vector */
      {"A" "B" "C"},        /* 2nd item is a character vector */
      [100, "Q", {3 6 2}]   /* 3rd item is a list */
    ];
```

When you create a list, you can use the '#' symbol to specify a name for one or more items. The '#' symbol must be followed by a character literal or a quoted string. Quotation marks are optional except for names that contain blanks or special characters.

The following example creates a list that has five items. Four items are named; the third and fourth items are empty. Names are not case sensitive.

```
S = [#"first"=1, #"Second Item"=2,
     #"Empty Item"= ,  , #"Fifth"=[0, "Z"] ];
```

Get and Set List Items

The list item operator (`$`) enables you to set or retrieve the value of an item. The following statements continue from the example in the previous section. The statements set and retrieve values of the list `L`:

```
L$1 = {3 4 5 2 2 1};        /* set 1st item in list */
i = 2;
A = L$i;                    /* get 2nd item in list */
```

Notice that the symbol `i` resolves to the value 2. You can also refer to items by their names by using a character vector or a character literal, as follows:

```
f = "FIRST";
S$f = -1;                    /* set item named "FIRST" */
B = S$"Second Item";         /* get item named "Second Item" */
```

In the previous example, the expression **L$i** resolves to **L$2**, and the expression **S$f** resolves to **S$"FIRST"**. Notice that **S$first** would not be a valid expression. Because there are no quotation marks, the $ operator will look for a symbol named **first** and report an error if the symbol is not defined. Quotation marks are required when you use the list item operator to specify a literal name, such as **S$"first"**.

You can use the $ operator multiple times to access elements of a sublist. For example, the following statements are valid:

```
C = L$3$2;                   /* get 2nd item of the 3rd item */
S$"fifth"$1 = 10;            /* set 1st item of the item named "fifth" */
```

You can also access subitems by using vectors of indices, vectors of names, or lists that combine names and indices. The following statements are equivalent to the previous statements:

```
C = L${3 2};                 /* get 2nd item of the 3rd item */
H = ["fifth", 1];
S$H = 10;                    /* set 1st item of the item named "fifth" */
```

If you use a literal list to specify an item, you must enclose the list in parentheses, as follows:

```
/* parentheses are required */
S$(["fifth", 1]) = 10;       /* S$["fifth", 1] is invalid syntax */
```

In general, you can use any expression that is enclosed in parentheses and that resolves to a valid matrix or list.

Get and Set Sublists

The sublist operator (**[]**) enables you to set or retrieve a sublist. The argument to the sublist operator is a vector of indices or a vector of names. When used on the right side of an assignment operator, the sublist operator returns a sublist, as shown by the following statements:

```
V = L[1:2];                  /* list with two items: [L$1, L$2] */
H = ["fifth", 1];
U = S[ H ];                  /* list with two items: [S$"fifth", S$1] */
```

The sublist operator always returns a list. For example, **L[1]** is always a list that contains one item. This is in contrast to **L$1**, which is the first item in **L**. Notice also the difference between the expressions **S$H** and **S[H]**. The first expression returns a single item, so **S$H** is the same as **S$"fifth"$1**. In contrast, **S[H]** is the list that contains the two items **S$"fifth** and **S$1**.

You can use the sublist operator on the left side of an assignment operator to assign new items to a sublist. The right side of the assignment operator must be a list that contains the same number of items, as shown by the following statements:

```
L[1:2] = ["A":"C", 1:3];       /* assign L$1 and L$2 */
H = ["fifth", 1];
S[ H ] = [{-1 1}, {X Y Z}];    /* assign S$"fifth" and S$1 */
```

Add Items to a List

SAS/IML lists are essentially arrays that can shrink and grow. You can use the ListAddItem subroutine to add a new item to the end of a list.

You can use the ListInsertItem subroutine to add a new item to an existing list. If the list has k items, you can insert a new item in positions $1, 2, \ldots, k+1$. If you insert an item at position i where $i \leq k$, then the existing items at positions $i, i+1, \ldots, k$ are shifted to the right to make room for the new item. In other words, the new item is inserted prior to the existing item in the specified position, and the length of the list increases by one.

It is more efficient to add a new item to the end of a list than to insert the item at the beginning or in the middle of a list. Inserting an item at position $k+1$ is equivalent to adding an item to the end of the list.

Delete Items from a List

You can use the ListDeleteItem subroutine to delete items from a list. If a list has k items and you delete m items, then the new length of the list is $k-m$.

Extract Items from a List

You can use the ListGetItem or ListGetSubItem function to extract one item from a list. These functions support options that control whether the extracted item is copied, moved, or deleted from the list. If the item is copied or moved, the length of the list does not change. If the item is deleted, then the length of the list or sublist decreases by one.

You can use the list item operator (**$**) to copy an item from a list. For example, **`A = L$1`**.

Release List Memory

If you no longer need a list, you can use the FREE statement to delete the list and all its items. The FREE statement also frees all memory that was used to store the list; there is no need to use the ListDeleteItem subroutine prior to running the FREE statement.

List Utilities: The ListUtil Package

SAS/IML software includes the ListUtil package, which is a system package that contains useful functions for working with lists. The ListUtil package is pre-installed, so the following statements load the package and print a short overview of the package in the SAS log:

```
proc iml;
package load ListUtil;
package help ListUtil;   /* display overview in SAS log */
quit;
```

For information about how to use packages, see Chapter 11, "Packages." Several examples in this chapter use the following modules from the ListUtil package:

- The ListPrint subroutine prints items in a list. If a list contains sublists, items of sublists are also printed.
- The Struct subroutine prints a table that shows the structure of a SAS/IML object. The Struct subroutine is especially useful for showing the structure of a list.

Examples

This section provides examples of creating complex list structures, including a list of lists, a stack structure, and a binary search tree.

Create a List of Lists

The Sashelp.Class data set contains data about 19 students. The data include each student's age and gender.

If you sort the data by age and gender, you have created a hierarchical structure. There are six ages (11 through 16) and two genders ("F" for females and "M" for males). You could represent these data as a list of named lists. The main list has six items, which are named "11" through "16". Each item in turn contains a two-item sublist. The sublists contain a vector of male names and a vector of female names. A vector can be empty if a particular age does not contain any students for a particular gender.

For example, in the list for 12-year-old students, the "F" item is the vector {Jane, Louise} and the "M" item is the vector {James, John, Robert}.

The following code builds a list of lists that contains the data. The outer list, `L`, contains the unique values of the ages. Each item of `L` is a two-item list that contains two vectors.

```
proc iml;
use Sashelp.Class;           /* read data */
read all var {Age Sex Name};
close Sashelp.Class;

Age = char(Age, 2);          /* convert to character vector */
ages = unique(Age);
L = ListCreate(ages);        /* outer list: items named "11":"16" */

gender = unique(Sex);
K = ListCreate(gender);      /* inner list: items named {"F" "M"} */

do i = 1 to ncol(ages);      /* For each age level... */
   idx = loc(Age=ages[i]);   /* Find observations for this age */
   do j = 1 to ncol(gender);              /* for each gender... */
      jdx = loc( Sex[idx]=gender[j] );    /* find this age and gender */
      if ncol(jdx)=0 then students = {};  /* no students found */
```

```
      else students = Name[idx[jdx]];     /* get student names */
      call ListSetItem(K, gender[j], students);  /* value of inner list */
   end;
   call ListSetItem(L, ages[i], K);  /* set sublist as value */
end;
```

Notice that the `K` sublist is used multiple times. For each age, the `K` list is updated in place. The ListSetItem subroutine copies the contents of the `K` list into the appropriate item of the `L` list.

Construct a Stack

A stack is linear array in which objects can be inserted and removed only at the beginning of the array. A push operation adds an item to the front of the list; a pop operation removes the item at the front of the list. A stack obeys the last-in first-out (LIFO) principle. You can access only the first element of a stack.

In the SAS/IML language, you can use the ListInsertItem operation to implement the push operation and use the ListGetItem function to implement the pop operation. For efficiency, this implementation adds and removes new items from the *end* of a list, so that the positions of other elements are unchanged. This prevents unnecessary copying of data.

The following SAS/IML modules define other useful operations on stacks:

- The StackCreate function creates and returns a list. When called with zero arguments, the function returns an empty stack. When called with one argument, it returns a stack that contains one element.
- The StackPush subroutine pushes an item onto an existing stack.
- The StackPop function returns the item on the top of the stack and removes the item from the stack.
- The StackPeek function returns the item on the top of the stack but does not change the stack.
- The StackIsEmpty function returns 1 (true) if the stack is empty and 0 (false) if the stack contains at least one element.

The modules are defined as follows. Each module is a thin wrapper around calls that manipulate lists.

```
proc iml;
/* implement a stack, which is a 1-D LIFO structure */
start StackCreate( item= );
   S = [];                        /* create empty list   */
   if ^IsSkipped(item) then       /* if item specified,  */
      call ListAddItem(S, item);  /*    add item to list */
   return S;
finish;

/* push an item onto the stack */
start StackPush(S, item);
   call ListAddItem(S, item);     /* add item to the end */
finish;

/* pop an item from the stack */
```

```
start StackPop(S);
   A = ListGetItem(S, ListLen(S), 'd'); /* get & remove last item */
   return A;
finish;

/* peek at the item at the top of the stack without removing it */
start StackPeek(S);
   A = ListGetItem(S, ListLen(S), 'c'); /* get last item */
   return A;
finish;

/* return 1 if stack is empty; 0 otherwise */
start StackIsEmpty(S);
   return (ListLen(S) = 0);
finish;

/* return number of elements in stack */
start StackLen(S);
   return ListLen(S);
finish;

store module=_all_;
quit;
```

In the same way, you could define helper functions for a queue, which is a one-dimensional first-in first-out (FIFO) data structure.

If you want to use the Stack or Queue functions, you can use the following statements to define the modules:

```
%include sampsrc(LstStack.sas);
%include sampsrc(LstQueue.sas);
proc iml;
load module = _all_;
/* use modules for stacks and queues */
quit;
```

You can specify the SOURCE option in the %INCLUDE statement to display the contents of the program file in the SAS log.

Reverse the Words in a Sentence

By using the helper functions in the previous section, you can use SAS/IML lists to simulate stacks. One of the simplest applications of stacks is to reverse the contents of a list. The following example loads the helper functions and uses the StackPush subroutine to create a list of words. The StackPop subroutine then retrieves the list elements. Because the elements are retrieved in LIFO order, the sequence of words is reversed.

```
proc iml;
load module=(StackCreate StackPush StackPop
             StackPeek StackIsEmpty StackLen);

/* Create sentence. Break into vector of words. */
str = "Now is the time for all good men to come to the aid of their party.";
n = countw(str, " .");          /* use blanks and period as delimiters */
words = scan(str, 1:n, " .");  /* character vector */
```

```
S = StackCreate();              /* create an empty stack */
do i = 1 to ncol(words);
   run StackPush(S, words[i]); /* push each element onto the stack */
end;

print (StackPeek(S))[L="Top of Stack"]; /* the last word is on top */

/* retrieve the data in reverse order */
w = j(1, StackLen(S), "     ");
do i = 1 to ncol(w);            /* pop each element; insert into stack */
   w[i] = StackPop(S);
end;
print w[L="Reversed Words"];

if StackIsEmpty(S) then
   print "Stack is empty";
else print (StackPeek(S))[L="Top of Stack"];
```

Figure 10.4 Using a Stack to Reverse Words

Top of Stack
party

Reversed Words
party their of aid the to come to men good all for time the is Now

Stack is empty

The output from the program is shown in Figure 10.4. The program illustrates pushing elements onto a stack and popping elements off the stack. In practice there are simpler ways to reverse the words in a sentence, such as the matrix expression **`words[,ncol(words):1]`**.

Implement a Postfix Calculator

Another elementary application of stacks is to implement a postfix calculator. Postfix notation (sometimes called reverse Polish notation in honor of its inventor, the logician Jan Lukasiewicz) is a way of writing arithmetic expressions in a way that does not require parentheses. In postfix notation, the operands (numbers) appear before the operators that act on them. This section implements a postfix expression calculator that can perform the following binary operations: addition, subtraction, multiplication, and division.

In postfix notation, an expression is read from right to left. Each binary operator operates on the two numbers that precede the operator. Thus the usual (infix) expression '3 + 4' is represented in postfix notation as '3 4 +'. The infix expression '2*(3+4)' is equivalent to the postfix expression '2 3 4 + *'.

The following SAS/IML program loads the helper functions for working with stacks. The program then defines two modules. The BinaryCalc function takes a binary operation and two numbers and returns the result of the operation. The PostfixCalc function takes a space-separated string that represents an expression in postfix notation. The function splits the string into tokens and then uses stacks to compute the result of the expression. The module assumes that the postfix expression is valid.

```
proc iml;
load module=(StackCreate StackPush StackPop);

/* Given a binary operator, return  the expression
  (L op R) where op is in the set {+, -, *, /} */
start BinaryCalc(operator, L, R);
   if      operator="+" then return L + R;
   else if operator="-" then return L - R;
   else if operator="*" then return L * R;
   else if operator="/" then return L / R;
   else return .;
finish;

/* Input a space-separated string that represents a valid
   arithmetic operation in postfix notation. The string
   can contain numbers and the binary operators {+, -, *, /}.
   The string must represent a valid operation; no error
   checking is performed. */
start PostfixCalc(str);
   n = countw(str, " ");
   tokens = scan(str, 1:n, " ");          /* character vector */
   binOps = {"+","-","*","/"};            /* four binary operators */
   S = StackCreate();                     /* create an empty stack */
   do i = 1 to ncol(tokens);
      token = tokens[i];                  /* get the token */
      if element(token, binOps) then do; /* it is a binary op */
         R = StackPop(S);                 /* retrieve the previous */
         L = StackPop(S);                 /*    two numbers        */
         result = BinaryCalc(token, num(L), num(R));
         run StackPush(S, char(result)); /* push result on stack */
      end;
      else
         run StackPush(S, token);         /* push number on stack */
   end;
   return num(StackPop(S));               /* return result as number */
finish;

/* examples of parsing postfix expressions */
str = {"2 2.8 7.2 + * 5 /",          /* 2*(2.8+7.2) / 5  =   4 */
       "4 5 7 2 + - *",              /* 4*(5 - (7+2))    = -16 */
       "4 -5 + 6 -2 -  *",           /* (4 + -5)*(6 - -2)=  -8 */
       "2 2 2 2 * * *"    };         /*  2**4            =  16 */
result = j(nrow(str), 1);
do i = 1 to nrow(str);
   result[i] = PostfixCalc(str[i]);
end;
print str result;
```

The output of the program in shown in Figure 10.5, which displays the result of parsing four postfix expressions. The comments in the program show the equivalent infix notation for each expression.

Figure 10.5 Using a Stack to Evaluate Postfix Expressions

str	result
2 2.8 7.2 + * 5 /	4
4 5 7 2 + - *	-16
4 -5 + 6 -2 - *	-8
2 2 2 2 * * *	16

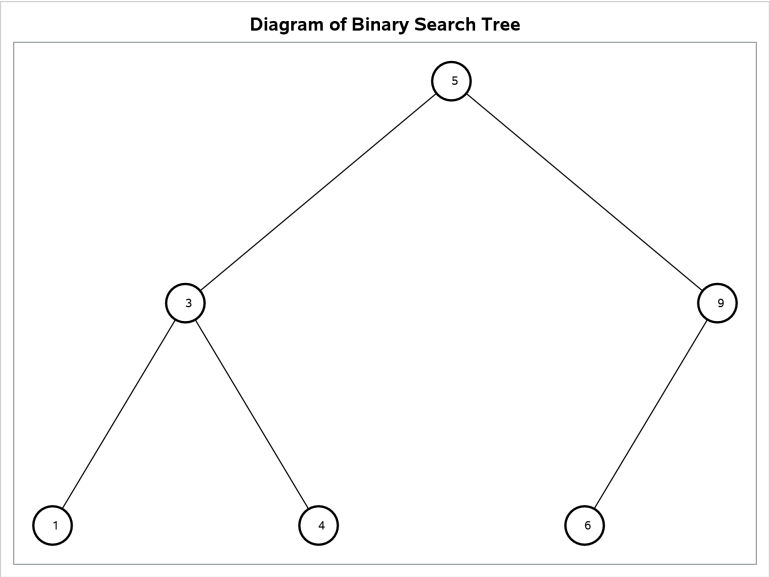
Construct a Binary Search Tree

A *tree* contains nodes and edges and starts with a root node. The root node is connected via branches to other nodes, called child nodes. Every node except the root node has exactly one parent node. A tree has no cycles: any two nodes can be connected by a unique path through the tree.

A binary tree is a tree in which every node has at most two child nodes. A binary search tree (BST) is a binary tree in which each node has a value (called a key), a link to a left child node, and a link to a right child node. Either or both child nodes might be null. By starting at the root node, you can quickly determine whether a value is in the tree. If not, you can insert the value into the tree by modifying one of the null left or right child nodes of an existing node.

Figure 10.6 shows the binary search tree that corresponds to the integer sequence {5, 3, 1, 9, 6, 4}.

Figure 10.6 Visualization of a Binary Search Tree



In terms of data structures, each node is a list that contains three items: the first item is the key value, and the second and third items are the child nodes. The child nodes are initially empty, but values can be assigned to them as the tree grows. In terms of SAS/IML lists, the following statements define a node in a BST. The call to PROC FORMAT defines a format that will be useful later in this section.

```
/* L[i] is key value, L[2] is left child, L[3] is right child */
proc format;
    value BSTFmt  1='Key'  2='Left'  3='Right';
run;

proc iml;
/* A node is a three-item list:
   node[1] contains the KEY   value
   node[2] contains the LEFT  value (or empty if null)
   node[3] contains the RIGHT value (or empty if null) */
start BSTNewNode(value);
   node = ListCreate(3);     /* create list with 3 null items */
   call ListSetItem(node, 1, value);   /* set KEY value */
   return node;
finish;
```

The following algorithm searches a binary tree to determine whether a target value is in the tree:

1. Set the current node to be the root node of the tree.
2. If the target value equals the key value for the current node, then the target value is found. Return information about the path to the node that contains the target value.
3. If the target value is less than the key value for a node, make the current node the left child. Otherwise, make the current node the right child.
4. If the current node is null, the target value is not in the tree. Return the path to the current node. You can create and insert a new node at that location to add the target value to the tree.
5. Go to Step 2.

In particular, the following SAS/IML function uses lists to look up a value in a binary search tree:

```
/* Search for a target value in a binary search tree.
   Input: root is the root node of a BST,
          value is the target value.
   Output: path contains the path to the node that contains the target
          value or the node where the target value can be inserted.
   Return: 1 if the target value is in the tree; 0 otherwise */
start BSTLookup(path, root, value);
   KEY = 1; LEFT = 2; RIGHT = 3;
   path = {};
   T = root;
   do while (1);
      if value = T$KEY then
         return 1;                  /* found it: return path to subitem */
      else if value < T$KEY then do;
         path = path || LEFT;       /* add to path */
         T = T$LEFT;                /* new root is left child */
      end;
      else do;
         path = path || RIGHT;      /* add to path */
         T = T$RIGHT;               /* new root is right child */
```

```
      end;
      if type(T)='U' then
         return 0;                    /* not found: return path to subitem */
   end;
finish;
```

For example, if the target value is 6 and you run the algorithm on the binary tree in Figure 10.6, the function does the following:

- The target value is greater than the root value (5), so the right child of the "5" node becomes the current node.
- The target value is less than the key value 9, so the left child of the "9" node becomes the current node.
- The target value equals the key value, so the target value has been found. Return the row vector {3, 3}, which represents unformatted values for the path {Right, Right}.

The previous BSTLookup function assumes that the binary search tree already exists. You can write additional functions to create a BST. The following statements define two functions:

- The BSTCreate function takes a vector of values and calls the BSTNewNode and BSTInsert functions to create a BST.
- The BSTInsert function takes one item and inserts it into the BST.

```
/* pass a vector of key values to this routine to create a BST
   that has those values as keys */
start BSTCreate(x);
   bst =  BSTNewNode( x[1] );
   do i = 2 to nrow(colvec(x));
      run BSTInsert(bst, x[i]);
   end;
   return bst;
finish;

/* Insert a new branch for a key value in a BST. If the value
   already exists, do nothing (so there are never duplicates) */
start BSTInsert(root, value);
   if ListLen(root)=0 then do;   /* List empty. Set root node */
      root = BSTNewNode(value);
      return;
   end;
   /* otherwise, search tree to find value */
   found = BSTLookup(path, root, value);  /* if found, return */
   if ^found then                         /* else add to sub-path */
      call ListSetSubItem(root, path, BSTNewNode(value));
finish;
```

To illustrate the process of building the tree in Figure 10.6, the following sequence traces through the algorithm for the key values {5, 3, 1, 9, 6, 4}:

- The first value is 5. Create the root node and assign the key value 5.

- The next value is 3, which is less than 5. Insert a new node with key value 3 as the left child of the "5" node.
- The next value is 1, which is less than 5 and less than 3. Insert a new node with key value 1 as the left child of the "3" node.
- The next value is 9, which is greater than 5. Insert a new node with key value 9 as the right child of the "5" node.
- The next value is 6, which is greater than 5 and less than 9. Insert a new node with key value 6 as the left child of the "9" node.
- The last value is 4, which is less than 5 and greater than 3. Insert a new node with key value 4 as the right child of the "3" node.

With these functions defined, you can create and search a BST. The following statements create the tree in Figure 10.6 and determine whether the target values 6 and 10 are in the tree:

```
x = {5 3 1 9 1 6 4}`;
bst = BSTCreate(x);

found = BSTLookup(path, bst, 6);
print found[L="Was 6 found?"], path[L="Path from root" F=BSTFmt.];
found = BSTLookup(path, bst, 10);
print found[L="Was 10 found?"], path[L="Path from root" F=BSTFmt.];
quit;
```

Figure 10.7 Look for Values in a Binary Search Tree

Was 6 found?
1

Path from root
Right Left

Was 10 found?
0

Path from root
Right Right

There are other algorithms that you can define to operate on binary trees. The *LstBST.sas* file, which is contained in the SAS/IML sample library, includes functions that produce a graph of a BST (like Figure 10.6), that compute the depth of a BST, and that return the set of all edges in a BST. For example, Figure 10.6 is created by using the following statements:

```
%include sampsrc(LstBST.sas);       /* define modules */
proc iml;
load module = _all_;                /* load modules */
x = {5 3 1 9 1 6 4}`;
bst = BSTCreate(x);
title "Diagram of Binary Search Tree";
call BSTPlot(bst);
quit;
```

You can specify the SOURCE option in the %INCLUDE statement to display the contents of the program file in the SAS log.

Chapter 11
Packages

Contents

Overview of Packages **171**
Installing, Loading, and Uninstalling a Package **172**
Displaying Information about Installed Packages **173**
Using a Package **175**
Collections of Packages **176**
Creating a Package **177**
Step 1: Create the Package Directory Structure 177
Step 2: Create the Package Information File 178
Step 3: Create the Source Files 180
Step 4: Create a Program File 181
Step 5: Create a Help File 181
Step 6: Create a ZIP File 182
Testing a Package **184**
Naming Source Files **185**

Overview of Packages

A SAS/IML package consists of SAS/IML source code, documentation, data sets, and sample programs. A package is a convenient way for a programmer to download and install a function that extends the functionality of SAS/IML software. Similarly, authoring a package is a convenient way for an expert to disseminate a SAS/IML module that implements complex computations.

You use the following PACKAGE statements to install, uninstall, and load packages:

- The PACKAGE HELP statement displays documentation for the package.
- The PACKAGE INFO statement displays information about an installed package.
- The PACKAGE INSTALL statement installs a package in a predetermined location.
- The PACKAGE LIBNAME statement creates a SAS libref to the data directory for a package.
- The PACKAGE LIST statement lists the installed packages.
- The PACKAGE LOAD statement loads modules that are defined by a package.
- The PACKAGE UNINSTALL statement uninstalls a package.

You might obtain a package by downloading it from a website, such as the SAS/IML File Exchange (https://communities.sas.com/sas-iml-file-exchange). The File Exchange is also a great place to post a package so that other SAS/IML programmers can easily download it.

The File Exchange contains the RightTriangle package that is used as an example in this documentation.

Installing, Loading, and Uninstalling a Package

Packages are distributed as ZIP files. The name of the ZIP file must be the package name. To install a package, specify the full path name of the ZIP file in the PACKAGE INSTALL statement. For example, if the file **C:\Packages\RightTriangle.zip** contains the source code for the RightTriangle package, you can install the package as follows:

```
proc iml;
package install "C:\Packages\RightTriangle.zip";
quit;
```

The PACKAGE INSTALL statement creates a subdirectory named **righttriangle** in a directory whose location is system-dependent and then unzips the contents of the ZIP file into that subdirectory. Although the SAS language is case-insensitive, some operating systems (notably Linux) are case-sensitive. When you specify a ZIP file on a case-sensitive operating system, be sure to match the case of the file name in the operating system.

By default, the PACKAGE INSTALL statement installs the package in the PRIVATE collection. For more information about the installation directory and the collections, see the section "Collections of Packages" on page 176.

The IML procedure also accepts a SAS fileref, as created by the FILENAME statement in Base SAS. Consequently, the following statements also install the RightTriangle package:

```
filename ThePkg "C:\Packages\RightTriangle.zip";
proc iml;
package install ThePkg;
quit;
```

You should install a package only once. The section "Displaying Information about Installed Packages" on page 173 describes how to display information about the installed package.

When you want to call modules in the package, use the PACKAGE LOAD statement to load the package's modules into a SAS/IML session, as described in the section "Using a Package" on page 175.

If you no longer need a package, you can use the PACKAGE UNINSTALL statement to uninstall the package.

Installing and uninstalling a package does not change or delete the ZIP file. Although you are free to delete the ZIP file from your local machine after you have installed the package, you might want to keep the ZIP file on your local machine if you are running SAS on a remote server. When SAS runs on a remote server, the help files and source files for the package are installed on the remote server. Consequently, it might be difficult to view or browse those files. If you keep the ZIP file on your local machine, you can locally access the help files and source files.

Displaying Information about Installed Packages

You can use the following PACKAGE statements to display information about packages that have been installed:

- The PACKAGE LIST statement displays the packages that are installed on your system. For each collection of packages, the statement produces a table that shows the names of packages in that collection. For more information about collections, see the section "Collections of Packages" on page 176. The following statement produces the output in Figure 11.1 for a hypothetical user and site. The results of the PACKAGE LIST statement will be different on your system.

```
proc iml;
package list;
```

Figure 11.1 Names of Installed Packages

Private Packages

Name	Version
RightTriangle	1.0
SimWithSAS	1.0

Public Packages

Name	Version
pharmaTools	2.1

System Packages

Name	Version
AboveBelow	1.0
ListUtil	1.0
wavelet	1.0

- The PACKAGE INFO statement displays information about a particular package that is installed. For example, the following statement produces the output in Figure 11.2, information about the AboveBelow package, which is distributed with SAS/IML software:

```
package info AboveBelow;
```

Figure 11.2 Information about an Installed Package

Package Information

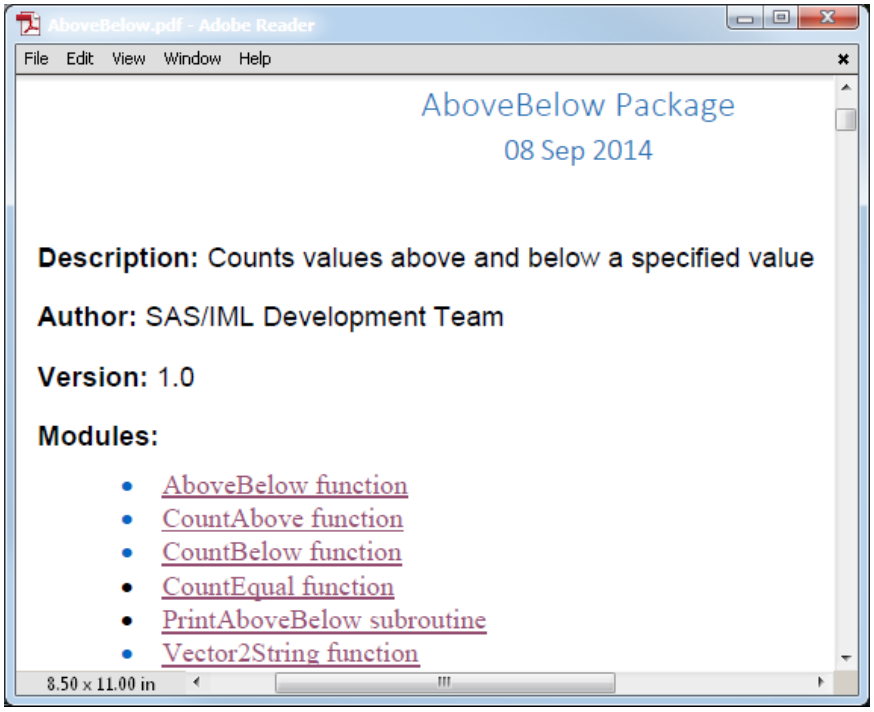
Name	AboveBelow
Description	Counts values above and below a specified value
Author	SAS/IML Development Team
Collection	System
Version	1.0
Requires IML	14.1
Directory	C:\Program Files\SASHome\SASFoundation\9.4\iml\sasmisc\packages\AboveBelow

The table in Figure 11.2 provides the name of the package, a brief description, the package author, the collection to which the package belongs, the package version, the version of SAS/IML software that is required to run the package, and the directory in which the package is installed. The location of the directory depends on the operating system.

- The PACKAGE HELP statement displays the help that was supplied by the author of the package. Usually the help includes information about the modules in the package and their calling syntax. The SAS/IML Studio application displays help in the form of a PDF file or HTML page, if these formats are provided by the package author. The IML procedure displays the function names and syntax in the SAS Log window. The following statement displays information about the functions in the AboveBelow package. The PDF output is shown in Figure 11.3.

```
package help AboveBelow;
```

Figure 11.3 Help File Displayed as PDF Document



Alternately, if you are running SAS on your local machine, you can view the documentation in PDF format by doing the following:

1. Use the PACKAGE INFO statement to display the package installation directory, as shown in Figure 11.2.
2. Use the operating system to navigate to the package installation directory.
3. Navigate to the **`help`** subdirectory of the installation directory.
4. Use an application such as Adobe Reader to display the PDF file in that directory.

If you install a package on a remote SAS server, you can keep the ZIP file on your local computer and use similar steps to open the ZIP file and view the documentation.

Using a Package

Before you can use a package, it must be installed, as described in the section "Installing, Loading, and Uninstalling a Package" on page 172.

Usually a package provides definitions of SAS/IML modules. To use a package, use the PACKAGE LOAD statement to load the module definitions. The following statement loads the AboveBelow package, which is distributed as part of SAS/IML software:

```
package load AboveBelow;
```

If the package includes a **data** directory, you can use the PACKAGE LIBNAME statement to create a libref that points to the data directory. (If the package does not include a **data** directory, the PACKAGE LIBNAME statement generates an error.) The following statements assign the libref ABDATA to point to the data directory for the AboveBelow package. The USE and READ statements read the data into the matrix **A**.

```
package libname abdata AboveBelow;
use abdata.example; read all var _NUM_ into A; close;
```

Loading a package defines all modules and variables in the package. (Use the PACKAGE HELP statement to view the module syntax, as shown in the section "Displaying Information about Installed Packages" on page 173.) The following statements call a module in the AboveBelow package and run it on the example data. Figure 11.4 illustrates the data and the module's output, which shows the numbers of negative values, zero values, positive values, and missing values for each of the three columns of the data matrix.

```
print A;
run PrintAboveBelow(A);     /* call module in package */
```

Figure 11.4 Matrix and Result of Calling a Module in the AboveBelow Package

A		
.	-9	-8
-7	0	.
0	.	.
-1	0	1
0	3	4
5	6	0

Counts			
Negative	2	1	1
Zero	2	2	1
Positive	1	2	2
Missing	1	1	2

In a similar way, you can call other functions in the AboveBelow package.

Collections of Packages

SAS/IML software supports the following *collections* of packages, which reflect different ways that SAS/IML software is used:

- The SYSTEM collection contains packages that are installed as part of SAS/IML software and are available to any user. These packages enable developers at SAS to write and distribute specialized packages of functions that might be of interest to some—but perhaps not all—SAS/IML programmers. The SYSTEM collection is a generalization of the IMLMLIB library of modules. The SYSTEM collection includes the AboveBelow package, which serves as an example of how to load and use packages.
- The PUBLIC collection contains packages that are installed by a system administrator in a public location and are available to any user. These packages enable SAS administrators at a site to install a library of common modules that are used by programmers who work together. These modules ensure that the programmers use the same module library.
- The PRIVATE collection contains packages that are installed by a user and are available only to that user. These packages enable a SAS/IML programmer to install libraries of modules for personal use. This collection enables an individual to extend the capabilities of SAS/IML software by using modules written by others.

If you specify the name of a package in the PACKAGE statement but you do not specify a collection, then the PACKAGE statement searches for the package first in the PRIVATE collection, then in the PUBLIC collection, and finally in the SYSTEM collection.

You can use this search behavior to control which version of a package is loaded. Suppose that your site administrator has installed version 1.0 of a package named Pkg1 in the PUBLIC collection. If the package author releases version 2.0 and you want to use that newer version, you can install the newer version in the PRIVATE collection. The following statement loads the newer version of the package:

```
package load Pkg1;    /* the PRIVATE collection is searched first */
```

If you discover that the newer version is not robust and you want to use the older version, you can use the following statement to explicitly load the older package from the PUBLIC collection:

```
package load Pkg1(public);  /* specify the PUBLIC collection */
```

Each collection is installed in a different directory. The IML procedure uses directory locations that are system specific:

- Private packages are installed in the directory that is specified by the IMLPACKAGEPRIVATE system option.
- Public packages are installed in the directory that is specified by the IMLPACKAGEPUBLIC system option.
- System packages are installed in the directory that is specified by the IMLPACKAGESYSTEM system option.

The following statement displays the location for private, public, and system packages:

```
proc options option=(imlpackageprivate imlpackagepublic imlpackagesystem) value;
run;
```

The SAS/IML Studio application uses the installation directories in Table 11.1.

Table 11.1 SAS/IML Studio Installation Directories

Collection	Default Directory
SYSTEM	***SASHOME*\SASIMLStudio\14.2\Packages**, where *SASHOME* is the root directory for the SAS system, such as **C:\Program Files\SASHome**
PUBLIC	**C:\ProgramData\SAS\IML Studio\Packages**
PRIVATE	**C:\Users*UserID*\Documents\My IML Studio Files\Packages**

Creating a Package

The previous sections of this chapter show how to install and use a package that was written by someone else. This section shows how you can author a new SAS/IML package. A package can contain source files that define modules, data sets, documentation, and programs that demonstrate how to use the package.

Suppose you have written some SAS/IML functions that compute certain facts about right triangles in the Euclidean plane. You decide to create a package called RightTriangle to share these functions with others.

This section guides you through the creation of the RightTriangle package. The steps are summarized as follows and described in more detail in subsequent sections:

1. Create a package directory structure.
2. Create a file named *info.txt*, which provides SAS/IML software with information about the package. The file name must contain only lowercase letters.
3. Create one or more source files that define the SAS/IML modules in the package.
4. Create a program file that demonstrates how to call the modules.
5. Create a help file that documents the purpose of the package and shows the syntax of the modules.
6. Create a ZIP file that preserves the directory structure and contains all the package files.

Step 1: Create the Package Directory Structure

To create a package, you must first create a directory hierarchy on your local computer.

If you install a package by using PROC IML, the root directory is the name of your package in lowercase. To match that behavior, create a directory named **righttriangle**.

Within the **righttriangle** directory, create four subdirectories: **data**, **help**, **programs**, and **source**. It is recommended that you create these directories even though one or more of them might be empty. The purpose of each subdirectory is as follows:

data	Contains all SAS data sets (sas7bdat files) that are used by the package. Use lowercase characters for the names of the data sets in this directory.
help	Contains documentation for the package. You should include a plain text file with a *.txt* extension. Optionally, you can include a PDF or HTML file in this subdirectory.
programs	Contains SAS/IML programs that call functions in the package. For example, this subdirectory can include demo programs, test programs, and programs that reproduce results in a journal article or conference proceedings.
source	Contains the source files that are read by the PACKAGE LOAD statement. These files usually contain module definitions. The files should not contain a PROC IML statement or a QUIT statement.

The RightTriangle package does not contain a data file, so the **data** subdirectory remains empty. The following sections describe how to create the *info.txt* file and the files for the other subdirectories.

A package can contain other subdirectories in addition to these four subdirectories.

Step 2: Create the Package Information File

The PACKAGE INSTALL statement reads the *info.txt* file, which must be in the root directory for the package. The file must be stored in UTF-8 format. The first line in the file, which tells the SAS/IML software how to interpret subsequent lines, must be as follows:

```
# SAS/IML Package Information File Format 1.0
```

Subsequent lines define keyword-value pairs. A keyword must begin in the first column and must be followed by a colon (:). The value that follows the colon is free-form text, unless otherwise noted. A value is ended by either a new keyword-value pair or a blank line. If a value requires more than one line of text, subsequent lines must be indented by at least one space or tab character.

If the first character on a line is the number sign (#), then the rest of the line is ignored. You can use the number sign to put comments in the *info.txt* file.

The SAS/IML language supports the following list of keyword-value pairs. Only the NAME keyword is required. If a keyword is marked "Reported," then that keyword-value pair (up to 100 characters) is displayed by the PACKAGE INFO statement.

NAME: (Reported) Specify the name of the package. The package name must be a valid SAS name, which means it must contain 32 characters or less, begin with a letter or underscore, and contain only letters, underscores, and digits.

The case (uppercase, lowercase, or mixed case) of this keyword value is important for the following reasons:

- The PACKAGE LIST and PACKAGE INFO statements display the value of the NAME keyword exactly as you specify it.
- The help system looks for a file of the name *PkgName.pdf* or *PkgName.txt*, where *PkgName* is the exact value of the NAME keyword.
- In the SAS/IML Studio application, the value determines the case of the name of the package's directory.

DESCRIPTION: (Reported) Specify a brief description of the package.

AUTHOR: (Reported) Specify the authors of the package. Optionally, you can include contact information such as an email address or website.

VERSION: (Reported) Specify the version of the package. The default value is 0.0.0.0. The version can have up to four levels, separated by decimal points (.). For example, valid values are 1.0, 2.7.1, and 3.1.4.1.

REQUIRESIML: (Reported) Specify the version of SAS/IML software that is required to run the package. The default value is 14.1.0.0. The specified value must be greater than or equal to 14.1.0.0. You can specify this value when your package relies on a new feature of SAS/IML software.

SOURCEFILES: Specify a list of the source file names for the package, with one file name per line. The case (uppercase, lowercase, or mixed case) of this keyword value is important because certain file systems (such as Linux) are case-sensitive. The source files usually have a *.sas* or *.iml* extension.

If a source file uses a specific character encoding, you can add an equal sign (=) and a value that indicates the character encoding after the name of the source file. For example, the following statement indicates that the file contains characters in the Windows Latin1 encoding:

```
File1.iml = wlatin1
```

It is not necessary to indicate the encoding of UTF-8 or UTF-16 files if the files begin with a byte-order mark. For more information about character encoding in SAS, see the section "Encoding Values in SAS Language Elements" in *SAS National Language Support (NLS): Reference Guide*.

Copyright: Specify copyright information for the package.

License: Specify an open-source license, such as those approved by the Open Source Initiative (https://opensource.org/licenses).

Dependencies: Specify other packages that your package requires. Each package must be written on a separate line. A user must install the packages in this dependency list before installing the current package.

Notes: Specify other information that the user needs to know in order to use the package.

For example, the keyword-value pairs in the following statements define a valid *info.txt* file for the RightTriangle package. To create this file, create a plain text file in the **righttriangle** directory, paste in the following text, and save the file.

```
# SAS/IML Package Information File Format 1.0
# Do NOT edit the line above!

Name: RightTriangle
Description: Computes side lengths for right triangles
Author: SAS/IML Development Team
Version: 1.0
SourceFiles: RightTriangle.iml
Notes: An example package from the SAS/IML documentation
```

Step 3: Create the Source Files

Usually the first files that you create are the source files that define the SAS/IML modules that you will distribute.

You can use *.sas* or *.iml* as the extension for a source file. For more information about file extensions, see the section "Naming Source Files" on page 185. This documentation uses the *.iml* extension.

In the **source** directory, create a file named *RightTriangle.iml*. (Notice that the case of this file name matches the case that was specified in the *info.txt* file.) Include the following SAS/IML statements in the file. Notice that the file does *not* contain the PROC IML or QUIT statements.

```
/* Modules for the RightTriangle package */
/* Do NOT include the PROC IML or QUIT statements! */

/* Given a vector that contains the three side lengths of a
   triangle, this function returns 1 if the triangle is a right
   triangle and 0 otherwise. */
start IsRightTriangle(vert);
   v = colvec(vert);
   call sort(v, 1);
   a = v[1]; b = v[2]; c = v[3];
   return ( a##2 + b##2 = c##2 );
finish;

/* Given the lengths of the hypotenuse and a leg of a right
   triangle, this function finds the length of the other leg. */
start FindLeg(v);
   b = min(v);  c = max(v);
   d = c##2 - b##2;
   if d>0 then return( sqrt(d) );
   else return( . );
finish;

/* Given the lengths of two legs of a right triangle,
   this function finds the length of the hypotenuse. */
```

```
start FindHypotenuse(v);
   b = max(v);
   a = min(v);
   return( sqrt(a##2 + b##2) );
finish;
```

Save the *RightTriangle.iml* file. When the package is loaded, this file will be read and these functions will be defined.

Step 4: Create a Program File

Examples are very useful to the programmers who will download and use your package. Create a file named *Example.sas* in the **programs** directory. Include the following SAS/IML statements in the file and save the file:

```
/* Examples of calling modules in the RightTriangle package. If you have
   not installed the package, modify and run the following statements:

   proc iml;
   package install "C:\<path>\righttriangle.zip";
   quit;
*/
proc iml;
package load RightTriangle;

Rt = IsRightTriangle({3 4 5});     /* 1 = valid lengths     */
NotRt = IsRightTriangle({5 6 7} ); /* 0 = not valid lengths */
print Rt NotRt;

leg = FindLeg({3 5});           /* find 2nd leg from hypotenuse and leg */
hypot = FindHypotenuse({3 4});  /* find hypotenuse from legs */
print leg hypot;                /* correct answer is {4 5} */
quit;
```

The *Example.sas* file contains an example of calling each function in the package, and the comments provide the correct answer to each call. Programmers who want to use the package can study the program and modify it for their own use. The program also serves as a simple test program.

Step 5: Create a Help File

Every package should contain help in at least two formats: a short plain text file that describes the syntax of functions, and a PDF or HTML file that contains the complete documentation. The root name for the PDF and plain-text help files must match the case that you specified for the NAME keyword in the *info.txt* file. For example, the help files for the RightTriangle package must be *RightTriangle.pdf* and *RightTriangle.txt*. If you create an HTML file, the name of the file must be *index.html* or *index.htm*. All help files should be in the **help** subdirectory.

An example of a PDF help file is included with the AboveBelow package, which is distributed with SAS/IML software and is installed in the SYSTEM collection.

For simplicity, this step creates only a plain text file that documents the syntax of the functions in the RightTriangle package. Navigate to the **help** subdirectory, and use a text editor to create a file called *RightTriangle.txt*. Insert the following text and save the file:

```
RightTriangle Package

Description: Computes side lengths for right triangles

Modules:
ISRIGHTTRIANGLE(v);
Given a three-element vector that contains the three side lengths
of a triangle, this function returns 1 if the triangle is a right
triangle and 0 otherwise.

FINDLEG(v);
Given a two-element vector that contains the lengths of the
hypotenuse and a leg of a right triangle, this function finds
the length of the other leg.

FINDHYPOTENUSE(v);
Given a two-element vector that contains the lengths of two legs of
a right triangle, this function finds the length of the hypotenuse.
```

The PACKAGE HELP statement displays a help file for an installed package. The file that is displayed depends on how you are running the SAS/IML software:

- The SAS/IML Studio application searches the **help** subdirectory for one of the following help files: *PkgName.pdf*, *index.html*, *index.htm*, or *PkgName.txt*, where *PkgName* is the name of your package. If a PDF or an HTML file is found, then that file is displayed.
- In PROC IML, the PACKAGE HELP statement displays the *PkgName.txt* file in the SAS Log window. This is a convenient way to see the syntax for the package functions.
- If SAS/IML is installed on a local or networked computer, you can use the PACKAGE INFO statement to discover the location of the directory where the package is installed. After you navigate to that directory, you can use a PDF viewer or a browser to view the documentation in the **help** subdirectory.
- If you are running a SAS/IML program on a remote SAS server (such as through SAS Enterprise Guide or SAS Studio), you might not be able to navigate to the location of the **help** subdirectory. You might want to keep a copy of the package files (or the ZIP file) in a local directory so that you can browse the help files and view the source code.

Step 6: Create a ZIP File

All packages are distributed as ZIP files. The file name of the ZIP file must be the name of the package followed by the *.zip* extension.

This section assumes that you have created a directory hierarchy and that you have created source files, help files, program files, and an *info.txt* file, as described in the previous sections. For the RightTriangle package, the directories and files are arranged as shown in the Figure 11.5.

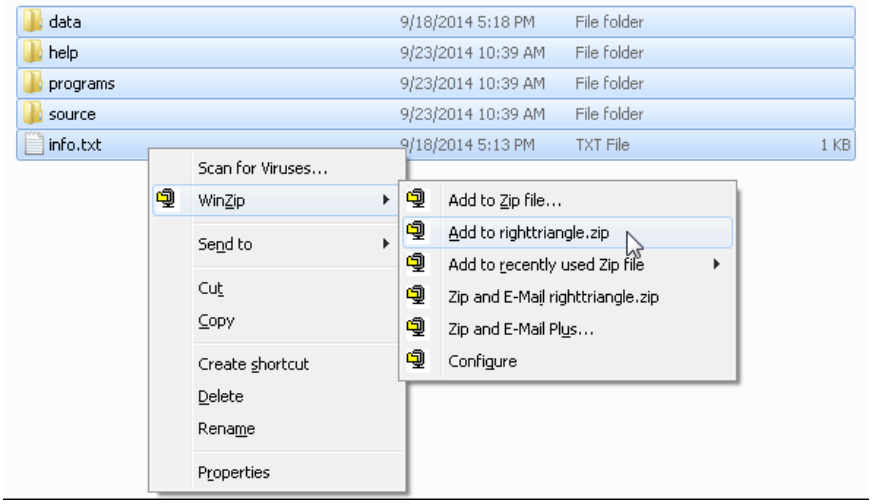
Figure 11.5 RightTriangle Directory Structure

```
|   info.txt
|
+---data
+---help
|       RightTriangle.txt
|
+---programs
|       example.sas
|
+---source
        RightTriangle.sas
```

This section uses WinZip software and the Windows operating system to illustrate how to create a ZIP file of the RightTriangle package. If you use other software to create ZIP file, the process will be different.

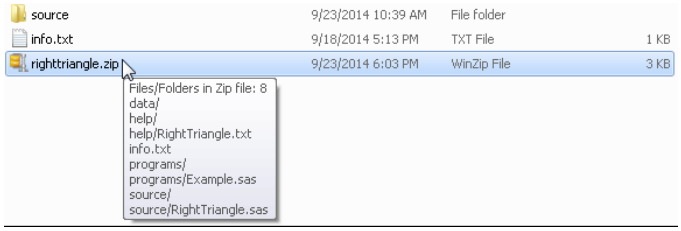
To create a ZIP file by using WinZip software, use Windows Explorer to navigate to the **`righttriangle`** directory. Select all the files and subdirectories in the **`righttriangle`** directory. Right-click on the selected files, and choose **WinZip ►Add to righttriangle.zip** from the pop-up menu, as shown in Figure 11.6.

Figure 11.6 Creating a ZIP File by Using WinZip



The program will create the file *righttriangle.zip*. You can examine the contents of the ZIP file to make sure that all the files are included and that the directory structure was preserved. The ZIP file will look similar to Figure 11.7.

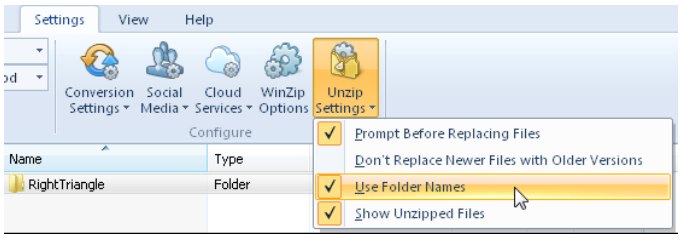
Figure 11.7 Checking the Contents of a ZIP File



So that the ZIP file can be correctly unzipped by the SAS/IML PACKAGE statement, make sure that the ZIP file satisfies the following requirements:

- The ZIP file should preserve the directory hierarchy when the files are unzipped. In WinZip 18.0, click the **Settings** tab and select the **Use Folder Names** option under the **Unzip Settings** menu, as shown in Figure 11.8.
- The ZIP file cannot be password-protected.
- The ZIP file must use the original ZIP compression scheme.

Figure 11.8 WinZip Options



Testing a Package

Before you send a ZIP file that contains a SAS/IML package to a colleague or post it to the SAS/IML File Exchange, you are strongly encouraged to install and run the package.

To test the RightTriangle package:

1. Store the ZIP file somewhere on your file system. This example assumes that the ZIP file is located in the directory `C:\Packages`.
2. Run the PACKAGE INSTALL statement to install the package from the ZIP file, as follows:

```
proc iml;
package install "C:\Packages\righttriangle.zip";
quit;
```

3. Run the following statements to ensure that the package installed correctly:

```
proc iml;
package list;
package info RightTriangle;
package help RightTriangle;
```

4 Load the package and run the programs in the **programs** directory, as described in the section "Step 4: Create a Program File" on page 181. For example, the following statements test each function in the RightTriangle package:

```
package load RightTriangle;
Rt = IsRightTriangle({3 4 5});  /* 1 = valid lengths     */
leg = FindLeg({3 5});           /* find 2nd leg from hypotenuse and leg */
hypot = FindHypotenuse({3 4});  /* find hypotenuse from legs */
print Rt leg hypot;             /* correct answer is {1 4 5} */
```

Naming Source Files

Some operating systems (notably Linux) are case-sensitive. Make sure that the names of all files in the **source** directory match the case of the names for the SourceFiles keyword in the *info.txt* file.

To ensure that data sets are found on all operating systems, lowercase the names of all data sets in the **data** directory of a package.

Knowing when to use lowercase versus mixed-case file names can be confusing. When it doubt, use lowercase file names.

Another potential source of confusion is whether to use *.iml* or *.sas* as the extension for source files in the **source** directory. The choice of a file name extension does not affect the functionality of a package. However, the choice of an extension does determine which application is used to edit the file. It also determines how that file is displayed by an application.

Under the Windows operating system, double-clicking a file automatically launches the default application that is associated with the file extension. A file with a *.sas* extension might be opened by a program such as the SAS windowing environment or SAS Enterprise Guide. A file with an *.iml* extension is opened by SAS/IML Studio, if it is installed.

Some text editors perform syntax highlighting based on a file's extension. A file with a *.sas* extension is opened as a SAS file by most program editors. Consequently, SAS/IML statements in the source files might not be colored correctly.

A file with an *.iml* extension is colored according to IMLPlus syntax when the file is opened by the SAS/IML Studio application. The same file might be treated as a plain-text file by other applications. For example, the enhanced editor in the SAS windowing environment does not apply syntax highlighting to a file that has an *.iml* extension.

Chapter 12
General Statistics Examples

Contents

Overview **187**
General Statistics Examples **188**
Example 12.1: Correlation Computation 188
Example 12.2: Newton's Method for Solving Nonlinear Systems of Equations 189
Example 12.3: Regression 191
Example 12.4: Alpha Factor Analysis 194
Example 12.5: Categorical Linear Models 196
Example 12.6: Regression of Subsets of Variables 200
Example 12.7: Response Surface Methodology 206
Example 12.8: Logistic and Probit Regression for Binary Response Models 208
Example 12.9: Linear Programming 211
Example 12.10: Quadratic Programming 215
Example 12.11: Regression Quantiles 216
Example 12.12: Simulations of a Univariate ARMA Process 220
Example 12.13: Parameter Estimation for a Regression Model with ARMA Errors . . 222
Example 12.14: Iterative Proportional Fitting 229
Example 12.15: Nonlinear Regression and Specifying a Model at Run Time 230
References **233**

Overview

Linear algebra is fundamental to regression, principal component analysis, and other multivariate statistical techniques. You can use the functions and high-level operators in SAS/IML software to implement these techniques. The similarity between the SAS/IML syntax and matrix algebra notation often makes it straightforward to translate an algorithm into a SAS/IML program.

The examples in this chapter demonstrate a variety of matrix computations. You can use these examples to gain insight into other complex problems you might need to solve. Some of the examples perform the same analyses as are 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 12.1: Correlation Computation

The following statements show how you can define modules to standardized columns for a matrix of numeric data. For a more robust implementation, see the STANDARD function.

```
proc iml;
/* Standardize data: Assume no column has 0 variance */
start stdMat(x);
   mean = mean(x);                          /* means for columns */
   cx = x - mean;                        /* center x to mean zero */
   std = std(x);                    /* standard deviation estimate*/
   y = cx / std(x);                      /* scaling to std dev 1 */
   return( y );
finish stdMat;

x = { 1 2 3,
      3 2 1,
      4 2 1,
      0 4 1,
     24 1 0,
      1 3 8};
nm = {age weight height};
std = stdMat(x);
print std[colname=nm label="Standardized Data"];
```

Output 12.1.1 Standardized Variables

Standardized Data		
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

The columns shown in Output 12.1.1 have zero mean and unit variance.

In a similar way, you can define a module that returns the correlation matrix of numeric data. The following module computes the correlation matrix according to a formula that you might see in a statistics textbook. For a more efficient implementation that supports missing values, use the built-in CORR function.

```
/* Compute correlations: Assume no missing values  */
start corrMat(x);
   n = nrow(x);                        /* number of observations */
   sum = x[+,];                           /* compute column sums */
   xpx = x`*x - sum`*sum/n;             /* compute sscp matrix   */
   s = diag(1/sqrt(vecdiag(xpx)));             /* scaling matrix */
```

```
   corr = s*xpx*s;                          /* correlation matrix */
   return( corr );
finish corrMat;

corr = corrMat(x);
print corr[rowname=nm colname=nm label="Correlation Matrix"];
```

Output 12.1.2 A Correlation Matrix

Correlation Matrix			
	AGE	WEIGHT	HEIGHT
AGE	1	-0.717102	-0.436558
WEIGHT	-0.717102	1	0.3508232
HEIGHT	-0.436558	0.3508232	1

There are many equivalent ways to compute a correlation matrix. If you have already written and debugged the STDMAT function, you might want to call that function during the computation of the correlation matrix. The following function is an alternative way to compute the correlation matrix of data:

```
/* Another way to compute correlations: Assume no missing values */
start corrMat2(x);
   y = StdMat(x);                          /* standardize columns */
   corr = (y`*y)/(nrow(x)-1);               /* correlation matrix */
   return( corr );
finish corrMat2;

c = corrMat2(x);
```

Example 12.2: 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 nonlinear function.

Newton's method is an iterative technique. The method starts with an initial estimate $\mathbf{x_0}$ of the root. The estimate is refined iteratively in an attempt to find a root of F. Given an estimate $\mathbf{x}_n$, the next estimate is given by

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}^{-1}(\mathbf{x}_n)F(\mathbf{x}_n)$$

where $\mathbf{J}(\mathbf{x})$ is the Jacobian matrix of partial derivatives of F with respect to $\mathbf{x}$. (For more efficient computations, use the built-in NLPNRA subroutine.)

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

$$\begin{aligned} \mathbf{x_1} + \mathbf{x_2} - \mathbf{x_1}\mathbf{x_2} + 2 &= 0 \\ \mathbf{x_1}\exp(-\mathbf{x_2}) - 1 &= 0 \end{aligned}$$

The following statements are 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 parameters: CONVERGE, MAXITER.                 */
/* On exit: return x such that FUN(x) is close to 0      */
proc iml;
/* User-supplied function evaluation */
start Fun(x);
   x1 = x[1];  x2 = x[2];           /* extract components */
   f1 = x1 + x2 - x1*x2 + 2;
   f2 = x1*exp(-x2) - 1;
   return( f1 // f2 );
finish Fun;

/* User-supplied derivatives of the function */
start Deriv(x);
   x1 = x[1]; x2 = x[2];
   df1dx1 = 1 - x2;
   df1dx2 = 1 - x1;
   df2dx1 = exp(-x2);
   df2dx2 = -x1 * exp(-x2);
   J = (df1dx1 || df1dx2 ) //
       (df2dx1 || df2dx2 );
   return( J );                    /* Jacobian matrix */
finish Deriv;

/* Implementation of Newton's method with default arguments */
/* By default, maximum iterations (maxiter) is 25           */
/*             convergence criterion (converge) is 1e-6     */
start NewtonMethod(x0, maxIter=25, converge=1e-6);
   x = x0;
   f = Fun(x);       /* evaluate function at starting values */
   do iter = 1 to maxiter           /* iterate until maxiter */
      while(max(abs(f))>converge);         /* or convergence */
      J = Deriv(x);                  /* evaluate derivatives */
      delta = -solve(J, f);   /* solve for correction vector */
      x = x + delta;                /* the new approximation */
      f = fun(x);                   /* evaluate the function */
   end;
   /* return missing if no convergence */
   if iter > maxIter then
      x = j(nrow(x0),ncol(x0),.);
   return( x );
finish NewtonMethod;

print "Solve the system: X1+X2-X1*X2+2=0, X1*EXP(-X2)-1=0" ,;
x0 = {.1, -2};     /* starting values */
x = NewtonMethod(x0);
```

```
f = Fun(x);
print x f;
```

Output 12.2.1 Newton's Method: Results

Solve 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

The results are shown in Output 12.2.1. Notice that the NEWTONMETHOD function was called with only a single argument, which causes the module to use the default number of iterations and the default convergence criterion. To change those parameter values, call the module with additional arguments, as follows:

```
x = NewtonMethod(x0, 15, 0.001);
```

Example 12.3: Regression

This example is a module that calculates statistics that are associated with a linear regression. The following module is similar to the REGRESS module, which is included in the IMLMLIB library:

```
proc iml;
start regress( x, y, name, tval=, l1=, l2=, l3= );
   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 - cdf("F",t#t,1,dfe);       /* significance probability */
   paramest = b || stdb || t || probt;
   print paramest[c={"Estimate" "StdErr" "t" "Pr>|t|"} r=name
                  l="Parameter Estimates" f=Best6.];

   s = diag(1/stdb);
   corrb = s * covb * s;                 /* correlation of estimates */
   reset fw=6 spaces=3;                  /* for proper formatting    */
   print covb[r=name c=name l="Covariance of Estimates"],
         corrb[r=name c=name l="Correlation of Estimates"];
```

```
   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 pred vals  */
   h = vecdiag(projx);                   /* hat leverage values      */
   lowerm = yhat - tval # sqrt(h*mse);   /* lower conf limit for mean*/
   upperm = yhat + tval # sqrt(h*mse);   /* upper CL for mean        */
   lower = yhat - tval # sqrt(h*mse+mse);/* lower CL for individual  */
   upper = yhat + tval # sqrt(h*mse+mse);/* upper CL                 */

   R = y || yhat || resid || h || lowerm || upperm || lower || upper;
   labels = {"y" "yhat" "resid" "h" "lowerm" "upperm" "lower" "upper"};
   reset fw=6 spaces=1;
   print R[c=labels label="Predicted values, Residuals, and Limits"];

   /* test hypoth that L*b = 0, where L is linear comb of estimates  */
   do i = 1 to 3;
      L = value("L"+ strip(char(i)));   /* get L matrix for L1, L2, and L3 */
      if nrow(L) = 0 then return;
      dfn = nrow(L);
      Lb = L * b;
      vLb = L * xpxi * L`;
      q = Lb` * inv(vLb) * Lb / dfn;
      f = q / mse;
      prob = 1 - cdf("F", f,dfn,dfe);
      test = dfn || dfe || f || prob;
      print L, test[c={"dfn" "dfe" "F" "Pr>F"} f=Best6.
                    l="Test Hypothesis that L*b = 0"];
   end;
finish;
```

The module accepts up to three matrices for testing the hypothesis that a linear combination of the parameters is zero. For more information about the computation, see the documentation for the TEST statement in the REG procedure in *SAS/STAT User's Guide*.

The following statements call the REGRESS module on data that describes the size of the US population during eight decades 1790–1860. The following statements fit a quadratic regression model to the data. The program also tests three hypotheses about the parameters in the model.

```
/* Quadratic regression on US population for decades beginning 1790 */
decade = T(1:8);
name={"Intercept", "Decade", "Decade**2" };
x= decade##0 || decade || decade##2;
y= {3.929,5.308,7.239,9.638,12.866,17.069,23.191,31.443};

/* n-p=5 dof at 0.025 level to get 95% confidence interval */
tval = quantile("T", 1-0.025, nrow(x)-ncol(x));
L1 = { 0 1 0 };   /* test hypothesis Lb=0 for linear coef */
L2 = { 0 1 0,     /* test hypothesis Lb=0 for linear,quad */
       0 0 1 };
L3 = { 0 1 1 };   /* test hypothesis Lb=0 for linear+quad */
option linesize=100;
run regress( x, y, name, tval, L1, L2, L3 );
```

The parameters estimates are shown in the first table in Output 12.3.1. The next two tables show the covariance and correlation of the estimates, respectively.

Output 12.3.1 Regression Results

Parameter Estimates

	Estimate	StdErr	t	Pr>|t|
Intercept	5.0693	0.9656	5.25	0.0033
Decade	-1.11	0.4923	-2.255	0.0739
Decade2**	0.5396	0.0534	10.106	0.0002

Covariance of Estimates

	Intercept	Decade	Decade**2
Intercept	0.9324	-0.436	0.0428
Decade	-0.436	0.2424	-0.026
Decade2**	0.0428	-0.026	0.0029

Correlation of Estimates

	Intercept	Decade	Decade**2
Intercept	1	-0.918	0.8295
Decade	-0.918	1	-0.976
Decade2**	0.8295	-0.976	1

The predicted values, residuals, leverage, and confidence limits for the mean and individual predictions are shown in Output 12.3.2.

Output 12.3.2 Regression Results: Predicted Values and Residuals

Predicted values, Residuals, and Limits

y	yhat	resid	h	lowerm	upperm	lower	upper
3.929	4.499	-0.57	0.7083	3.0017	5.9964	2.1737	6.8244
5.308	5.008	0.3	0.2798	4.067	5.949	2.9954	7.0207
7.239	6.5963	0.6427	0.2321	5.7391	7.4535	4.6214	8.5711
9.638	9.2638	0.3742	0.2798	8.3228	10.205	7.2511	11.276
12.866	13.011	-0.145	0.2798	12.07	13.952	10.998	15.023
17.069	17.837	-0.768	0.2321	16.979	18.694	15.862	19.812
23.191	23.742	-0.551	0.2798	22.801	24.683	21.729	25.755
31.443	30.727	0.7164	0.7083	29.229	32.224	28.401	33.052

The results of the hypothesis tests are shown in Output 12.3.3. The first hypothesis is that the coefficient of the linear term is 0. This hypothesis is not rejected at the 0.05 significance level. The second hypothesis is that the coefficients of the linear and quadratic terms are simultaneously 0. This hypothesis is soundly rejected. The third hypothesis is that the linear coefficient is equal to the negative of the quadratic coefficient. Given the data, this hypothesis is not rejected.

Output 12.3.3 Regression Results: Hypothesis Tests

L
0 1 0

Test Hypothesis that L*b = 0			
dfn	dfe	F	Pr>F
1	5	5.0832	0.0739

L
0 1 0
0 0 1

Test Hypothesis that L*b = 0			
dfn	dfe	F	Pr>F
2	5	666.51	854E-9

L
0 1 1

Test Hypothesis that L*b = 0			
dfn	dfe	F	Pr>F
1	5	1.6775	0.2518

Example 12.4: Alpha Factor Analysis

This example shows how an algorithm for computing alpha factor patterns (Kaiser and Caffrey 1965) could be implemented in the SAS/IML language. This algorithm is similar to that provided by the METHOD=ALPHA option in the FACTOR procedure.

The following statements define a SAS/IML module for computing an alpha factor analysis. The input is a matrix of correlations. The module computes eigenvalues, communalities, and a factor pattern.

```
proc iml;
/*                Alpha Factor Analysis                        */
/*  Ref: Kaiser et al., 1965 Psychometrika, pp. 12-13          */
/*  Input:  r = correlation matrix                             */
/*  Output: m = eigenvalues                                    */
/*          h = communalities                                  */
/*          f = factor pattern                                 */
start alpha(m, h, f, r);
   p = ncol(r);
   q = 0;
   h = 0;                                        /* initialize */
   h2 = I(p) - diag(1/vecdiag(inv(r)));/* smc=sqrd mult corr */
   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);                /* communalities as vector */
   f = hi*e*mm;                          /* resulting pattern */
finish;
```

The following statements call the ALPHA module on a sample correlation matrix. The results are shown in Output 12.4.1.

```
/* Correlation Matrix from Harmon, Modern Factor Analysis, */
/* Second edition, page 124, "Eight Physical Variables"    */
nm = {Var1 Var2 Var3 Var4 Var5 Var6 Var7 Var8};
r ={ 1.00 .846 .805 .859 .473 .398 .301 .382 ,
     .846 1.00 .881 .826 .376 .326 .277 .415 ,
     .805 .881 1.00 .801 .380 .319 .237 .345 ,
     .859 .826 .801 1.00 .436 .329 .327 .365 ,
     .473 .376 .380 .436 1.00 .762 .730 .629 ,
     .398 .326 .319 .329 .762 1.00 .583 .577 ,
     .301 .277 .237 .327 .730 .583 1.00 .539 ,
     .382 .415 .345 .365 .629 .577 .539 1.00};
run alpha(Eigenvalues, Communalities, Factors, r);
print Eigenvalues,
      Communalities[rowname=nm],
      Factors[label="Factor Pattern" rowname=nm];
```

Output 12.4.1 Alpha Factor Analysis: Results

Eigenvalues
5.937855
2.0621956
0.1390178
0.0821054
0.018097
-0.047487
-0.09148
-0.100304

Communalities	
VAR1	0.8381205
VAR2	0.8905717
VAR3	0.81893
VAR4	0.8067292
VAR5	0.8802149
VAR6	0.6391977
VAR7	0.5821583
VAR8	0.4998126

Factor Pattern		
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 12.5: Categorical Linear Models

This example fits a linear model to a function of the response probabilities

$$\mathbf{K} \log \pi = \mathbf{X}_{\boldsymbol{\epsilon}} + \epsilon$$

where $\mathbf{K}$ is a matrix that compares each response category to the last category.

First, the Grizzle-Starmer-Koch approach (Grizzle, Starmer, and Koch 1969) is used to obtain generalized least squares estimates of $\boldsymbol{\beta}$. 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 (Cox 1970).

```
proc iml ;
/* Subroutine to compute new probability estimates   */
/* Last column not needed since sum of each row is 1 */
start prob(x, beta, q);
   la = exp(x*shape(beta,0,q));
   pi = la / ((1+la[,+])*repeat(1,1,q));
   return( colvec(pi) );
```

```
finish prob;

/* Categorical Linear Models                    */
/* by Least Squares and Maximum Likelihood      */
/*  Input:                                      */
/*     n the s by p matrix of response counts   */
/*     x the s by r design matrix               */
start catlin(n, x);
   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 */

   /* initial (emprical) probability estimates */
   rown = n[,+];                                /* row totals */
   pr = n/rown;                       /* probability estimates */
   print pr[label="Initial Probability Estimates"];

   /* function of probabilities */
   p = colvec(pr[,1:q]);          /* cut and shaped to vector */
   f = log(p) - log(pr[,r])@repeat(1,q,1);

   /* estimate by the GSK method */
   /* 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 */
   pi = prob(x, beta, q);
   est = beta || stderr;
   pr = shape(pi, 0, q);
   print est[colname={"beta" "stderr"} label="GSK Estimates"], pr;

   /* 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);  /* correction via Newton's method */
      beta = beta+delta;               /* apply the correction */
      pi = prob(x, beta, q);         /* compute prob estimates */
      crit = max(abs(delta));         /* convergence criterion */
   end;
   stderr = sqrt(vecdiag(inv(h)));         /* standard errors */
   est = beta || stderr;
   pr = shape(pi, 0, q);
   print est[colname={"beta" "stderr"} label="ML Estimates"], pr;
   print it[label="Iterations"]  crit[label="Criterion"];
finish catlin;
```

The following statements call the CATLIN module to analyze data from Kastenbaum and Lamphiear (1959):

```
/* frequency counts*/
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};

/* design matrix */
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};

run catlin(n, x);
```

The maximum likelihood estimates are shown in Output 12.5.1.

Output 12.5.1 Maximum Likelihood Estimates

Initial Probability Estimates		
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

Output 12.5.1 *continued*

GSK Estimates	
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

pr	
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

ML Estimates	
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

Output 12.5.1 *continued*

pr	
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

Iterations	Criterion
3	0.0004092

Example 12.6: Regression of Subsets of Variables

This example performs regression along with variable selection. Some of the methods used in this example are also used in the REG procedure in SAS/STAT software. The GLMSELECT procedure implements these and other variable selection techniques.

To simplify communication between modules, the modules in this example do not take any arguments. This means that the modules do not use a local symbol table: all variables are defined at the main scope of the program. In this programming technique, modules are used to organize the algorithm and, potentially, to enable code reuse.

```
proc iml;
 /*-------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      0-1 vector of whether variable is in model   |
 | 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*/
  csave=(xpx || xpy) //
        (xpy`|| ypy);             /* save copy of crossproducts*/
finish;
```

```
/*-----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;

/*-----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;

/*-----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;
     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;

/*----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;

/*-----search all possible models----------------------------*/
start all;
   /*---use method of Schatzoff et al. for search technique--*/
   betak=repeat(0,k,k); /* record estimates for best 1-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 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];
finish;

/*-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;

/*-----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;

/*-----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;

/*--------------stepwise methods---------------------*/
/* after a call to the initial routine, which sets up*/
/* 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;          /* forward selection method   */
   run backward;         /* backward elimination method*/
   run stepwise;         /* stepwise method            */
finish;
```

The following statements call the SEQ module, which in turn calls modules that perform all-subset regression, forward selection, backward selection, and stepwise selection. The results are shown in Output 12.6.1.

```
/*-----------------------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  };
y=data[,3];
x=data[,{1 2 4 5 6 7 }];
free data;
```

```
varnames={age weight runtime rstpuls runpuls maxpuls};
reset fw=6 linesize=87;
run seq;
```

Output 12.6.1 Model Selection: Results

Initial Probability Estimates		
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

GSK Estimates	
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

pr	
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

Output 12.6.1 *continued*

ML Estimates	
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

pr	
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

Iterations	Criterion
3	0.0004092

Example 12.7: Response Surface Methodology

A regression model that has a complete quadratic set of regressions across several factors can be processed to yield the estimated critical values that can optimize a response. First, the regression is performed for two variables according to the following model:

$$y = c + b_1x_1 + b_2x_2 + a_{11}x_1^2 + a_{12}x_1x_2 + a_{22}x_2^2 + e$$

The estimates are then divided into a vector of linear coefficients (estimates), $\mathbf{b}$, and a matrix of quadratic coefficients, $\mathbf{A}$. 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. For more information about response surface modeling, see the documentation for the RSREG procedure in *SAS/STAT User's Guide*.

```
proc iml;
/*          Quadratic Response Surface Regression            */
/* This matrix routine reads in the factor variables and     */
/* the response, forms the quadratic regression model and    */
/* estimates the parameters, and then solves for the optimal */
/* response, prints the optimal factors and response, and    */
/* 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:                                                    */
/* d contains the factor variables                           */
/* y contains the response variable                          */
/*                                                           */

start rsm(d, y);
   n=nrow(d);
   k=ncol(d);                                  /* dimensions */
   x=j(n,1,1) || d;                  /* set up design matrix */
   do i=1 to k;                     /* add quadratic effects */
      x = x || d[,i] #d[,1:i];
   end;
   beta=solve(x`*x, x`*y);             /* estimate parameters */
   names = "b0":("b"+strip(char(nrow(beta)-1)));
   print beta[rowname=names label="Parameter Estimates"];

   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`)/2;                                 /* symmetrize */
   xx = -0.5*solve(a,b);          /* solve for critical value */
   print xx[label="Critical Factor Values"];

   /* Compute response at critical value */
   yopt=c + b`*xx + xx`*a*xx;
   print yopt[label="Response at Critical Value"];

   call eigen(eval,evec,a);
   if min(eval)>0 then print "Solution Is a Minimum";
   if max(eval)<0 then print "Solution Is a Maximum";
finish rsm;
```

The following statements run the RSM module and use sample data that represent the result of a designed experiment with two factors. The results are shown in Output 12.7.1

```
/* 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(d,y);
```

Output 12.7.1 Response Surface Regression: Results

Parameter Estimates	
b0	81.222222
b1	1.9666667
b2	0.2166667
b3	-3.933333
b4	-2.225
b5	-1.383333

Critical Factor Values
0.2949376
-0.158881

Response at Critical Value
81.495032

Solution Is a Maximum

Output 12.7.1 displays the parameter estimates from the regression and shows that the values $(0.295, -0.159)$ are values of the factors that result in a maximum response, based on a quadratic fit of the data. The maximum value of the response is predicted to be about 81.5.

Example 12.8: 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. In this example, the normal and logistic distributions are used.

The regression computes parameter estimates by using maximum likelihood via iteratively reweighted least squares, as described in Charnes, Frome, and Yu (1976); Jennrich and Moore (1975); Nelder and Wedderburn (1972). Rows are scaled by the derivative of the distribution, which is the density. The weights are assigned by using the expression $w/p(1-p)$, where w is a count or some other weight.

The following statements define the module BINEST, which computes logistic and probit regressions for binary response models:

```
proc iml ;
/* compute density function (PDF) */
start Density(model, z);
   if upcase(model)='LOGIT'  then
      return( pdf("Logistic", z) );
   else /* "PROBIT" */
      return( pdf("Normal", z) );
finish;

/* compute cumulative distribution function (CDF) */
start Distrib(model, z);
   if upcase(model)='LOGIT'  then
      return( cdf("Logistic", z) );
   else /* "PROBIT" */
      return( cdf("Normal", z) );
finish;

/* routine for estimating binary response models             */
/* model is "logit" or "probit"                              */
/* varNames has the names of the regressor variables         */
start BinEst(nEvents, nTrials, data, model, varNames);
   /* set up design matrix */
   n = nrow(data);
   x = j(n,1,1) || data;                        /* add intercept */
   x = x // x;                                     /* regressors */
   y = j(n,1,1) // j(n,1,0);      /* binary response: 1s and 0s */
   wgt = nEvents // (nTrials-nEvents);          /* count weights */

   parms = "Intercept" || rowvec(varNames);
   k = ncol(x);
   b = j(k,1,0);                               /* starting values */
   oldb = b+1;
   results = j(20, 2+k, .);           /* store iteration history */
   do iter=1 to nrow(results) while(max(abs(b-oldb))>1e-8);
      oldb = b;
      z = x*b;
      p = Distrib(model, z);
      loglik = sum( wgt#((y=1)#log(p) + (y=0)#log(1-p)) );
      results[iter, ] = iter || loglik || b`;
      w = wgt / (p#(1-p));
      f = Density(model, z);
      xx = f#x;
      xpxi = inv(xx`*(w#xx));
      b = b + xpxi*(xx`*(w#(y-p)));
   end;
   idx = loc(results^=.);    /* trim results if few iterations */
   results = shape(results[idx],0,2+ncol(parms));
   colnames = {"Iter" "LogLik"} || parms;
   lbl = "Iteration History: " + model + " Model";
   print results[colname=colnames label=lbl];
```

```
   p0 = sum((y=1)#wgt) / sum(wgt);           /* average response */
   loglik0 = sum( wgt#((y=1)#log(p0) + (y=0)#log(1-p0)) );
   chisq = 2#(loglik-loglik0);
   df = k-1;
   prob = 1 - cdf("ChiSq", chisq, df);
   stats = chisq || df || prob;
   print stats[colname={'ChiSq' 'DF' 'Prob'}
               label='Likelihood Ratio, Intercept-only Model'];

   stderr = sqrt(vecdiag(xpxi));
   tRatio = b/stderr;
   print (parms`)[label="parms"] b stderr tRatio;
finish;
```

The following statements call the BINEST module to compute a logistic regression for data that appear in Cox and Snell (1989, pp. 10–11). The data consist of the number of ingots that are not ready for rolling (nReady) and the total number tested (nTested) for a number of combinations of heating time and soaking time. The results are shown in Output 12.8.1.

```
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};
x = data[, 1:2];
parms = {"Heat" "Soak"};
nReady = data[, 3];
nTotal = data[, 4];

run BinEst(nReady, nTotal, x, "Logit", parms);     /* run logit model  */
```

Output 12.8.1 Logistic Regression: Results

Iteration History: Logit Model

Iter	LogLik	Intercept	Heat	Soak
1	-268.248	0	0	0
2	-76.29481	-2.159406	0.0138784	0.0037327
3	-53.38033	-3.53344	0.0363154	0.0119734
4	-48.34609	-4.748899	0.0640013	0.0299201
5	-47.69191	-5.413817	0.0790272	0.04982
6	-47.67283	-5.553931	0.0819276	0.0564395
7	-47.67281	-5.55916	0.0820307	0.0567708
8	-47.67281	-5.559166	0.0820308	0.0567713

Likelihood Ratio, Intercept-only Model		
ChiSq	DF	Prob
11.64282	2	0.0029634

parms	b	stderr	tRatio
Intercept	-5.559166	1.1196947	-4.964895
Heat	0.0820308	0.0237345	3.4561866
Soak	0.0567713	0.3312131	0.1714042

You can use the LOGISTIC procedure in SAS/STAT software to perform a similar analysis. See the section "Getting Started: Logistic Procedure" in *SAS/STAT User's Guide*.

In a similar way, you can call the BINEST module and request a probit-model regression. The results, which appear in Output 12.8.2, are consistent with results from the PROBIT procedure.

```
run BinEst(nReady, nTotal, x, "Probit", parms);    /* run probit model */
```

Output 12.8.2 Probit Regression: Results

Iteration History: Logit Model

Iter	LogLik	Intercept	Heat	Soak
1	-268.248	0	0	0
2	-76.29481	-2.159406	0.0138784	0.0037327
3	-53.38033	-3.53344	0.0363154	0.0119734
4	-48.34609	-4.748899	0.0640013	0.0299201
5	-47.69191	-5.413817	0.0790272	0.04982
6	-47.67283	-5.553931	0.0819276	0.0564395
7	-47.67281	-5.55916	0.0820307	0.0567708
8	-47.67281	-5.559166	0.0820308	0.0567713

Likelihood Ratio, Intercept-only Model

ChiSq	DF	Prob
11.64282	2	0.0029634

parms	b	stderr	tRatio
Intercept	-5.559166	1.1196947	-4.964895
Heat	0.0820308	0.0237345	3.4561866
Soak	0.0567713	0.3312131	0.1714042

Example 12.9: Linear Programming

You can solve the following general linear programming problem by using the LPSOLVE call:

$$\max \mathbf{c}'\mathbf{x}$$
$$\text{st. } \mathbf{A}\mathbf{x} \leq, =, \geq \mathbf{b}$$
$$\mathbf{x} \geq 0$$

Consider the following product mix example (Hadley 1962). A shop that has three machines, A, B, and C, turns out four different products. 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:

	Product			
Machine	**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 2,000, 8,000, and 5,000 hours, respectively. The products contribute 5.24, 7.30, 8.34, and 4.18 to profit, respectively. What mixture of products can be manufactured to maximize profit?

The following SAS/IML program calls the LPSOLVE routine and displays a summary of the optimization results:

```
proc iml;
names={'product 1' 'product 2' 'product 3' 'product 4'};
/* coefficients of the linear objective function             */
c = {5.24 7.30 8.34 4.18};
/* coefficients of the constraint equation                   */
A = { 1.5  1  2.4  1   ,
      1    5  1    3.5 ,
      1.5  3  3.5  1   };

/* right-hand side of constraint equation                    */
b = { 2000, 8000, 5000};
/* operators: 'L' for <=, 'G' for >=, 'E' for =              */
ops = { 'L', 'L', 'L' };

n=ncol(A);                              /* number of variables */
cntl = j(1,7,.);                             /* control vector */
cntl[1] = -1;               /* 1 for minimum; -1 for maximum */
call lpsolve(rc, value, x, dual, redcost,
             c, A, b, cntl, ops);

print x[r=names L='Optimal Product Mix'];
print value[L='Maximum Profit'];
lhs = A*x;
Constraints = lhs || b;
print Constraints[r={"Machine1" "Machine2" "Machine3"}
                  c={"Actual" "Upper Bound"}
                  L="Time Constraints"];
```

The results from this example are shown in Output 12.9.1. The optimal mix of products is to produce 295 units of product 1, 1,500 units of product 2, no units of product 3, and 58 units of product 4. Manufacturing that product mix results in an optimal profit and also utilizes the maximum availability of the three machines.

Output 12.9.1 Product Mix: Optimal Solution

Optimal Product Mix	
product 1	294.11765
product 2	1500
product 3	0
product 4	58.823529

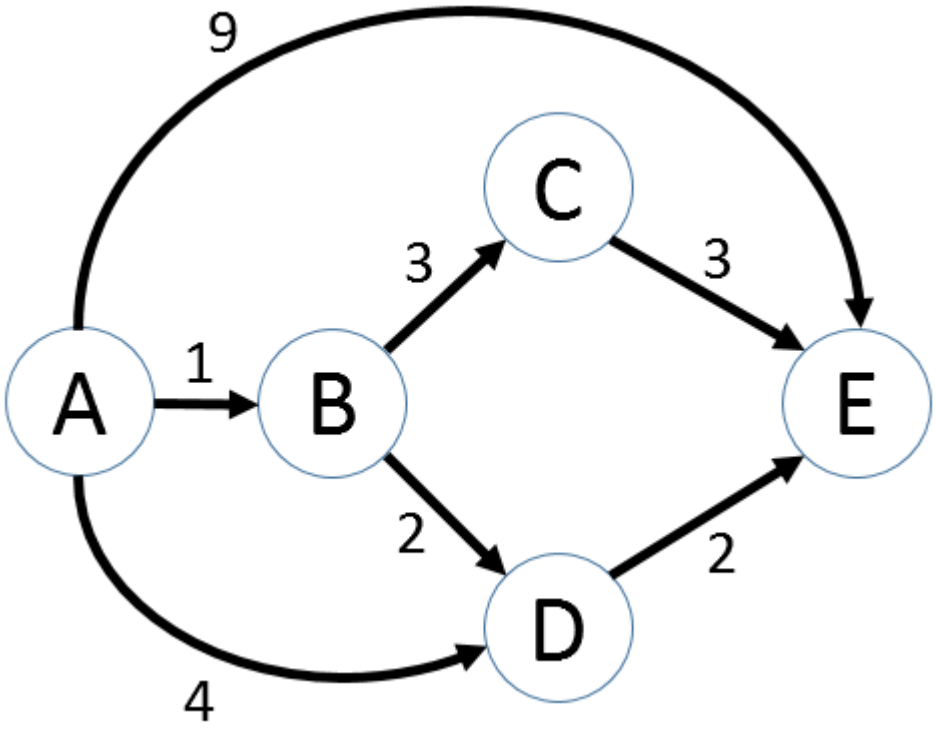
Output 12.9.1 *continued*

Maximum Profit
12737.059

Time Constraints		
	Actual	Upper Bound
Machine1	2000	2000
Machine2	8000	8000
Machine3	5000	5000

The next example shows how to find the minimum cost flow through a network by using linear programming. The network consists of five nodes, named A, B, C, D, and E. Seven arcs connect certain nodes. The arcs are named by the tail and head nodes that define the arc. Output 12.1 shows the network.

Figure 12.1 A Network of Nodes and Arcs



Suppose that some nodes have an excess supply of goods whereas others have a deficit of goods, and suppose that you want to transport goods from the nodes that have excess supply to nodes that have the demand. Each route (arc) between nodes has a cost that is associated with it. In Output 12.1, the cost is represented by a number next to an arc. Given a distribution of goods, what is the optimal way to move goods through the network?

This example sets up the problem and calls the LPSOLVE subroutine to find an optimal solution. In the example, two units of goods are located at node A. One unit needs to be moved to node D; the other unit needs to travel to node E.

The first part of the problem requires generating the node-arc incidence matrix.

```
arcs = { 'ab' 'bd' 'ad' 'bc' 'ce' 'de' 'ae' }; /* decision variables  */
n=ncol(arcs);                                  /* number of variables */
nodes = {'a', 'b', 'c', 'd', 'e'};
inode = substr(arcs, 1, 1);
onode = substr(arcs, 2, 1);
/* coefficients of the constraint equation */
A = j(nrow(nodes), n, 0);
do j = 1 to n;
   A[,j] = (inode[j]=nodes) - (onode[j]=nodes);
end;
```

The matrix **A** constrains the goods to flow through the existing arcs between nodes. A solution to the problem is a vector that contains the number of goods that flow through each arc. The cost of moving goods is a linear function of a solution. The following statements define the supply and demand of goods within the network and call the LPSOLVE subroutine to obtain a solution that minimizes the cost:

```
/* coefficients of the linear objective function             */
cost = { 1 2 4 3 3 2 9 };
/* right-hand side of constraint equation                    */
supply = { 2, 0, 0, -1, -1 };
/* operators: 'L' for <=, 'G' for >=, 'E' for =               */
ops = repeat('E',nrow(nodes),1);

cntl = j(1,7,.);                            /* control vector */
cntl[1] = 1;                /* 1 for minimum; -1 for maximum */
call lpsolve(rc, value, x, dual, redcost,
             cost, A, supply, cntl, ops);

print value[L='Minimum Cost'];
print x[r=arcs L='Optimal Flow'];
```

The solution is shown in Output 12.9.2. The optimal solution is to move both units along arc AB and then along arc BD. One unit stays at node D, while the other proceeds along arc DE. The minimum cost is 8.

Output 12.9.2 Minimum Cost Flow: Optimal Solution

Minimum Cost
8

	Optimal Flow
ab	2
bd	2
ad	0
bc	0
ce	0
de	1
ae	0

Example 12.10: Quadratic Programming

The following quadratic program can be solved by solving an equivalent linear complementarity problem when $\mathbf{H}$ is positive semidefinite:

$$\begin{aligned} &\min \mathbf{c}'\mathbf{x} + \mathbf{x}'\mathbf{H}\mathbf{x}/2 \\ &\text{st. } \mathbf{G}\mathbf{x} \leq, =, \geq \mathbf{b} \\ &\mathbf{x} \geq 0 \end{aligned}$$

This approach is outlined in the discussion of the LCP subroutine.

The following routine solves the quadratic problem:

```
proc iml;
start qp( names, c, H, G, rel, b, activity);
   if min(eigval(h))<0 then do;
      error={'The minimum eigenvalue of the H matrix is negative.',
             'Thus it is not positive semidefinite.',
             'QP is terminating.'};
      print 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 */
   print ({'*************Solution is optimal***************',
           '*********No solution possible*****************',
           ' ', ' ', ' ',
           '**********Solution is numerically unstable*****',
           '***********Not enough memory*******************',
           '**********Number of iterations exceeded********'}[rc+1]);
   activity = z[1:nc];
   objval = c`*activity + activity`*H*activity/2;
   print objval[L='Objective Value'],
```

```
          activity[r=names L= 'Decision Variables'];
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; 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 };

/* constraints: proportions sum to 1; gain at least 10% */
b = { 1 , .10 };
rel = { '=', '>='};
names = 'Asset1':'Asset4';
run qp(names, c, h, g, rel, b, activity);
```

The results in Output 12.10.1 show that the minimum variance portfolio that achieves the 0.10 expected gain is composed of Asset2 and Asset3 in proportions of 0.933 and 0.067, respectively.

Output 12.10.1 Portfolio Selection: Optimal Solution

**********Solution is optimal**********

Objective Value
1.0966667

Decision Variables	
Asset1	0
Asset2	0.9333333
Asset3	0.0666667
Asset4	0

Example 12.11: Regression Quantiles

The technique of estimating parameters in linear models by using 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 $\boldsymbol{\beta}$ in the model

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

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 that are produced by the LAE. The following routine finds this estimate by using linear programming.

This subroutine follows the approach given in Koenker and Bassett (1978); Bassett and Koenker (1982). When $q = 0.5$, this is equivalent to minimizing the sum of the absolute deviations, which is also known as L1 regression. For L1 regression, a faster and more accurate algorithm is available in the SAS/IML LAV subroutine, which is based on the approach given in Madsen and Nielsen (1993). For more information about quantile regression, see the documentation for the QUANTREG procedure in *SAS/STAT User's Guide*.

```
proc iml;
/*-------------------------------------------------------------*/
/* 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                                                 */
/*-------------------------------------------------------------*/
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(0,1,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)) ;
  /*----------------find the optimal solution---------------*/
  cost = j(1,n+m+2,0);
  cost[n+m+1] = 1.0;
  call lpsolve(rc,optimum,p,d,rcost,
               cost,a,r,-1,j(m+2,1,'E'),,l,u);
```

```
  /*---------------- 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) );

  label = 'Estimation for ' + yname;
  desc = q // n // wsum;
  rownames = {'Regression Quantile', 'Number of Observations',
              'Sum of Weighted Absolute Errors'};
  print desc[L=label r=rownames];
  print b[r=variable];
  print X y predict error;
finish rq;
```

The following example uses data on the United States population from 1790 to 1970, and compares the L1 residuals to the least square residuals:

```
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);
/* Compare L1 residuals with least squares residuals  */
/* Compute the least squares residuals                */
LSResid=y-x*inv(x`*x)*x`*y;
L1Resid = resid;
t = z[,2];
create Residuals var{t L1Resid LSResid}; append; close Residuals;
quit;

proc sgplot data=Residuals;
  scatter x=t y=L1Resid / LEGENDLABEL= "L(1) residuals";
```

```
scatter x=t y=LSResid / LEGENDLABEL= "Least squares residuals";
yaxis label="Residuals";
refline 0 / axis=y;
run;
```

The results are shown in Output 12.11.1.

Output 12.11.1 Regression Quantiles: Results

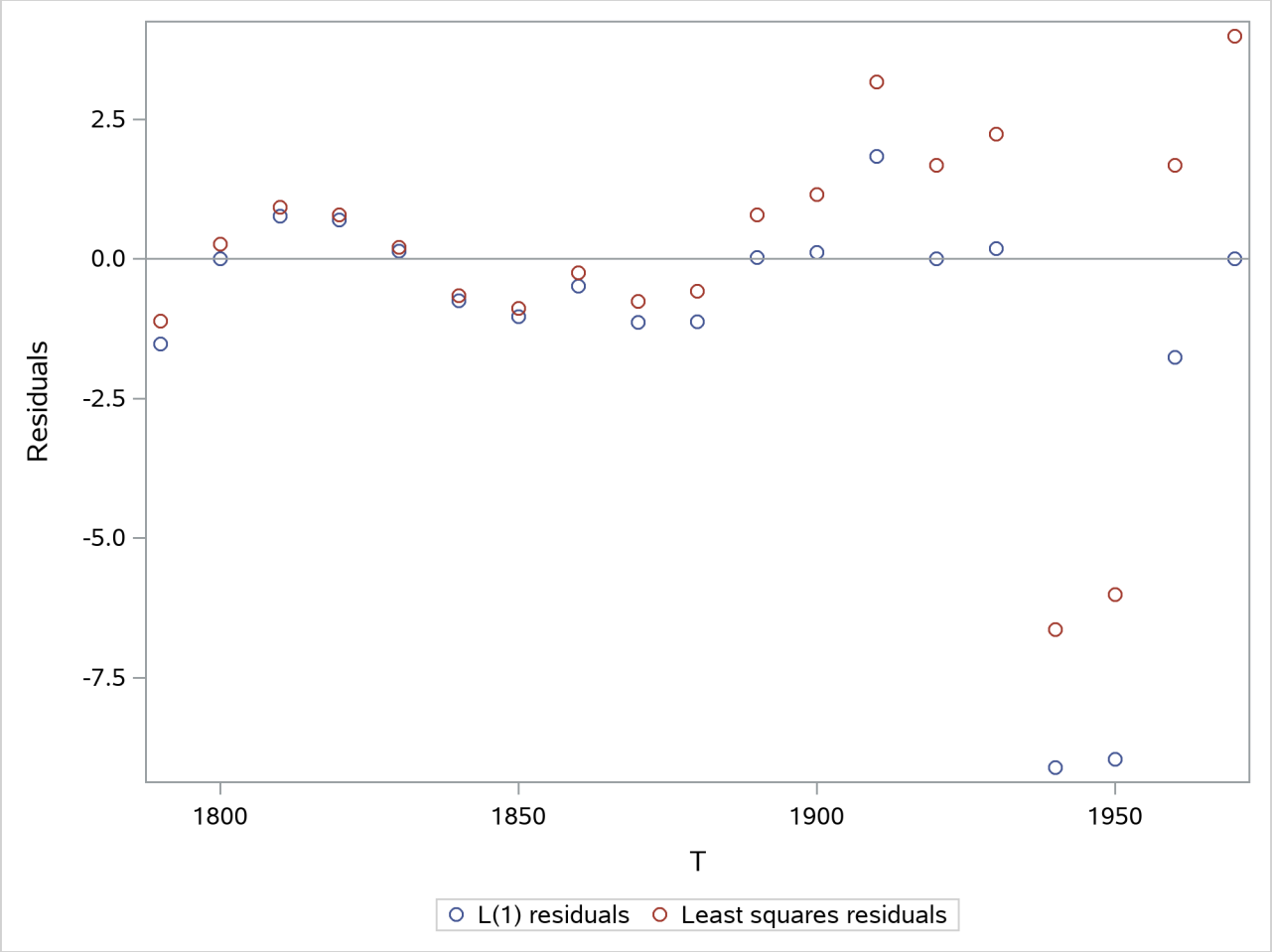
Estimation for pop	
Regression Quantile	0.5
Number of Observations	19
Sum of Weighted Absolute Errors	14.826429

	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	-9.02E-13
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	4.505E-12
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	6.821E-13

The L1 norm (when $q = 0.5$) tends to cause the fit to be better at more points at the expense of causing the fit to be worse at some points, as shown in Output 12.11.2, which shows a plot that compares the L1 residuals with the least squares residuals.

Output 12.11.2 L1 Residuals versus Least Squares Residuals



When q =0.5, the results of this module can be compared with the results of the LAV routine, as follows:

```
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 12.12: Simulations of a Univariate ARMA Process

Simulations of time series with known autoregressive moving average (ARMA) structure are often needed as part of other simulations or as sample data sets for developing skills in time series analysis. You can use the ARMASIM function to simulate a univariate series from an ARMA model. The following module shows some of the computations that are required to simulate data from an ARMA model. The module uses many

SAS/IML functions, including the ARMACOV, HANKEL, PRODUCT, RATIO, TOEPLITZ, and ROOT functions. A short simulated ARMA(1,1) series is shown in Output 12.12.1.

```
proc iml;
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;
call randseed(seed, 1);
y = randfun(1 || (n+q), "Normal");

/* 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.',
              '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;

   phi1 = phi[,p+1:2]`;
   y=startup[,1:p];

   /* Use difference equation to generate remaining values */
   do i = 1 to n-p;
      y = y || (e[,i]-y[,i:i+p-1]*phi1);
   end;

end;
y = y`;
finish armasim; /* ARMASIM */

run armasim(y,10,{1 -0.8},{1 0.5}, 1234321);
print y[label="Simulated Series"];
```

Output 12.12.1 Simulated Series

Simulated Series
0.7480769
1.1786332
0.9478736
1.1041563
4.4027236
4.1308281
3.9587348
5.08593
5.1387329
5.6760884

Example 12.13: 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 that have an ARMA structure. Three estimation methods used by the ARIMA procedure in SAS/ETS software are implemented in the following SAS/IML program. The implemented algorithms are slightly different from those used by PROC ARIMA, but the results should be similar. This example uses the ARMALIK, PRODUCT, and RATIO functions to perform the estimation. Note the interactive nature of this example, illustrating how you can adjust the estimates when they venture outside 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  ---*/
 datalines;
 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
 ;
 run;

proc iml;
/*-------------------------------------------------------------*/
/*  Estimation for regression model with ARMA errors           */
/*  The ARMAREG module uses the following global parameters:   */
/*  x      - matrix of predictors.                             */
/*  y      - response vector.                                  */
/*  iphi   - defines indices of nonzero AR parameters,         */
/*           omit the index 0 which corresponds to the zero    */
/*           order constant one.                               */
/*  itheta - defines indices of nonzero MA parameters,         */
/*           omit the index 0 which corresponds 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         */
/*                                                             */
/*  Invertibility and stationarity are not checked during     */
/*  theestimation process. The parameter array PAR can be     */
/*  modified after running ARMAREG to place estimates         */
/*  in the stationary and invertible regions, and then        */
/*  ARMAREG can 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 can be modified.                             */
/*                                                             */
/*  T-ratios are only approximate and can be undependable,    */
/*  especially for small series.                              */
/*                                                             */
/*  The notation is the same as for the ARMALIK function.     */
/*  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`)[,nrow(y)+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 ss[L='Gradient Matrix'];
   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);
      ss=sssave;
      if sse<oldsse then do;      /* reduction, no iteration */
         lambda=max(lambda/10,1e-12);
         k=21;
      end;
      else do;                                /* no reduction */
                              /* increase lambda and iterate */
         if nopr^=0 then
            print lambda[L='Lambda='] sse oldsse,
                  ss[L='Gradient Matrix'];
         lambda=min(10*lambda,1e12);
         if k=20 then do;
            print ({'GETSS: No improvement in SSE after twenty iterations.',
```

```
                    'Possible Ridge Problem.'});
            return;
         end;
      end;
   end;
   if nopr^=0 then print ss[L='Gradient Matrix'];
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:q,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 par [colname=parmname L='Parameter Starting Values'];
   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 par[colname=parmname L='Parameter Update'];
         print lambda[L='Lambda='];
      end;
```

```
   end;

   sighat=sqrt(sse/(nrow(y)-ncol(par)));
   print sighat[L='Innovation Standard Deviation'];
   ss=sweep(ss,2:ncol(ss));          /* Gaussian elimination */
   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 label='Maximum Likelihood Estimation Results';
   else if ml=-1 then label='Conditional Least Squares Estimation Results';
   else label='Unconditional Least Squares Estimation Results';
   print estm [rowname=parmname colname=pname L=label] ;
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;
close grunfeld;

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 results are shown in Output 12.13.1.

Output 12.13.1 Conditional Least Squares Results

Parameter Starting Values				
AR 1	MA 1	B 1	B 2	B 3
-0.5	0.5	-9.956306	0.0265512	0.1516939

GETSS: No improvement in SSE after twenty iterations.
Possible Ridge Problem.

GETSS: No improvement in SSE after twenty iterations.
Possible Ridge Problem.

GETSS: No improvement in SSE after twenty iterations.
Possible Ridge Problem.

Innovation Standard Deviation
22.653769

Output 12.13.1 *continued*

Conditional Least Squares Estimation Results			
	Estimate	Std. Error	T-Ratio
AR 1	-0.230905	0.3429525	-0.673287
MA 1	0.69639	0.2480617	2.8073252
B 1	-20.87774	31.241368	-0.668272
B 2	0.038706	0.0167503	2.3107588
B 3	0.1216554	0.0441722	2.7541159

```
/*----  With CLS estimates as starting values,  ----*/
/*----  perform ML estimation.                  ----*/
ml=1;
maxiter=10;
run armareg;
```

The results are shown in Output 12.13.2.

Output 12.13.2 Maximum Likelihood Results

Parameter Starting Values				
AR 1	MA 1	B 1	B 2	B 3
-0.230905	0.69639	-20.87774	0.038706	0.1216554

Innovation Standard Deviation
23.039253

Maximum Likelihood Estimation Results			
	Estimate	Std. Error	T-Ratio
AR 1	-0.196224	0.3510868	-0.558904
MA 1	0.6816033	0.2712043	2.5132468
B 1	-26.47514	33.752826	-0.784383
B 2	0.0392213	0.0165545	2.3692242
B 3	0.1310306	0.0425996	3.0758622

Example 12.14: Iterative Proportional Fitting

The classical use of iterative proportional fitting is to adjust frequencies to conform to new marginal totals. You can 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.

This example is a census study that estimates a population distribution according to age and marital status (Bishop, Fienberg, and Holland 1975). Estimates of the distribution are known for the previous year, but only estimates of marginal totals are known for the current year. The following program adjusts the distribution of the previous year to fit the estimated marginal totals of the current year:

```
proc iml;
mod={0.01 15};  /* Stopping criteria */
dim={3 8};      /* Marital status has 3 levels; age has 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};

/* Use known distribution for initial 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 initab[colname=c rowname=r format=8.0
             label='Known Distribution (Previous Year)'],
      fit[colname=c rowname=r format=8.2
          label='Adjusted Estimates (Current Year)'];
```

The results are shown in Output 12.14.1.

Output 12.14.1 Iterative Proportional Fitting: Results

Known Distribution (Previous Year)			
	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

Adjusted Estimates (Current Year)			
	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 12.15: Nonlinear Regression and Specifying a Model at Run Time

This example demonstrates two techniques: The first is an iterative statistical technique for fitting a nonlinear regression model (Hartley 1961). The second is a programming technique for generating modules at run time by using the QUEUE subroutine.

The typical nonlinear regression program defines modules for the regression model and its derivative as follows:

```
start nlfit;
   /* fit model, residuals, and SSE for current parameter values */
finish;
start nlderiv;
   /* evaluate derivatives of model w.r.t parameters */
finish;
```

However, there might be situations in which the regression model is not known until run time. For example, the model might be specified in a file or from an equation that is typed into a dialog box.

In this situation, you can use the QUEUE subroutine to write the NLFIT and NLDERIV modules at run time. You can insert equations for the model and its derivative into the module definitions by using the techniques that are described in the section "Statements That Define and Execute Modules" on page 65 in Chapter 6, "Programming Statements."

The following module specifies the model and its derivatives as character strings:

```
proc iml;

/* _FUN and _DER are text strings that define model and deriv */
/* _parm contains parm names                                  */
/* _beta contains initial values for parameters               */
/* _k is the number of parameters                             */
start nlinit;
   _dep = "uspop";                      /* dependent variable */
   _fun = "a0*exp(a1*time)";    /* nonlinear regression model */
   /* deriv w.r.t. parameters */
   _der  = {"exp(a1*time)", "time*a0*exp(a1*time)"};
   _parm = {"a0", "a1"};               /* names of parameters */
   _beta = {3.9, 0};          /* initial guess for parameters */
   _k= nrow(_parm);                   /* number of parameters */
finish nlinit;
```

All variables are global in scope. Consequently, their names are prefixed by an underscore in order to reduce the likelihood of conflicting with other variables in your program.

The following statements use equations for the model to write the NLFIT and NLDERIV modules:

```
/* Generate the following modules at run time:                */
/* NLFIT: evaluate the model. After RUN NLFIT:                */
/*    _y contains response,                                   */
/*    _p contains predictor after call                        */
/*    _r contains residuals                                   */
/*    _sse contains sse                                       */
/* NLDERIV: evaluate derivs w.r.t params. After RUN NLDERIV:  */
/*    _x contains jacobian                                    */
start nlgen;
   call change(_fun, '*', '#', 0);  /* substitute '#' for '*' */
   call change(_der, '*', '#', 0);

   /* Write the NLFIT module at run time */
   call queue('start nlfit;');
   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;" );

   /* Write the NLDERIV function at run time */
   call queue('start nlderiv; free _NULL_; _x = ');
   do i=1 to _k;
      call queue("(", _der[i], ")||");
   end;
   call queue("_NULL_; finish;");

   call queue("resume;");    /* Pause to compile the functions */
   pause *;
finish nlgen;
```

The program proceeds by calling the NLFIT and NLDERIV modules. The algorithm uses a Gauss-Newton nonlinear regression with step-halving to solve the nonlinear least squares estimation problem:

```
/* Gauss-Newton nonlinear regression with Hartley step-halving */
start nlest;
   run nlfit;                /* f, r, and sse for initial beta */

   /* Gauss-Newton iterations to estimate parameters */
   do _iter=1 to 30 until(_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 nlfit;                        /* compute residual */
      _eps = abs((_lastsse-_sse)) / (_sse+1e-6);
      /* Hartley subiterations */
      do _subit=1 to 10 while(_sse>_lastsse);
         _delta = _delta/2;   /* halve the correction vector */
         _beta = _old+_delta; /* apply the halved correction */
         run nlfit;                         /* find sse et al */
      end;
      /* if no improvement after 10 halvings, exit iter loop */
      if _subit>10 then _eps=0;
   end;

   /* display table of results  */
   if _iter < 30 then do; /* convergence */
      _dfe = nrow(_y) - _k;
      _mse = _sse/_dfe;
      _std = sqrt(vecdiag(_xpxi)#_mse);
      _t = _beta/_std;
      _prob = 1 - cdf("F", _t#_t, 1, _dfe);
      print _beta[label="Estimate"] _std[label="Std Error"]
            _t[label="t Ratio"]     _prob[format=pvalue6.];
      print _iter[label="Iterations"] _lastsse[label="SSE"];
   end;
   else print "Convergence failed";
finish nlest;
```

Finally, the following statements define the data for the problem. The dependent variable is the US population from 1790–1970. The explanatory variable is the number of years since 1790. The program fits an exponential model $y = a_0 \exp(a_1 t) + \epsilon$, where ϵ is an error term. The program estimates the parameters a_0 and a_1. The results are shown in Output 12.15.1.

```
/* main program: run nonlinear regression on data */
uspop = {3929, 5308, 7239, 9638, 12866, 17069, 23191, 31443, 39818,
         50155, 62947, 75994, 91972, 105710, 122775, 131669, 151325,
         179323, 203211}/1000;  /* US population, in thousands */
year = do(1790,1970,10)`;
time = year - 1790;
```

```
run nlinit; /* define strings that define the regression model */
run nlgen;  /* write modules that evaluate the model          */
run nlest;  /* compute param estimates, std errs, and p-values */
```

Output 12.15.1 Nonlinear Regression Estimates

Estimate	Std Error	t Ratio	_prob
11.72004	1.2287001	9.5385689	<.0001
0.0160908	0.0006682	24.081729	<.0001

Iterations	SSE
10	1087.2447

Output 12.15.1 shows that the US population data are best fit by the model $y = 11.7 \exp(0.016 \times t)$.

References

Bassett, G. W., and Koenker, R. (1982). "An Empirical Quantile Function for Linear Models with iid Errors." *Journal of the American Statistical Association* 77:401–415.

Bishop, Y. M. M., Fienberg, S. E., and Holland, P. W. (1975). *Discrete Multivariate Analysis: Theory and Practice*. Cambridge, MA: MIT Press.

Charnes, A., Frome, E. L., and Yu, P. L. (1976). "The Equivalence of Generalized Least Squares and Maximum Likelihood Estimation in the Exponential Family." *Journal of the American Statistical Association* 71:169–171.

Cox, D. R. (1970). *Analysis of Binary Data*. London: Metheun.

Cox, D. R., and Snell, E. J. (1989). *The Analysis of Binary Data*. 2nd ed. London: Chapman & Hall.

Grizzle, J. E., Starmer, C. F., and Koch, G. G. (1969). "Analysis of Categorical Data by Linear Models." *Biometrics* 25:489–504.

Hadley, G. (1962). *Linear Programming*. Reading, MA: Addison-Wesley.

Hartley, H. O. (1961). "The Modified Gauss-Newton Method for the Fitting of Non-linear Regression Functions by Least Squares." *Technometrics* 3:269–280.

Jennrich, R. I., and Moore, R. H. (1975). "Maximum Likelihood Estimation by Means of Nonlinear Least Squares." In *Proceedings of the Statistical Computing Section*, 57–65. Washington, DC: American Statistical Association.

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* 15:107–115.

Koenker, R., and Bassett, G. W. (1978). "Regression Quantiles." *Econometrica* 46:33–50.

Madsen, K., and Nielsen, H. B. (1993). "A Finite Smoothing Algorithm for Linear L_1 Estimation." *SIAM Journal on Optimization* 3:223–235.

Nelder, J. A., and Wedderburn, R. W. M. (1972). "Generalized Linear Models." *Journal of the Royal Statistical Society, Series A* 135:370–384.

Chapter 13
Submitting SAS Statements

Contents

Introduction to Submitting SAS Statements 235
Calling a Procedure 235
Passing Parameters from SAS/IML Matrices 237
Details of Parameter Substitution 238
Creating Graphics in a SUBMIT Block 240
Handling Errors in a SUBMIT Block 242

Introduction to Submitting SAS Statements

In 2002, the IML Workshop application (now known as SAS/IML Studio) introduced a mechanism for submitting SAS statements from programs written in the IMLPlus language. As of SAS/IML 9.22, this feature is also available in PROC IML. This chapter shows you how to submit SAS statements from PROC IML by using the SUBMIT and ENDSUBMIT statements. By using these statements, SAS/IML programmers can call any SAS procedure without losing the state of their PROC IML session.

The statements between the SUBMIT and the ENDSUBMIT statements are referred to as a *SUBMIT block*. The SUBMIT block is processed by the SAS language processor. You can use the SUBMIT statement to call DATA steps, macros, and SAS procedures.

This chapter covers the following topics:

- calling a SAS procedure from PROC IML
- passing parameters into the SUBMIT block
- creating ODS graphics in a SUBMIT block
- handling errors in the SUBMIT block

Calling a Procedure

This section describes how to call a procedure from PROC IML.

Suppose you have data in a SAS/IML matrix that you want to analyze by using a statistical procedure. In general, you can use the following steps to analyze the data:

1. Write the data to a SAS data set by using the CREATE and APPEND statements.
2. Use the SUBMIT statement to call a SAS procedure that analyzes the data.
3. Read the results of the analysis into SAS/IML matrices by using the USE and READ statements.
4. Use the results in further computations.

Of course, if the data are already in a SAS data set, you can skip the first step. Similarly, if you are solely interested in the printed output from a procedure, you can skip the third and fourth steps.

The following example calls the UNIVARIATE procedure in Base SAS software to compute a regression analysis. In order to tell the SAS/IML language interpreter that you want certain statements to be sent to the SAS System, you must enclose your SAS statements with SUBMIT and ENDSUBMIT statements. The ENDSUBMIT statement must appear on a line by itself.

1 The following statements create a SAS data set from data in a vector:

```
proc iml;
q = {3.7, 7.1, 2, 4.2, 5.3, 6.4, 8, 5.7, 3.1, 6.1, 4.4, 5.4, 9.5, 11.2};

create MyData var {q};
append;
close MyData;
```

The MyData data set is used in the rest of this chapter.

2 You can call the UNIVARIATE procedure to analyze these data. The following statements use the ODS SELECT statement to limit the output from the UNIVARIATE procedure. The output is shown in Figure 13.1.

```
submit;
ods select Moments;
proc univariate data=MyData;
   var q;
   ods output Moments=Moments;
run;
endsubmit;
```

Figure 13.1 Output from the UNIVARIATE Procedure

The UNIVARIATE Procedure
Variable: Q

Moments			
N	14	**Sum Weights**	14
Mean	5.86428571	**Sum Observations**	82.1
Std Deviation	2.49387161	**Variance**	6.2193956
Skewness	0.66401924	**Kurtosis**	0.34860956
Uncorrected SS	562.31	**Corrected SS**	80.8521429
Coeff Variation	42.5264343	**Std Error Mean**	0.66651522

3 The previous statements also used the ODS OUTPUT statement to create a data set named Moments that contains the statistics shown in Figure 13.1. In the data set, the first column of Figure 13.1 is contained in a variable named Label1 and the second column is contained in a variable named nValue1. The following statements read those variables into SAS/IML vectors of the same names and print the values:

```
use Moments;
read all var {"nValue1" "Label1"};
close Moments;

labl = "Statistics for " + name(q);
print nValue1[rowname=Label1 label=labl];
```

Figure 13.2 Statistics Read into SAS/IML Vectors

Statistics for q	
N	14
Mean	5.8642857
Std Deviation	2.4938716
Skewness	0.6640192
Uncorrected SS	562.31
Coeff Variation	42.526434

4 By using this technique, you can read the value of any statistic that is created by any SAS procedure. You can then use these values in subsequent computations in PROC IML. For example, if you want to standardize the **y** vector, you can use the mean and standard deviation as computed by the UNIVARIATE procedure, as shown in the following statements:

```
mean = nValue1[2];
stddev = nValue1[3];
stdQ = (q - mean)/stddev;
```

Passing Parameters from SAS/IML Matrices

The SUBMIT statement enables you to substitute the values of a SAS/IML matrix into the statements that are submitted to the SAS System. For example, the following program calls the UNIVARIATE procedure to analyze data in the MyData data set that was created in the section "Calling a Procedure" on page 235. The program submits SAS statements that are identical to the SUBMIT block in that section:

```
table = "Moments";
varName = "q";

submit table varName;
ods select &table;
```

```
proc univariate data=MyData;
   var &varName;
   ods output &table=&table;
run;
endsubmit;
```

You can list the names of SAS/IML matrices in the SUBMIT statement and refer to the contents of those matrices inside the SUBMIT block. The syntax is reminiscent of the syntax for macro variables: an ampersand (&) preceding an expression means "substitute the value of the expression." However, the substitution takes place before the SUBMIT block is sent to the SAS System; no macro variables are actually created.

You can substitute values from character or numeric matrices and vectors. If **x** is a vector, then **&x** lists the elements of **x** separated by spaces. For example, the following statements compute trimmed means for three separate values of the TRIM= option:

```
table = "TrimmedMeans";
varName = "q";
n = {1, 3, 5};          /* number of observations to trim */

submit table varName n;
ods select &table;
proc univariate data=MyData trim=&n;
   var &varName;
run;
endsubmit;
```

The output is shown in Figure 13.3. The values in the column labeled "Number Trimmed in Tail" correspond to the values in the **n** matrix. These values were substituted into the TRIM= option in the PROC UNIVARIATE statement.

Figure 13.3 Statistics Read into SAS/IML Vectors

The UNIVARIATE Procedure
Variable: Q

Trimmed Means

Percent Trimmed in Tail	Number Trimmed in Tail	Trimmed Mean	Std Error Trimmed Mean	95% Confidence Limits		DF	t for H0: Mu0=0.00	Pr > |t|
7.14	1	5.741667	0.664486	4.279142	7.204191	11	8.64076	<.0001
21.43	3	5.575000	0.587204	4.186483	6.963517	7	9.49414	<.0001
35.71	5	5.625000	0.408613	4.324612	6.925388	3	13.76609	0.0008

Details of Parameter Substitution

The SUBMIT statement supports two kinds of parameter substitution: full substitution and specific substitution.

Full Substitution

If you want to substitute many values into a SUBMIT block, it can be tedious to explicitly list the name of every SAS/IML matrix that you reference. You can use an asterisk (*) in the SUBMIT statement as a "wildcard character" to indicate that all SAS/IML matrices are available for parameter substitution. This is called *full substitution* and is shown in the following statements:

```
proc iml;
DSName = "Sashelp.Class";
NumObs = 1;

submit *;
proc print data=&DSName(obs=&NumObs);
run;
endsubmit;
```

Figure 13.4 Full Substitution

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69	112.5

If the SUBMIT block contains a parameter reference (that is, a token that begins with an ampersand (&) for which there is no matching SAS/IML matrix, the parameter reference is not modified prior to being sent to the SAS language processor. In this way, you can reference SAS macro variables in a SUBMIT block.

Specific Substitution

A SUBMIT statement that contains an explicit list of parameters is easier to understand than a SUBMIT statement that contains only the asterisk wildcard character (*). Specifying an explicit list of parameters is called *specific substitution*. These—and only these—parameters are used to make substitutions into the SUBMIT block.

```
proc iml;
DSName = "Sashelp.Class";
NumObs = 2;

submit DSName NumObs;
proc print data=&DSName(obs=&NumObs);
run;
endsubmit;
```

Figure 13.5 Specific Substitution

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0

If the SUBMIT block contains a parameter reference (that is, a token that begins with an ampersand (&) for which there is no matching parameter, the parameter reference is not modified prior to being sent to the SAS language processor. In this way, you can reference SAS macro variables in a SUBMIT block.

With specific substitution, you have additional options for specifying the value of a parameter. You can use any of the following ways to specify the value of a parameter:

- Specify the name of a SAS/IML matrix to use for the value of a parameter, as shown in the following statements:

```
s = "Sashelp.Class";   n = 2;

submit DSName=s NumObs=n;
proc print data=&DSName(obs=&NumObs);
run;
endsubmit;
```

- Specify a literal value to use for the value of a parameter, as shown in the following statements:

```
submit DSName="Sashelp.Class" NumObs=2;
proc print data=&DSName(obs=&NumObs);
run;
endsubmit;
```

- Specify a matrix expression that is enclosed in parentheses, as shown in the following statements:

```
libref = "Sashelp";
fname  = "Class";
NumObs = 2;

submit DSName=(libref+"."+fname) NumObs;
proc print data=&DSName(obs=&NumObs);
run;
endsubmit;
```

Creating Graphics in a SUBMIT Block

If you use the SUBMIT statement to call a SAS procedure that creates a graph, that graph is sent to the current ODS destination. The following statements call the UNIVARIATE procedure, which creates a histogram as part of the analysis:

```
ods graphics on;

proc iml;
msg1 = "First PRINT Statement in PROC IML";
msg2 = "Second PRINT Statement in PROC IML";
print msg1;

submit;
ods select Moments Histogram;
```

```
proc univariate data=Sashelp.Class;
   var Height;
   histogram / kernel;
run;
endsubmit;

print msg2;
ods graphics off;
```

When you run the program, the PROC UNIVARIATE output is interleaved with the PROC IML output. The output from the program is shown in Figure 13.6 through Figure 13.8.

Figure 13.6 Output from PROC IML and from SUBMIT Block

msg1
First PRINT Statement in PROC IML

The UNIVARIATE Procedure
Variable: Height

Moments			
N	19	**Sum Weights**	19
Mean	62.3368421	**Sum Observations**	1184.4
Std Deviation	5.12707525	**Variance**	26.2869006
Skewness	-0.2596696	**Kurtosis**	-0.1389692
Uncorrected SS	74304.92	**Corrected SS**	473.164211
Coeff Variation	8.22479143	**Std Error Mean**	1.17623173

Figure 13.7 Graphic Created in a SUBMIT Block

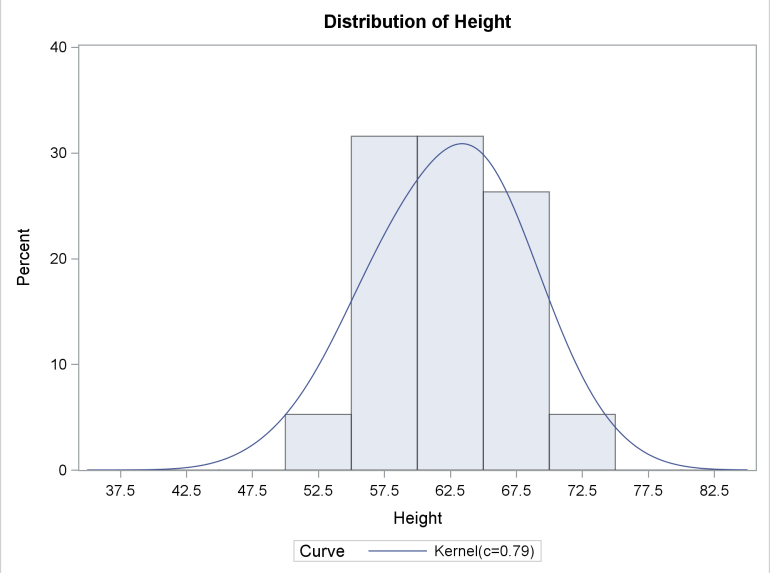


Figure 13.8 Further PROC IML Output

msg2
Second PRINT Statement in PROC IML

Handling Errors in a SUBMIT Block

After executing a SUBMIT block, PROC IML continues to execute the remaining statements in the program. However, if there is an error in the SUBMIT block, it might make sense to abort the program or to handle the error in some other way.

The OK= option in the SUBMIT statement provides a limited form of error handling. If you specify the OK= option, then PROC IML sets a matrix to the value 1 if the SUBMIT block executes without error. Otherwise, the matrix is set to the value 0.

The following statements contain an error in a SUBMIT block: two letters are transposed when specifying the name of a data set. Consequently, the **isOK** matrix is set to 0, and the program handles the error.

```
DSName = "Sashelp.calss";   /* mistyped name; data set does not exist */

submit DSName / ok=isOK;
proc univariate data=&DSName;
   var Height;
   ods output Moments=Moments;
run;
endsubmit;

if isOK then do;           /* handle the no-error case */
   use Moments;
   read all var {"nValue1"} into m;
   close Moments;
   skewness = m[4];        /* get statistic from procedure output */
end;
else
   skewness = .;           /* handle an error */

print skewness;
```

Figure 13.9 The Result of Handling an Error in a SUBMIT Block

skewness
.

Chapter 14
Calling Functions in the R Language

Contents

Overview of Calling Functions in the R Language **243**
Installing the R Statistical Software **244**
The RLANG System Option **244**
Submit R Statements **245**
Transferring Data between SAS and R Software **246**
Transfer from a SAS Source to an R Destination 247
Transfer from an R Source to a SAS Destination 247
Call an R Analysis from PROC IML **248**
Using R to Analyze Data in SAS/IML Matrices 248
Using R to Analyze Data in a SAS Data Set 250
Passing Parameters to R **250**
Call R Packages from PROC IML **251**
Call R Graphics from PROC IML **254**
Handling Errors from R **256**
Details of Data Transfer **256**
Numeric Data Types 256
Logical Data Types 256
Unsupported Data Types 256
Special Numeric Values 256
Date, Time, and Datetime Values 257
Time Series Data 258
Data Structures 258
Differences from SAS/IML Studio **258**

Overview of Calling Functions in the R Language

R is a freely available language and environment for statistical computing and graphics. Like the SAS/IML language, the R language has features suitable for developers of statistical algorithms: the ability to manipulate matrices and vectors, a large number of built-in functions for computing statistical quantities, and the capability to extend the basic function library by writing user-defined functions. There are also a large number of user-contributed packages in R that implement specialized computations.

In 2009, the SAS/IML Studio application introduced a mechanism for calling R functions from programs written in the IMLPlus language. As of SAS/IML 9.22, this feature is available in PROC IML. This chapter shows you how to call R functions from PROC IML by using the SUBMIT and ENDSUBMIT statements.

This chapter describes how to configure the SAS system so that you can call functions in the R language. The chapter also describes how to do the following:

- transfer data to R
- call R functions from PROC IML
- transfer the results from R to a number of SAS data structures

You cannot call R from the free SAS University Edition. The SAS University Edition runs on a virtual machine that does not have R installed.

Installing the R Statistical Software

SAS does not distribute R software. In order to call R software, you must first install R on the same computer that runs SAS software. If you access a SAS workspace server through client software such as SAS Enterprise Guide, then R must be installed on the SAS server.

You can download R from The Comprehensive R Archive Network Web site: http://cran.r-project.org. If you experience problems installing R, consult the R FAQ: http://cran.r-project.org/doc/FAQ/R-FAQ.html. SAS Technical Support does not provide support for installing or configuring third-party software.

In SAS/IML, the interface to R is supported on computers that run a 32-bit or 64-bit Windows operating system or Linux operating systems. If you are using SAS software in a 64-bit Linux environment, you must download a 64-bit binary distribution of R. Otherwise, download a 32-bit binary distribution.

The document "Installing R on Linux Operating Systems" is available on support.sas.com and includes pointers for installing R on Linux that it works with the SAS interface to R.

The RLANG System Option

The RLANG system option determines whether you have permission to call R from the SAS system. You can determine the value of the RLANG option by submitting the following SAS statements:

```
proc options option=RLANG;
run;
```

The result is one of the following statements in the SAS log:

NORLANG **Do not support access to R language interfaces**
If the SAS log contains this statement, you do not have permission to call R from the SAS system.

RLANG **Support access to R language interfaces**
If the SAS log contains this statement, you can call R from the SAS system.

The RLANG option can be changed only at SAS start-up. In order to call R, the SAS system must be launched with the -RLANG option. (It is often convenient to insert this option in a SASV9.CFG file.) For security reasons, some system administrators configure the SAS system to start with the -NORLANG option. The RLANG option is similar to the XCMD option in that both options enable SAS users to potentially write or delete important data and system files.

If you attempt to submit R statements on a system that was not launched with the -RLANG option, you get the following error message:

> ERROR: The RLANG system option must be specified in the SAS configuration file or on the SAS invocation command line to enable the submission of R language statements.

Some operating systems do not support the RLANG system option. The RLANG system option is currently supported for the Windows and Linux operating systems. If you attempt to submit R statements on a host that does not support the RLANG option, you get the following warning message:

> WARNING: SAS option RLANG is not supported on this host.

Submit R Statements

In order to call R from the SAS system, the R statistical software must be installed on the SAS workspace server and the RLANG system option must be enabled. (See the section "The RLANG System Option" on page 244.)

Chapter 13, "Submitting SAS Statements," describes how to submit SAS statements from PROC IML. Submitting R statements is similar. You use a SUBMIT statement, but add the R option: SUBMIT / R. All statements in the program between the SUBMIT statement and the next ENDSUBMIT statement are sent to R for execution. The ENDSUBMIT statement must appear on a line by itself.

The simplest program that calls R is one that does not transfer any data between the two environments. In the following program, SAS/IML is used to compute the product of a matrix and a vector. The result is printed. Then the SUBMIT statement with the R option is used to send an equivalent set of statements to R.

```
proc iml;
/* Comparison of matrix operations in IML and R */
print "----------  SAS/IML Results  -----------------";
x = 1:3;                                /* vector of sequence 1,2,3 */
m = {1 2 3, 4 5 6, 7 8 9};              /* 3 x 3 matrix */
q = m * t(x);                           /* matrix multiplication */
print q;
```

```
print "-------------  R Results  ---------------------";
submit / R;
  rx <- matrix( 1:3, nrow=1)              # vector of sequence 1,2,3
  rm <- matrix( 1:9, nrow=3, byrow=TRUE) # 3 x 3 matrix
  rq <- rm %*% t(rx)                      # matrix multiplication
  print(rq)
endsubmit;
```

The printed output from R is automatically routed to the SAS output window, as shown in Figure 14.1. As expected, the result of the computation is the same in R as in SAS/IML.

Figure 14.1 Output from SAS/IML and R

---------- SAS/IML Results ----------

q
14
32
50

---------- R Results ----------

```
     [,1]
[1,]   14
[2,]   32
[3,]   50
```

Notice that SAS/IML automatically starts the R process when it processes the first SUBMIT / R statement. The R software does not terminate when the ENDSUBMIT statement is reached. Instead, R continues to run until PROC IML exits. This means that you can call R multiple times in a single SAS/IML program and any R variables that were created during an earlier invocation are available for subsequent invocations. Similarly, if you load a package, the package remains loaded until PROC IML exits.

No startup options are sent to R by the SUBMIT statement. In particular, you cannot set command-line options such as **`--no-restore`** or **`--vanilla`**. However, you can set some command-line options by using environment variables. For example, instead of using the command line options **`--min-nsize`** and **`--min-vsize`** you can specify values for the environment variables R_NSIZE and R_VSIZE, respectively, before R starts. You can set R-specific environment variables in the file **`.Renviron`** or **`~/.Renviron`**. For more details about the R startup process, see the R documentation for **`?Startup`**.

Transferring Data between SAS and R Software

Many research statisticians take advantage of special-purpose functions and packages written in the R language. When you call an R function, the data must be accessible to R, either in a data frame or in an R matrix. This section describes how you can transfer data and statistical results (for example, fitted values or parameter estimates) between SAS and R data structures.

You can transfer data to and from the following SAS data structures:

- a SAS data set in a libref
- a SAS/IML matrix
- a SAS/IML table

In addition, you can transfer data to and from the following R data structures:

- an R data frame
- an R matrix

Transfer from a SAS Source to an R Destination

Table 14.1 summarizes the subroutines that copy data from a SAS source to an R destination. For more information, see the section "Details of Data Transfer" on page 256.

Table 14.1 Transferring from a SAS Source to an R Destination

Subroutine	SAS Source	R Destination
ExportDataSetToR	SAS data set	R data frame
ExportMatrixToR	SAS/IML matrix	R matrix
ExportTableToR	SAS/IML table	R data frame

As a simple example, the following program transfers a data set from the Sashelp libref into an R data frame named df. The program then submits an R statement that displays the names of the variables in the data frame.

```
proc iml;
call ExportDataSetToR("Sashelp.Class", "df" );
submit / R;
   names(df)
endsubmit;
```

The R **names** function produces the output shown in Figure 14.2.

Figure 14.2 Result of Sending Data to R

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

Transfer from an R Source to a SAS Destination

You can transfer data and results from R data frames or matrices to a SAS data set, a SAS/IML matrix, or a SAS/IML table. Table 14.2 summarizes the frequently used methods that copy from an R source to a SAS destination.

Table 14.2 Transferring from an R Source to a SAS Destination

Subroutine	R Source	SAS Destination
ImportDataSetFromR	R expression	SAS data set
ImportMatrixFromR	R expression	SAS/IML matrix
ImportTableFromR	R expression	SAS/IML table

The next section includes an example of calling an R analysis. Some of the results from the analysis are then transferred into SAS/IML matrices.

The result of an R analysis can be a complicated structure. In order to transfer an R object via the previously mentioned methods and modules, the object must be coercible to a data frame. (The R object `m` can be coerced to a data frame provided that the function `as.data.frame(m)` succeeds.) There are many data structures that cannot be coerced into data frames. As the example in the next section shows, you can use R statements to extract and transfer simpler objects.

Call an R Analysis from PROC IML

You can use the techniques in Chapter 13, "Submitting SAS Statements," to perform a linear regression by calling a regression procedure (such as REG, GLM, or MIXED) in SAS/STAT software. This section presents examples of submitting statements to R to perform a linear regression. The first example performs a linear regression on data that are transferred from SAS/IML vectors. The second example performs an identical analysis on data that are transferred from a SAS data set.

Using R to Analyze Data in SAS/IML Matrices

The program in this section consists of four parts:

1. Read the data into SAS/IML vectors.
2. Transfer the data to R.
3. Call R functions to analyze the data.
4. Transfer the results of the analysis into SAS/IML vectors.

1 Read the data. The following statements read the Weight and Height variables from the Sashelp.Class data set into SAS/IML vectors with the same names:

```
proc iml;
use Sashelp.Class;
read all var {Weight Height};
close Sashelp.Class;
```

2 Transfer the data to R. The following statements run the ExportMatrixToR subroutine in order to transfer data from a SAS/IML matrix into an R matrix. The names of the corresponding R vectors that contain the data are **w** and **h**.

```
/* send matrices to R */
call ExportMatrixToR(Weight, "w");
call ExportMatrixToR(Height, "h");
```

3 Call R functions to perform some analysis. The SUBMIT statement with the R option is used to send statements to R. Comments in R begin with a hash mark (#, also called a number sign or a pound sign).

```
submit / R;
   Model    <- lm(w ~ h, na.action="na.exclude")          # a
   ParamEst <- coef(Model)                                # b
   Pred     <- fitted(Model)
   Resid    <- residuals(Model)
endsubmit;
```

The R program consists of the following steps:

a. The **lm** function computes a linear model of w as a function of h. The **na.action=** option specifies how the model handles missing values (which in R are represented by NA). In particular, the **na.exclude** option specifies that the **lm** function should not omit observations with missing values from residual and predicted values. This option makes it easier to merge the R results with the original data when the data contain missing values.

b. Various information is retrieved from the linear model and placed into R vectors named **ParamEst**, **Pred**, and **Resid**.

4 Transfer the data from R. The ImportMatrixFromR subroutine transfers the **ParamEst** vector from R into a SAS/IML vector named **pe**. This vector is printed by the SAS/IML PRINT statement. The predicted values (**Pred**) and residual values (**Resid**) can be transferred similarly. The parameter estimates are used to compute the predicted values for a series of hypothetical heights, as shown in Figure 14.3.

```
call ImportMatrixFromR(pe, "ParamEst");
print pe[r={"Intercept" "Height"}];

ht = T( do(55, 70, 5) );
A = j(nrow(ht),1,1) || ht;
pred_wt = A * pe;
print ht pred_wt;
```

Figure 14.3 Results from an R Analysis

	pe
Intercept	-143.0269
Height	3.8990303

Figure 14.3 *continued*

ht	pred_wt
55	71.419746
60	90.914898
65	110.41005
70	129.9052

You cannot directly transfer the contents of the **Model** object. Instead, various R functions are used to extract portions of the **Model** object, and those simpler pieces are transferred.

Using R to Analyze Data in a SAS Data Set

As an alternative to the data transfer statements in the previous section, you can call the ExportDataSetToR subroutine to transfer the entire SAS data set to an R data frame. For example, you could use the following statements to create an R data frame named Class and to model the Weight variable:

```
call ExportDataSetToR("Sashelp.Class", "Class");
submit / R;
   Model <- lm(Weight ~ Height, data=Class, na.action="na.exclude")
endsubmit;
```

The R language is case-sensitive so you must use the correct case to refer to variables in a data frame. You can use the CONTENTS function in the SAS/IML language to obtain the names and capitalization of variables in a SAS data set. If the data are in a SAS/IML table, you can use the TableGetVarName function to obtain the column names.

Passing Parameters to R

The SUBMIT statement supports parameter substitution from SAS/IML matrices as detailed in the section "Passing Parameters from SAS/IML Matrices" on page 237. For example, you can substitute the names of analysis variables into a SUBMIT block by using the following statements:

```
YVar = "Weight";
XVar = "Height";
submit XVar YVar / R;
   Model <- lm(&YVar ~ &XVar, data=Class, na.action="na.exclude")
   print (Model$call)
endsubmit;
```

Figure 14.4 shows the result of the **print(Model$call)** statement. The output shows that the values of the **YVar** and **XVar** matrices were substituted into the SUBMIT block.

Figure 14.4 Parameter Substitutions in a SUBMIT Block

```
lm(formula = Weight ~ Height, data = Class, na.action = "na.exclude")
```

Call R Packages from PROC IML

You do not need to do anything special to call an R package. Provided that an R package is installed, you can call **library(***package***)** from inside a SUBMIT block to load the package. You can then call the functions in the package.

The example in this section calls an R package and imports the results into a SAS data set. This example is similar to the example in the section "Creating Graphics in a SUBMIT Block" on page 240, which calls the UNIVARIATE procedure to create a kernel density estimate. The program in this section consists of the following steps:

1. Define the data and transfer the data to R.
2. Call R functions to analyze the data.
3. Transfer the results of the analysis into SAS/IML vectors.

1 Define the data in the SAS/IML vector **q** and then transfer the data to R by using the ExportMatrixToR subroutine. In R, the data are stored in a vector named **rq**.

```
proc iml;
q = {3.7, 7.1, 2, 4.2, 5.3, 6.4, 8, 5.7, 3.1, 6.1, 4.4, 5.4, 9.5, 11.2};
RVar = "rq";
call ExportMatrixToR( q, RVar );
```

2 Load the KernSmooth package. Because the functions in the KernSmooth package do not handle missing values, the nonmissing values in **q** must be copied to a matrix **p**. (There are no missing values in this example.) The Sheather-Jones plug-in bandwidth is computed by calling the **dpik** function in the KernSmooth package. This bandwidth is used in the **bkde** function (in the same package) to compute a kernel density estimate.

```
submit RVar / R;
   library(KernSmooth)
   idx <-which(!is.na(&RVar))        # must exclude missing values (NA)
   p <- &RVar[idx]                   #    from KernSmooth functions
   h = dpik(p)                       # Sheather-Jones plug-in bandwidth
   est <- bkde(p, bandwidth=h)       # est has 2 columns
endsubmit;
```

3 Copy the results into a SAS data set or a SAS/IML matrix, and perform additional computations. For example, the following statements use the trapezoidal rule to numerically estimate the density that is contained in the tail of the density estimate of the data:

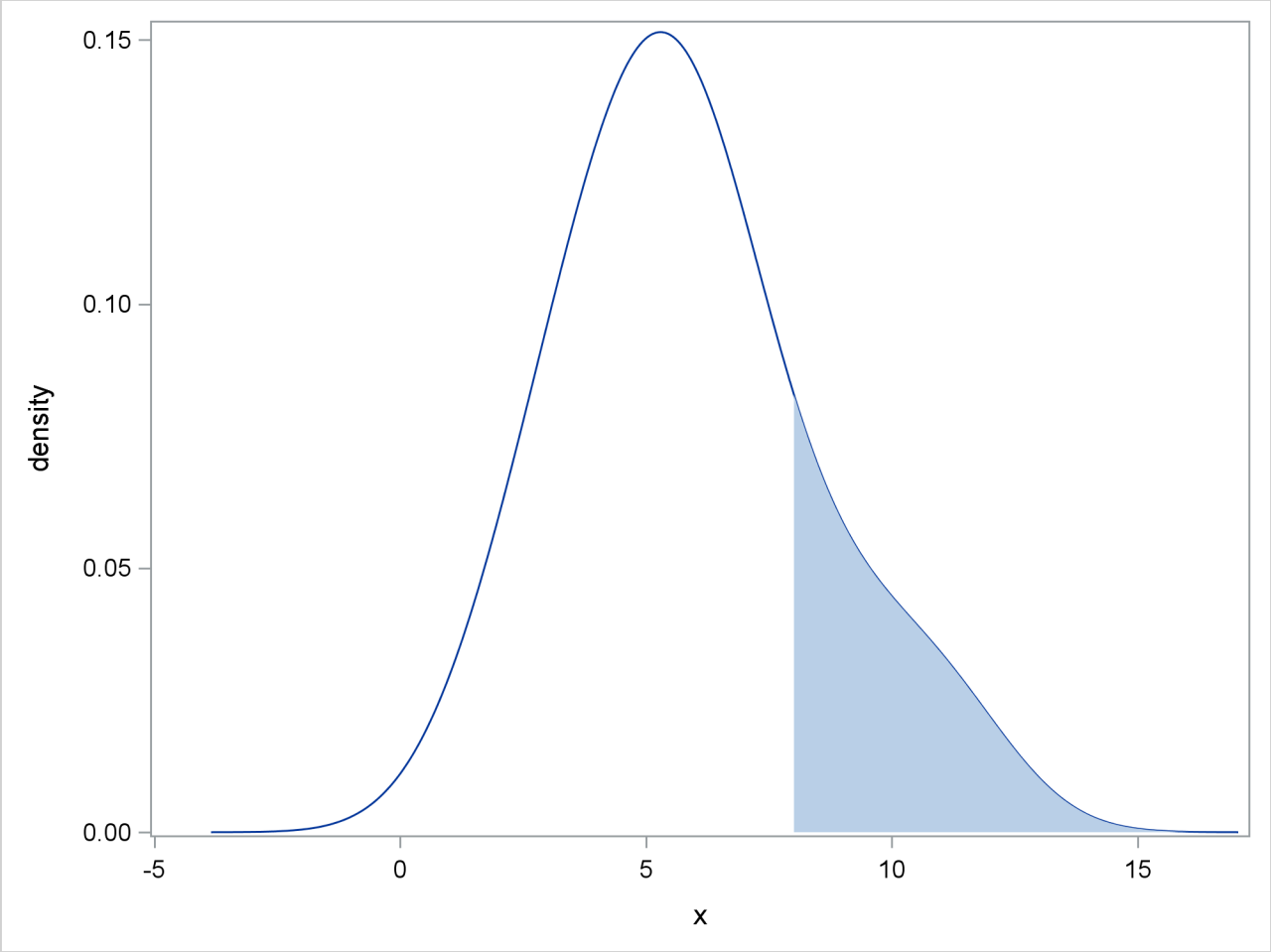
```
call ImportMatrixFromR( m, "est" );
/* estimate the density for q >= 8 */
x = m[,1];                 /* x values for density */
idx = loc( x>=8 );         /* find values x >= 8 */
y = m[idx, 2];             /* extract corresponding density values */

/* Use the trapezoidal rule to estimate the area under the density curve.
   The area of a trapezoid with base w and heights h1 and h2 is
   w*(h1+h2)/2. */
w = m[2,1] - m[1,1];
h1 = y[1:nrow(y)-1];
h2 = y[2:nrow(y)];
Area = w * sum(h1+h2) / 2;
print Area;
```

The numerical estimate for the conditional density is shown in Figure 14.5. The estimate is shown graphically in Figure 14.6, where the conditional density corresponds to the shaded area in the figure. Figure 14.6 was created by using the SGPLOT procedure to display the density estimate computed by the R package.

Figure 14.5 Computation That Combines SAS/IML and R Computations

Area
0.2118117

Figure 14.6 Estimated Density for $x \geq 8$



Call R Graphics from PROC IML

R can create graphics in a separate window which, by default, appears on the same computer on which R is running. If you are running PROC IML and R locally on your desktop or laptop computer, you can display R graphics. However, if you are running client software that connects with a remote SAS server that is running PROC IML and R, then R graphics might be disabled.

The following statements describe some common scenarios for running a PROC IML program:

- If you run PROC IML through a SAS Display Manager Session (DMS), you can create R graphics from your PROC IML program. The graph appears in the standard R graphics window.
- If you run PROC IML through SAS Enterprise Guide, the display of R graphics is disabled because, in general, the SAS server (and therefore R) is running on a different computer than the SAS Enterprise Guide application.
- If you run PROC IML from interactive line mode or from batch mode, then R graphics are disabled.

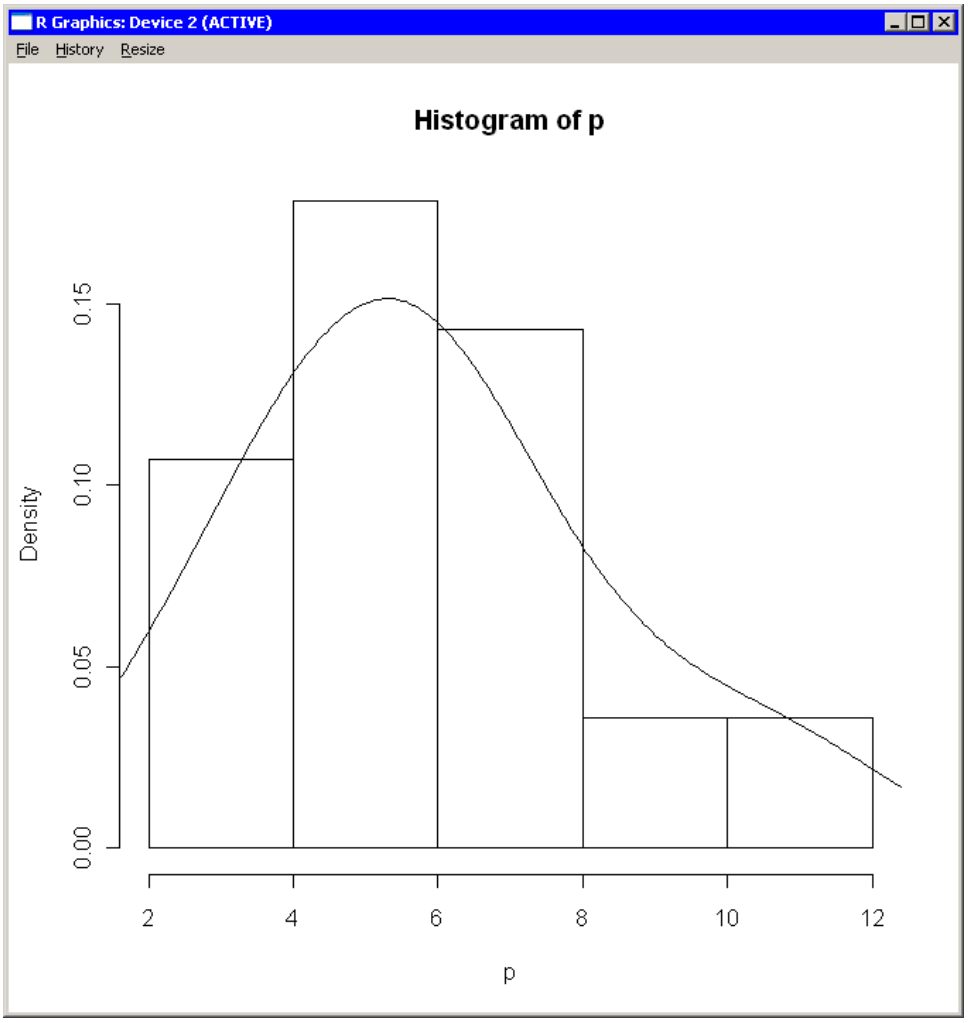
You can determine whether R graphics are enabled by calling the **interactive** function in the R language.

For example, the previous section used R to compute a kernel density estimate for some data. If you are running PROC IML through SAS DMS, you can create a histogram and overlay the kernel density estimate by using the following statements:

```
submit / R;
   hist(p, freq=FALSE) # histogram
   lines(est)          # kde overlay
endsubmit;
```

The **hist** function creates a histogram of the data in the **p** matrix, and the **lines** function adds the kernel density estimate contained in the **est** matrix. The R graphics window contains the histogram, which is shown in Figure 14.7.

Figure 14.7 R Graphics



Handling Errors from R

If you submit R code that causes an error, you can attempt to handle the error by using the OK= option in the SUBMIT statement, as described in the section "Handling Errors in a SUBMIT Block" on page 242.

Details of Data Transfer

This section describes how data are transferred between SAS and R software. It includes a discussion of numerical data types, missing values, and data that represent dates and times.

Numeric Data Types

R can store numeric data in either an integer or a double-precision data type. When transferring R data to a SAS data type, integers types are converted to double precision.

Logical Data Types

R provides a logical data type for storing the values **TRUE** and **FALSE**. When logical data are transferred to a SAS data type, the value **TRUE** is converted to the number 1 and the value **FALSE** to the number 0.

Unsupported Data Types

R provides two data types that are not converted to a SAS data type: complex and raw. It is an error to attempt to transfer data stored in either of these data types to a SAS data type.

Special Numeric Values

The R language has four symbols that are used to represent special numerical values.

- The symbol **`NA`** represents a missing value.
- The symbol **`Inf`** represents positive infinity.
- The symbol **`-Inf`** represents negative infinity.
- The symbol **`NaN`** represents a "NaN," which is a floating-point value that represents an undefined value such as the result of the division $0/0$.

The SAS language has 28 symbols that are used to represent special numerical values.

- The symbol `.` represents a generic missing value.
- The symbols `.A`–`.Z` and `._` are also missing values. Some applications use `.I` to represent positive infinity and use `.M` to represent negative infinity.

The following table shows how special numeric values in R are converted to SAS missing values:

Value in R	SAS Missing Value
Inf	.I
–Inf	.M
NA	.
NaN	.

The following table shows how SAS missing values are converted when data are transferred to R:

SAS Missing Value	Value in R
.I	Inf
.M	–Inf
All others	NA

Date, Time, and Datetime Values

R supports date and time data differently than does SAS software. In SAS software, variables that represent dates or times are assigned a format such as DATE9. or TIME5. In R, classes are used to represent dates and times.

When a variable in a SAS data set is transferred to R software, the variable's format is examined and the following occurs:

- If the format is in the family of date formats (for example, DATE*w*.*d*), the variable in R is assigned the "Date" class.
- If the format is in the family of datetime formats (for example, DATETIME*w*.*d*) or time formats (for example, TIME*w*.*d*), the variable in R is assigned the "POSIXct" and "POSIXt" classes.
- In all other cases, the variable in R is assigned the "numeric" class.

When a variable in an R data frame is transferred to SAS software, the variable's class is examined and the following occurs:

- If the variable's class is "Date," the corresponding SAS variable is assigned the DATE9. format.
- If the variable's class is "POSIXt," the corresponding SAS variable is assigned the DATETIME19. format.
- In all other cases, the SAS variable is not assigned a format.

Time Series Data

In SAS, the sampling times for time series data are often stored in a separate variable. In R, the sampling times for a time series object are specified by the **tsp** attribute. When a time series object in R is transferred to SAS software, the following occurs:

- The R **time** function is used to generate a vector of the times at which the time series is sampled.
- A new variable named *VarName*_ts is created, where *VarName* is the name of the time series object in R. The variable contains sampling times for the time series.

No special processing of time series data is performed when data are transferred from SAS to R software.

Data Structures

R provides a wide range of built-in and user-defined data structures. When data are transferred from R to SAS software, the data are coerced to a data frame prior to the transfer. If the coercion fails, the data are not transferred.

The section "Using R to Analyze Data in SAS/IML Matrices" on page 248 presents an example of an R object that cannot be directly imported to SAS software and shows how to use R functions to extract simpler data structures from the R object.

Differences from SAS/IML Studio

This section lists differences between the R option in the SUBMIT statement as implemented in SAS/IML Studio and the same option in PROC IML:

- In PROC IML, R must be installed on the computer that runs the SAS server. In SAS/IML Studio, R must be installed on the computer that runs the SAS/IML Studio application.
- If R is installed on a SAS workspace server and is accessed through SAS Enterprise Guide, everyone that connects to that server uses the same version of R and the same set of installed packages. In SAS/IML Studio, R is installed locally on the client computer, so each user can potentially have a different version of R and different packages.

Chapter 15
Robust Regression Examples

Contents

Overview **259**
Using the LMS and LTS Subroutines **261**
Example 15.1: Robust Regression and Leverage Points 261
Example 15.2: Comparison of LMS and LTS Algorithms 265
Example 15.3: LMS and LTS Univariate (Location) Problem 269
Using the MVE and MCD Subroutines **270**
Example 15.4: Relationship between Brain Mass and Body Mass 270
Example 15.5: Multivariate Location, Scale, and Outliers 275
Diagnostic Plots for Robust Regression **282**
References **284**

Overview

SAS/IML has four subroutines that you can use for robust estimation of location and scale, for outlier detection, and for robust regression. The least median of squares (LMS) and least trimmed squares (LTS) subroutines perform *robust regression* (sometimes called *resistant regression*). These subroutines can detect outliers and perform a least squares regression on the remaining observations. You can use the minimum volume ellipsoid estimation (MVE) and minimum covariance determinant estimation (MCD) subroutines to find a robust location and a robust covariance matrix that you can use to construct confidence regions, to detect multivariate outliers and leverage points, and to conduct robust canonical correlation and principal component analyses.

The LMS, LTS, MVE, and MCD methods were developed by Rousseeuw (1984) and Rousseeuw and Leroy (1987). All these methods have a high breakdown value. The breakdown value is a measure of the proportion of contamination that a procedure can withstand and still maintain its robustness.

The algorithm that the LMS subroutine uses is based on the program for robust regression (PROGRESS) of Rousseeuw and Hubert (1996), which is an updated version of Rousseeuw and Leroy (1987). In the special case of regression through the origin for a single regressor, Barreto and Maharry (2006) show that the PROGRESS algorithm does not, in general, find the slope that yields the least median of squares. Starting with SAS/IML 9.2, the LMS subroutine includes the algorithm of Barreto and Maharry (2006) as a special case.

The algorithm that the LTS subroutine uses is based on the FAST-LTS algorithm of Rousseeuw and Van Driessen (2000). The MCD algorithm is based on the FAST-MCD algorithm of Rousseeuw and Van Driessen (1999), which is similar to the FAST-LTS algorithm. The MVE algorithm is based on the

algorithm that is used in the MINVOL program by Rousseeuw (1984). LTS estimation has higher statistical efficiency than LMS estimation. Using the FAST-LTS algorithm, LTS is also faster than LMS for large data sets. Similarly, MCD is faster than MVE for large data sets.

In addition to LTS estimation and LMS estimation, there are other methods for robust regression and outlier detection. For more information, see the documentation of the ROBUSTREG procedure in *SAS/STAT User's Guide*. A summary of these robust tools in SAS can be found in Chen (2002).

The four SAS/IML subroutines are designed for the following tasks:

- LMS minimizes the *h*th ordered squared residual.
- LTS minimizes the sum of the *h* smallest squared residuals.
- MCD minimizes the determinant of the covariance of *h* points.
- MVE minimizes the volume of an ellipsoid that contains *h* points.

The value *h* is the number of observations to use. The value is in the range

$$\frac{N}{2}+1 \;\le\; h \;\le\; \frac{3N}{4}+\frac{n+1}{4}$$

where N is the number of observations and n is the number of regressors. (The value of h can be specified, but in most applications the default value works well and the results seem to be quite stable for different choices of h.) 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). Here, $T(Z)$ indicates the result of applying an estimator T to a sample Z that contains h observations.

For a linear regression model that includes the parameter vector $\mathbf{b} = (b_1, \ldots, b_n)$, the residual of observation i is $r_i = y_i - \mathbf{x}_i\mathbf{b}$. Let $(r^2)_{1:N} \le \ldots \le (r^2)_{N:N}$ be the ordered, squared residuals. The objective functions for the LMS, LTS, MCD, and MVE optimization problems are defined as follows. Each algorithm strives to find the parameter vector that minimizes the objective function.

- The objective function for the LMS optimization problem is the *h*th ordered squared residual:

  $$F_{\mathrm{LMS}} = (r^2)_{h:N}$$

  For $h = N/2+1$, the *h*th quantile is the median of the squared residuals. The default h in PROGRESS is an optimal value $h = \left[\frac{N+n+1}{2}\right]$, which yields the breakdown value $(N-h+1)/n$, where $[k]$ denotes the integer part of k.

- The objective function for the LTS optimization problem is the sum of the h smallest ordered squared residuals:

  $$F_{\mathrm{LTS}} = \sqrt{\frac{1}{h}\sum_{i=1}^{h}(r^2)_{i:N}}$$

- The objective function for the MCD optimization problem is based on the determinant of the covariance of the selected h points,

  $$F_{\mathrm{MCD}} = \det(\mathbf{C}_h)$$

  where $\mathbf{C}_h$ is the covariance matrix of the selected h points.

- The objective function for the MVE optimization problem is based on the hth quantile $d_{h:N}$ of the Mahalanobis-type distances $\mathbf{d} = (d_1, \ldots, d_N)$,

  $$F_{\mathrm{MVE}} = \sqrt{d_{h:N} \det(\mathbf{C})}$$

  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

  $$\mathbf{d} = \mathrm{diag}(\sqrt{(\mathbf{X} - T)^T \mathbf{C}^{-1} (\mathbf{X} - T)})$$

  where T is the location estimate.

Because of the nonsmooth form of these objective functions, the estimates cannot be obtained by using 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 specified 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 might not obtain the global optimum.

The LMS, LTS, MCD, and MVE subroutines require that the number of observations, N, be more than twice the number of explanatory variables, n (including the intercept). That is, the require $N > 2n$.

Using the LMS and LTS Subroutines

Because of space considerations, the tables that contain residuals and resistant diagnostics are not displayed in this document. The LMS and LTS routines have options for displaying tables. However, you can also obtain relevant information by examining the return values of the subroutines' first three arguments. Both techniques are shown in this chapter.

Example 15.1: Robust Regression and Leverage Points

A Hertzsprung-Russell diagram is a scatter plot that shows the relationship between the luminosity of stars and their effective temperatures. The following data correspond to 47 stars of the CYG OB1 cluster in the direction of the constellation Cygnus (Rousseeuw and Leroy 1987; Humphreys 1978; Vansina and De Greve 1982). The regressor variable x (column 2) is the logarithm of the effective temperature at the surface of the star, and the response variable y (column 3) is the logarithm of its light intensity. This data set is remarkable in that it contains four substantial leverage points (observations 11, 20, 30, and 34) that greatly affect the results of L_2, and even L_1, regression. The high leverage points, which represent giant stars, are shown in Output 15.1.2.

The following SAS/IML statements define the data:

```
proc iml;
/* Hertzsprung-Russell Star Data */
/*  ObsNum LogTemp LogIntensity  */
hr = { 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 } ;

x = hr[,2]; y = hr[,3];
```

You can call the LMS subroutine to carry out a least median squares regression analysis. In the following statements, the ODS SELECT statement limits the number of tables that are produced by the subroutine:

```
optn = j(9,1,.);
optn[2]= 1;    /* do not print residuals, diagnostics, or history */
optn[3]= 3;    /* compute LS, LMS, and weighted LS regression */

ods select LSEst EstCoeff RLSEstLMS;
call lms(sc, coef, wgt, optn, y, x);
ods select all;
```

Output 15.1.1 shows the parameter estimates for three regression models: the least squares (LS) model, the LMS model, and a weighted robust least squares (RLS) model.

Output 15.1.1 Parameter Estimates from the LMS Subroutine

LS Parameter Estimates

VAR1	Intercep
-0.413303861	6.7934672987

Estimated Coefficients

VAR1	Intercep
3.9705882353	-12.62794118

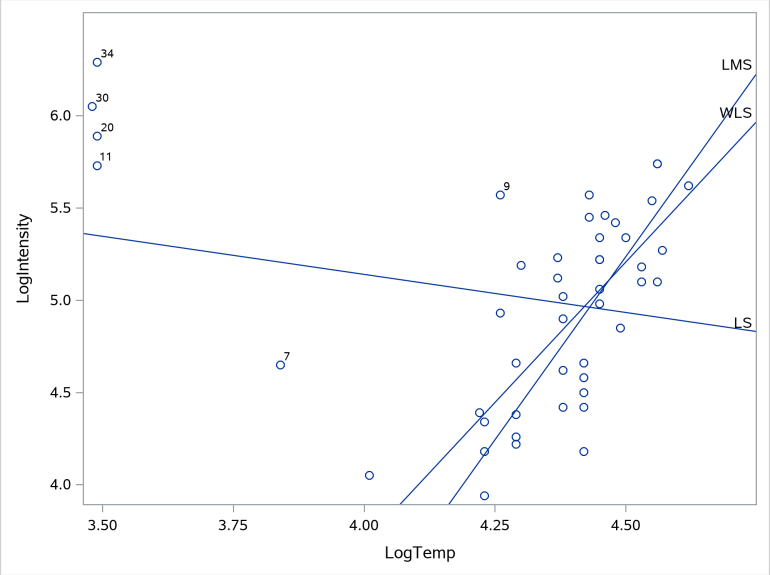
RLS Parameter Estimates Based on LMS

VAR1	Intercep
3.0461569368	-8.500054884

The three regression lines are plotted in Output 15.1.2. The least squares line has a negative slope and a positive intercept. It is highly influenced by the four leverage points in the upper left portion of Output 15.1.2. In contrast, the LMS regression line (whose parameter estimates are shown in the "Estimated Coefficients" table) fits the bulk of the data and ignores the four leverage points.

Similarly, the weighted least squares line (in which the observations 7, 9, 11, 20, 30, and 34 are given zero weight) is less affected by the leverage points. The weights are determined by the size of the scaled residuals for the LMS regression.

Output 15.1.2 Three Regression Lines



In addition to the printed output, the LMS subroutine returns information about the fitted models in the **sc**, **coef**, and **wgt** matrices. The following statements display some of the values in the **sc** matrix. See Output 15.1.3.

```
r1 = {"Quantile", "Number of Subsets", "Number of Singular Subsets",
       "Number of Nonzero Weights", "Objective Function",
       "Preliminary Scale Estimate", "Final Scale Estimate",
       "Robust R Squared", "Asymptotic Consistency Factor"};
r2 = { "WLS Scale Estimate", "Weighted Sum of Squares",
       "Weighted R-squared", "F Statistic"};
sc1 = sc[1:9];
sc2 = sc[11:14];
print sc1[r=r1 L="LMS Information and Estimates"],
      sc2[r=r2 L="Weighted Least Squares"];
```

Output 15.1.3 Details of LMS Regression

LMS Information and Estimates	
Quantile	25
Number of Subsets	1081
Number of Singular Subsets	45
Number of Nonzero Weights	41
Objective Function	0.2620588
Preliminary Scale Estimate	0.3987302
Final Scale Estimate	0.3645644
Robust R Squared	0.5813149
Asymptotic Consistency Factor	1.3781732

Weighted Least Squares	
WLS Scale Estimate	0.3407456
Weighted Sum of Squares	4.5281945
Weighted R-squared	0.5543574
F Statistic	48.514066

Output 15.1.3 shows summary statistics for the analysis. The analysis tries to minimize the hth ordered residual, where $h = \left[\frac{N+n+1}{2}\right] = \left[\frac{47+2+1}{2}\right] = 25$. The LMS algorithm randomly selects 1,081 subsets of three observations. Of these, 45 are singular. The subset that minimizes the 25th ordered residual is found. Based on this subset, six observations are classified as outliers.

The **coef** matrix contains as many columns as there are regressor variables. Rows of the **coef** matrix contain parameter estimates and related statistics. The **wgt** matrix contains as many columns as there are observations. Rows of the **wgt** matrix contain an indicator variable for outliers and residuals for the robust regression.

An alternative to LMS regression is least trimmed squares (LTS) regression. The LTS subroutine implements the FAST-LTS regression algorithm, which improves the Rousseeuw and Leroy (1987) algorithm (called V7 LTS in this chapter) by 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.

The LTS subroutine performs least trimmed squares (LTS) robust regression by minimizing the sum of the h smallest squared residuals. The following statements compute the LTS regression for the Hertzsprung-Russell star data:

```
optn = j(9,1,.);
optn[2]= 3;    /* print a maximum amount of information */
optn[3]= 3;    /* compute LS, LTS, and weighted LS regression */

ods select BestHalf EstCoeff;
call lts(sc, coef, wgt, optn, y, x);
ods select all;
```

The line of best fit for the LTS regression has slope 4.23 and intercept −13.624 as shown in the "Estimated Coefficients" table in Output 15.1.4. This is a steeper line than for the LMS regression, which is shown in Output 15.1.1.

Output 15.1.4 LTS Parameter Estimates

Estimated Coefficients	
VAR1	Intercep
4.219182102	-13.6239903

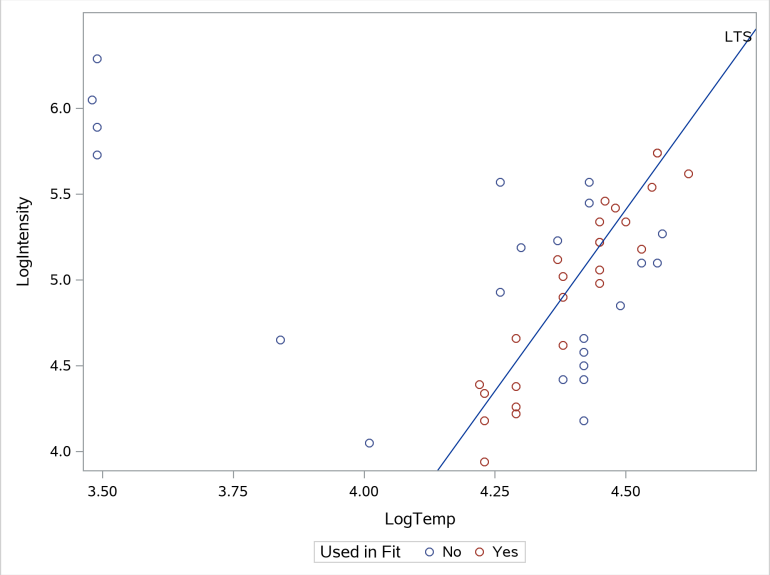
Output 15.1.5 shows the best subset of observations for the Hertzsprung-Russell data. There are $h = 25$ observations in Output 15.1.5.

Output 15.1.5 Observations of the Best Subset

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

Output 15.1.6 shows the geometric meaning of the best subset. In the graph, the selected observations determine the regression line. The observations that are not contained in the best subset are not used to fit the regression line.

Output 15.1.6 LTS Regression Line and Best Subset



Example 15.2: Comparison of LMS and LTS Algorithms

The following example compares the LMS and FAST-LTS subroutines. The data are the stack loss data of Brownlee (1965). The three explanatory variables correspond to measurements for a plant that oxidizes ammonia to nitric acid on 21 consecutive days:

- x_1 represents the air flow to the plant.
- x_2 represents the temperature of cooling water.
- x_3 represents the acid concentration.

The response variable gives the permillage of ammonia lost (stack loss). The following data are also given in Rousseeuw and Leroy (1987) and Osborne (1985):

```
proc iml;
   /* Obs X1  X2  X3   Y  Stack Loss data */
SL = { 1  80  27  89  42,
       2  80  27  88  37,
       3  75  25  90  37,
       4  62  24  87  28,
       5  62  22  87  18,
       6  62  23  87  18,
       7  62  24  93  19,
       8  62  24  93  20,
       9  58  23  87  15,
      10  58  18  80  14,
      11  58  18  89  14,
      12  58  17  88  13,
      13  58  18  82  11,
      14  58  19  93  12,
      15  50  18  89   8,
      16  50  18  86   7,
      17  50  19  72   8,
      18  50  19  79   8,
      19  50  20  80   9,
      20  56  20  82  15,
      21  70  20  91  15 };
x = SL[, 2:4]; y = SL[, 5];
```

Rousseeuw and Leroy (1987) cite a large number of papers in which the preceding data were analyzed. 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.

LMS Regression with 2,000 Random Subsets

For $N = 21$ and $n = 4$ (three explanatory variables plus an intercept), there are a total of $\binom{21}{4}$ =5,985 different subsets of four observations. If you do not specify OPTN[5], the LMS subroutine draws 2,000 random sample subsets. A large number of subsets are collinear and therefore lead to singular linear systems. To suppress printing these subsets and to reduce other output, choose OPTN[2]=2 as in the following statements:

```
/* Use 2000 Random Subsets for LMS */
optn = j(9,1,.);
optn[2]= 2;    /* print a moderate amount of output */
optn[3]= 1;    /* compute only LMS regression */

ods select IterHist0 BestSubset EstCoeff;
call lms(sc, coef, wgt, optn, y, x);
ods select all;
```

Summary statistics are shown in Output 15.2.1. The "IterHist0" table summarizes the process of choosing subsets of four observations. A total of 2,103 subsets are chosen in order to obtain 2,000 nonsingular subsets. The subset that yields the best regression fit consists of observations 10, 11, 15, and 19. The parameter estimates for the LMS regression are $\beta_1 = 0.75$, $\beta_2 = 0.5$, $\beta_3 = 0.0$, and $\beta_0 = -39.25$.

Output 15.2.1 LMS Regression Output

Subset	Singular	Best Criterion	Percent
500	23	0.163262	25
1000	55	0.140519	50
1500	79	0.140519	75
2000	103	0.126467	100

Observations of Best Subset			
15	11	19	10

Estimated Coefficients			
VAR1	VAR2	VAR3	Intercep
0.75	0.5	0	-39.25

The three matrices that are returned by the LMS subroutine contain detailed information about the regression. A few of the results are shown in Output 15.2.2, which is produced by the following statements:

```
r1 = {"Quantile", "Number of Subsets", "Number of Singular Subsets",
      "Number of Nonzero Weights", "Min Objective Function",
      "Preliminary Scale Estimate", "Final Scale Estimate",
      "Robust R Squared", "Asymptotic Consistency Factor"};
sc1 = sc[1:9];
print sc1[r=r1 L="LMS Information and Estimates"];
```

The matrix that is shown in Output 15.2.2 includes the following information:

- The LMS algorithm minimizes the square of the 13th smallest residual.
- Of the 21 observations in the data, 17 are assigned nonzero weights. Equivalently, four are classified as influential observations and are assigned zero weights.
- The other statistics are described in the documentation for the LMS subroutine.

Output 15.2.2 LMS Regression Output

LMS Information and Estimates	
Quantile	13
Number of Subsets	2103
Number of Singular Subsets	103
Number of Nonzero Weights	17
Min Objective Function	0.75
Preliminary Scale Estimate	1.0478511
Final Scale Estimate	1.2076147
Robust R Squared	0.9648438
Asymptotic Consistency Factor	1.1413664

You can print the **wgt** vector to discover that the observations 1, 3, 4, and 21 have scaled residuals larger than 2.5 (output not shown) and so are classified as outliers.

LTS Regression with 500 Random Subsets

The FAST-LTS algorithm uses only 500 random subsets and gets better optimization results, as measured by the sum of the squared residuals criterion. The following statements call the LTS subroutine:

```
/* Use 500 random subsets for FAST-LTS algorithm */
optn = j(9,1,.);
optn[2]= 0;    /* suppress output */
optn[3]= 0;    /* compute only LTS regression */
optn[9]= 0;    /* FAST-LTS */

call lts(sc, coef, wgt, optn, y, x);
```

The following statements display information about the LTS algorithm, parameter estimates, and outliers:

```
r1 = {"Quantile", "Number of Subsets", "Number of Singular Subsets",
      "Number of Nonzero Weights", "Min Objective Function",
      "Preliminary Scale Estimate", "Final Scale Estimate",
      "Robust R Squared", "Asymptotic Consistency Factor"};
sc1 = sc[1:9];
print sc1[r=r1 L="LTS Information and Estimates"];

print (coef[1,])[L="Estimated Coefficients"
                 c={"x1" "x2" "x3" "Intercept"}];

outliers = loc(wgt[1,]=0);
print outliers;
```

The results are shown in Output 15.2.3. The LTS algorithm examines 517 subsets of observations, for which 17 are singular, and classifies six observations as outliers. The output also shows the parameter estimates for the regression model.

Output 15.2.3 Results for LTS Algorithm

LTS Information and Estimates	
Quantile	13
Number of Subsets	517
Number of Singular Subsets	17
Number of Nonzero Weights	15
Min Objective Function	0.4749406
Preliminary Scale Estimate	0.9888436
Final Scale Estimate	1.0360273
Robust R Squared	0.974552
Asymptotic Consistency Factor	2.0820364

Estimated Coefficients			
x1	x2	x3	Intercept
0.7409211	0.3915267	0.0111345	-37.32333

outliers
1 2 3 4 13 21

Robust Regression with All 5,985 Subsets

For a small number of observations, you can generate regression results by considering all possible subsets of observations. For the LMS subroutine, you can set OPTN[5] = −1 to generate all subsets. For the stack loss data, the parameter estimates are identical to Output 15.2.2.

Example 15.3: LMS and LTS Univariate (Location) Problem

If you do not specify a design matrix X for the last input argument, the regression problem reduces to the problem of estimating the location parameter. That is, the "intercept-only" regression model is equivalent to estimating the location parameter for the response variable. For ordinary least squares regression, an intercept-only regression model estimates the mean. For robust regression, it estimates a robust measure of location.

The following example is described in Rousseeuw and Leroy (1987); Barnett and Lewis (1994).

```
proc iml;
y = { 3, 4, 7, 8, 10, 949, 951 };

optn = j(9,1,.);
call lms(scLMS, coefLMS, wgtLMS, optn, y);
call lts(scLTS, coefLTS, wgtLTS, optn, y);

LMSOutliers = loc(wgtLMS[1,]=0);
LTSOutliers = loc(wgtLTS[1,]=0);
print LMSOutliers, LTSOutliers;

rLoc = {"Mean", "Median", "LMS Location", "LTS Location"};
Loc  = mean(y) // median(y) // coefLMS[1] // coefLTS[1];
print Loc[r=rLoc L="Location Estimates"];

rScale = {"StdDev", "MAD", "LMS Scale", "LTS Scale"};
Scale =    std(y) // mad(y) // scLMS[7] // scLTS[7];
print Scale[r=rScale L="Scale Estimates"];
```

Output 15.3.1 shows that the LMS and LTS subroutines both classify observations 6 and 7 as outliers.

Output 15.3.1 Estimates of Location and Scale for Univariate Data

LMSOutliers	
6	7

LTSOutliers	
6	7

Location Estimates	
Mean	276
Median	8
LMS Location	5.5
LTS Location	5.5

Output 15.3.1 *continued*

Scale Estimates	
StdDev	460.43603
MAD	4
LMS Scale	3.0516389
LTS Scale	3.0516389

Output 15.3.1 shows several estimates of the central location of the data. The classical mean (276) is highly influenced by the two large values. In contrast, the median of the data is 8, and the LMS and LTS estimates are both 5.5. Output 15.3.1 also shows estimates of the scale of the data. The classical standard deviation (460.4) is influenced by the two large values. In contrast, the MAD function computes the median absolute deviation to be 4. The LMS and LTS estimates are both 3.05. The scale estimate in the univariate problem is a resistant (high-breakdown) estimator for the dispersion of the data (Rousseeuw and Leroy 1987).

Using the MVE and MCD Subroutines

The MVE subroutine computes the robust estimation of multivariate location and scatter, which are obtained by minimizing the volume of an ellipsoid that contains h points. The MCD subroutine is similar. It minimizes the determinant of the covariance matrix that is computed from h points. In general, the MCD subroutine is faster than the MVE subroutine.

You can use these robust locations and covariance matrices to detect multivariate outliers and leverage points. Both subroutines provide a table of robust distances.

The MVESCATTER and MCDSCATTER modules are used in these examples for plotting the results. These routines are in the *RobustMC.sas* file, which is contained in the SAS/IML sample library.

Example 15.4: Relationship between Brain Mass and Body Mass

This section creates graphs that illustrate the results of the MVE and MCD procedures. The following statements load the *RobustMC.sas* program, which is included in the SAS/IML sample library. The LOAD MODULES=_ALL_ statement loads modules that are defined in the program. In particular, this section uses the MVESCATTER and MCDSCATTER modules.

```
%include sampsrc(robustmc);    /* define graphing modules */
proc iml;
load module=_all_;             /* load graphing modules */
```

Jerison (1973) reported data for the body mass (in kilograms) and brain mass (in grams) of $N = 28$ animals. These data were further analyzed in Rousseeuw and Leroy (1987). Instead of the original data, the following example uses the logarithms of the measurements of the two variables:

```
/* Log(Body Mass) Log(Brain Mass) */
mass={ 0.1303338  0.9084851,  2.6674530  2.6263400,
       1.5602650  2.0773680,  1.4418520  2.0606980,
       0.0170333  0.7403627,  4.0681860  1.6989700,
       3.4060290  3.6630410,  2.2720740  2.6222140,
       2.7168380  2.8162410,  1.0000000  2.0606980,
       0.5185139  1.4082400,  2.7234560  2.8325090,
       2.3159700  2.6085260,  1.7923920  3.1205740,
       3.8230830  3.7567880,  3.9731280  1.8450980,
       0.8325089  2.2528530,  1.5440680  1.7481880,
      -0.9208187  0.0000000, -1.6382720 -0.3979400,
       0.3979400  1.0827850,  1.7442930  2.2430380,
       2.0000000  2.1959000,  1.7173380  2.6434530,
       4.9395190  2.1889280, -0.5528420  0.2787536,
      -0.9136401  0.4771213,  2.2833010  2.2552720};
```

By default, the MVE subroutine uses randomly selected subsets rather than all subsets. The following statements specify that all 3,276 subsets of three observations out of 28 observations be generated and evaluated. Output 15.4.1 shows partial results of the analysis.

```
optn = j(5,1,.);
optn[1] = 1;              /* print basic output */
optn[2] = 1;              /* print covariance matrices */
optn[5]= -1;              /* nrep: use all subsets */

ods exclude EigenRobust Distances DistrRes;
call mve(sc, xmve, dist, optn, mass);
ods select all;
```

Output 15.4.1 Results of MVE Robust Estimation

Minimum Volume Ellipsoid (MVE) Estimation

Consider Ellipsoids Containing 15 Cases.

Classical Covariance Matrix		
	VAR1	VAR2
VAR1	2.6816512381	1.3300846941
VAR2	1.3300846941	1.0857537552

Classical Mean	
VAR1	1.6378572179
VAR2	1.9219465964

Output 15.4.1 *continued*

There are 3276 subsets of 3 cases out of 28 cases.

All 3276 subsets will be considered.

Complete Enumeration for MVE

25 % of calculations have been executed.

75 % of calculations have been executed.

Minimum Criterion= 0.439709106

Among 3276 subsets 0 are singular.

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

Final MVE Estimates (Using Local Improvement)

Number of Points with Nonzero Weight=24

Robust MVE Location Estimates	
VAR1	1.2952823792
VAR2	1.8733722792

Robust MVE Scatter Matrix

	VAR1	VAR2
VAR1	2.056659296	1.5290250177
VAR2	1.5290250177	1.2041353589

Distribution of Robust Distances

Cutoff Value = 2.7162030315

The cutoff value is the square root of the 0.975 quantile of the chi square distribution with 2 degrees of freedom.

There are 5 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.

The MVE routine also returns information in the **sc**, **xmve**, and **dist** matrices. The following statements print some of that information:

```
r1 = {"Quantile", "Number of Subsets", "Number of Singular Subsets",
      "Number of Nonzero Weights", "Min Objective Function",
      "Min Distance", "Chi-Square Cutoff Value"};
RobustCenter = xmve[1,];
RobustCov =    xmve[3:4,];
print sc[r=r1],
      RobustCenter[c={"X1","X2"}],
      RobustCov[r={"X1","X2"} c={"X1","X2"}];
MVEOutliers = loc(dist[3,]=0);
print MVEOutliers;
```

Output 15.4.2 Robust Estimates and Outliers

sc	
Quantile	15
Number of Subsets	3276
Number of Singular Subsets	0
Number of Nonzero Weights	23
Min Objective Function	0.4397091
Min Distance	1.4755584
Chi-Square Cutoff Value	2.716203

RobustCenter	
X1	X2
1.2952824	1.8733723

RobustCov		
	X1	X2
X1	2.0566593	1.529025
X2	1.529025	1.2041354

MVEOutliers				
6	14	16	17	25

The first table in Output 15.4.2 shows some summary statistics about the combinatorial optimization (complete subset sampling) during the MVE algorithm. The algorithm considers subsets of three observations and strives to find the subset that minimizes the volume of an ellipsoid that contains 15 points. The MVE algorithm considers 3,276 subsets, of which zero are singular. The minimum volume is found to be 1.47.

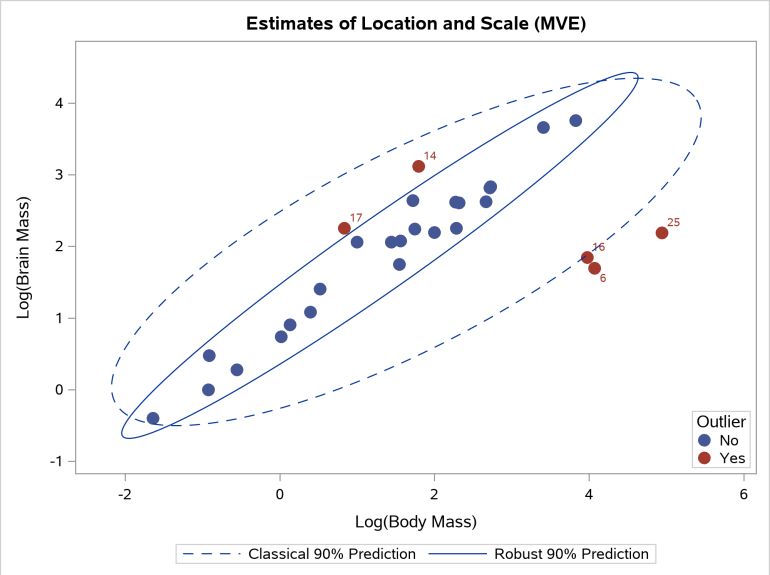
The mean vector is contained in the `RobustCenter` vector. The covariance matrix is contained in the `RobustCov` matrix.

Based on that center and covariance matrix, a Mahalanobis-type distance (called the *robust distance*) that measures each observation's distance from the robust center is computed. Observations that are more than a certain distance (the "Cutoff Value") from the center are classified as outliers. For these data, observations 6, 14, 16, 17, and 25 are classified as outliers.

You can call the MVESCATTER subroutine, which is included in the sample library in the file *RobustMC.sas*, to plot the classical and robust confidence ellipsoids. The LOAD MODULES=_ALL_ statement loads modules that are defined in the program. The following statements create the scatter plot that is shown in Output 15.4.3.

```
optn = j(5,1,.); optn[5]= -1;
vnam = {"Log(Body Mass)","Log(Brain Mass)"};
titl = "Estimates of Location and Scale (MVE)";
call MVEScatter(mass, optn, 0.9, vnam, titl);
```

Output 15.4.3 BrainLog Data: Classical and Robust Ellipses (MVE)



MCD is another subroutine that can be used to compute the robust location and the robust covariance of multivariate data sets. The following statements call the MCD subroutine and produce Output 15.4.4

```
/* MCD: Use Random Subsets */
optn = j(5,1,.);
call mcd(sc, xmve, dist, optn, mass);

r1 = {"Quantile", "Number of Subsets", "Number of Singular Subsets",
      "Number of Nonzero Weights", "Min Objective Function",
      "Min Distance", "Chi-Square Cutoff Value"};
RobustCenter = xmve[1,];
RobustCov =    xmve[3:4,];
print sc[r=r1],
      RobustCenter[c={"X1","X2"}],
      RobustCov[r={"X1","X2"} c={"X1","X2"}];
```

Output 15.4.4 Results of MCD Robust Estimation

sc	
Quantile	15
Number of Subsets	500
Number of Singular Subsets	0
Number of Nonzero Weights	23
Min Objective Function	0.0174302
Min Distance	1.9190823
Chi-Square Cutoff Value	2.716203

Output 15.4.4 *continued*

RobustCenter	
X1	X2
1.315403	1.8568731

RobustCov		
	X1	X2
X1	2.1399861	1.6068557
X2	1.6068557	1.2520385

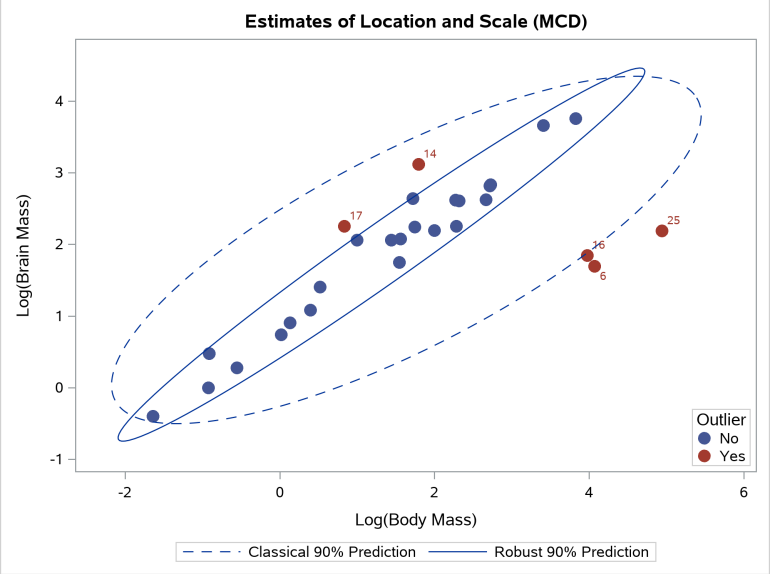
The results are similar to Output 15.4.1. For the MCD subroutine, 500 random subsets are used for the optimization. The robust center and covariance matrices are slightly different from those found by the MVE subroutine. However, the observations that are classified as outliers (not shown) are the same.

You can call the MCDSCATTER subroutine, which is included in the SAS/IML sample library, to plot the classical and robust confidence ellipsoids, as follows:

```
optn = j(5,1,.); optn[5]= -1;
vnam = { "Log(Body Mass)","Log(Brain Mass)" };
titl = "Estimates of Location and Scale (MCD)";
call MCDScatter(mass, optn, 0.9, vnam, titl);
```

The plot is shown in Output 15.4.5. It looks very similar to Output 15.4.3.

Output 15.4.5 BrainLog Data: Classical and Robust Ellipsoid (MCD)



Example 15.5: Multivariate Location, Scale, and Outliers

This section analyzes the three regressors in the stack loss data of Brownlee (1965), which are defined in Example 15.2. As in the previous section, the LOAD MODULES=_ALL_ statement loads modules that are defined in the *RobustMC.sas* file.

```
%include sampsrc(robustmc);    /* define graphing modules */
proc iml;
load module=_all_;             /* load graphing modules */
```

By default, the MVE subroutine generates 2,000 randomly selected subsets in its search. The following call to the MVE subroutine uses all 5,985 subsets of four observations that can be chosen from the 21 observations:

```
   /* Obs X1  X2  X3   Y  Stack Loss data */
SL = { 1  80  27  89  42,
       2  80  27  88  37,
       3  75  25  90  37,
       4  62  24  87  28,
       5  62  22  87  18,
       6  62  23  87  18,
       7  62  24  93  19,
       8  62  24  93  20,
       9  58  23  87  15,
      10  58  18  80  14,
      11  58  18  89  14,
      12  58  17  88  13,
      13  58  18  82  11,
      14  58  19  93  12,
      15  50  18  89   8,
      16  50  18  86   7,
      17  50  19  72   8,
      18  50  19  79   8,
      19  50  20  80   9,
      20  56  20  82  15,
      21  70  20  91  15 };
x = SL[, 2:4]; y = SL[, 5];

optn = j(5,1,.);
optn[1] = 1;              /* print basic output */
optn[2] = 1;              /* print covariance matrices */
optn[5]= -1;              /* nrep: use all subsets */

ods select ClassicalMean ClassicalCov RobustMVELoc RobustMVEScatter;
call mve(sc, xmve, dist, optn, x);
ods select all;
```

Output 15.5.1 shows the classical and robust estimates of the location. Output 15.5.2 shows the classical and robust estimates of the scatter.

Output 15.5.1 Classical and Robust Estimates of the Location

	Classical Mean
VAR1	60.428571429
VAR2	21.095238095
VAR3	86.285714286

Output 15.5.1 *continued*

Robust MVE Location Estimates	
VAR1	56.705882353
VAR2	20.235294118
VAR3	85.529411765

Output 15.5.2 Classical and Robust Estimates of the Scatter

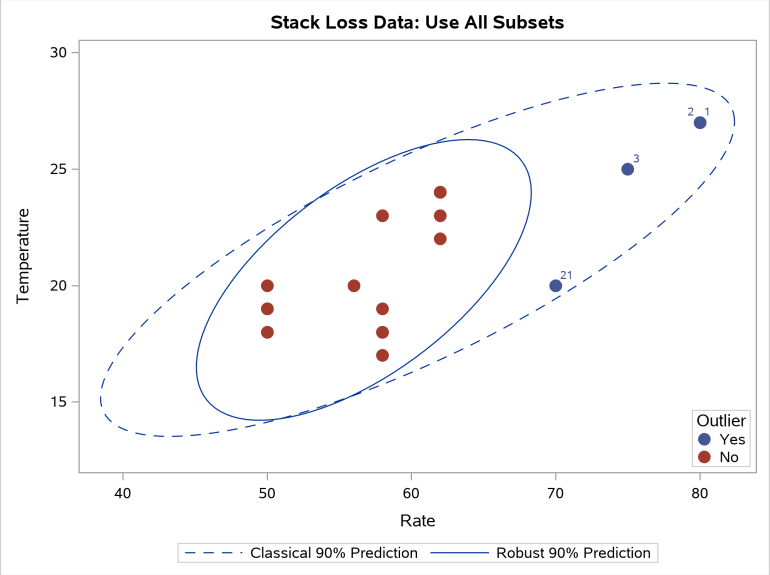
Classical Covariance Matrix			
	VAR1	**VAR2**	**VAR3**
VAR1	84.057142857	22.657142857	24.571428571
VAR2	22.657142857	9.9904761905	6.6214285714
VAR3	24.571428571	6.6214285714	28.714285714

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

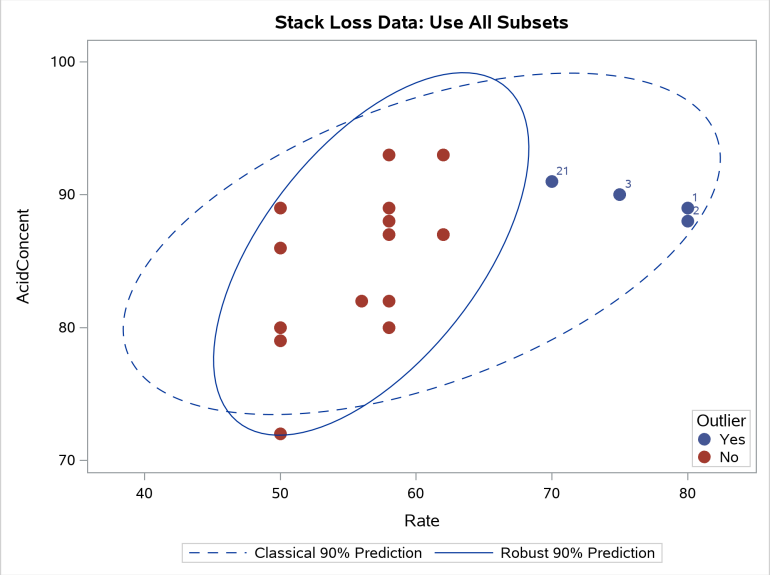
The following statements generate three bivariate scatter plots of the classical and robust tolerance ellipsoids. The plots are shown in Output 15.5.3, Output 15.5.4, and Output 15.5.5, one plot for each pair of variables.

```
optn = j(5,1,.); optn[5]= -1;
vnam = {"Rate", "Temperature", "AcidConcent"};
titl = "Stack Loss Data: Use All Subsets";
call MVEScatter(x, optn, 0.9, vnam, titl);
```

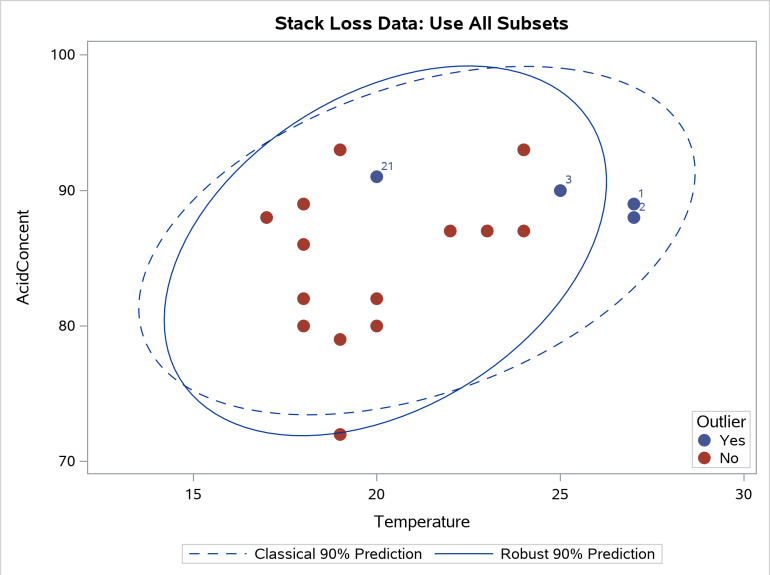
Output 15.5.3 Stack Loss Data: Rate versus Temperature (MVE)



Output 15.5.4 Stack Loss Data: Rate versus Acid Concentration (MVE)



Output 15.5.5 Stack Loss Data: Temperature versus Acid Concentration (MVE)



You can also use the MCD method for the stack loss data as follows:

```
optn = j(5,1,.);
optn[1]= 2;              /* print distances */
optn[2]= 1;              /* print covariance matrices */
optn[5]= -1 ;            /* nrep: use all subsets */

call mcd(sc, xmcd, dist, optn, x);
```

The optimization results are displayed in Output 15.5.6. The reweighted results are displayed in Output 15.5.7.

Output 15.5.6 MCD Results of Optimization

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

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

Output 15.5.7 Final Reweighted MCD Results

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

The MCD robust distances and outlying diagnostic are displayed in Output 15.5.8. MCD identifies more leverage points than MVE identifies.

Output 15.5.8 MCD Robust Distances

Classical Distances and Robust (Rousseeuw) Distances

Unsquared Mahalanobis Distance and

Unsquared Rousseeuw Distance of Each Observation

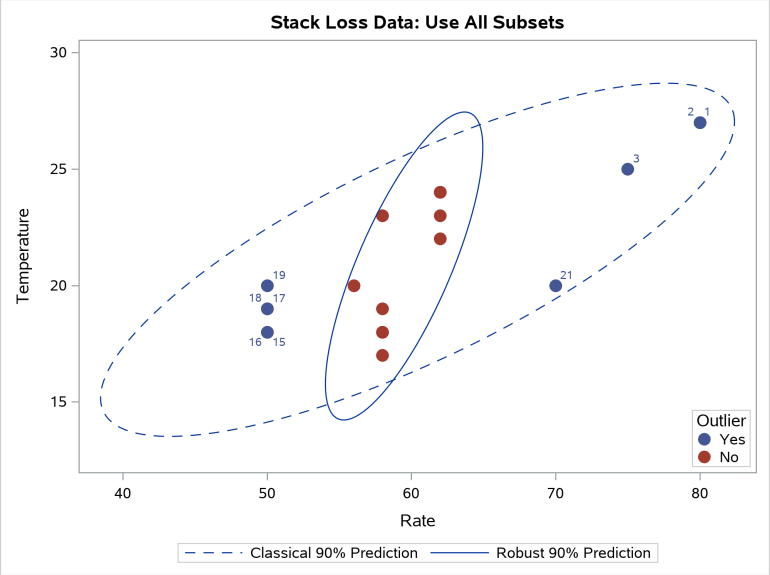
N	Mahalanobis Distances	Robust Distances	Weight
1	2.253603	12.173282	0
2	2.324745	12.255677	0
3	1.593712	9.263990	0
4	1.271898	1.401368	1.000000
5	0.303357	1.420020	1.000000
6	0.772895	1.291188	1.000000
7	1.852661	1.460370	1.000000
8	1.852661	1.460370	1.000000
9	1.360622	2.120590	1.000000
10	1.745997	1.809708	1.000000
11	1.465702	1.362278	1.000000
12	1.841504	1.667437	1.000000
13	1.482649	1.416724	1.000000
14	1.778785	1.988240	1.000000
15	1.690241	5.874858	0
16	1.291934	5.606157	0
17	2.700016	6.133319	0
18	1.503155	5.760432	0
19	1.593221	6.156248	0
20	0.807054	2.172300	1.000000
21	2.176761	7.622769	0

The following statements generate three bivariate scatter plots of the classical and robust tolerance ellipsoids:

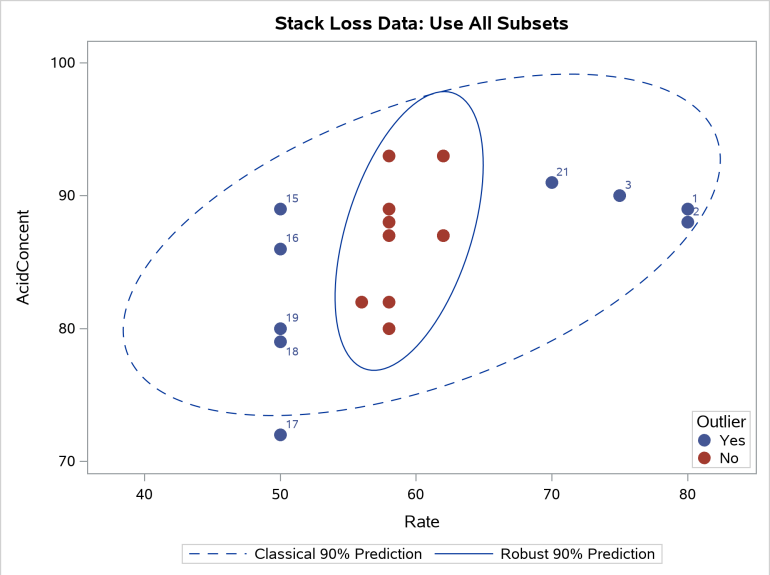
```
optn = j(5,1,.); optn[5]= -1;
vnam = {"Rate", "Temperature", "AcidConcent"};
titl = "Stack Loss Data: Use All Subsets";
call MCDScatter(x, optn, 0.9, vnam, titl);
```

Output 15.5.9, Output 15.5.10, and Output 15.5.11 display these plots, one plot for each pair of variables:

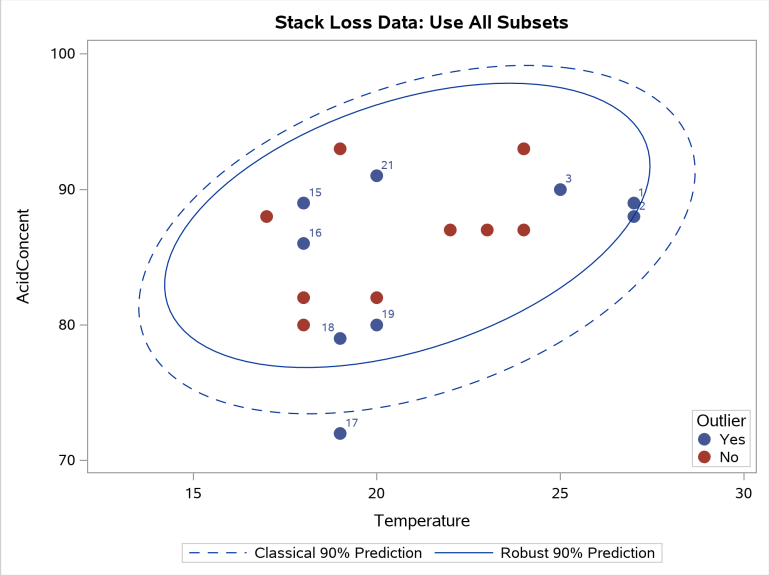
Output 15.5.9 Stack Loss Data: Rate versus Temperature (MCD)



Output 15.5.10 Stack Loss Data: Rate versus Acid Concentration (MCD)



Output 15.5.11 Stack Loss Data: Temperature versus Acid Concentration (MCD)



Diagnostic Plots for Robust Regression

This section is based 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 and MCD subroutines compute 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.

Therefore, the definition of a leverage point is based entirely on the extreme value 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.

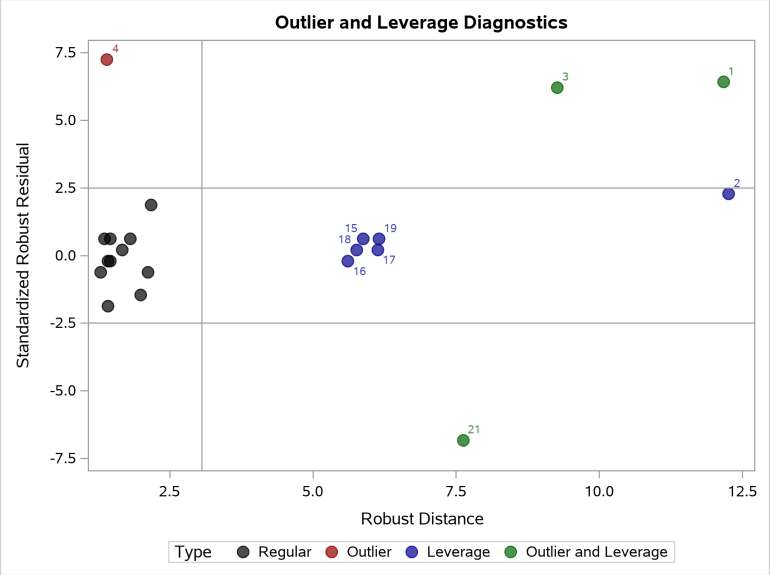
Rousseeuw and Van Zomeren (1990) propose plotting the standardized residuals of robust regression (LMS or LTS) versus the robust distances that are obtained from MVE or MCD. Two horizontal lines that correspond to residual values of $+2.5$ and -2.5 are useful for distinguishing between small and large residuals, and one vertical line that corresponds to the $\sqrt{\chi^2_{n,.975}}$ is used to distinguish between small and large distances.

For example, once again consider the stack loss data from Brownlee (1965). The following statements call the RDPLOT module, which is distributed in the *RobustMC.sas* file. As in the previous section, the LOAD MODULES=_ALL_ statement loads modules that are defined in the *RobustMC.sas* file.

```
%include sampsrc(robustmc);   /* define graphing modules */
proc iml;
load module=_all_;            /* load graphing modules */

   /* Obs X1  X2  X3   Y  Stack Loss data */
SL = { 1  80  27  89  42,
       2  80  27  88  37,
       3  75  25  90  37,
       4  62  24  87  28,
       5  62  22  87  18,
       6  62  23  87  18,
       7  62  24  93  19,
       8  62  24  93  20,
       9  58  23  87  15,
      10  58  18  80  14,
      11  58  18  89  14,
      12  58  17  88  13,
      13  58  18  82  11,
      14  58  19  93  12,
      15  50  18  89   8,
      16  50  18  86   7,
      17  50  19  72   8,
      18  50  19  79   8,
      19  50  20  80   9,
      20  56  20  82  15,
      21  70  20  91  15 };
x = SL[, 2:4]; y = SL[, 5];

LMSOpt = j(9,1,.);
MCDOpt = j(5,1,.);
MCDOpt[5]= -1;              /* nrep: all subsets */

run RDPlot("LMS", LMSOpt, MCDOpt, y, x);
```

The diagnostic plot is shown in Output 15.5.12. The graph shows the standardized LMS residuals plotted against the robust distances RD_i. The plot shows that observation 4 is a regression outlier but not a leverage point, so it is a vertical outlier. Observations 1, 3, and 21 are bad leverage points, whereas observation 2 is a good leverage point. Notice that observation 2 is very close to the boundary between good and bad leverage points.

Output 15.5.12 Stack Loss Data: LMS Residuals versus Robust Distances



If you use the LTS algorithm instead of the LMS algorithm, observation 13 is classified as a vertical outlier and observation 2 is classified as a bad leverage point.

References

Afifi, A. A., and Azen, S. P. (1972). *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.

Barnett, V., and Lewis, T. (1994). *Outliers in Statistical Data*. 3rd ed. New York: John Wiley & Sons.

Barreto, H., and Maharry, D. (2006). "Least Median of Squares and Regression through the Origin." *Computational Statistics and Data Analysis* 50:1391–1397.

Brownlee, K. A. (1965). *Statistical Theory and Methodology in Science and Engineering*. New York: John Wiley & Sons.

Chen, C. (2002). "Robust Tools in SAS." In *Developments in Robust Statistics: International Conference on Robust Statistics*, edited by R. Dutter, P. Filzmoser, U. Gather, and P. J. Rousseeuw, 125–133. Heidelberg: Springer-Verlag.

Ezekiel, M., and Fox, K. A. (1959). *Methods of Correlation and Regression Analysis*. New York: John Wiley & Sons.

Humphreys, R. M. (1978). "Studies of Luminous Stars in Nearby Galaxies, Part 1: Supergiants and O Stars in the Milky Way." *Astrophysical Journal Supplement Series* 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.

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*, edited by W. Grossmann, G. Pflug, I. Vincze, and W. Wertz, 283–297. Dordrecht, Netherlands: D. Reidel Publishing.

Rousseeuw, P. J., and Hubert, M. (1996). "Recent Development in PROGRESS." *Computational Statistics and Data Analysis* 21:67–85.

Rousseeuw, P. J., and Leroy, A. M. (1987). *Robust Regression and Outlier Detection*. New York: John Wiley & Sons.

Rousseeuw, P. J., and Van Driessen, K. (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." In *Data Analysis: Scientific Modeling and Practical Application*, edited by W. Gaul, O. Opitz, and M. Schader, 335–346. New York: Springer-Verlag.

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." *Astrophysics and Space Science* 87:377–401.

Chapter 16
Time Series Analysis and Examples

Contents

Overview **287**
Basic Time Series Subroutines **288**
Kalman Filter Subroutines **290**
Getting Started 290
Syntax 291
Example 16.1: Kalman Filtering: Likelihood Function Evaluation 292
Example 16.2: Kalman Filtering: SSM Estimation with the EM Algorithm 295
Example 16.3: Diffuse Kalman Filtering 301
Vector Time Series Analysis Subroutines **303**
Getting Started 303
Syntax 306
Fractionally Integrated Time Series Analysis **307**
Getting Started 307
Syntax 310
Time Series Analysis and Control Subroutines **310**
References **311**

Overview

This chapter describes SAS/IML subroutines that are related to univariate, multivariate, and fractional time series analysis and subroutines for Kalman filtering and smoothing. You can use these subroutines to analyze economic and financial time series. You can develop a model of univariate time series and a model of the relationships between 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 functionality:

- 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 by using Kalman filters

- fitting time series models, including the AR, periodic AR, time-varying coefficient AR, VAR, and ARFIMA models
- handling Bayesian seasonal adjustment models

In addition, SAS/IML software provides decomposition analysis, forecasting of an ARMA model, and fractional differencing of a time series.

Basic Time Series Subroutines

In classical linear regression analysis, the underlying process can often be represented simply by an intercept and slope parameters. A time series can be modeled by a type of regression analysis.

The following subroutines and functions 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.

The ARMACOV subroutine provides the pattern of the autocovariance function of AR, MA, and ARMA models and helps identify and fit a proper model.

The ARMALIK subroutine provides the log likelihood of an ARMA model and helps estimate the parameters of an ARMA regression model.

The ARMASIM function generates various time series from the underlying AR, MA, and ARMA models. Simulations of time series that have a known ARMA structure are often needed as part of other simulations or as learning data sets for developing time series analysis skills.

Consider a time series of length 100 from the ARMA(2,1) model

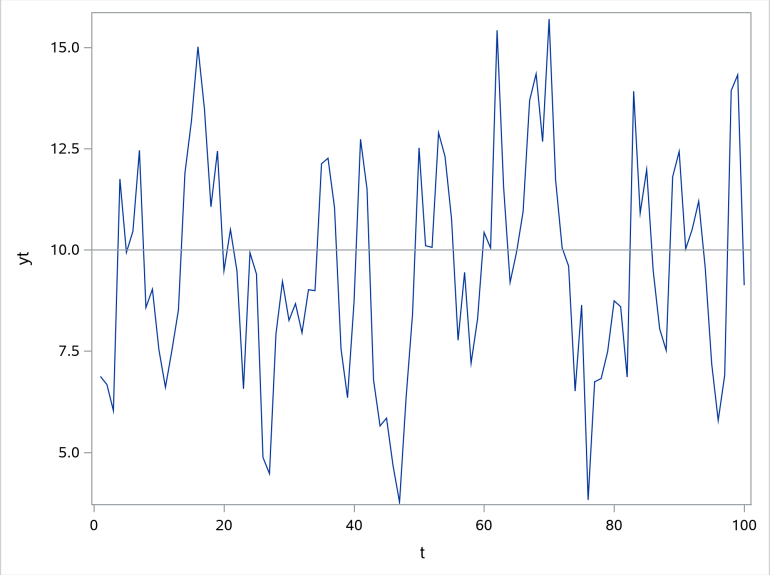
$$y_t = 0.5y_{t-1} - 0.04y_{t-2} + e_t + 0.25e_{t-1}$$

where the error series follows a normal distribution with mean 10 and standard deviation 2.

The following statements generate 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);
```

Figure 16.1 Plot of Generated ARMA(2,1) Process (ARMASIM)



The ARMASIM function generates the data shown in Figure 16.1. The following statements compute 10 lags of the autocovariance function of the series. Figure 16.2 displays the autocovariance functions of the ARMA(2,1) model, the covariance functions of the moving-average term with lagged values of the process, and the autocovariance functions of the moving-average term.

```
call armacov(autocov, cross, convol, phi, theta, lag);
autocov = autocov`;  cross = cross`;  convol = convol`;
print autocov cross convol;
```

Figure 16.2 Autocovariance Functions of ARMA(2,1) Model (ARMACOV)

autocov	cross	convol
1.6972803	1.1875	1.0625
1.0563848	0.25	0.25
0.4603012		
0.1878952		
0.0755356		
0.030252		
0.0121046		
0.0048422		
0.0019369		
0.0007748		

The following statements call the ARMALIK subroutine. The first column of Figure 16.3 contains the log-likelihood function, the estimate of the innovation variance, and the log of the determinant of the variance matrix. The next two columns display part of the standardized residuals and the scale factors that are used to standardize the residuals.

```
call armalik(lnl, resid, std, yt, phi, theta);
resid=resid[1:9];
std=std[1:9];
print lnl resid std;
```

Figure 16.3 Log-Likelihood Function of ARMA(2,1) Model (ARMALIK)

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

Another example that uses the ARMACOV and ARMALIK subroutines is provided in "Example 12.12: Simulations of a Univariate ARMA Process" on page 220.

Kalman Filter Subroutines

This section describes a collection of Kalman filtering and smoothing subroutines for time series analysis; immediately following are three examples that demonstrate how to use Kalman filtering subroutines. The state space model (SSM) is a method of analyzing a wide range of time series models. When the time series is represented by the state space model, the Kalman filter is used for filtering, prediction, and smoothing of the state vector. The state space model consists of the measurement and transition equations.

SAS/IML software supports the following Kalman filtering and smoothing subroutines:

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 as

$$\mathbf{y}_t = \mathbf{b}_t + \mathbf{H}_t \mathbf{z}_t + \epsilon_t$$

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 by 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

$$\delta_{ts} = \begin{cases} 1 & \text{if } t = s \\ 0 & \text{if } t \neq s \end{cases}$$

De Jong (1991) 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 as

$$\begin{aligned} \mathbf{y}_t &= \mathbf{X}_t \beta + \mathbf{H}_t \mathbf{z}_t + \epsilon_t \\ \mathbf{z}_{t+1} &= \mathbf{W}_t \beta + \mathbf{F}_t \mathbf{z}_t + \eta_t \\ \mathbf{z}_0 &= \mathbf{a} + \mathbf{A}\delta \\ \beta &= \mathbf{b} + \mathbf{B}\delta \end{aligned}$$

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 $\mathbf{P}_{t|t}$, by using forward recursions. You can obtain the k-step prediction $\mathbf{z}_{t+k|t}$ and its covariance matrix $\mathbf{P}_{t+k|t}$ by using 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 along with 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 16.1: Kalman Filtering: Likelihood Function Evaluation

In the following example, the log-likelihood function of the SSM is computed by using a prediction-error decomposition. The data are the annual real gross national product (GNP) for the years 1909–1969. The GNP series y_t can be decomposed as

$$y_t = \mu_t + \epsilon_t$$

where μ_t is a trend component and $\epsilon_t \sim (0, \sigma_\epsilon^2)$ is a white noise error term. For more information about these data, see Nelson and Plosser (1982) The trend component is assumed to be generated from the stochastic equations

$$\begin{aligned} \mu_t &= \mu_{t-1} + \beta_{t-1} + \eta_{1t} \\ \beta_t &= \beta_{t-1} + \eta_{2t} \end{aligned}$$

where $\eta_{1t} \sim (0, \sigma_{\eta_1}^2)$ and $\eta_{2t} \sim (0, \sigma_{\eta_2}^2)$ are independent white noise disturbances.

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

$$\begin{aligned} \mathbf{H} &= (1, 0) \\ \mathbf{F} &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \\ \mathbf{z}_t &= (\mu_t, \beta_t)' \\ \eta_t &= (\eta_{1t}, \eta_{2t})' \\ \mathrm{Var}\left(\begin{bmatrix} \eta_t \\ \epsilon_t \end{bmatrix}\right) &= \begin{bmatrix} \sigma_{\eta_1}^2 & 0 & 0 \\ 0 & \sigma_{\eta_2}^2 & 0 \\ 0 & 0 & \sigma_\epsilon^2 \end{bmatrix} \end{aligned}$$

When the observation noise ϵ_t is normally distributed, the average log-likelihood function of the SSM is

$$\begin{aligned} \ell &= \frac{1}{T}\sum_{t=1}^{T} \ell_t \\ \ell_t &= -\frac{N_y}{2}\log(2\pi) - \frac{1}{2}\log(|\mathbf{C}_t|) - \frac{1}{2}\hat{\epsilon}_t'\mathbf{C}_t^{-1}\hat{\epsilon}_t \end{aligned}$$

where $\mathbf{C}_t$ is the mean square error matrix of the prediction error $\hat{\epsilon}_t$, such that $\mathbf{C}_t = \mathbf{H}\mathbf{P}_{t|t-1}\mathbf{H}' + \mathbf{R}_t$.

As an example, consider the following annual real GNP data for 1909–1969:

```
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
;
```

In the following program, the LIK module computes the average log-likelihood function. First, the average log-likelihood function is computed by using the default initial values: 0 for Z0 and the diagonal matrix $10^6 I$ for VZ0. The second call of the module LIK produces the average log-likelihood function with the given initial conditions: Z0 = 0 and VZ0 = $10^{-3} I$. Output 16.1.1 shows a sizable difference between the uncertain initial condition (VZ0 = $10^6 I$) and the almost deterministic initial condition (VZ0 = $10^{-3} I$).

```
proc iml;
start lik(y,a,b,f,h,var,z0,vz0);
    nz = nrow(f);  n = nrow(y);   k = ncol(y);
    pi = constant('pi');
    const = k*log(2*pi);
    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(-.5*const-.5*(sum1+sum2)/n);
finish;

use gnp;
read all var {y};
close gnp;

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 provided', 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 provided', logl;
```

Output 16.1.1 Average Log Likelihood of SSM

Likelihood Evaluation of SSM
DATA: Annual Real GNP 1909-1969

No initial values are provided

logl
-26313.74

Initial values are provided

logl
-91883.49

The following statements compute one-step predictions, filtered values, and real GNP series under the moderate initial condition (VZ0 = $10I$). Output 16.1.2 shows the observed data, the predicted state vectors, and the filtered state vectors for the first 16 observations.

```
z0 = j(1,nrow(f),0);
vz0 = 10#i(nrow(f));
call kalcvf(pred0,vpred,filt0,vfilt,y,1,a,f,b,h,var,z0,vz0);

/* print results for the first few observations */
y0 = y;
y    = y0[1:16];
pred = pred0[1:16,];
filt = filt0[1:16,];
print y pred filt;
```

Output 16.1.2 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 16.2: Kalman Filtering: SSM Estimation with the EM Algorithm

The following example estimates the normal SSM of the mink-muskrat data (Harvey 1989) by using the EM algorithm. The mink-muskrat data are log-counts that have been detrended.

```
title 'SSM Estimation Using EM Algorithm';
data MinkMuskrat;
   input muskrat mink @@;
datalines;
 0.10609  0.16794 -0.16852  0.06242 -0.23700 -0.13344
-0.18022 -0.50616  0.18094 -0.37943  0.65983 -0.40132
 0.65235  0.08789  0.21594  0.23877 -0.11515  0.40043
-0.00067  0.37758 -0.00387  0.55735 -0.25202  0.34444
-0.65011 -0.02749 -0.53646 -0.41519 -0.08462  0.02591
-0.05640 -0.11348  0.26630  0.20544  0.03641  0.16331
-0.26030 -0.01498 -0.03995  0.09657  0.33612  0.31096
-0.11672  0.30681 -0.69775 -0.69351 -0.07569 -0.56212
 0.36149 -0.36799  0.42341 -0.24725  0.26721  0.04478
-0.00363  0.21637  0.08333  0.30188 -0.22480  0.29493
-0.13728  0.35463 -0.12698  0.05490 -0.18770 -0.52573
 0.34741 -0.49541  0.54947 -0.26250  0.57423 -0.21936
 0.57493 -0.12012  0.28188  0.63556 -0.58438  0.27067
-0.50236  0.10386 -0.60766  0.36748 -1.04784 -0.33493
-0.68857 -0.46525 -0.11450 -0.63648  0.22005 -0.26335
 0.36533  0.07017 -0.00151 -0.04977  0.03740 -0.02411
 0.22438  0.30790 -0.16196  0.41050 -0.12862  0.34929
 0.08448 -0.14995  0.17945 -0.03320  0.37502  0.02953
 0.95727  0.24090  0.86188  0.41096  0.39464  0.24157
 0.53794  0.29385  0.13054  0.39336 -0.39138 -0.00323
-1.23825 -0.56953 -0.66286 -0.72363
;
```

Because this EM algorithm uses filtering and smoothing, you can use the KALCVF and KALCVS calls to analyze the data. Consider the bivariate SSM,

$$\begin{aligned} \mathbf{y}_t &= \mathbf{H}\mathbf{z}_t + \epsilon_t \\ \mathbf{z}_t &= \mathbf{F}\mathbf{z}_{t-1} + \eta_t \end{aligned}$$

where $\mathbf{H}$ 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

$$\begin{bmatrix} 0.1 & 0.0 \\ 0.0 & 0.1 \end{bmatrix}$$

whereas 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, because 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 by using the M-step of the EM maximization,

$$\begin{aligned} \mathbf{F}^{i+1} &= \mathbf{S}_t(1)[\mathbf{S}_{t-1}(0)]^{-1} \\ \mathbf{V}^{i+1} &= \frac{1}{T}\left(\mathbf{S}_t(0) - \mathbf{S}_t(1)[\mathbf{S}_{t-1}(0)]^{-1}\mathbf{S}_t'(1)\right) \\ \mathbf{R}^{i+1} &= \frac{1}{T}\sum_{t=1}^{T}\left[(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|T})(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|T})' + \mathbf{H}\mathbf{P}_{t|T}\mathbf{H}'\right] \\ \mu^{i+1} &= \mathbf{z}_{0|T} \end{aligned}$$

where the index i represents the current iteration number, and

$$\begin{aligned} \mathbf{S}_t(0) &= \sum_{t=1}^{T}(\mathbf{P}_{t|T} + \mathbf{z}_{t|T}\mathbf{z}_{t|T}') \\ \mathbf{S}_t(1) &= \sum_{t=1}^{T}(\mathbf{P}_{t,t-1|T} + \mathbf{z}_{t|T}\mathbf{z}_{t-1|T}') \end{aligned}$$

It is necessary to compute the value of $\mathbf{P}_{t,t-1|T}$ recursively such that

$$\mathbf{P}_{t-1,t-2|T} = \mathbf{P}_{t-1|t-1}\mathbf{P}_{t-2}^{*\prime} + \mathbf{P}_{t-1}^{*}(\mathbf{P}_{t,t-1|T} - \mathbf{F}\mathbf{P}_{t-1|t-1})\mathbf{P}_{t-2}^{*\prime}$$

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 by using the formula

$$\mathbf{P}_{T,T-1|T} = \left[\mathbf{I} - \mathbf{P}_{t|t-1}\mathbf{H}'(\mathbf{H}\mathbf{P}_{t|t-1}\mathbf{H}' + \mathbf{R})^{-1}\mathbf{H}\right]\mathbf{F}\mathbf{P}_{T-1|T-1}$$

Note that the initial value of the state vector is updated for each iteration,

$$\begin{aligned} \mathbf{z}_{1|0} &= \mathbf{F}\mu^i \\ \mathbf{P}_{1|0} &= \mathbf{F}^i\,\Sigma\mathbf{F}^{i\prime} + \mathbf{V}^i \end{aligned}$$

The objective function value is computed as -2ℓ in the SAS/IML module LIK. The log-likelihood function is written as

$$\ell = -\frac{1}{2}\sum_{t=1}^{T}\log(|\mathbf{C}_t|) - \frac{1}{2}\sum_{t=1}^{T}(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|t-1})\mathbf{C}_t^{-1}(\mathbf{y}_t - \mathbf{H}\mathbf{z}_{t|t-1})'$$

where $\mathbf{C}_t = \mathbf{H}\mathbf{P}_{t|t-1}\mathbf{H}' + \mathbf{R}$.

The EM algorithm is implemented by the following statements. The iteration history is shown in Output 16.2.1.

```
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;

use MinkMuskrat;
read all into y var {muskrat mink};
close MinkMuskrat;

/*-- 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;
logl0 = 0.0;
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;
diflog = logl - logl0;
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((diflog)/(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];
```

Output 16.2.1 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

The following statements compute the eigenvalues of $\mathbf{F}$. As shown in Output 16.2.2, the eigenvalues of $\mathbf{F}$ are within the unit circle, indicating that the SSM is stationary. However, the muskrat series is reported to be difference stationary. The estimated parameters are almost identical to those of the VAR(1) estimates. See Harvey (1989).

```
eval = eigval(f0);
colnm = {'Real' 'Imag' 'MOD'};
eval = eval || sqrt((eval#eval)[,+]);
print eval[colname=colnm];
```

Output 16.2.2 Eigenvalues of F Matrix

Real	Imag	MOD
0.6547534	0.438317	0.7879237
0.6547534	-0.438317	0.7879237

Finally, multistep forecasts of $\mathbf{y}_t$ are computed by calling the KALCVF subroutine. The predicted values of the state vector $\mathbf{z}_t$ and their standard errors are shown in Output 16.2.3.

```
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];
quit;
```

Output 16.2.3 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 16.3: Diffuse Kalman Filtering

The nonstationary SSM is simulated to analyze the diffuse Kalman filter call KALDFF. The transition equation is generated by using the 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 because the transition matrix $\mathbf{F}$ has one unit root. The following program simulates a time series:

```
title 'Diffuse Kalman Filtering';
proc iml;
T = 20;
y = j(T,1);
burnIn = 10;
z_1 = 0; z_2 = 0;
do i = 1-burnIn to T;
   z = 1.5*z_1 - 0.5*z_2 + rannor(1234567);
   z_2 = z_1;  z_1 = z;
   x = z + 0.8*rannor(1234567);
   if ( i > 0 ) then
      y[i] = x;
end;
```

The KALDFF and KALCVF calls produce one-step prediction, and the following program shows that two predictions coincide after the fifth observation. See Output 16.3.1.

```
h = { 1 0 };
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 16.3.1 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 as

$$\lambda(\mathbf{y}) = -\frac{1}{2}[\mathbf{y}^{\#}\log(\hat{\sigma}^2) + \sum_{t=1}^{T}\log(|\mathbf{D}_t|)]$$

where $\mathbf{y}^{\#}$ is the dimension of the matrix $(\mathbf{y}_1', \ldots, \mathbf{y}_T')'$. The likelihood function for the diffuse Kalman filter under the diffuse initial covariance matrix ($\Sigma_\delta \rightarrow \infty$) 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 16.3.2 displays the log likelihood and the diffuse log likelihood, as computed by the following statements:

```
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[L='Log L']  dff_logl[L='Diffuse Log L'];
quit;
```

Output 16.3.2 Diffuse Likelihood Function

Log L	Diffuse Log L
-11.42547	-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 subroutine generates various time series from the underlying VARMA models. Simulation of time series that have a known VARMA structure enables you to develop analytical skills for vector time series.

The VARMACOV subroutine provides the pattern of the autocovariance function of VARMA models and helps you identify and fit a proper model.

The VARMALIK subroutine provides the log likelihood of a VARMA model and helps you obtain estimates of the parameters of a regression model.

The following subroutines are supported:

VARMACOV	computes the theoretical cross covariances for a multivariate ARMA model.
VARMALIK	evaluates the log-likelihood function for a multivariate ARMA model.
VARMASIM	generates a multivariate ARMA time series.
VNORMAL	generates a multivariate normal random series.
VTSROOT	computes the characteristic roots of a multivariate ARMA model.

Getting Started

Stationary VAR Process

The following equation describes a first-order stationary vector autoregressive model with zero mean:

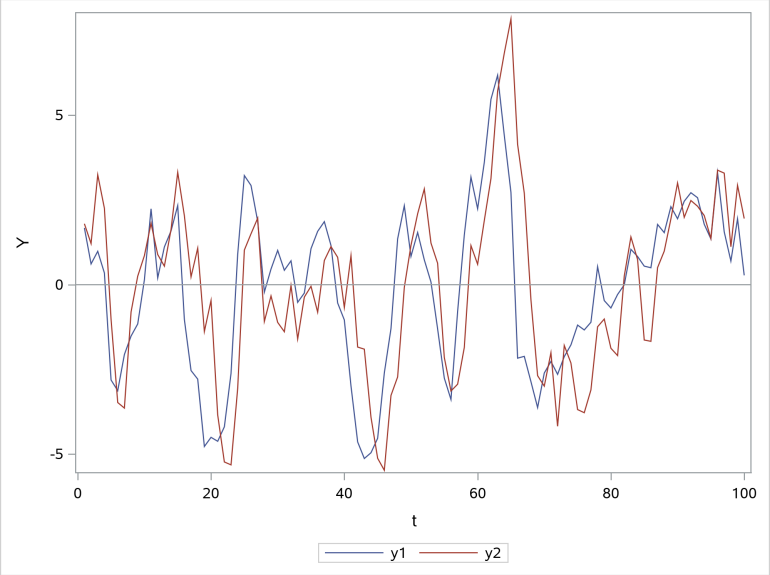
$$\mathbf{y}_t = \begin{pmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{pmatrix} \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t \quad \text{with} \quad \Sigma = \begin{pmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{pmatrix}$$

The following statements simulate 100 observations for the 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;
```

The stationary VAR(1) process is shown in Output 16.3.3.

Output 16.3.3 Plot of Generated VAR(1) Process (VARMASIM)



The following statements compute the roots of the characteristic function:

```
call vtsroot(root,phi);
print root[c={R I 'Mod' 'ATan' 'Deg'}];
```

Output 16.3.4 Roots of VAR(1) Model (VTSROOT)

root				
R	I	Mod	ATan	Deg
0.75	0.3122499	0.8124038	0.3945069	22.603583
0.75	-0.31225	0.8124038	-0.394507	-22.60358

In Output 16.3.4, the first column displays the real part (R) of the root of the characteristic function, and the second column shows the imaginary part (I). The third column displays the modulus, the square root of $R^2 + I^2$. The fourth column shows the $\tan^{-1}(I/R)$, measured in radians, and the last column shows the same measurement in degrees. The third column shows that the moduli are less than 1, so the series is stationary.

The following statements compute five lags of cross-covariance matrices:

```
call varmacov(crosscov,phi) sigma=sig lag=5;
lag = {'0','','1','','2','','3','','4','','5',''};
print lag crosscov;
```

Output 16.3.5 Cross-Covariance Matrices of VAR(1) Model (VARMACOV)

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

In each matrix in Output 16.3.5, the diagonal elements correspond to the autocovariance functions of each time series. The off-diagonal elements correspond to the cross-covariance functions between the two series.

The following statements evaluate the log-likelihood function of the VAR(1) model:

```
call varmalik(lnl,yt,phi) sigma=sig;
labl = {"LogLik", "SumLogDet", "SSE"};
print lnl[rowname=labl];
```

Output 16.3.6 Log-Likelihood Function of VAR(1) Model (VARMALIK)

	lnl
LogLik	-113.4708
SumLogDet	2.5058678
SSE	224.43567

In Output 16.3.6, the first row displays the value of log-likelihood function; the second row shows the sum of the log determinant of the innovation variance; the last row displays the weighted sum of squares of residuals.

Nonstationary VAR Process

The following equation describes an 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

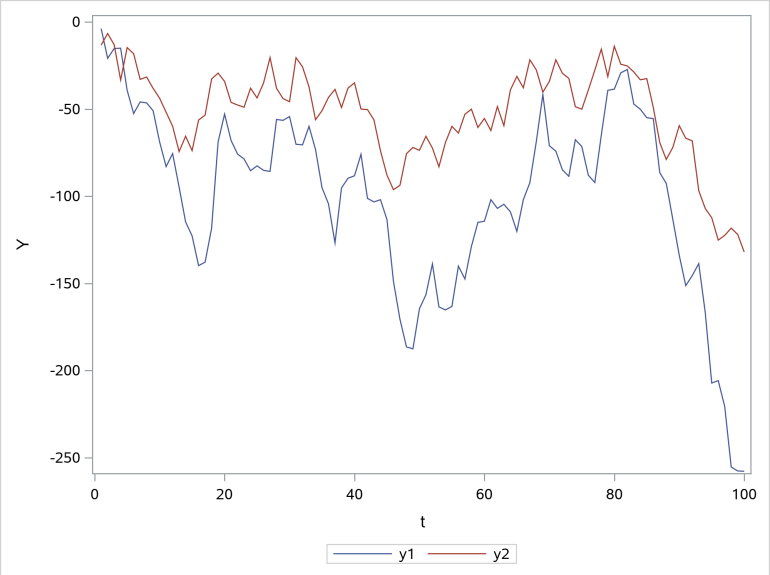
$$\Sigma = \begin{pmatrix} 100 & 0 \\ 0 & 100 \end{pmatrix} \text{ and } \mathbf{y}_0 = 0$$

In the equation, $\mathbf{y}_t$ is a 2×1 vector. On the right hand side of the equation, the 1×2 row vector $(1 \ \ -2)$ multiplies the vector $\mathbf{y}_{t-1}$ to form a scalar linear combination of components.

The following statements generate simulated data:

```
proc iml;
/* nonstationary  model */
sig = 100*i(2);
phi = {0.6 0.8, 0.1 0.8};   /* derived model */
call varmasim(yt,phi) sigma=sig n=100 seed=1324;
```

Output 16.3.7 Plot of Generated Nonstationary Vector Process (VARMASIM)



The nonstationary correlated processes are shown in Output 16.3.7.

The following statements compute the roots of the characteristic function:

```
call vtsroot(root,phi);
print root[c={R I 'Mod' 'ATan' 'Deg'}];
```

Output 16.3.8 Roots of Nonstationary VAR(1) Model (VTSROOT)

root				
R	I	Mod	ATan	Deg
1	0	1	0	0
0.4	0	0.4	0	0

In Output 16.3.8, the first column displays the real part (R) of the root of the characteristic function, and the second column displays the imaginary part (I). The third column shows that a modulus is greater than or equal to 1, so the series is 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 that are related to fractionally integrated time series analysis. The phenomenon of long memory can be observed in hydrology, finance, economics, and other fields. Unlike what occurs in a stationary process, the correlations between observations of a long-memory series slowly decay 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 0 and variance σ_ϵ^2, and B is the backshift operator such that $B^j \mathbf{y}_t = \mathbf{y}_{t-j}$. An ARFIMA$(p, d, q)$ model extends the ARFIMA$(0, d, 0)$ model and combines fractional differencing with an ARMA(p, q) model.

Consider an ARFIMA$(0, 0.4, 0)$ model that is represented as $(1 - B)^{0.4} y_t = \epsilon_t$, where $\epsilon_t \sim \text{iid } N(0, 2)$. The following statements accomplish several tasks:

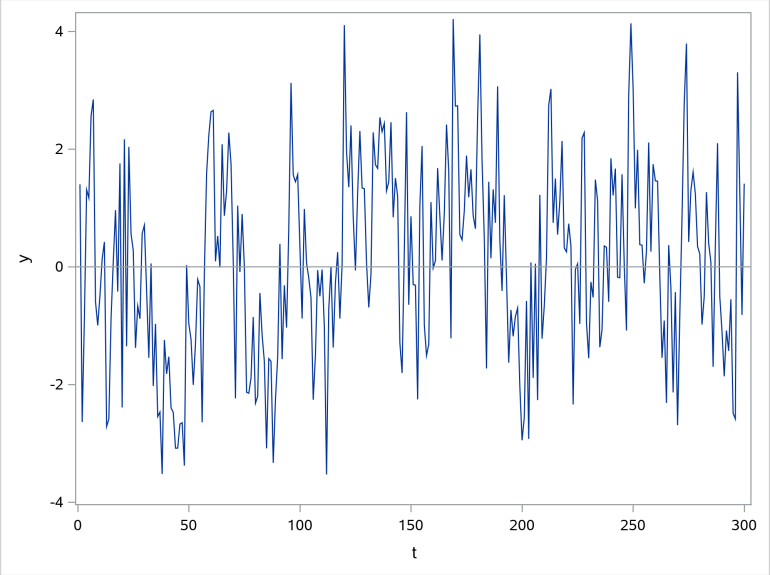
- generate 300 observations of simulated data
- obtain the fractionally differenced data
- compute the autocovariance function
- compute the log-likelihood function
- fit a fractionally integrated time series model to the data

The output is shown in Output 16.3.9 through Output 16.3.13.

```
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;
call fdif(zt, yt, d);

call farmalik(lnl, yt, d);
print lnl;

call farmafit(d, ar, ma, sigma, yt);
print d sigma;
```

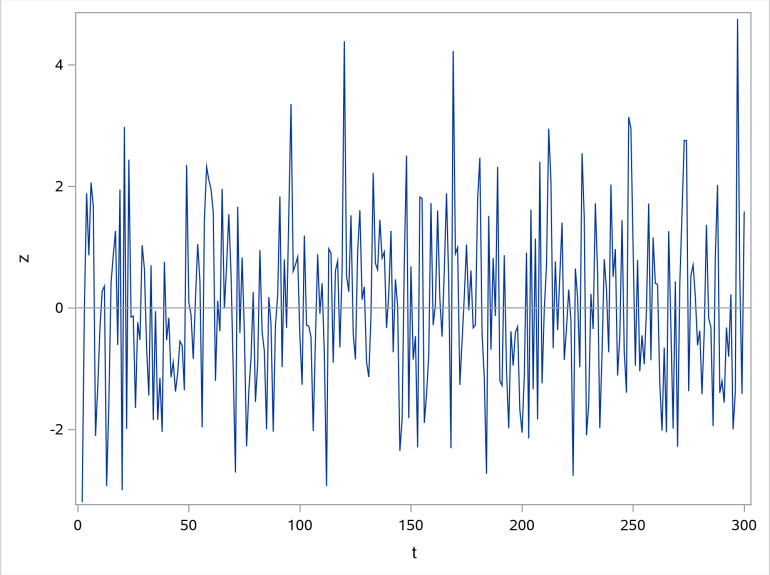
The FARMASIM function generates the data shown in Output 16.3.9.

Output 16.3.9 Plot of Generated ARFIMA(0,0.4,0) Process (FARMASIM)



The FDIF function creates the fractionally differenced process. Output 16.3.10 shows a white noise series.

Output 16.3.10 Plot of Fractionally Differenced Process (FDIF)



In Output 16.3.11, the first column displays the autocovariance function of the ARFIMA(0,0.4,0) model, and the second column displays the autocovariance function of the ARFIMA(0,0.05,0) model. The first column decays to zero more slowly than the second column.

Output 16.3.11 Autocovariance Functions of ARFIMA(0,0.4,0) and ARFIMA(0,0.05,0) Models (FARMACOV)

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

In Output 16.3.12, the first row value is the log-likelihood function of the ARFIMA(0,0.4,0) model. Because the default option of the estimates method is the conditional sum of squares, the last two rows of Output 16.3.12 contain missing values.

Output 16.3.12 Log-Likelihood Function of ARFIMA(0,0.4,0) Model (FARMALIK)

lnl
-101.0599
.
.

Output 16.3.13 shows parameter estimates for the simulated data. The parameter estimates are $d = 0.387$ and $\sigma^2 = 1.96$, whereas the true parameters of the data generating process are $d = 0.4$ and $\sigma^2 = 2$.

Output 16.3.13 Parameter Estimation of ARFIMA(0,0.4,0) Model (FARMAFIT)

d	sigma
0.386507	1.9610507

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)* ;

Time Series Analysis and Control Subroutines

The following SAS/IML functions from the **Tim**e **S**eries **A**nalysis and **C**ontrol (TIMSAC) package (Kitagawa and Akaike 1981; Ishiguro 1987) are deprecated:

CALL TSBAYSEA	Bayesian seasonal adjustment modeling
CALL TSDECOMP	time series decomposition analysis
CALL TSMLOCAR	locally stationary univariate AR model fitting
CALL TSMLOMAR	locally stationary multivariate AR model fitting
CALL TSMULMAR	multivariate AR model fitting
CALL TSPEARS	periodic AR model fitting
CALL TSPRED	ARMA model forecasting and forecast error variance
CALL TSROOT	polynomial roots or ARMA coefficients computation
CALL TSTVCAR	time-varying coefficient AR model estimation
CALL TSUNIMAR	univariate AR model fitting

The SAS/ETS product supports most of this functionality through procedures. For examples and documentation of these subroutines, see the SAS/IML 9.3 documentation.

References

De Jong, P. (1991). "The Diffuse Kalman Filter." *Annals of Statistics* 19:1073–1083.

Harvey, A. C. (1989). *Forecasting, Structural Time Series Models, and the Kalman Filter*. Cambridge: Cambridge University Press.

Ishiguro, M. (1987). "TIMSAC-84: A New Time Series Analysis and Control Package." In *Proceedings of the Business and Economic Section*, 33–42. Alexandria, VA: American Statistical Association.

Kitagawa, G., and Akaike, H. (1981). "On TIMSAC-78." In *Applied Time Series Analysis II*, edited by D. F. Findley, 499–547. New York: Academic Press.

Nelson, C. R., and Plosser, C. I. (1982). "Trends and Random Walks in Macroeconomic Time Series: Some Evidence and Implications." *Journal of Monetary Economics* 10:139–162.

Chapter 17
Nonlinear Optimization Examples

Contents

Overview **313**
Getting Started **315**
Details **324**
Global versus Local Optima 324
Kuhn-Tucker Conditions 325
Definition of Return Codes 326
Objective Function and Derivatives 326
Finite-Difference Approximations of Derivatives 331
Parameter Constraints 333
Options Vector 335
Termination Criteria 339
Control Parameters Vector 346
Printing the Optimization History 348
Nonlinear Optimization Examples **349**
Example 17.1: Chemical Equilibrium 349
Example 17.2: Network Flow and Delay 353
Example 17.3: Compartmental Analysis 356
Example 17.4: MLEs for Two-Parameter Weibull Distribution 366
Example 17.5: Profile-Likelihood-Based Confidence Intervals 368
Example 17.6: Survival Curve for Interval Censored Data 370
Example 17.7: A Two-Equation Maximum Likelihood Problem 376
Example 17.8: Time-Optimal Heat Conduction 380
References **384**

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 OPTMODEL procedure or the NLP procedure in SAS/OR software.

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 subsequent 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;
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 about the NLPTR subroutine and its arguments, see the section "NLPTR Call" on page 853. For details about the options vector, which is given by the OPTN vector in the preceding code, see the section "Options Vector" on page 335.

A portion of the output produced by the NLPTR subroutine is shown in Figure 17.1.

Figure 17.1 NLPTR Solution to the Rosenbrock Problem

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	-1.200000	-107.800000
2	X2	1.000000	-44.000000

Value of Objective Function = 12.1

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 Element	107.8	Radius	1

Figure 17.1 *continued*

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Trust Region 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
4	0	9	0	1.43279	0.3111	6.5134	0.918	0.372
5	0	10	0	1.13242	0.3004	4.9245	0	0.373
6	0	11	0	0.86905	0.2634	2.9302	0	0.291
7	0	12	0	0.66711	0.2019	3.6584	0	0.205
8	0	13	0	0.47959	0.1875	1.7354	0	0.208
9	0	14	0	0.36337	0.1162	1.7589	2.916	0.132
10	0	15	0	0.26903	0.0943	3.4089	0	0.270
11	0	16	0	0.16280	0.1062	0.6902	0	0.201
12	0	19	0	0.11590	0.0469	1.1456	0	0.316
13	0	20	0	0.07616	0.0397	0.8462	0.931	0.134
14	0	21	0	0.04873	0.0274	2.8063	0	0.276
15	0	22	0	0.01862	0.0301	0.2290	0	0.232
16	0	25	0	0.01005	0.00858	0.4553	0	0.256
17	0	26	0	0.00414	0.00590	0.4297	0.247	0.104
18	0	27	0	0.00100	0.00314	0.4323	0.0453	0.104
19	0	28	0	0.0000961	0.000906	0.1134	0	0.104
20	0	29	0	1.67872E-6	0.000094	0.0224	0	0.0569
21	0	30	0	6.9581E-10	1.678E-6	0.000336	0	0.0248
22	0	31	0	1.3127E-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	1.312677E-16	**Max Abs Gradient Element**	1.9772068E-7
Lambda	0	**Actual Over Pred Change**	0
Radius	0.0031401752		

ABSGCONV convergence criterion satisfied.

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	**X1**	1.000000	0.000000198
2	**X2**	1.000000	-0.000000105

Value of Objective Function = 1.312677E-16

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 Figure 17.2.

```
proc iml;
start F_ROSEN_LS(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_LS;

x = {-1.2 1.};
optn = {2 2};
call nlplm(rc,xres,"F_ROSEN_LS",x,optn);
quit;
```

Figure 17.2 NLPLM Solution Using the Least Squares Technique

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	-1.200000	-107.799999
2	X2	1.000000	-44.000000

Value of Objective Function = 12.1

Levenberg-Marquardt Optimization

Scaling Update of More (1978)

Gradient Computed by Finite Differences

CRP Jacobian Computed by Finite Differences

Parameter Estimates	2
Functions (Observations)	2

Optimization Start			
Active Constraints	0	Objective Function	12.1
Max Abs Gradient Element	107.7999987	Radius	2626.5613171

Figure 17.2 *continued*

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Ratio Between Actual and Predicted Change
1	0	4	0	2.18185	9.9181	17.4704	0.00804	0.964
2	0	6	0	1.59370	0.5881	3.7015	0.0190	0.988
3	0	7	0	1.32848	0.2652	7.0843	0.00830	0.678
4	0	8	0	1.03891	0.2896	6.3092	0.00753	0.593
5	0	9	0	0.78943	0.2495	7.2617	0.00634	0.486
6	0	10	0	0.58838	0.2011	7.8837	0.00462	0.393
7	0	11	0	0.34224	0.2461	6.6815	0.00307	0.524
8	0	12	0	0.19630	0.1459	8.3857	0.00147	0.469
9	0	13	0	0.11626	0.0800	9.3086	0.00016	0.409
10	0	14	0	0.0000396	0.1162	0.1781	0	1.000
11	0	15	0	0	0.000040	0	0	1.000

Optimization Results			
Iterations	11	**Function Calls**	16
Jacobian Calls	12	**Active Constraints**	0
Objective Function	0	**Max Abs Gradient Element**	0
Lambda	0	**Actual Over Pred Change**	1
Radius	0.0178062912		

GCONV2 convergence criterion satisfied.

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	**X1**	1.000000	0
2	**X2**	1.000000	0

Value of Objective Function = 0

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 about the NLPLM subroutine, see the section "NLPLM Call" on page 835.

Constrained Betts Function

The linearly constrained Betts function (Hock and Schittkowski 1981) is defined as

$$f(x) = 0.01x_1^2 + x_2^2 - 100$$

The boundary constraints are

$$\begin{aligned} 2 \leq \quad & x_1 \quad \leq 50 \\ -50 \leq \quad & x_2 \quad \leq 50 \end{aligned}$$

The linear constraint is

$$10x_1 - x_2 \geq 10$$

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 17.3.

```
proc iml;
start F_BETTS(x);
   f = .01 * x[1] * x[1] + x[2] * x[2] - 100.;
   return(f);
finish F_BETTS;

con = {  2. -50.  .   .,
        50.  50.  .   .,
        10.  -1. 1. 10.};
x = {-1. -1.};
optn = {0 2};
ods select ParameterEstimates LinCon ProblemDescription
    IterStart IterHist IterStop LinConSol;
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 about the NLPCG subroutine, see the section "NLPCG Call" on page 825. For details about the constraint matrix, which is represented by the CON matrix in the preceding code, see the section "Parameter Constraints" on page 333.

Figure 17.3 NLPCG Solution to Betts Problem

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

Linear Constraints

1	59.00000 :	10.0000 <= + 10.0000 * X1 - 1.0000 * X2

Parameter Estimates	2
Lower Bounds	2
Upper Bounds	2
Linear Constraints	1

Figure 17.3 *continued*

Optimization Start			
Active Constraints	0	Objective Function	-98.5376
Max Abs Gradient Element	2		

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
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 Element	0
Slope of Search Direction	-7.398365E-6		

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	X1	2.000000	0.040000	Lower BC
2	X2	-1.24028E-10	0	

Linear Constraints Evaluated at Solution

1 10.00000 = -10.0000 + 10.0000 * X1 - 1.0000 * X2

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 about the printed output, see the section "Printing the Optimization History" on page 348.

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;
ods select ProblemDescription IterStart IterHist IterStop ParameterEstimates;
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 335 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] = , 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 about the CALL NLPQN statement, see the section "NLPQN Call" on page 845.

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 17.4.

Figure 17.4 Solution to the Rosen-Suzuki Problem by the NLPQN Subroutine

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function
1	X1	1.000000	-3.000000	-3.000000
2	X2	1.000000	-3.000000	-3.000000
3	X3	1.000000	-17.000000	-17.000000
4	X4	1.000000	9.000000	9.000000

Parameter Estimates	4
Nonlinear Constraints	3

Optimization Start			
Objective Function	-19	**Maximum Constraint Violation**	0
Maximum Gradient of the Lagran Func	17		

Figure 17.4 *continued*

Iteration	Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Predicted Function Reduction	Step Size	Maximum Gradient Element of the Lagrange Function
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

Optimization Results			
Iterations	7	**Function Calls**	9
Gradient Calls	9	**Active Constraints**	2
Objective Function	-44.00000026	**Maximum Constraint Violation**	9.1176306E-8
Maximum Projected Gradient	0.0002265341	**Value Lagrange Function**	-44
Maximum Gradient of the Lagran Func	0.00022158	**Slope of Search Direction**	-5.097332E-7

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange 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

In addition to the standard iteration history, the NLPQN subroutine includes the following 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 can 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;
```

Figure 17.5 Two Minima

xra	
1	-6.517E-9

xrb	
4.0000002	-7.451E-9

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 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:

$$\begin{aligned} &\text{minimize } f(x), && x \in \mathcal{R}^n, \text{ subject to} \\ &c_i(x) = 0, && i = 1, \ldots, m_e \\ &c_i(x) \geq 0, && i = m_e + 1, \ldots, m \end{aligned}$$

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

$$\begin{aligned} &c_i(x^*) = 0, && i = 1, \ldots, m_e \\ &c_i(x^*) \geq 0, \quad \lambda_i^* \geq 0, \quad \lambda_i^* c_i(x^*) = 0, && i = m_e + 1, \ldots, m \\ &\nabla_x L(x^*, \lambda^*) = 0 \end{aligned}$$

- second-order condition

  Each nonzero vector $y \in \mathcal{R}^n$ with

$$y^T \nabla_x c_i(x^*) = 0 i = 1, \ldots, m_e, \quad \text{and } \forall i \in m_e + 1, \ldots, m; \lambda_i^* > 0$$

  satisfies

$$y^T \nabla_x^2 L(x^*, \lambda^*) y > 0$$

In practice, you cannot expect the constraint functions $c_i(x^*)$ to vanish within machine precision, and determining the set of active constraints at the solution x^* might 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 17.1 gives the reason for termination associated with each return code.

Table 17.1 Summary of Return Codes

Code	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 where it indicates 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*") accept 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 section "Numerical Considerations" on page 356 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 might 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 $x_j \geq$ 1E–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 might not ensure feasibility.
- Sometimes the preferred way is difficult to implement. An alternative is to make the function module return a missing value for infeasible inputs. This can 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 $\mathbf{J}$ 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 are very precise. Occasionally, difficult objective functions and zero x coordinates cause problems. You can use the *par* argument to specify the number of accurate digits 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 331.

NOTE: For some difficult applications, the finite-difference approximations of derivatives that are generated by default might not be precise enough to solve the optimization or least squares problem. In such cases, you might be able to specify better derivative approximations by using a better approximation formula. You can submit your own finite-difference approximations by using the IML module "*grd,*" "*hes,*" "*jac,*" or "*jacnlc.*" See Example 17.3 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 call 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;
```

Similarly, the following statements specify a module for the Rosenbrock function *when considered as a least squares problem*. They also specify the Jacobian matrix of the least squares functions.

```
start F_ROSEN_LS(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_LS;

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 can 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 as follows:

```
proc iml;
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 = sqrt(5) * r2;
      r3 = x[i+1] - 2. * x[i+2];
      f3 = r3 * r3;
      r4 = x[i] - x[i+3];
      f4 = sqrt(10) * r4 * r4;
      f = f + f1 * f1 + r2 * f2 + f3 * f3 + r4 * r4 * f4;
   end;
   f = 0.5 * 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 = sqrt(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 = sqrt(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 = sqrt(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 = sqrt(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 module "*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 et al. (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 et al. (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.

$$\begin{aligned}
\frac{\partial^2 f}{\partial x_i^2} &= \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2} \\
\frac{\partial^2 f}{\partial x_i \partial x_j} &= \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}
\end{aligned}$$

- 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 et al. (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 switches 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 $\max(1E{-}6, 100 \times \text{GTOL threshold})$. The 1E–6 ensures that the switch is performed even if you set the GTOL threshold to zero.

The algorithm of Gill et al. (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.

$$(\nabla c_i) = \left(\frac{\partial c_i}{\partial x_j} \right), \quad i = 1, \ldots, nc, \quad j = 1, \ldots, n$$

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 might 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:

$$\sum_{j=1}^{n} a_{ij} x_j \quad (\le \mid = \mid \ge) \quad b_i, \quad i = 1, \ldots, c$$

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 $\geq$ inequality constraint.
- blc$[n+1] = -1$ indicates a $\leq$ 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:

$$\begin{array}{rcrcl} 2 & \leq & x_1 & \leq & 100 \\ & & x_2 & \leq & 40 \\ 0 & \leq & x_4 & & \end{array}$$

$$\begin{array}{rcrcrcrcl} 4x_1 & + & 3x_2 & - & x_3 & & & \leq & 30 \\ & & x_2 & & & + & 6x_4 & \geq & 17 \\ x_1 & - & x_2 & & & & & = & 8 \end{array}$$

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

$$\begin{array}{rcll} c_i(x) & = & 0, & i = 1, \ldots, nec \\ c_i(x) & \geq & 0, & i = nec+1, \ldots, nc \end{array}$$

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 17.2 gives a summary of the available options.

Table 17.2 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 $\mathbf{D}^{(0)} = \mathrm{diag}(d_i^{(0)})$, 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 $\mathbf{D}^{(0)} = \mathrm{diag}(d_i^{(0)})$ is updated as indicated in the following table.

Value of *opt*[3]	Scaling Update
0	No scaling is done.
1	Moré (1978) scaling update: $d_i^{(k+1)} = \max\left(d_i^{(k)}, \sqrt{\max(\lvert G_{i,i}^{(k)}\rvert, \epsilon)}\right)$
2	Dennis, Gay, and Welsch (1981) scaling update: $d_i^{(k+1)} = \max\left(0.6 * d_i^{(k)}, \sqrt{\max(\lvert G_{i,i}^{(k)}\rvert, \epsilon)}\right)$
3	d_i is reset in each iteration: $d_i^{(k+1)} = \sqrt{\max(\lvert G_{i,i}^{(k)}\rvert, \epsilon)}$

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 section "Control Parameters Vector" on page 346). 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) and Powell (1982b). You can specify the approach of Powell (1978a) 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 by using an algorithm of Gill et al. (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 et al. (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 et al. (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 section "Finite-Difference Approximations of Derivatives" on page 331 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 section "Objective Function and Derivatives" on page 326 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 can 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 that correspond 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 17.3 and Table 17.4 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 17.3 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 17.4 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_{\mathrm{end}} =$1E–4. Convergence to small values of ρ_{end} can take many calls of the function and constraint modules and might 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.

$$\frac{|f_{hi}^{(k)} - f_{lo}^{(k)}|}{\max(|f_{hi}^{(k)})|, \mathrm{FSIZE})} \leq \mathrm{FTOL}$$

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 $x_0^{(k)}, \ldots, x_n^{(k)}$.

$$\sqrt{\frac{1}{n+1}\sum_l (f(x_l^{(k)}) - \overline{f}(x^{(k)}))^2} \leq \text{FTOL2}$$

  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.

$$|f_{hi}^{(k)} - f_{lo}^{(k)}| \leq \text{ABSFTOL}$$

  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.

$$\frac{\max_j |x_j^{lo} - x_j^{hi}|}{\max(|x_j^{lo}|, |x_j^{hi}|, \text{XSIZE})} \leq \text{XTOL}$$

  The default is *tc*[8] = 1E–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, $\delta^{(k)}$.

$$\begin{aligned} \alpha^{(k)} &\leq \text{ABSXTOL} \\ \delta^{(k)} &\leq \text{ABSXTOL} \end{aligned}$$

  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] = 1E–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.

  $$\frac{g(x^{(k)})^T [G^{(k)}]^{-1} g(x^{(k)})}{\max(|f(x^{(k)})|, \mathrm{FSIZE})} \leq \mathrm{GTOL}$$

  where FSIZE is defined by *tc*[10]. For the NLPCG technique (where a reliable Hessian estimate is not available),

  $$\frac{\| g(x^{(k)}) \|_2^2 \quad \| s(x^{(k)}) \|_2}{\| g(x^{(k)}) - g(x^{(k-1)}) \|_2 \max(|f(x^{(k)})|, \mathrm{FSIZE})} \leq \mathrm{GTOL}$$

  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.

  $$\max_j \frac{|g_j(x^{(k)})|}{\sqrt{f(x^{(k)}) G^{(k)}_{j,j}}} \leq \mathrm{GTOL2}$$

  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.

  $$\max_j |g_j(x^{(k)})| \leq \mathrm{ABSGTOL}$$

  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.

  $$\frac{|f(x^{(k)}) - f(x^{(k-1)})|}{\max(|f(x^{(k-1)})|, \mathrm{FSIZE})} \leq \mathrm{FTOL}$$

  where FSIZE is defined by *tc*[10]. The default is *tc*[7] = $10^{-\mathrm{FDIGITS}}$, 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:

$$\begin{aligned} df^{(k)} &= -g^{(k)T} s^{(k)} - \frac{1}{2} s^{(k)T} G^{(k)} s^{(k)} \\ &= -\frac{1}{2} s^{(k)T} g^{(k)} \\ &\leq \text{FTOL2} \end{aligned}$$

  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.

$$|f(x^{(k-1)}) - f(x^{(k)})| \leq \text{ABSFTOL}$$

  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.

$$\frac{\max_j |x_j^{(k)} - x_j^{(k-1)}|}{\max(|x_j^{(k)}|, |x_j^{(k-1)}|, \text{XSIZE})} \leq \text{XTOL}$$

  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.

$$\| x^{(k)} - x^{(k-1)} \|_2 \leq \text{ABSXTOL}$$

  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 et al. (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.

  $$\frac{|g(x^{(k)})s(x^{(k)})| + \sum_{i=1}^{m} |\lambda_i c_i(x^{(k)})|}{\max(|f(x^{(k)})|, \mathrm{FSIZE})} \leq \mathrm{GTOL}$$

  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.

  $$\max_j |\{\nabla_x L(x^{(k)}, \lambda^{(k)})\}_j| \leq \mathrm{ABSGTOL}$$

  The default is *tc*[6] = 1E–5.

- **tc[8]**
  specifies the FTOL2 criterion, which requires that the predicted function reduction be small.

  $$|g(x^{(k)})s(x^{(k)})| + \sum_{i=1}^{m} |\lambda_i c_i| \leq \mathrm{FTOL2}$$

  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 17.5 summarizes the uses of the *par* argument for the optimization and least squares subroutines.

Table 17.5 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 *i*th constraint is considered an active constraint if the point $x^{(k)}$ satisfies the condition

$$\left| \sum_{j=1}^{n} a_{ij} x_j^{(k)} - b_i \right| \leq \text{LCEPS}(|b_i| + 1)$$

where LCEPS is the value of *par*[3] and a_{ij} and b_i are defined as in the section "Parameter Constraints" on page 333. 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 never becomes 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

$$par[4] = \pm \min\left(0.01, \max\left(0.1 \times \text{ABSGTOL},\ \ 0.001 \times \text{gmax}^{(k)}\right)\right)$$

  where the positive value is for maximization and the negative value is for minimization. ABSGTOL is the value of the absolute gradient criterion, and $\text{gmax}^{(k)}$ 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 can 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 335.

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:

- Iteration: the iteration number.
- Restarts: the number of iteration restarts.
- Function Calls: the number of function calls.
- Active Constraints: the number of active constraints.
- Objective Function: the value of the optimization criterion.
- Objective Function Change: the difference between adjacent function values.
- Max Abs Gradient Element: 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 that corresponds to each subroutine in the section "Nonlinear Optimization and Related Subroutines" on page 822.

- **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 17.1: 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.

$$x_j \geq 0, \quad j = 1, \ldots, n$$

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.

Optimization Start			
Active Constraints	3	**Objective Function**	-45.05516448
Max Abs Gradient Element	4.4710303342	**Radius**	1

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Trust Region Radius
1	0	2	3	-47.33413	2.2790	4.3611	2.456	1.000
2	0	3	3	-47.70051	0.3664	7.0045	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	90.185	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	8.806E-6	0	0.00332

Optimization Results			
Iterations	10	**Function Calls**	13
Hessian Calls	11	**Active Constraints**	3
Objective Function	-47.76109086	**Max Abs Gradient Element**	6.8221615E-6
Lambda	0	**Actual Over Pred Change**	0
Radius	0.0033215668		

The output lists the optimal parameters with the gradient.

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	0.040668	-9.785056
2	X2	0.147730	-19.570111
3	X3	0.783154	-34.792169
4	X4	0.001414	-12.968920
5	X5	0.485247	-25.937841
6	X6	0.000693	-22.753977
7	X7	0.027399	-28.190991
8	X8	0.017947	-15.222060
9	X9	0.037314	-30.444120
10	X10	0.096871	-25.007114

The three equality constraints are satisfied at the solution.

Linear Constraints Evaluated at Solution

```
1 ACT -4.996E-16 = -2.0000 + 1.0000 * X1 + 2.0000 * X2 + 2.0000 * X3 + 1.0000 * X6 + 1.0000 * X10
2 ACT  1.249E-16 = -1.0000 + 1.0000 * X4 + 2.0000 * X5 + 1.0000 * X6 + 1.0000 * X7
3 ACT -3.886E-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

Active Constraint	Lagrange Multiplier
Linear EC [1]	-9.785055
Linear EC [2]	-12.968922
Linear EC [3]	-15.222061

Projected Gradient

Free Dimension	Projected Gradient
1	-0.000000106
2	0.000000645
3	0.000000644
4	-0.000006822
5	-0.000003999
6	-0.000003761
7	-0.000001901

Example 17.2: Network Flow and Delay

The following example is taken from the user's guide of the GINO program (Liebman et al. 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

$$\sum_i x_{ij} = \sum_i x_{ji} \quad , \quad j = 1, \ldots, n$$

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

$$\begin{aligned} 0 \leq \quad & x_{12}, x_{32}, x_{34} \quad \leq 10 \\ 0 \leq \quad & x_{13}, x_{24} \quad \leq 30 \end{aligned}$$

and the flow constraints are

$$\begin{aligned} x_{13} &= x_{32} + x_{34} \\ x_{24} &= x_{12} + x_{32} \\ x_{12} + x_{13} &= x_{24} + x_{34} \end{aligned}$$

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. The following code finds the maximum flow. The optimal solution is shown in Output 17.2.1.

```
proc iml;
start MAXFLOW(x);
   f = x[3] + x[4];
   return(f);
finish MAXFLOW;

/* constraints: lower and upper traffic limits; */
```

```
/* what flows into an intersection must flow out */
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 2};
ods select ParameterEstimates#2;
CALL NLPCG(xres,rc,"MAXFLOW",x,optn,con);
```

Output 17.2.1 Optimizing Flow

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound 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

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

$$\begin{aligned} 0 \leq \quad x_{12}, x_{32}, x_{34} \quad &\leq 10 \\ 0 \leq \quad x_{13}, x_{24} \quad &\leq 30 \end{aligned}$$

$$
\begin{aligned}
x_{13} &= x_{32} + x_{34} \\
x_{24} &= x_{12} + x_{32} \\
x_{24} + x_{34} &= F = 5
\end{aligned}
$$

In the following code, the NLPNRR subroutine is used to solve the minimization problem. The optimal solution is shown in Output 17.2.2.

```
title 'Minimize Total Delay in Network';
proc iml;
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);
```

Output 17.2.2 Parameter Estimates

Minimize Total Delay in Network

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound 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	

The active constraints and corresponding Lagrange multiplier estimates (costs) are shown in Output 17.2.3.

Output 17.2.3 Constraints and Lagrange Multiplier Estimates

Linear Constraints Evaluated at Solution

1	ACT	0	=	0	+	1.0000	*	X2	- 1.0000	* X3 - 1.0000 * X5
2	ACT	0	=	0	+	1.0000	*	X1	+ 1.0000	* X3 - 1.0000 * X4
3	ACT	4.4409E-16	=	-5.0000	+	1.0000	*	X4	+ 1.0000	* X5

Output 17.2.3 *continued*

First Order Lagrange Multipliers	
Active Constraint	Lagrange Multiplier
Lower BC X3	0.924702
Linear EC [1]	5.702479
Linear EC [2]	5.777777
Linear EC [3]	11.480257

Example 17.3: 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 of the finite differences worthless.

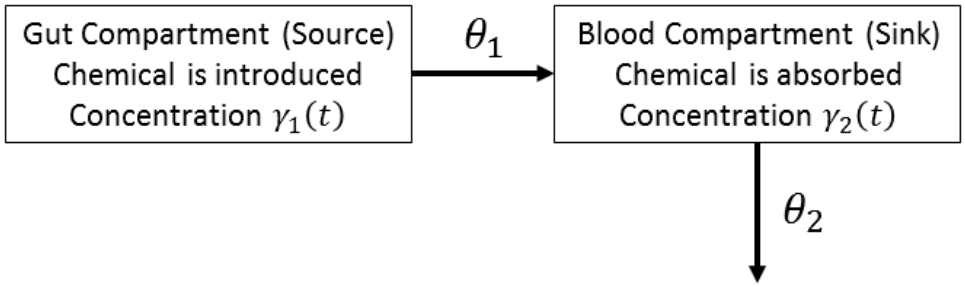
A coarse computation of the solution of the differential equation cannot be accompanied by very fine computation of the finite-difference estimates of the gradient and the Hessian. That is, you cannot set the precision of the differential equation solver to be 1E–4 and perform the finite difference estimation 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 $O(\epsilon^{\frac{1}{3}})$. (You can specify the step size h in the finite-difference formulas.)

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 consists 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:

Output 17.3.1 Model of Diffusion



The rates of flow of the drug are described by the following pair of ordinary differential equations:

$$\begin{aligned} \frac{d\gamma_1(t)}{dt} &= -\theta_1\gamma_1(t) \\ \frac{d\gamma_2(t)}{dt} &= \theta_1\gamma_1(t) - \theta_2\gamma_2(t) \end{aligned}$$

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 are given by

$$\begin{aligned} \gamma_1(0) &= \theta_3 \\ \gamma_2(0) &= 0 \end{aligned}$$

Also, a nonnegativity constraint is imposed on the parameters θ_1, θ_2, and θ_3, although for numerical purposes, you might 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 about the ODE call, see the section "ODE Call" on page 856. The output from the optimization process is shown in Output 17.3.2.

```
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;
close 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 . . . . };
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;
```

Output 17.3.2 Printed Output for Tetracycline Diffusion Problem

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound Constraint
1	X1	0.100000	76.460655	0	.
2	X2	0.300000	-48.210699	0	.
3	X3	10.000000	1.660423	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**	4.1469872335
Max Abs Gradient Element	76.460655087	**Radius**	1

Output 17.3.2 *continued*

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Slope of Search Direction
1	0	5	0	3.11516	1.0318	124.0	67.173	-8.016
2	0	6	0	0.89568	2.2195	14.1753	1.885	-5.011
3	0	7	0	0.32332	0.5724	3.7036	1.188	-0.788
4	0	8	0	0.14723	0.1761	2.6339	0	-0.121
5	0	9	0	0.07419	0.0730	2.2782	0	-0.0589
6	0	10	0	0.06545	0.00873	1.5767	0	-0.0071
7	0	11	0	0.06413	0.00132	1.0894	0	-0.0009
8	0	12	0	0.06331	0.000825	0.5409	0	-0.0006
9	0	13	0	0.06286	0.000448	0.0801	0	-0.0004
10	0	14	0	0.06278	0.000082	0.0167	0.191	-0.0001
11	0	15	0	0.06276	0.000020	0.0225	0	-134E-7
12	0	16	0	0.06274	0.000021	0.0158	0	-154E-7
13	0	18	0	0.06270	0.000034	0.0348	0.119	-289E-7
14	0	19	0	0.06265	0.000049	0.0432	0.831	-386E-7
15	0	21	0	0.06207	0.000585	0.0719	0	-0.0003
16	0	23	0	0.05916	0.00291	0.4169	0	-0.0025
17	0	24	0	0.05708	0.00208	0.5178	0	-0.0011
18	0	27	0	0.04852	0.00856	1.0888	0	-0.0083
19	0	28	0	0.04478	0.00373	1.8060	0.646	-0.0064
20	0	29	0	0.04115	0.00363	0.6907	0	-0.0030
21	0	30	0	0.03954	0.00161	0.4207	0.866	-0.0012
22	0	31	0	0.03703	0.00250	0.5907	0.400	-0.0031
23	0	32	0	0.03630	0.000730	0.1192	0	-0.0008
24	0	35	0	0.03587	0.000438	0.0947	0.218	-0.0004
25	0	36	0	0.03570	0.000169	0.0472	0	-0.0001
26	0	37	0	0.03565	0.000042	0.0406	0	-307E-7
27	0	39	0	0.03565	5.29E-6	0.0288	0.972	-148E-7
28	0	40	0	0.03565	1.135E-6	0.0227	0	-519E-8
29	0	41	0	0.03565	9.587E-7	0.00777	0	-387E-8
30	0	43	0	0.03565	8.37E-8	0.00415	1.300	-778E-9
31	0	44	0	0.03565	8.723E-8	0.000589	0	-124E-9
32	0	53	0	0.03565	6.92E-12	0.00193	401.8	-67E-12

Optimization Results			
Iterations	32	**Function Calls**	54
Gradient Calls	34	**Active Constraints**	0
Objective Function	0.0356470793	**Max Abs Gradient Element**	0.0019283169
Slope of Search Direction	-6.68927E-11	**Radius**	1

GCONV convergence criterion satisfied.

Note: At least one element of the (projected) gradient is greater than 1e-3.

Output 17.3.2 *continued*

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	0.183312	0.001928
2	X2	0.433839	0.001240
3	X3	5.983952	-0.000029320

Value of Objective Function = 0.0356470793

The differential equation model is linear, and in fact, it can be solved by 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 enable 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. The revised code follows:

```
proc iml;
use tetra;
read all into tetra;
close tetra;

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;

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 . . . . };
con    = j(1,3,0.);
itheta = { .1  .3  10};
eps    = 1.e-11;

ods select ParameterEstimates#2;
call nlpdd(rc,rx,"f",itheta) blc=con opt=opt tc=tc par=par;
```

The parameter estimates, which are shown in Output 17.3.3, are close to those obtained by the first solution.

Output 17.3.3 Second Set of Parameter Estimates for Tetracycline Diffusion

Optimization Results			
Parameter Estimates			
N	Parameter	Estimate	Gradient Objective Function
1	X1	0.183024	0.000000101
2	X2	0.434484	-0.000000117
3	X3	5.995273	8.8535431E-9

Because of the nature of the closed form of the solution, you might want to add an additional constraint to guarantee that $\theta_2 \neq \theta_1$ at any time during the optimization. This prevents 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 byproducts. The model is given by

$$\begin{aligned}
\frac{d\gamma_1(t)}{dt} &= -(\theta_1 + \theta_4)\gamma_1(t)\iota(t,\theta_5) \\
\frac{d\gamma_2(t)}{dt} &= [\theta_1\gamma_1(t) - (\theta_2 + \theta_3)\gamma_2(t)]\iota(t,\theta_5) \\
\frac{d\gamma_3(t)}{dt} &= [\theta_4\gamma_1(t) + \theta_2\gamma_2(t)]\iota(t,\theta_5)
\end{aligned}$$

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 program illustrates how to combine the NLPQN subroutine and the ODE subroutine to estimate the parameters θ_i in this model. The parameter estimates are shown in Output 17.3.4.

```
data oil(drop=temp);
   input temp time bitumen oil;
   cards;
673     5      0.      0.
```

```
673     7      2.2     0.
673    10     11.5      .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;
close oil;


/*************************************************************/
/* 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};
   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.};

ods select ParameterEstimates#2;
call nlpqn(rc,rx,"f",itheta)  blc=con opt=opt tc=tc par=par;
```

Output 17.3.4 Parameter Estimates for Oil Shale Pyrolysis

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	0.022602	1371870
2	X2	0.007750	-50656
3	X3	0.017021	7012.613014
4	X4	0.010851	1318815
5	X5	0.998383	-6.056502

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. The revised code follows. The parameter estimates are shown in Output 17.3.5.

```
proc iml;
use oil;
read all into a;
close oil;

start ins(from,into) global(permm);
```

```
   in    =  into // from;
   x     =  i(nrow(in));
   permm = inv(x[rank(in),]);
   return(permm*in);
finish;

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;

opt = {0 2 0 1 };
tc = repeat(.,1,12);
tc[1] = 100;
tc[7] = 1.e-7;
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;
```

Output 17.3.5 Second Set of Parameter Estimates for Oil Shale Pyrolysis

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	0.017178	-0.019007
2	X2	0.008912	0.098220
3	X3	0.020007	-0.041127
4	X4	0.010494	0.298366
5	X5	7.771452	-0.000147

Example 17.4: 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;
minx = carcin[><];
rang = carcin[<>] - minx;
```

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

$$L(\theta,\sigma,c) = \frac{c^m}{\sigma^m}\prod_{i\in D}\left(\frac{t_i-\theta}{\sigma}\right)^{c-1}\prod_{i=1}^{p}\exp\left\{-\left(\frac{t_i-\theta}{\sigma}\right)^c\right\}$$

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 $(\hat{\sigma},\hat{c})$. The following statements define the function:

```
/*--- 1. Two parameter estimation: for thet = 0 ---*/
start f_weib2(x) global(carcin,thet);
   /* use x[1]=sig and x[2]=c */
   p = ncol(carcin); m = p - 2;
   temp = carcin - thet;
   sum1 = sum(log(temp[1:m]));
   sum2 = sum((temp / x[1])##x[2]);
   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);
   /* use x[1]=sig and x[2]=c */
   p = ncol(carcin); m = p - 2;
   g = j(1,2,0.);
   temp = carcin - thet;
   sum1 = sum(log(temp[1:m]));
   sum2 = sum((temp / x[1])##x[2]);
   sum3 = sum(((temp / x[1])##x[2]) # (log(temp / x[1])));
   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 $c, \sigma \geq 10^{-6}$. 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,xopt,"f_weib2",x0,optn,con,,,,"g_weib2");
fopt = f_weib2(xopt);
```

The results shown in Output 17.4.1 are the same as those given in Lawless (1982).

Output 17.4.1 Parameter Estimates for Carcinogen Data

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	234.318611	1.3362896E-9
2	X2	6.083147	-7.850939E-9

The following code computes confidence intervals based on the asymptotic normal distribution. These are compared with the profile-likelihood-based confidence 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.

```
/*--- Compute confidence interval for x at optimum ---*/
/* compute Hessian at optimum */
xopt = xopt`;
call nlpfdd(f,g,hes2,"f_weib2",xopt,,"g_weib2");
hin2 = inv(hes2);

/* quantile of normal distribution */
prob = 0.05;
stderr = sqrt(abs(vecdiag(hin2)));
z = quantile("Normal", 1 - prob/2);
xlb = xopt - z * stderr;
xub = xopt + z * stderr;
results = xlb || xopt || xub;
print results[L="Normal Distribution Confidence Interval"
              c={"Lower Bound" "Estimate" "UpperBound"}
              r={"sigma" "c"}];
```

Output 17.4.2 Confidence Interval Based on Normal Distribution

Normal Distribution Confidence Interval			
	Lower Bound	Estimate	UpperBound
sigma	215.41298	234.31861	253.22425
c	3.9894574	6.0831471	8.1768368

Example 17.5: 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 $x = (\beta, \omega)$:

$$\begin{bmatrix} \ell(\theta) - \ell^* \\ \frac{\partial \ell}{\partial \omega}(\theta) \end{bmatrix} = 0$$

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 statements define the module for the system of $n = 2$ nonlinear equations to be solved in terms of the modules that are defined in the previous section:

```
start f_plwei2(x) global(carcin,ipar,lstar);
   /* use x[1]=sig, x[2]=c, thet */
   like = f_weib2(x);
   grad = g_weib2(x);
   grad[ipar] = like - lstar;
   return(grad`);
finish f_plwei2;
```

The following statements implement 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. The results, shown in Output 17.5.1, are close to the results shown in Output 17.4.2.

```
/* quantile of chi**2 distribution */
chqua = cinv(1-prob,1); lstar = fopt - .5 * chqua;
/* set number of equations = 2, and print parameter = 0 */
optn = {2 0};                    /* Dont print in NLPLM */
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];
   alfa = - (hes2[ipar,ipar] - delt` * hes2[ind,ipar]);
   if alfa > 0 then do;
   alfa = .5 * sqrt(chqua / alfa);
   end; 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;
results = xlb || xopt || xub;
print results[L="Profile-Likelihood Confidence Interval"
              c={"Lower Bound" "Estimate" "UpperBound"}
              r={"sigma" "c"}];
```

Output 17.5.1 Confidence Interval Based on Profile Likelihood

Profile-Likelihood Confidence Interval			
	Lower Bound	Estimate	UpperBound
sigma	215.1963	234.31861	255.2157
c	4.1344126	6.0831471	8.3063797

Example 17.6: 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 $I_j = (q_j, p_j],\ 1 \le j \le m$, 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 $\{T_i \in (L_i, R_i], 1 \le i \le n\}$ can be constructed in terms of the pseudo-parameters $\theta_j = \Pr[T_i \in I_j], 1 \le i \le m$. The conditional likelihood of $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$ 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, $\hat{\theta}_j, 1 \le j \le m$, yield an estimator $\hat{S}(t)$ of the survival function $S(t)$, which is given by

$$\hat{S}(t) = \begin{cases} 1 & t \le q_1 \\ \sum_{i=j+1}^{m} \hat{\theta}_i & p_j \le t \le q_{j+1}, 1 \le j \le m-1 \\ 0 & t \ge p_m \end{cases}$$

$\hat{S}(t)$ remains undefined in the intervals (q_j, p_j) where the function can decrease in an arbitrary way. The asymptotic covariance matrix of $\widehat{\theta}$ 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
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};
```

```
close carcin;

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;
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
*************************************************************/
q= q`;
p= p`;
theta= rx`;
ParamEst = q || p || theta;
print ParamEst[L="Parameter Estimates"
              colname={Q P THETA} format=BEST9.];

/*************************************************************
 print survival curve estimates and confidence limits
*************************************************************/
left= {0} // p;
right= q // p[nparm];
sdf= {1} // sdf // {0};
lcl= {.} // lcl //{.};
ucl= {.} // ucl //{.};
SurvEst = left || right || sdf || lcl || ucl;
print SurvEst[L="Survival Curve Estimates and 95% Confidence Intervals"
          colname={left right estimate lower upper}];
quit;
```

The iteration history produced by the NLPQN subroutine is shown in Output 17.6.1.

Output 17.6.1 Iteration History for the NLPQN Subroutine

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		

Output 17.6.1 *continued*

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
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**	0.0010060789
Slope of Search Direction	-3.033154E-7		

GCONV convergence criterion satisfied.

The estimates of the pseudo-parameter for the nonoverlapping intervals are shown in Output 17.6.2.

Output 17.6.2 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 17.6.3.

Output 17.6.3 Survival Estimates and Confidence Intervals

Survival Curve Estimates and 95% Confidence Intervals					
	LEFT	RIGHT	ESTIMATE	LOWER	UPPER
ROW1	0	20	1	.	.
ROW2	22	22	0.9500003	0.8824601	1
ROW3	24	26	0.8750015	0.7725135	0.9774894
ROW4	28	28	0.7750037	0.6455975	0.9044099
ROW5	30	30	0.6716688	0.5251576	0.81818
ROW6	32	32	0.5910674	0.4363312	0.7458036
ROW7	34	34	0.3492651	0.1972562	0.501274
ROW8	36	36	0.2619499	0.1194032	0.4044967
ROW9	38	38	0.203738	0.072003	0.3354731
ROW10	40	42	0.1455261	0.0293244	0.2617279
ROW11	44	46	0.058211	0	0.1360926
ROW12	48	48	0.0291055	0	0.0851723
ROW13	50	50	0	.	.

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 17.7: A Two-Equation Maximum Likelihood Problem

The following 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. The data can be entered by using the following statements:

```
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.

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:

```
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;
      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;
         return(.);
      end;
      lndet = lndet + log(abs(detj));
   end;

   /* 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 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. Part of the iteration history is shown in Output 17.7.1.

```
pr = j(1,5,0.001);
optn = {0 2};
tc = {. . . 0};
call nlpdd(rc, xr,"fiml", pr, optn,,tc);
quit;
```

Output 17.7.1 Iteration History for Two-Equation ML Problem

Optimization Start			
Active Constraints	1	**Objective Function**	-93.3278404
Max Abs Gradient Element	65.361558529		

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
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**	0.0010060789
Slope of Search Direction	-3.033154E-7		

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 17.7.2 Parameter Estimates

Optimization Results Parameter Estimates			
N	**Parameter**	**Estimate**	**Gradient Objective Function**
1	**X1**	0.050000	40.000241
2	**X2**	0.074999	40.000622
3	**X3**	0.099998	40.000887
4	**X4**	0.103335	39.999422
5	**X5**	0.080601	39.999337
6	**X6**	0.241802	39.999631
7	**X7**	0.087315	40.000599
8	**X8**	0.058212	39.999533
9	**X9**	0.058212	39.999533
10	**X10**	0.087315	40.000599
11	**X11**	0.029105	40.000082
12	**X12**	0.029105	40.000082

Example 17.8: Time-Optimal Heat Conduction

The following 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 and 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

$$\begin{aligned}
h(x) &= \int_0^1 \left(\sum_{i=1}^{30} \alpha_i(s)\rho_i(x) - k_0(s) \right)^2 ds \\
\alpha_i(s) &= \mu_i^2 A_i \cos(\mu_i s) \\
\rho_i(x) &= -\mu_i^2 \left[\exp\left(-\mu_i^2 \sum_{j=1}^{n} x_j^2 \right) - 2\exp\left(-\mu_i^2 \sum_{j=2}^{n} x_j^2 \right) + \cdots \right. \\
&\qquad \left. + (-1)^{n-1} 2\exp\left(-\mu_i^2 x_n^2\right) + (-1)^n \right] \\
k_0(s) &= 0.5(1 - s^2) \\
A_i &= \frac{2\sin\mu_i}{\mu_i + \sin\mu_i \cos\mu_i}, \\
\mu &= (\mu_1, \ldots, \mu_{30})', \text{ where } \mu_i \tan(\mu_i) = 1
\end{aligned}$$

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;
/* Vector mu[30] found by solving mu[j] * tan(mu[j]) = 1 */
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, as follows:

```
/* Vector a[nmu] depends only on mu and is computed once */
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 about the QUAD call, see the section "QUAD Call" on page 904. Here is the code:

```
/* 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;
```

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] = 1E–4. Here is the code:

```
title '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";
quit;
```

Part of the iteration history and the parameter estimates are shown in Output 17.8.1.

Output 17.8.1 Iteration History and Parameter Estimates

Hock & Schittkowski Problem #91 (1981) n=5, INSTEP=1

Parameter Estimates	5
Nonlinear Constraints	1

Optimization Start			
Objective Function	1.25	**Maximum Constraint Violation**	0.0952775105
Maximum Gradient of the Lagran Func	1.1433393501		

Output 17.8.1 *continued*

Iteration		Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Predicted Function Reduction	Step Size	Maximum Gradient Element of the Lagrange 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	0.34576	0.0690	0.9318	1.000	0.640
4		0	6	0.77689	0.0132	0.3496	1.000	1.328
5		0	7	0.54995	0.0240	0.3403	1.000	1.334
6		0	9	0.55322	0.0212	0.4327	0.242	0.423
7		0	10	0.75884	0.00828	0.1545	1.000	0.630
8		0	12	0.76720	0.00681	0.3810	0.300	0.248
9		0	13	0.94843	0.00237	0.2186	1.000	1.857
10		0	15	0.95661	0.00207	0.3707	0.184	0.795
11		0	16	1.11201	0.000690	0.2575	1.000	0.591
12		0	18	1.14602	0.000494	0.2810	0.370	1.483
13		0	20	1.19821	0.000311	0.1991	0.491	2.014
14		0	22	1.22804	0.000219	0.2292	0.381	1.155
15		0	23	1.31506	0.000056	0.0697	1.000	1.578
16		0	25	1.31759	0.000051	0.0876	0.100	1.007
17		0	26	1.35547	7.337E-6	0.0133	1.000	0.912
18		0	27	1.36136	1.644E-6	0.00259	1.000	1.064
19		0	28	1.36221	4.263E-7	0.000902	1.000	0.0325
20		0	29	1.36266	9.79E-10	2.063E-6	1.000	0.00273
21		0	30	1.36266	5.72E-12	1.031E-6	1.000	0.00011
22		0	31	1.36266	0	2.044E-6	1.000	0.00001

Optimization Results			
Iterations	22	**Function Calls**	32
Gradient Calls	24	**Active Constraints**	1
Objective Function	1.3626578337	**Maximum Constraint Violation**	0
Maximum Projected Gradient	3.4047917E-6	**Value Lagrange Function**	1.3626568149
Maximum Gradient of the Lagran Func	3.1335034E-6	**Slope of Search Direction**	-2.043677E-6

Hock & Schittkowski Problem #91 (1981) n=5, INSTEP=1

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Gradient Lagrange Function
1	X1	0.860193	1.720385	0.000001811
2	X2	6.8255933E-8	0.000000137	0.000000555
3	X3	0.643607	1.287214	-0.000000197
4	X4	-0.456614	-0.913228	0.000003134
5	X5	-6.02693E-9	-1.205386E-8	2.5536607E-8

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., eds. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* 10th printing. New York: Dover.

Al-Baali, M., and Fletcher, R. (1985). "Variational Methods for Nonlinear Least Squares." *Journal of the Operations Research Society* 36:405–421.

Al-Baali, M., and Fletcher, R. (1986). "An Efficient Line Search for Nonlinear Least Squares." *Journal of Optimization Theory and Applications* 48:359–377.

Anderson, B. D., and Moore, J. B. (1979). *Optimal Filtering*. Englewood Cliffs, NJ: 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.

Beale, E. M. L. (1972). "A Derivation of Conjugate Gradients." In *Numerical Methods for Nonlinear Optimization*, edited by F. A. Lootsma, 39–43. 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.

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 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 and Applications* 28:453–482.

Dennis, J. E., and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall.

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*. 2nd ed. Chichester, UK: John Wiley & Sons.

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 Transactions on Mathematical Software* 9:503–524.

George, J. A., and Liu, J. W. (1981). *Computer Solutions of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Gill, P. E., Murray, W., Saunders, M. A., and Wright, M. H. (1983). "Computing Forward-Difference Intervals for Numerical Optimization." *SIAM Journal on Scientific and Statistical Computing* 4:310–321.

Gill, P. E., 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.

Gill, P. E., Murray, W., and Wright, M. H. (1981). *Practical Optimization*. New York: Academic Press.

Goldfeld, S. M., Quandt, R. E., and Trotter, H. F. (1966). "Maximisation by Quadratic Hill-Climbing." *Econometrica* 34:541–551.

Hartmann, W. M. (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.* Vol. 187 of Lecture Notes in Economics and Mathematical Systems. Berlin: 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 Methods and Methods for Lifetime Data*. New York: John Wiley & Sons.

Liebman, J., Lasdon, L., Schrage, L., and Waren, A. (1986). *Modeling and Optimization with GINO*. Redwood City, CA: Scientific Press.

Lindström, P., and Wedin, P.-A. (1984). "A New Line-Search 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." In *Lecture Notes in Mathematics*, vol. 30, edited by G. A. Watson, 105–116. Berlin: Springer-Verlag.

Moré, J. J., Garbow, B. S., and Hillstrom, K. E. (1981). "Testing Unconstrained Optimization Software." *ACM Transactions on Mathematical Software* 7:17–41.

Moré, J. J., and Sorensen, D. C. (1983). "Computing a Trust-Region Step." *SIAM Journal on Scientific and Statistical Computing* 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." *Journal of the Royal Statistical Society, Series C* 22:86–91.

Polak, E. (1971). *Computational Methods in Optimization*. New York: Academic Press.

Powell, M. J. D. (1977). "Restart Procedures for the Conjugate Gradient Method." *Mathematical Programming* 12:241–254.

Powell, M. J. D. (1978a). "Algorithms for Nonlinear Constraints That Use Lagrangian Functions." *Mathematical Programming* 14:224–248.

Powell, M. J. D. (1978b). "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations." In *Lecture Notes in Mathematics*, vol. 630, edited by G. A. Watson, 144–175. Berlin: Springer-Verlag.

Powell, M. J. D. (1982a). "Extensions to Subroutine VF02AD." In *Systems Modeling and Optimization, Lecture Notes in Control and Information Sciences*, vol. 38, edited by R. F. Drenick and F. Kozin, 529–538. Berlin: Springer-Verlag.

Powell, M. J. D. (1982b). *VMCWD: A Fortran Subroutine for Constrained Optimization*. Technical Report DAMTP 1982/NA4, Department of Applied Mathematics and Theoretical Physics, University of Cambridge.

Powell, M. J. D. (1992). *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*. Technical Report DAMTP 1992/NA5, Department of Applied Mathematics and Theoretical Physics, University of Cambridge.

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." In *Optimization Techniques: Proceedings of the Eighth IFIP Conference on Optimization Techniques*. Berlin: Springer-Verlag.

Schittkowski, K. (1987). *More Test Examples for Nonlinear Programming Codes.* Vol. 282 of Lecture Notes in Economics and Mathematical Systems. Berlin: Springer-Verlag.

Schittkowski, K., and Stoer, J. (1979). "A Factorization Method for the Solution of Constrained Linear Least Squares Problems Allowing Subsequent Data Changes." *Numerische Mathematik* 31:431–463.

Turnbull, B. W. (1976). "The Empirical Distribution Function with Arbitrarily Grouped, Censored, and Truncated Data." *Journal of the Royal Statistical Society, Series B* 38:290–295.

Venzon, D. J., and Moolgavkar, S. H. (1988). "A Method for Computing Profile-Likelihood-Based Confidence Intervals." *Journal of the Royal Statistical Society, Series C* 37:87–94.

Wedin, P. A., and Lindström, P. (1987). *Methods and Software for Nonlinear Least Squares Problems*. Technical Report No. UMINF 133.87, Umeå University, Sweden.

Ziegel, E. R., and Gorman, J. W. (1980). "Kinetic Modelling with Multipurpose Data." *Technometrics* 27:352–357.

Chapter 18
Statistical Graphics

Contents

Overview of Statistical Graphics **387**
How the Graphs Are Created **388**
Summary of Graph Options **389**
Limitations of the ODS Statistical Graphics Subroutines **390**
Examples of Creating Graphs **391**
- Bar Charts 391
- Box Plots 393
- Histograms 395
- Scatter Plots 397
- Series Plots 399
- Matrix Heat Maps 400

Overview of Statistical Graphics

This chapter describes SAS/IML subroutines that enable you to create high-level statistical graphs. These subroutines use the SUBMIT statement and ENDSUBMIT statement to call the SGPLOT procedure, which displays the graph in the current ODS destination. These subroutines are implemented as part of the IMLMLIB library.

The following subroutines create ODS statistical graphs from data in SAS/IML matrices:

BAR call	creates a bar chart.
BOX call	creates a box plot.
HISTOGRAM call	creates a histogram.
SCATTER call	creates a scatter plot.
SERIES call	creates a series plot.

In addition, the following subroutines create a heat map to visualize data in a matrix:

HEATMAPCONT call	creates a heat map with a continuous color ramp.
HEATMAPDISC call	creates a heat map with a discrete color ramp.

The heat map subroutines are described and documented in Chapter 25, "Language Reference."

The traditional graphics subroutines (GSTART, GOPEN, GDRAW, GSHOW, and so forth) are deprecated as of SAS/IML 14.3. The subroutines have been removed from this documentation. The traditional graphics routines are no longer developed or recommended, although they are still supported. The SAS programmer is encouraged to use ODS graphics, either by calling PROC SGPLOT (as described in the next section) or by using the pre-written ODS graphics that are described in this chapter.

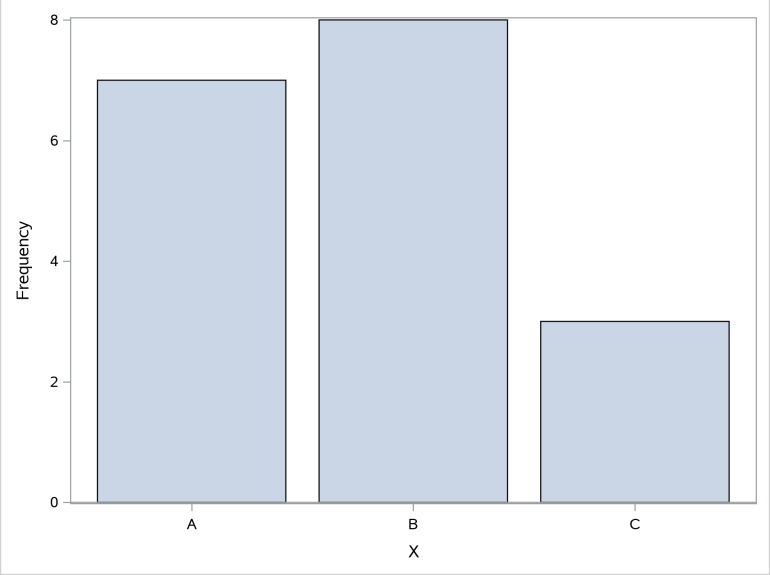
How the Graphs Are Created

This section provides a simple example that demonstrates how the ODS statistical graphics subroutines work.

Suppose you want to create a bar chart of some discrete data that are contained in a SAS/IML matrix. One way to create the bar chart would be to write the data to a SAS data set, quit PROC IML, and call the SGPLOT procedure. However, you might prefer to create the bar chart without exiting from PROC IML. You can do this by using the SUBMIT and ENDSUBMIT statements, as follows:

```
proc iml;
x = {[7]A [8]B [3]C};  /* repetition factors: 7 As, 8 Bs, and 3 Cs */
create Bar var {x}; append; close Bar;       /* write SAS data set */

submit;
   proc sgplot data=Bar;
      vbar x;
   run;
endsubmit;
```

Figure 18.1 Bar Chart



The result is shown in Figure 18.1. The graph is created by calling the SGPLOT procedure from within a SAS/IML program. Of course, you can also encapsulate these statements into a module that creates a bar chart from the data, as follows:

```
/* module to create a bar chart from data in X */
start BarChart(x);
   create Bar var {x}; append; close Bar;  /* write to SAS data set  */
   submit;
      proc sgplot data=Bar;                /* create the plot     */
         vbar x;
      run;
   endsubmit;
   call delete("Bar");                     /* delete the data set */
finish;

run BarChart(x);                           /* call the module     */
```

This program illustrates the basic idea of the ODS statistical graphics subroutines that are available in the IMLMLIB module library. The modules write data to a data set and call PROC SGPLOT to create a graph. The subroutines also accept additional parameters that determine options in the graph.

Summary of Graph Options

For each graph type, you must specify a vector of data. The BAR, BOX, and HISTOGRAM subroutines require at least one vector of data; the SCATTER and SERIES subroutines require two vectors of data. The remaining arguments are optional and specify additional data or parameters that set options in the SGPLOT procedure.

Some options are common to all ODS statistical graphics subroutines. The following common options specify options in the XAXIS and YAXIS statements in the SGPLOT procedure:

GRID= specifies whether grid lines are displayed for the X and Y axes. This option corresponds to the GRID option in the XAXIS and YAXIS statements.

LABEL= specifies axis labels for the X or Y axis. If the argument is a scalar, the value of the argument is used for the X-axis label. If the argument has two elements, the first is used for the X-axis label and the second for the Y-axis label. If this option is not specified, the labels “X” and “Y” are used for labels.

XVALUES= specifies a vector of values for ticks for the X axis.

YVALUES= specifies a vector of values for ticks for the Y axis.

PROCOPT= specifies a character matrix or string literal. The value is used verbatim to specify options in the PROC SGPLOT statement.

OTHER= specifies a character matrix or string literal. You can use this option to specify one or more complete statements in the SGPLOT procedure. For example, you can specify multiple REFLINE statements and an INSET statement.

Table 18.1 summarizes options that apply to only certain graph types.

Table 18.1 Options for ODS Statistical Graphical Subroutines

Option	BAR	BOX	HISTOGRAM	SCATTER	SERIES
FREQ=	X				
CATEGORY=		X			
SCALE=			X		
DENSITY=			X		
REBIN=			X		
LINEPARM=				X	
ORDER=	X	X			
TYPE=	X	X			
GROUPOPT=	X	X			
GROUP=	X	X		X	X
DATALABEL=		X		X	
OPTION=		X		X	X

Specify mandatory arguments (the data) in parentheses after the name of the subroutine. Specify options outside the parentheses.

You can add a title and footnotes to a plot by using the global TITLE and FOOTNOTE statements.

Limitations of the ODS Statistical Graphics Subroutines

The ODS statistical graphics subroutines are intended to enable you to quickly and easily construct basic graphics. They are not intended to be a complete interface to the SGPLOT procedure. If you need to construct a complicated graph from within the SAS/IML language, use the technique that is shown in "How the Graphs Are Created" on page 388.

One of the limitations of the ODS statistical graphics subroutines is that only certain variables are written to a data set by the subroutines. The *x* and *y* arguments (the data) are written to a SAS data set. So, too, are the data that are specified by the FREQ=, CATEGORY=, GROUP=, and DATALABEL= options. If you want to use other variables (for example, for the YERRORLOWER= and YERRORUPPER= options in the SCATTER statement), you need to use the method that is shown in "How the Graphs Are Created" on page 388 to write the data set and to create the graph.

Although the ODS statistical graphics subroutines are not comprehensive, they do show how to write modules that create graphs by calling the SGPLOT procedure from the SAS/IML language. The source code for the subroutines are available in the Sashelp.iml catalog.

Examples of Creating Graphs

Bar Charts

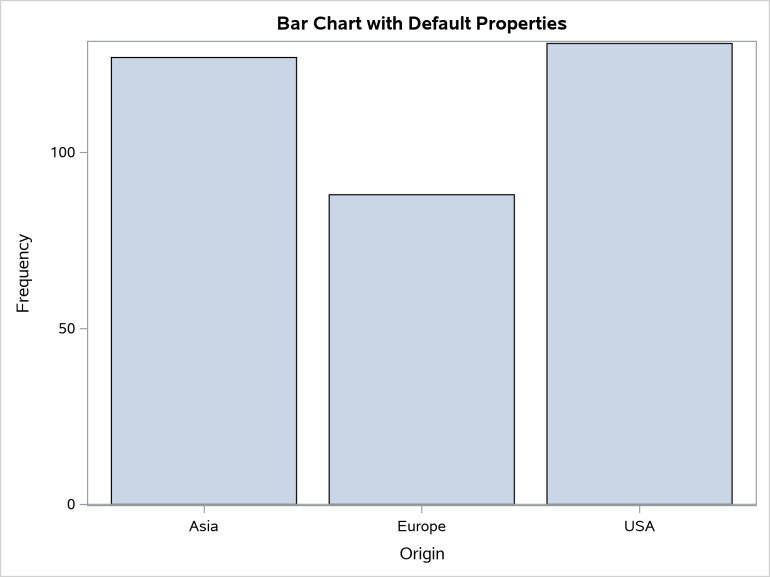
You can use the BAR subroutine to create a bar chart. The required argument is a vector that contains values of a discrete variable. These values are used to form the categories of a bar chart. The following statements read the Origin and Type variables from a subset of the Sashelp.Cars data set:

```
proc iml;
use Sashelp.Cars where(type ? {"SUV" "Truck" "Sedan"});
read all var {origin type};
close Sashelp.Cars;
```

The following statements create a simple bar chart of the Origin variable, which is shown in Figure 18.2:

```
title "Bar Chart with Default Properties";
call Bar(origin);
```

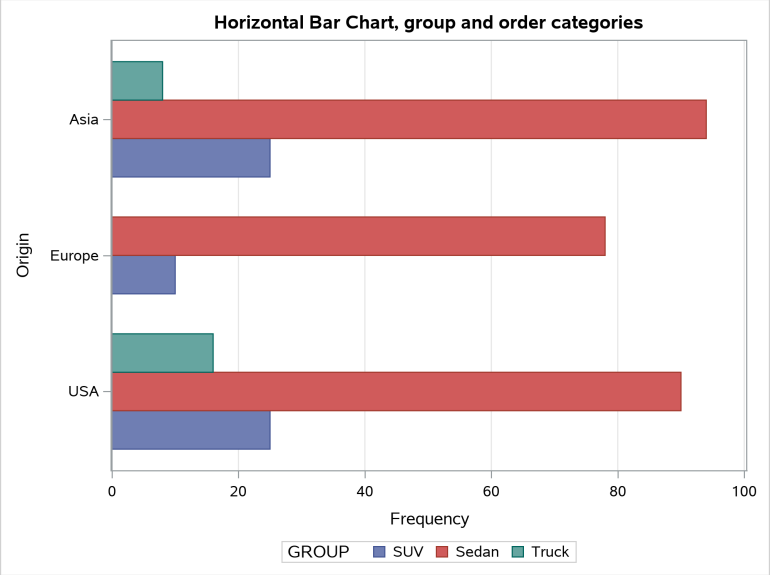
Figure 18.2 Bar Chart



For a more complicated example, the following statements create a bar chart by using the TYPE=, GROUP=, GROUPOPT=, GRID=, and LABEL= options. The result is shown in Figure 18.3.

```
title "Horizontal Bar Chart, group and order categories";
call Bar(origin) type="HBar" group=type groupopt="Cluster"
         grid="X" label="Origin";
```

Figure 18.3 Clustered Bar Chart



The following list explains the options that are used to create Figure 18.3:

- The TYPE= option specifies whether to create a vertical bar chart or a horizontal bar chart. Figure 18.3 shows a horizontal bar chart.
- The GROUP= option specifies a vector of values that determine groups in the plot. Figure 18.3 shows the bar chart grouped according to a subset of values for the Type variable.
- The GROUPOPT= option specifies a character vector of values that determine how groups are displayed. Figure 18.3 shows a clustered bar chart.
- The GRID= option specifies whether grid lines are displayed for the X and Y axes. Figure 18.3 shows grid lines for the X axis.
- The LABEL= option specifies axis labels for the X or Y axis. Figure 18.3 shows that the label "Origin" is used for the X axis.

Box Plots

You can use the BOX subroutine to create a box plot. The required argument is a vector that contains values of a continuous variable. Optionally, you can specify a categorical variable in order to obtain multiple box plots.

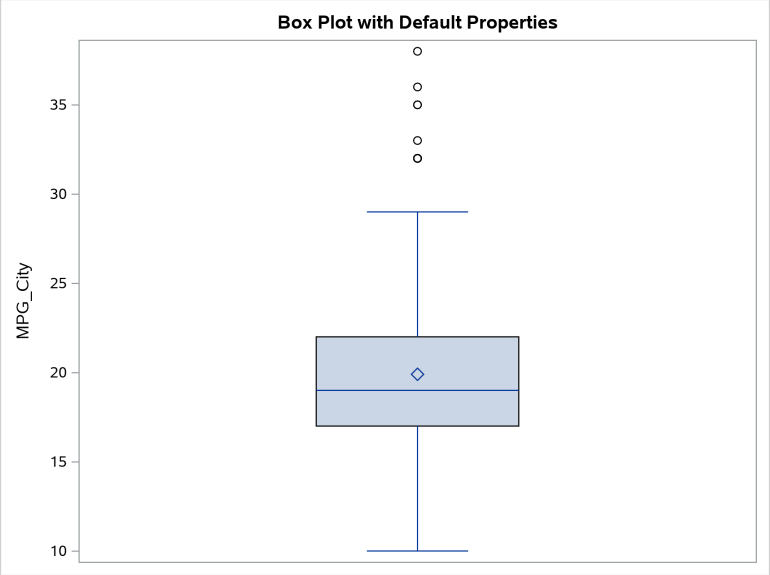
The following statements read several variables from a subset of the Sashelp.Cars data set:

```
proc iml;
use Sashelp.Cars where(type ? {"SUV" "Truck" "Sedan"});
read all var {MPG_City Origin Type Make Model};
close Sashelp.Cars;
```

The following statements create a simple box plot of the MPG_City variable. The box plot is shown in Figure 18.4.

```
title "Box Plot with Default Properties";
call Box(MPG_City);
```

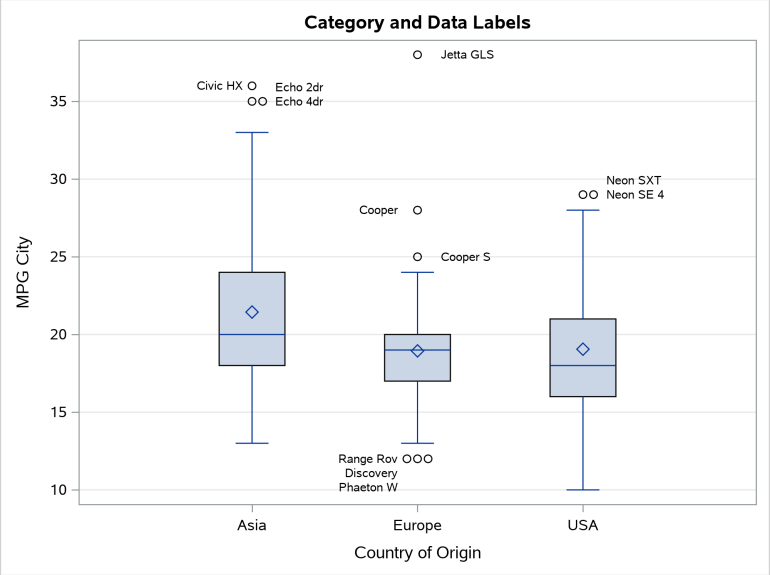
Figure 18.4 Box Plot with Default Options



For a more complicated example, the following statements create a box plot by using the CATEGORY=, GRID=, LABEL=, DATALABEL=, and OPTION= options. The result is shown in Figure 18.5.

```
title "Category and Data Labels";
call Box(MPG_City) Category=Origin grid="y"
                   label={"Country of Origin" "MPG City"}
                   datalabel=putc(Model,"$10.") option="spread";
```

Figure 18.5 Box Plot with Categorical Variable and Data Labels



The following list explains the options that are used to create Figure 18.5:

- The CATEGORY= option specifies a category variable. A box plot is created for each distinct value of the category variable. Figure 18.5 displays three box plots: one for vehicles that are manufactured in Asia, one for vehicles that are manufactured in Europe, and one for vehicles that are manufactured in the United States.
- The GRID=option specifies whether grid lines are displayed for the X and Y axes. Figure 18.5 displays grid lines for the Y axis.
- The LABEL= option specifies labels for the X and Y axes.
- The DATALABEL= option specifies a vector of values that are used to label outliers. In Figure 18.5, the labels are the first 10 characters of the Model variable in the Sashelp.Cars data set.
- The OPTION= option specifies options in the HBOX or VBOX statement. In this example, the SPREAD option is specified. This option has the effect, shown in Figure 18.5, of separating markers that would otherwise be overplotted.

Histograms

You can use the HISTOGRAM subroutine to create a histogram. The required argument is a vector that contains values of a continuous variable.

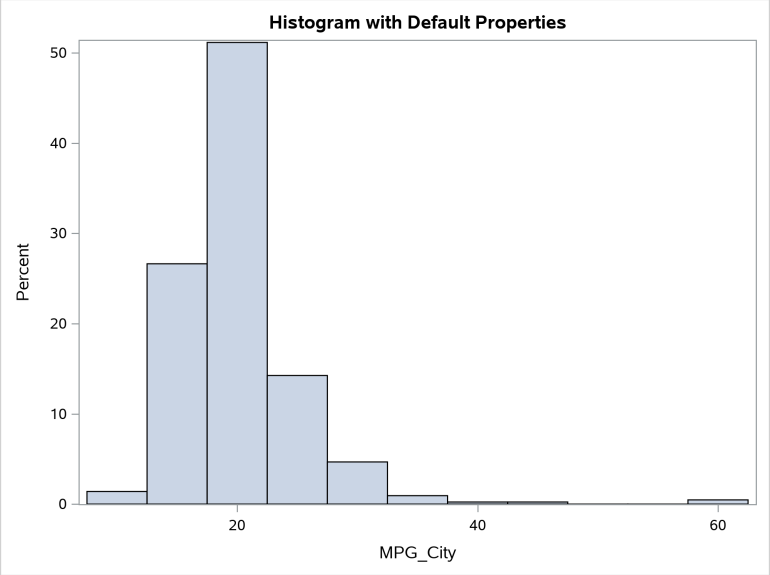
The following statements read the MPG_City variable from the Sashelp.Cars data set:

```
proc iml;
use Sashelp.Cars;
read all var {MPG_City};
close Sashelp.Cars;
```

The following statements create a simple histogram, which is shown in Figure 18.6:

```
title "Histogram with Default Properties";
call Histogram(MPG_City);
```

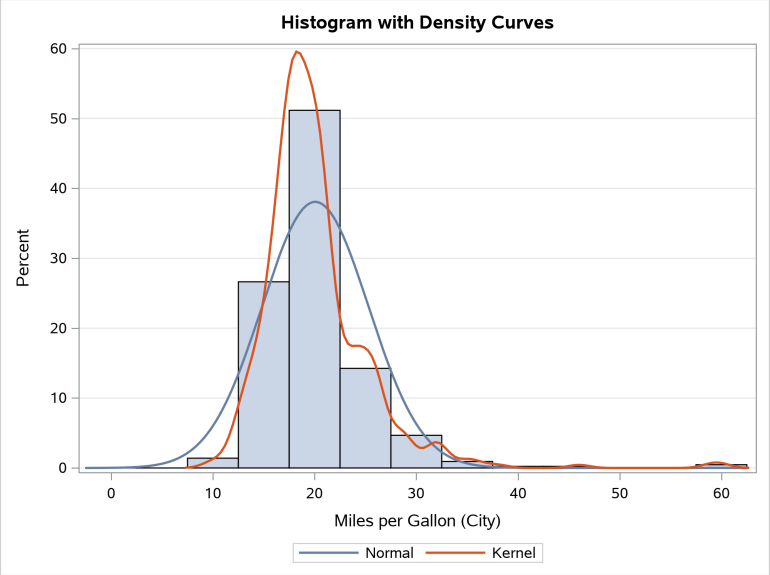
Figure 18.6 Histogram with Default Options



For a more complicated example, the following statements create a histogram by using the SCALE=, DENSITY=, REBIN=, GRID=, LABEL=, and XVALUES= options: The result is shown in Figure 18.7.

```
title "Histogram with Density Curves";
call Histogram(MPG_City)
          scale="Percent"
          density={"Normal" "Kernel"}
          rebin={0 5}
          grid="y"
          label="Miles per Gallon (City)"
          xvalues=do(0, 60, 10);
```

Figure 18.7 Histogram with Overlaid Densities



The following list explains the options that are used to create Figure 18.7:

- The SCALE= option specifies the scaling that is applied to the vertical axis of the histogram. In Figure 18.6, the vertical axis is scaled to represent the percentage of observations in each bar.
- The DENSITY= option specifies whether to overlay a density estimate on the histogram. In Figure 18.6, a normal density estimate and a kernel density estimate are overlaid.
- The REBIN= option specifies two numerical values that set the location of the first bins and the width of bins. In Figure 18.6, the bins are anchored at the value 0 and have a width of 5 units.
- The GRID= option specifies whether grid lines are displayed for the X and Y axes. Figure 18.6 shows grid lines for the Y axis.
- The LABEL= option specifies axis labels. In Figure 18.6, a label is specified for the X axis.
- The XVALUES= option specifies a vector of values for ticks for the X and Y axes. In Figure 18.6, tick marks are placed every 10 units in the interval [0, 60].

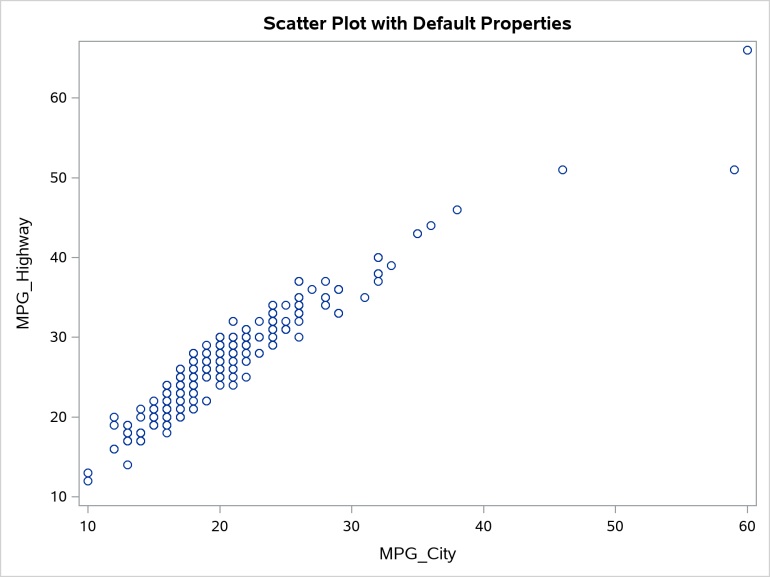
Scatter Plots

You can use the SCATTER subroutine to create a scatter plot. The subroutine requires two vector arguments: values for the X variable and values for the Y variable.

The following statements read the MPG_City and MPG_Highway variables from the Sashelp.Cars data set and create a simple scatter plot. The plot is shown in Figure 18.8.

```
proc iml;
use Sashelp.Cars;
read all var {MPG_City MPG_Highway Origin};
close Sashelp.Cars;

title "Scatter Plot with Default Properties";
run Scatter(MPG_City, MPG_Highway);
```

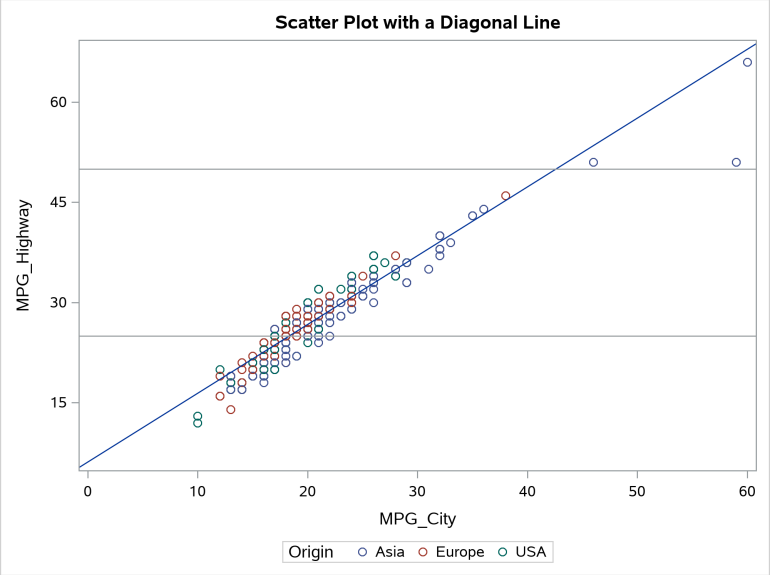
Figure 18.8 Scatter Plot



For a more complicated example, the following statements create a scatter plot by using the GROUP=, OTHER=, LABEL=, LINEPARM=, and YVALUES= options. The result is shown in Figure 18.9.

```
title "Scatter Plot with a Diagonal Line";
run Scatter(MPG_City, MPG_Highway)
    group=Origin                  /* assign color/marker shape */
    other="refline 25 50 /axis=y" /* add reference line */
    label={"MPG_City" "MPG_Highway"}
    lineparm={0 6.15 1.03}        /* diagonal line */
    yvalues=do(15,60,15);
```

Figure 18.9 Marker and Axis Attributes



The following list explains the options that are used to create Figure 18.9:

- The GROUP= option specifies a vector of values that determine groups in the plot. In Figure 18.9, the marker attributes correspond to values of the Origin variable.
- The OTHER= option specifies statements in the SGPLOT procedure. In Figure 18.9, the REFLINE statement draws two horizontal lines in the plot.
- The LABEL= option specifies axis labels for the X or Y axis. In Figure 18.9, both axes are labeled.
- The LINEPARM= option specifies a three-element vector whose elements specify a point and a slope for a line. In Figure 18.9, the line goes through the point $(0, 6.15)$ and has a slope of 1.03.
- The YVALUES= option specifies a vector of values for ticks for the Y axes. In Figure 18.9, the tick marks on the vertical axis are spaced 15 units apart in the interval $[15, 60]$.

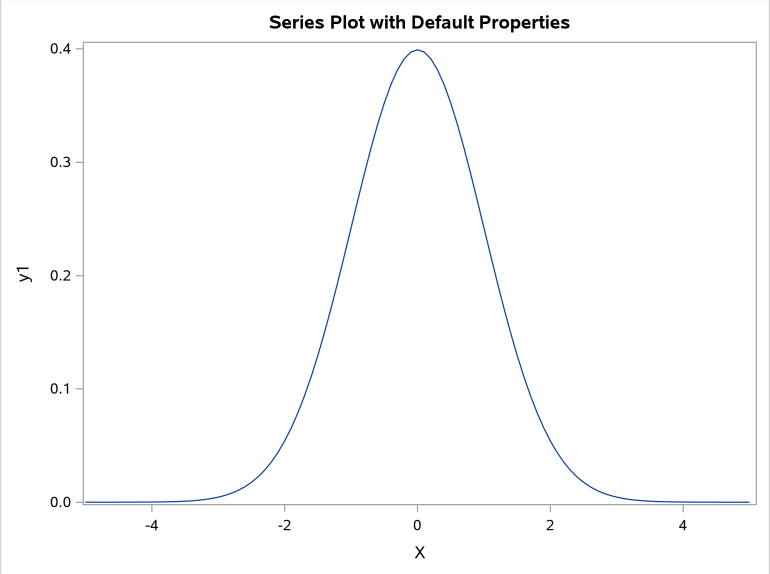
Series Plots

You can use the SERIES subroutine to create a series plot, which is also known as a line plot. The subroutine requires two vector arguments: values for the X variable and values for the Y variable.

The following statements provide a simple example of creating a series plot. The PDF function evaluates the normal density function at evenly spaced points in the interval $[-5, 5]$. The SERIES subroutine creates the graph that is shown in Figure 18.10.

```
proc iml;
x = do(-5, 5, 0.1);
y1 = pdf("Normal", x, 0, 1);

title "Series Plot with Default Properties";
run Series(x, y1);
```

Figure 18.10 Series Plot



For a more complicated example, the following statements create a series plot by using the GROUP=, OTHER=, GRID=, LABEL=, XVALUES=, and YVALUES= options. The result is shown in Figure 18.11.

```
title "Series Plot with Groups and Reference Lines";
y2 = pdf("Normal", x, 0, 1.5);
g = repeat({1,2}, 1, ncol(x));   /* 1,1,1,...,2,2,2 */
x = x  || x ;
y = y1 || y2;

call Series(x, y) group=g                  /* assign color/marker shape */
                  other="refline -2 2 / axis=x"   /* add reference line */
                  grid={X Y}
                  label={"X" "Normal Density"}
                  xvalues=-4:4
                  yvalues=do(0,0.4,0.05);
```

Figure 18.11 Two Curves in a Series Plot



The following list explains the options that are used to create Figure 18.11:

- The GROUP= option specifies a vector of values that determine groups in the plot. In Figure 18.11, the two curves have different values of the grouping variable, which is set by using the `g` matrix.
- The OTHER= option specifies statements in the SGPLOT procedure. In Figure 18.11, the REFLINE statement draws two horizontal lines on the plot.
- The GRID= option specifies whether grid lines are displayed for the X and Y axes. Figure 18.11 shows grid lines for the Y axis.
- The LABEL= option specifies axis labels for the X and Y axes. In Figure 18.11, both axes are labeled.
- The XVALUES= option specifies a vector of values for ticks for the X axis. In Figure 18.11, the tick marks on the horizontal axis are spaced one unit apart in the interval $[-4, 4]$.
- The YVALUES= option specifies a vector of values for ticks for the Y axis. In Figure 18.11, the tick marks on the vertical axis are spaced 0.05 units apart in the interval $[0, 0.4]$.

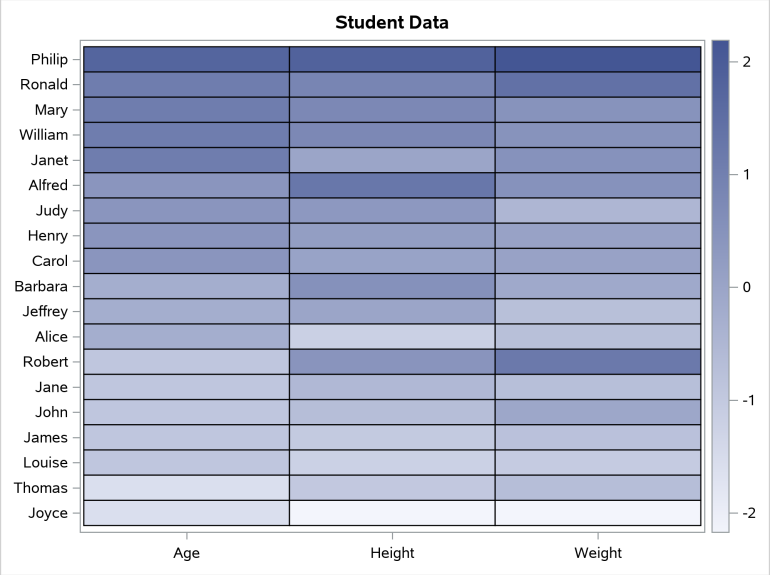
Matrix Heat Maps

You can use the HEATMAPCONT subroutine and the HEATMAPDISC subroutine to display a heat map that visualizes a matrix. The HEATMAPCONT subroutine displays a heat map of a numeric matrix whose values are assumed to vary continuously. The HEATMAPDISC subroutine displays a heat map of a numeric or character matrix whose values are assumed to have a small number of discrete values.

The following statements provide a simple example of creating a heat map that shows the relative ages, heights, and weights of 19 children. The SCALE="Col" option standardizes each column to have zero mean and unit standard deviation.

```
proc iml;
use Sashelp.Class;
read all var _NUM_ into Students[c=varNames r=Name];
close Sashelp.Class;

/* sort data in descending order according to Age and Height */
call sortndx(idx, Students, 1:2, 1:2);
Students = Students[idx,];
Name = Name[idx];

/* standardize each column */
call HeatmapCont(Students) scale="Col"
        xvalues=varNames yvalues=Name title="Student Data";
```

Figure 18.12 Heat Map of a Data Matrix

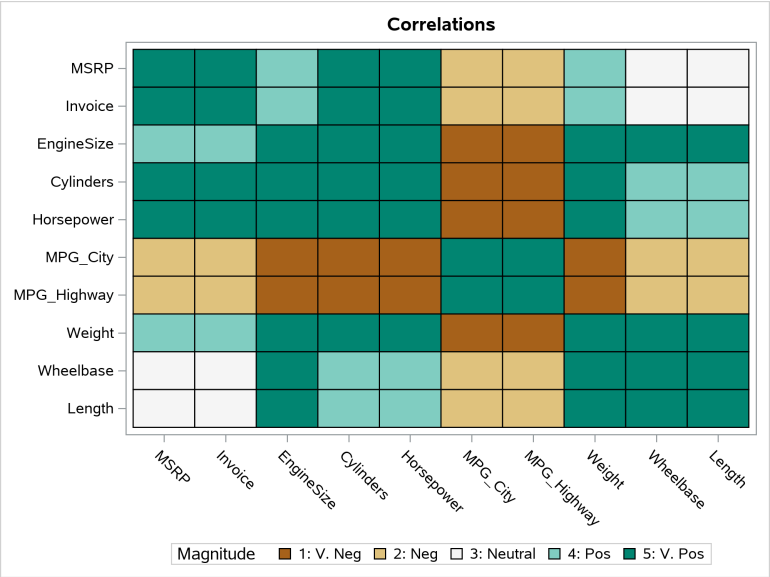


In Figure 18.12, you can see that Philip is the biggest student, Joyce is the smallest, Robert is heavy for his age, and Alfred is tall for his age.

You can also create a heat map of a matrix that contain discrete values. For example, the following statements compute the correlation matrix for variables in the Sashelp.Cars data set. The correlations are then binned into five discrete categories:

```
use Sashelp.Cars;
read all var _NUM_ into Y[c=varNames];
close Sashelp.Cars;
corr = corr(Y);

/* You can visualize the correlations as a continous heat map:
      call HeatmapCont(corr) xvalues=varNames yvalues=varNames;
   Alternatively, bin the values into five categories, as follows: */
Bins = {"1: V. Neg", "2: Neg", "3: Neutral", "4: Pos", "5: V. Pos"};
idx = bin(corr, {-1, -0.6, -0.2, 0.2, 0.6, 1});
disCorr = shape(Bins[idx], nrow(corr));
call HeatmapDisc(disCorr) title="Correlations"
                xvalues=varNames yvalues=varNames
                LegendTitle="Magnitude";
```

Figure 18.13 A Heat Map of a Discrete Matrix



The heat map shows strong negative correlations between fuel efficiency and three variables that indicate the size of a vehicle's engine. There is almost no correlation between the size of a vehicle and the price of the vehicle. There are large positive correlations between the size of a vehicle and the size of its engine.

Chapter 19
Storage Features

Contents

Overview **403**
Storage Catalogs **403**
Catalog Management **404**
Restoring Matrices and Modules 404
Removing Matrices and Modules 405
Specifying the Storage Catalog 405
Listing Storage Entries 406
Storing Matrices and Modules 406

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 25).

By 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 by 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 as follows:

```
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 following statement:

```
load module=_all_;
```

If you want to load all matrices and modules in storage, use the LOAD command by itself, as follows:

```
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 by using the REMOVE command. The REMOVE command has the same form as the LOAD command. Some examples of valid REMOVE statements are as follows:

```
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 following statement:

```
remove _all_;
```

If you want to remove everything from storage, use the following statement:

```
remove module=(_all_) _all_;
```

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 by using the following statement:

```
reset storage=mystor;
```

To change the libref as well, use the following statement:

```
reset storage=mylib.mystor;
```

Listing Storage Entries

You can list all modules and matrices in the current storage catalog by 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 by using the STORE command. The STORE command has the same general form as the LOAD command. Several examples of valid STORE statements are as follows:

```
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 following statement:

```
store _all_ module=_all_;
```

Alternatively, to store everything, you can also enter the STORE command by itself, as follows:

```
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 20
Using SAS/IML Software to Generate SAS/IML Statements

Contents

Overview **407**
Generating and Executing Statements **407**
Executing a String Immediately 408
Feeding an Interactive Program 409
Calling the Operating System 409
Calling the SAS Windowing Environment 410
Executing Any Command in an EXECUTE Call 410
Making Operands More Flexible 411
Interrupt Control 411
Specific Error Control 412
General Error Control 413
Macro Interface 415
IML Line Pushing Contrasted with Using the Macro Facility 416
Summary **416**

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. Here is the code:

```
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 following 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 preceding module DELALL:

```
   /* 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 as follows:

```
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 following statements:

```
dm 'log; color source red'; resume;
```

These statements 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 by 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 following 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. Here is the code:

```
/* 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 following 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 iteration 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 the following example, if a singular matrix is passed to the INV function, the IML procedure pauses and executes the pushed code to set the result for the inverse 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 as follows:

```
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 might 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 exceeds some maximum. Often, you might 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 as follows:

```
  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);                          */
    /* macro-processes the text in x                       */
    /* and returns 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 as follows:

```
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 by 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.

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 21
Wavelet Analysis

Contents

Overview **417**
Some Brief Mathematical Preliminaries 417
Diagnostic Plots for Wavelet Analysis 419
Getting Started **419**
Creating the Wavelet Decomposition 421
Wavelet Coefficient Plots 423
Multiresolution Approximation Plots 426
Multiresolution Decomposition Plots 428
Wavelet Scalograms 429
Reconstructing the Signal from the Wavelet Decomposition 431
Details **433**
Using Symbolic Names 433
Obtaining Help for the Wavelet Macros and Modules 435
References **436**

Overview

Wavelets are a versatile tool for understanding and analyzing data; they have 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* ϕ (also called the *father wavelet*), and the *mother wavelet* ψ. These translations and dilations are defined as follows:

$$\begin{aligned} \phi_{j,k}(x) &= 2^{j/2}\phi(2^j x - k) \\ \psi_{j,k}(x) &= 2^{j/2}\psi(2^j x - k) \end{aligned}$$

The index j defines the dilation (*level*), and the index k defines the translation. 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

$$\begin{aligned} c_k^j &= \int f(x)\phi_{j,k}(x)\,dx \\ d_k^j &= \int f(x)\psi_{j,k}(x)\,dx \end{aligned}$$

The key to obtaining fast numerical algorithms for computing the scaling and detail coefficients for a specified 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 formulas are

$$\begin{aligned} c_k^{j-1} &= \sum_i h_{i-2k} c_i^j \\ d_k^{j-1} &= \sum_i g_{i-2k} c_i^j \end{aligned}$$

The coefficients h_k and g_k that appear in these formulas are called *filter coefficients.* The h_k are determined by the father wavelet and form a low-pass filter. The coefficients g_k are defined by $g_k = (-1)^k h_{1-k}$ and form a high-pass filter. The preceding summations represent sums 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 nonzero. So in this case the sums in the recurrence relationships for the detail and scaling coefficients are finite.

Conversely, if you know the scaling and detail coefficients at level $j-1$, then you can obtain the scaling coefficients at level j by 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 the following data values at $N = 2^J$ equally spaced points x_k:

$$y_k = f(x_k), \qquad k = 0, 1, 2, \ldots, N-1$$

It turns out that the values $2^{-J/2} y_k$ are good approximations of the scaling coefficients c_k^J. Then, by using the recurrence formula, you can find c_k^{J-1} and d_k^{J-1}, $k = 0, 1, 2, \ldots, N/2 - 1$. 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 when the recurrence relationships are applied to finite data, a few values of the c_k^J for $k < 0$ or $k \geq N$ might be needed. One way to cope with this difficulty is to extend the sequence c_k^J to the left and right by using some specified boundary treatment.

If you replace the scaling coefficients at any level j by the scaling and detail coefficients at level $j-1$, you obtain 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

$$f(x) \approx \sum_k c_k^{j_0} \phi_{j_0,k}(x) + \sum_{j=j_0}^{J-1} \sum_k d_k^j \psi_{j,k}(x)$$

Diagnostic Plots for Wavelet Analysis

SAS/IML provides modules that produce wavelet diagnostic plots. The modules are defined by calling the autocall WAVINIT macro. The WAVINIT macro, which must be called from within PROC IML, also defines symbolic macro variables that you can use to improve the readability of your code.

There are two sets of modules, which have the same names and signatures. The newer modules, which use ODS statistical graphics to create the diagnostic plots, are defined if you enable ODS graphics prior to calling the WAVINIT macro. For example, to use the newer graphics, run the following statements:

```
/* define modules that use ODS graphics */
ods graphics on; /* ODS graphics enabled */
proc iml;
%wavinit;        /* define modules for ODS graphics */
```

The older modules use the traditional graphics subroutines, catalogs, and the ODS LISTING destination. The older modules are defined if you disable ODS graphics before calling the WAVINIT macro. To use the traditional graphics, you must call the WAVGINIT autocall macro prior to calling PROC IML, as follows:

```
/* define modules that use traditionaly graphics */
ods graphics off; /* ODS graphics disabled */
%wavginit;        /* define graphical options */
proc iml;
%wavinit;         /* define modules for traditional graphics */
```

The WAVGINIT macro defines several macro variables that are used to adjust the size, aspect ratio, and font size for the plots the wavelet plot modules produce. This macro can also take several optional arguments that control the positioning and size of the wavelet diagnostic plots. For information about getting help for this macro call, see the section "Obtaining Help for the Wavelet Macros and Modules" on page 435.

The newer ODS graphics are used in this chapter. The older graphics are still available, but are not recommended.

Getting Started

Fourier transform infrared (FT-IR) spectroscopy is an important tool in analytic chemistry. The following example demonstrates wavelet analysis applied to an FT-IR spectrum of quartz (Sullivan 2000). The following

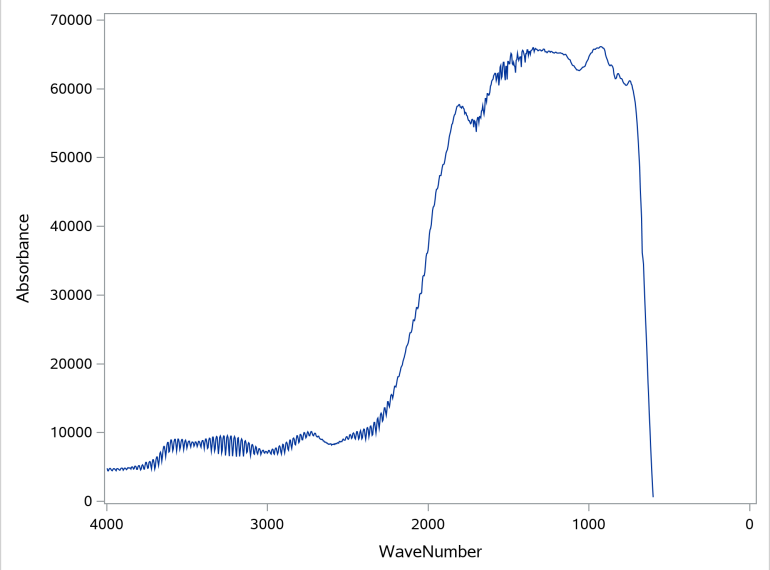
DATA step creates a data set that contains the spectrum, expressed as an absorbance value for each of 850 wave numbers:

```
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

   ... more lines ...

 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, which is displayed in Figure 21.1:

```
proc sgplot data=quartzInfraredSpectrum;
   series x=WaveNumber y=Absorbance;
   xaxis reverse min=0;
   yaxis values=(0 to 70000 by 10000);
run;
```

Figure 21.1 FT-IR Spectrum of Quartz



These data contain information at two distinct scales, namely a high-frequency oscillation superimposed on a low-frequency curve. Notice that the oscillation is not uniform but occurs in several distinct bands. Wavelet analysis is an appropriate tool for providing insight into this type of data, because 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/IML statements begin a wavelet analysis by enabling ODS graphics and by running the WAVINIT autocall macro:

```
ods graphics on; /* ODS graphics enabled */
proc iml;
%wavinit;        /* define modules for ODS graphics */
```

You can use the SAS/IML wavelet functions without using the WAVINIT macro. However, the macro defines symbolic macro variables that you can use to improve the readability of your code. The macro also defines modules that you can use to produce several standard wavelet diagnostic plots.

The following statements read the Absorbance variable into a vector:

```
use quartzInfraredSpectrum;
read all var "absorbance";
close;
```

You can now begin the wavelet analysis. The first step is to set up the options vector that specifies which wavelet and which 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] = 2;   */
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 shown 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 that is 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 SAS/IML wavelet subroutines and plot modules. For example, you use the WAVPRINT subroutine to print the information that is encapsulated in a wavelet decomposition. The following code produces the output in Figure 21.2:

```
call wavprint(decomp, &summary);
call wavprint(decomp, &detailCoeffs, 1, 4);
```

Figure 21.2 Output of WAVPRINT Calls

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	-1.70985E-9	1.31649E-10	-8.6402E-12	5.10454E-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

You can use the WAVGET call to obtain the wavelet detail coefficients in a vector so that you can perform additional computations. For example, the following statements estimate the noise level of the data:

```
call wavget(tLevel, decomp, &topLevel);
call wavget(noiseCoeffs, decomp, &detailCoeffs, tLevel-1);

noiseScale = mad(noiseCoeffs, "nmad");
print noiseScale[label="Noise Scale"];
```

The result is shown in Figure 21.3.

Figure 21.3 Scale of Noise in the Absorbance Data

Noise Scale
169.18717

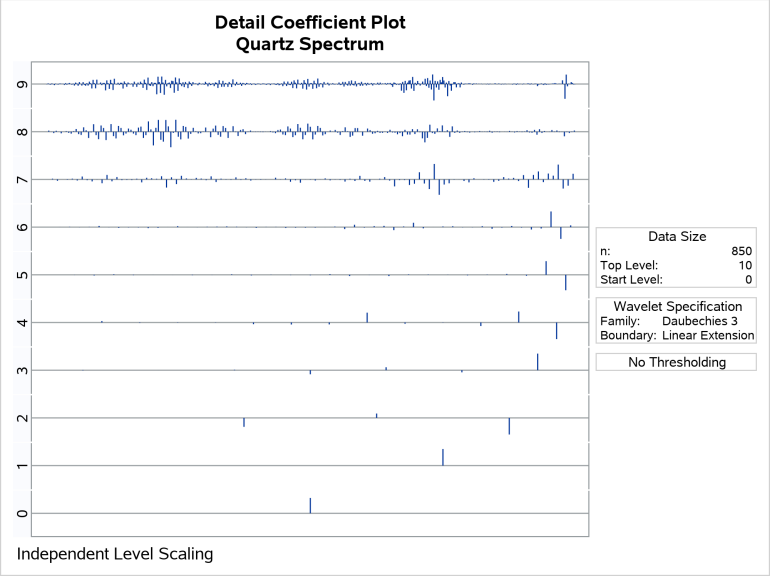
The first WAVGET call obtains the top level number in the wavelet decomposition decomp. The highest level of detail coefficients is defined at one level below the top level in the decomposition. The second WAVGET call returns these coefficients in the vector noiseCoeffs. Finally, the MAD function computes 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 sequence of detail coefficients by level. By using a module that is defined by the WAVINIT macro call, you can produce the plot shown in Figure 21.4 as follows:

```
call coefficientPlot(decomp) header="Quartz Spectrum";
```

Figure 21.4 Detail Coefficients Scaled by Level



The first argument to the COEFFICIENTPLOT module is required; it specifies the wavelet decomposition. Other arguments are optional and need not be specified. This example uses a named argument to specify the header that appears at the top of the coefficient plot.

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 the autocall WAVINIT macro. For example, the following call to the WAVHELP macro writes a short description of the COEFFICIENTPLOT module to the SAS log, as shown in Figure 21.5.

```
%wavhelp(coefficientPlot);
```

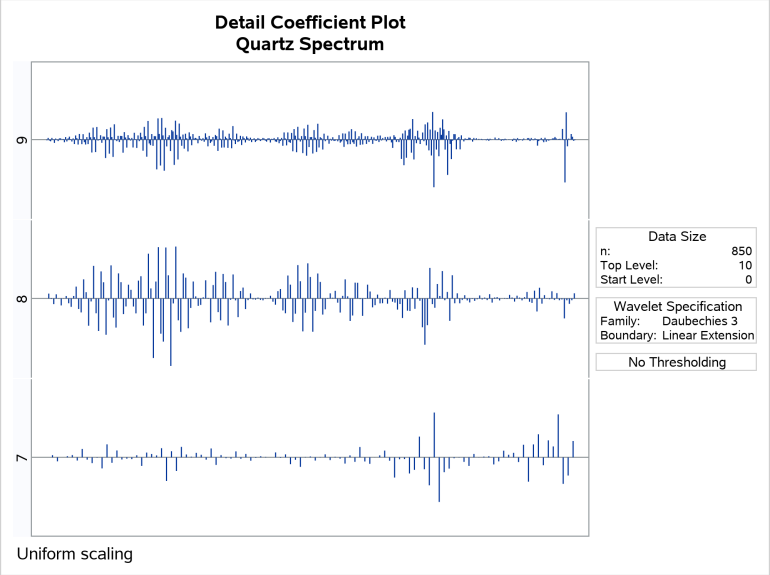
Figure 21.5 Log Output Produced by %wavhelp(coefficientPlot) Call

```
      coefficientPlot Module

Function: Plots wavelet detail coefficients

Usage: call coefficientPlot(decomposition,
                            threshopt,
                            startLevel,
                            endLevel,
                            howScaled,
                            header);

Arguments:
   decomposition - (required) valid wavelet decomposition 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
```

In Figure 21.4, 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. The plot is shown in Figure 21.6.

```
call coefficientPlot(decomp, , 7, , 'uniform', "Quartz Spectrum");
```

Figure 21.6 Uniformly Scaled Detail Coefficients

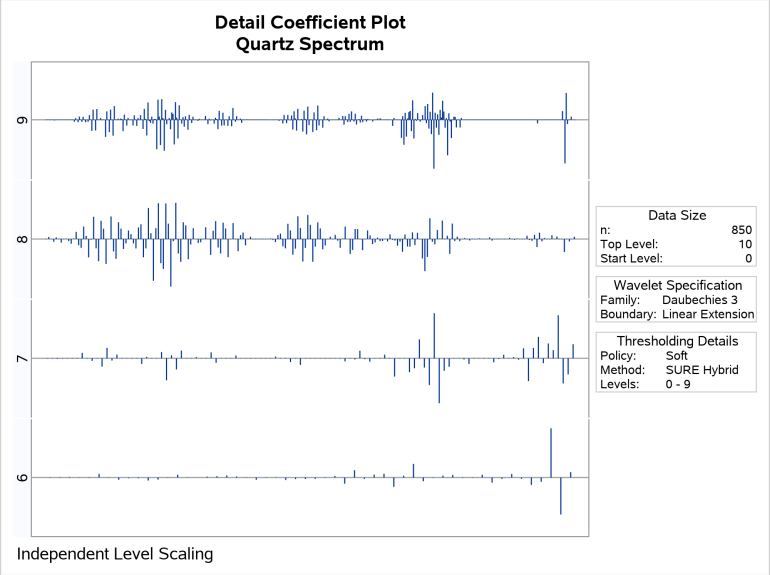


As mentioned earlier, noise in the data is captured in the detail coefficients, particularly in the small coefficients at higher levels in the decomposition. By zeroing or shrinking these coefficients, you can get smoother reconstructions of the input data. This is done by specifying a threshold value for each level of detail coefficients and then zeroing or shrinking all the detail coefficients below this threshold value. The SAS/IML wavelet functions and modules support several policies both for how this thresholding is performed and for selecting the thresholding value at each level. For more information, see the section "WAVIFT Call" on page 1097.

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). The plot is shown in Figure 21.7.

```
call coefficientPlot(decomp, &SureShrink, 6, , , "Quartz Spectrum");
```

Figure 21.7 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 input signal that use the thresholded detail coefficients still capture the essential features of the data, but are smoother because much of the very fine-scale detail has been eliminated.

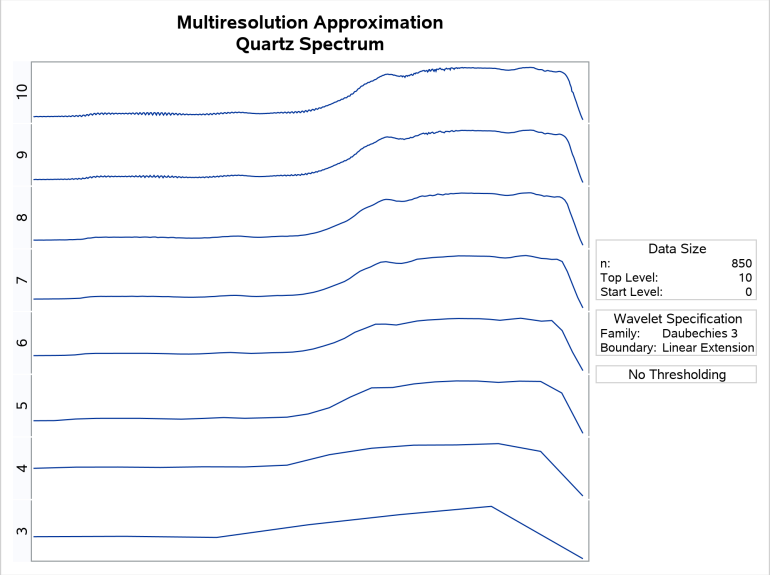
Multiresolution Approximation Plots

One way of presenting reconstructions is in a multiresolution approximation plot. In this plot, reconstructions of the input data are shown by level. The reconstruction at any level uses only the detail and scaling coefficients that are defined below that level.

The following statement produces such a plot, starting at level 3. The results are shown in Figure 21.8.

```
call mraApprox(decomp, , 3, , "Quartz Spectrum");
```

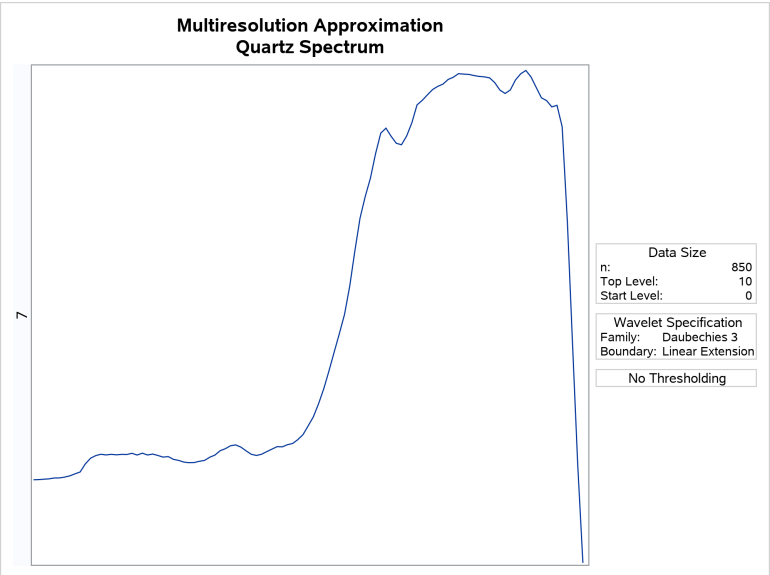
Figure 21.8 Multiresolution Approximation



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 higher than level 7. Thus, the reconstructions at level 7 and lower are largely free of oscillation because 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. The results are shown in Figure 21.9.

```
call mraApprox(decomp, , 7, 7, "Quartz Spectrum");
```

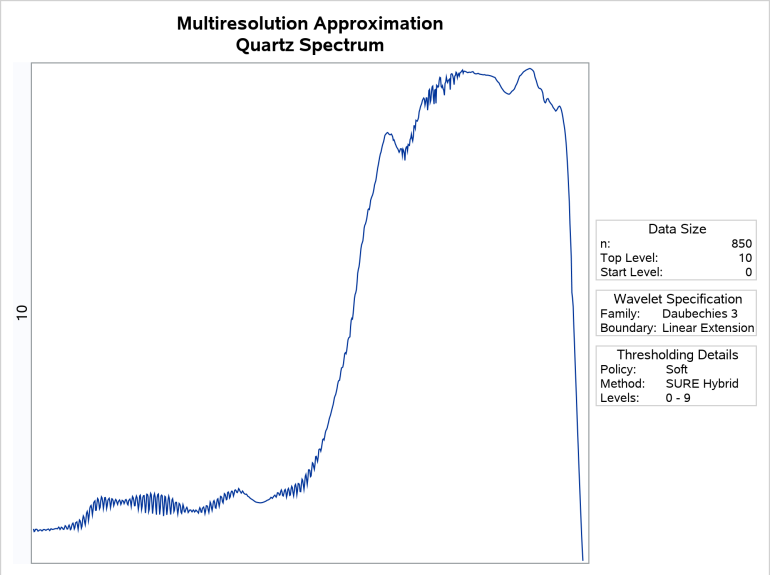
Figure 21.9 Level 7 of the Multiresolution Approximation



You can also plot the multiresolution approximations that are obtained by using thresholded detail coefficients. For example, the following statement plots the top-level reconstruction that is obtained by using the "SureShrink" threshold. The results are shown in Figure 21.10.

```
call mraApprox(decomp, &SureShrink, 10, 10, "Quartz Spectrum");
```

Figure 21.10 Top Level of Multiresolution Approximation with SureShrink Thresholding Applied



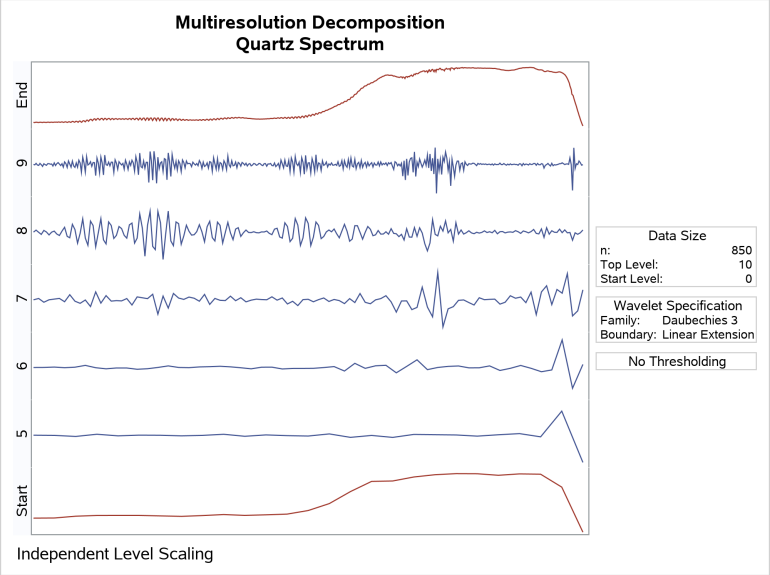
Notice that the high-frequency oscillation is still present in the reconstruction even with "SureShrink" thresholding applied.

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 (labeled "Start") and the reconstruction at the highest level (labeled "End") are also shown. Adding suitably scaled versions of the detail levels to the starting-level reconstruction results in the final reconstruction. The following statement produces such a plot, which is shown in Figure 21.11:

```
call mraDecomp(decomp, ,5, , , "Quartz Spectrum");
```

Figure 21.11 Multiresolution Decomposition



Wavelet Scalograms

Wavelet scalograms visualize the time frequency localization property of the discrete wavelet transform. In a scalogram, 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. Coefficients at low levels are plotted as wide, 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 are not 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.

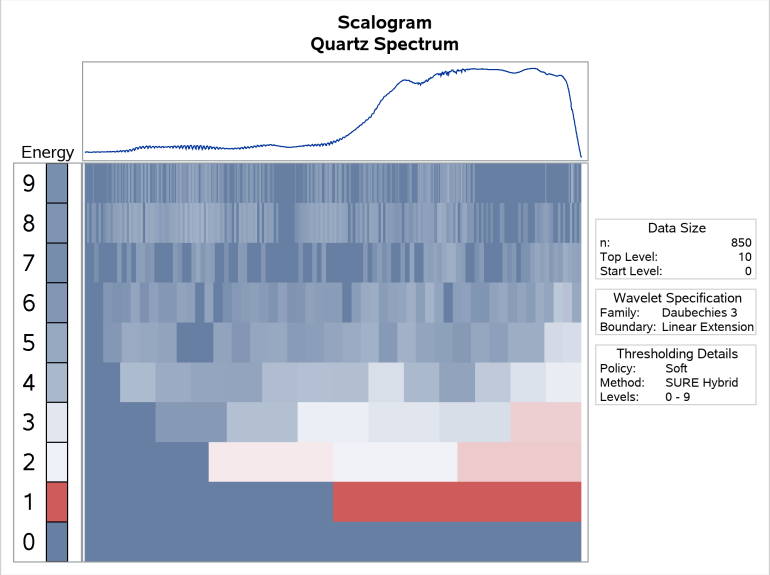
The following statement produces a scalogram plot of all levels with "SureShrink" thresholding applied:

```
call scalogram(decomp, &SureShrink, , , 0.25, 'log',"Quartz Spectrum");
```

The sixth argument requests that the rectangle heights 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 because 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 21.12.

Figure 21.12 Scalogram Showing All Levels

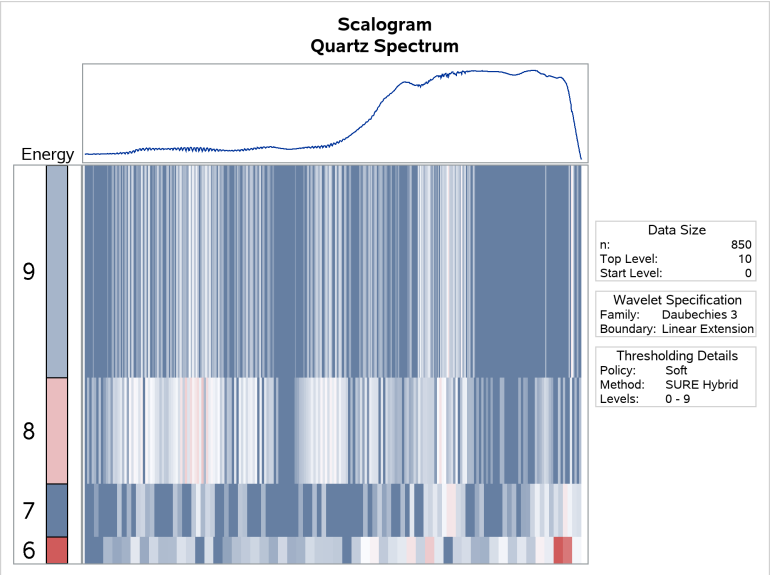


The bar on the left side of the scalogram plot indicates the overall energy of each level. This energy is defined as the sum of the squares of the detail coefficients for each level. These energies are amplified by the same algorithm that amplifies the magnitudes of the detail coefficients. The energy bar in Figure 21.12 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 higher by using the default coefficient amplification and a uniform scale for the rectangle heights. The result is shown in Figure 21.13.

```
call scalogram(decomp, &SureShrink, 6, , , , "Quartz Spectrum");
```

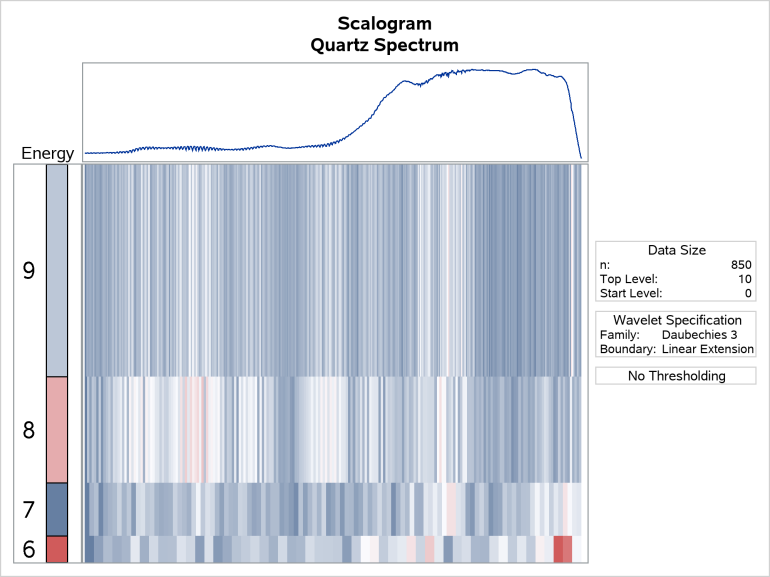
Figure 21.13 Scalogram of Levels 6 and Higher Using “SureShrink” Thresholding



The scalogram in Figure 21.13 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 21.13 with Figure 21.14, which shows the corresponding scalogram except that no thresholding is done. The following statement produces Figure 21.14:

```
call scalogram(decomp, , 6, , , , "Quartz Spectrum");
```

Figure 21.14 Scalogram of Levels 6 and Higher Using No Thresholding



Reconstructing the Signal from the Wavelet Decomposition

You can use the WAVIFT subroutine to invert a wavelet transformation that is computed by the WAVFT subroutine. If no thresholding is specified, then this inversion is exact up to numerical rounding error, as shown by the following statements:

```
call wavift(reconstructedAbsorbance, decomp);
errorSS=ssq(absorbance-reconstructedAbsorbance);
print errorSS[label="Reconstruction Error Sum of Squares"];
```

The output is shown in Figure 21.15.

Figure 21.15 Exact Reconstruction Property of WAVIFT

Reconstruction Error Sum of Squares
1.288E-16

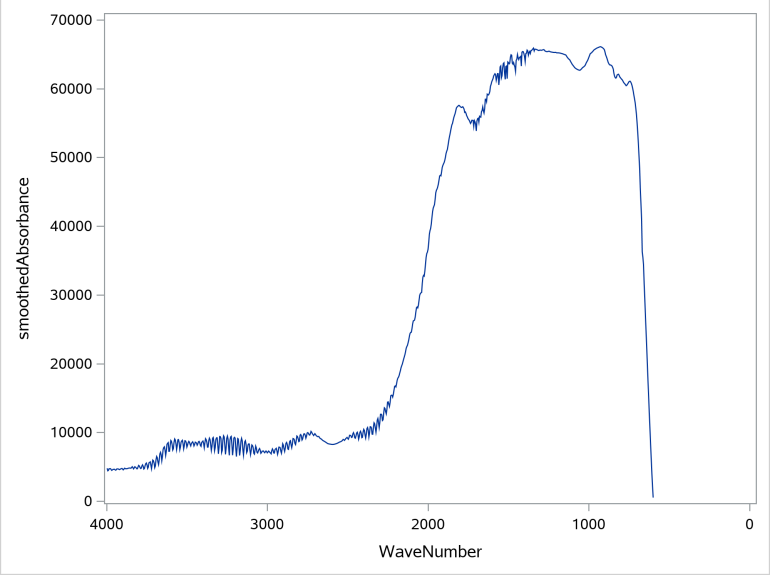
Usually you use the WAVIFT subroutine with thresholding to produce a smoothed reconstruction of the input data. You can use the following statements to create a smoothed reconstruction of absorbance and add this variable to the QuartzInfraredSpectrum data set:

```
call wavift(smoothedAbsorbance, decomp, &Sureshrink);
create temp var "smoothedAbsorbance";
   append;
close temp;
quit;

data Spectrum;
  merge quartzInfraredSpectrum temp;
run;
```

The following statements produce the line plot of the smoothed absorbance data shown in Figure 21.16:

```
proc sgplot data=Spectrum;
   series x=WaveNumber y=smoothedAbsorbance;
   xaxis reverse min=0;
   yaxis values=(0 to 70000 by 10000);
run;
```

Figure 21.16 Smoothed FT-IR Spectrum of Quartz



You can see by comparing Figure 21.1 with Figure 21.16 that the wavelet smoothing of the absorbance data has preserved all the essential features of these data.

Details

Using Symbolic Names

Several of the wavelet subroutines take 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 to use in computing the wavelet transform. You could write the following statements to define the options vector:

```
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. You can improve readability by using symbolic names. SAS macro variables provide a convenient mechanism for creating such symbolic names. For example, the previous statements could be replaced by the following statements, which use symbolic macro variables (names with a preceding ampersand) to resolve the relevant quantities:

```
optn            = &waveSpec;
optn[&boundary] = &zeroExtension;
optn[&family]   = &daubechies;
optn[&member]   = 3;
```

Symbolic names also improve code readability when they are substituted for integer arguments that control which actions a multipurpose subroutine performs. Consider the following statements:

```
call wavget(n, decomposition, 1);
call wavget(fWavelet, decomposition, 8);
```

These statements can be replaced by the following:

```
call wavget(n, decomposition, &numPoints);
call wavget(fWavelet, decomposition, &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 21.1 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 21.2 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 21.3 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 21.4 Macro Variables for the Second Argument of WAVPRINT

Name	Value
&summary	1
&detailCoeffs	2
&scalingCoeffs	3
&thresholdedDetailCoeffs	4

Table 21.5 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 21.6 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 as follows:

%WAVHELP < *(name)* > **;**

In the macro call, *name* is either 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. You can obtain help for the built-in SAS/IML wavelet subroutines by using the SAS Online Help.

References

Daubechies, I. (1992). *Ten Lectures on Wavelets.* Vol. 61 of CBMS-NSF Regional Conference Series in Applied Mathematics. Philadelphia: SIAM.

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.

Chapter 22
Genetic Algorithms

Contents

Overview **437**
Formulating a Genetic Algorithm Optimization **439**
Choosing the Problem Encoding 439
Defining the Objective Function 440
Controlling the Selection Process 441
Using Crossover and Mutation Operators 442
Executing a Genetic Algorithm **447**
Using the GA Modules to Implement an Optimization 447
Incorporating Local Optimization 449
Handling Constraints 449
Examples **449**
Example 22.1: The Traveling Salesman Problem 450
Example 22.2: Genetic Algorithm with Local Optimization 453
Example 22.3: Real-Valued Objective Optimization with Constraints 456
Example 22.4: Integer Programming Knapsack Problem 459
Example 22.5: Optimization with Linear Constraints Using Repair Strategy 461
References **464**

Overview

Genetic algorithms (hereafter referred to as GAs) are a family of search algorithms that seek optimal solutions to problems by using the principles of natural selection and evolution. GAs can be applied to almost any optimization problem and are especially useful for problems where calculus-based techniques do not work, such as when the objective function has many local extrema, is not differentiable, or is not continuous. GAs are also useful when the solution vector is constrained to be a sequence of integers.

In many cases GAs require more computation than specialized techniques that take advantage of specific problem structures. However, when no such techniques are available, GAs provide a robust general method of optimization.

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 is evaluated for each member of the initial population.

Regeneration: A new population is generated from the current population. First, individual members are chosen stochastically to parent the next generation such that those who are "fit" (have good values for the objective function) are more likely to be chosen. This process is called *selection*. Those chosen "parents" are either copied directly to the next generation or can be passed to a crossover operator. 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, is passed to a mutation operator that 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 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 members that were generated by selection, crossover, and mutation. The objective function values are evaluated for the new generation. A common variation on this approach is to pass one or more of the best solutions from the current population on to the next population unchanged. This can lead 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 criterion, such as the number of iterations. If the stopping criterion 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, because the convergence of GAs can be sensitive to the choice of genetic operators, mutation probability, and selection criteria, some experimentation and fine-tuning of these parameters is often required.

In the traditional formulation of GAs, the set of parameters is mapped into finite-length bit strings, and the genetic operators that act on these strings are based on biological processes. Although there is a theoretical basis for the effectiveness of GAs formulated in this way (Goldberg 1989), in practice most problems do not fit naturally into this paradigm. Research has shown that optimizations can be set up by using the natural solution domain (for example, a real vector or integer sequence) and then applying crossover and mutation operators that are analogous to the traditional genetic operators but that are more appropriate to the natural formulation of the problem (Michalewicz 1996). This latter approach is sometimes called *evolutionary computing*. SAS/IML implements the evolutionary computing approach, which makes it much easier to formulate practical problems that have realistic constraints. Throughout this documentation, the term "genetic algorithm" is to be interpreted as evolutionary computing.

SAS/IML provides a flexible framework for implementing GAs. You can write your own modules for the genetic operators and objective function. You can use standard genetic operators or define your own. This framework enables you to use variations of the usual GA, such as modifying the optimization parameters during the optimization or incorporating problem-specific local optimizations to enhance convergence.

A SAS/IML program that implements GA optimization is structured differently from a program that uses the nonlinear programming (NLP) routines to perform nonlinear optimization. When you use the NLP routines, you define the objective function and constraints and make a single call that runs an optimization algorithm to completion.

In contrast, 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 create an initial population. After that, you advance the optimization process by calling GAREGEN (the regeneration step) within a loop. Within the loop you can use GAGETMEM and GAGETVAL calls to retrieve and examine population members and objective function values. This strategy enables you to monitor the convergence of the GA, adjust optimization parameters within the loop, and exit the loop if the GA is not improving the objective function. The next section explains the optimization parameters in more detail.

Formulating a Genetic Algorithm Optimization

To formulate a GA, you must provide the following parameters:

Encoding:	The general structure and form of the solution.
Objective function:	The function to be optimized. You can specify whether the function is to be minimized or maximized.
Selection:	How members of the current solution population are chosen to be parents for the next generation.
Crossover operator:	How the attributes of parent solutions are combined to produce new offspring.
Mutation operator:	How random variation will be introduced into the new offspring to maintain genetic diversity.

The following sections discuss each of these items.

Choosing the Problem Encoding

Problem encoding refers to the structure or type of solution space that is to be searched, such as real-valued fixed-length vectors or integer sequences. The GA routines offer the following built-in encoding options:

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. When you use this encoding, there are no assumptions about the form of the solution, so you must specify user modules for crossover and mutation operators and a user module for creating the initial solution population.

Real-valued fixed-length row vector: If you use this encoding, you must also specify the number of components in the solution vector. When you use this encoding, you can use built-in crossover and mutation operators, or you can supply custom modules. You can specify upper and lower bounds for each component and SAS/IML will generate a random initial population between the bounds. If you do not explicitly set crossover and mutation operators, default operators are used. This type of encoding is often used for general nonlinear optimization problems.

Integer-valued fixed-length row vector: This option is similar to the real-valued fixed-length encoding, except that the built-in genetic operators and initialization process will preserve and generate integer solutions. This type of encoding is useful for an assignment problem in which the positions within the vector represent

different tasks and the integer values represent different machines or other resources that can be applied to each task.

Integer fixed-length sequence: In this encoding, each solution is composed of a sequence of integers ranging from 1 to the length of the sequence. Different solutions correspond to different permutations of the elements. For example, the following $s1$ 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 (such as the traveling salesman problem, where each element represents a city in a circular route) or scheduling problems.

Defining the Objective Function

Before executing a GA, you must specify the objective function. Two options are currently available: a user-defined function module and a built-in traveling salesman problem (TSP) objective function.

User function module: The module must take exactly one parameter, which is a potential solution, and return a scalar value, which is the objective function evaluated at the parameter. The module can have a global clause, which can be used to pass in other information that is necessary to evaluate the objective function. 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 the GAREEVAL call) to ensure that the values are consistent with the changed global parameter.

The parameter that is passed to the routine is copied to the solution population when the module exits, so take care not to unintentionally modify the parameter. However, it is permissible (and might be very effective) to add logic to the module to improve the solution through some heuristic technique or local optimization, and deliberately copy that improved solution to the solution population by updating the parameter before returning. Using this hybrid approach might significantly improve the convergence of the GA, especially in later stages when solutions might be near an optimum.

TSP objective function: An objective function for the traveling salesman problem can be specified by using integer sequence encoding. For the TSP, a solution sequence represents a circular route. For example, a solution s that has the following value represents a route going from location 2 to location 4, then to 3, to 1, to 5, and finally back to 2:

```
s = {2 4 3 1 5};
```

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 that is determined by s, and is equivalent to the following SAS/IML statements:

```
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 built-in "order operator" for crossover and "invert operator" for mutation are especially appropriate for the TSP and other routing problems.

Controlling the Selection Process

Two competing factors 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 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. However, with too little selective pressure the GA might 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.

The GA routines offer two variants of *tournament selection* (Miller and Goldberg 1995), a standard technique for the selection process. In general, the tournament selection process randomly chooses a group of members from the current population, compares their objective values, and chooses the one that has the best objective value to be a parent for the next generation. Tournament selection is one of the fastest selection methods and offers good control over the selection pressure.

In the first variant of tournament selection, you can control the selective pressure by specifying the tournament size, which is the number of members that are chosen to compete for parenthood in each tournament. This number should be two or greater, where smaller numbers provide less selection pressure. Tournament sizes from two to 10 have been successfully applied to various GA optimizations; sizes over four or five are considered to represent strong selective pressure.

The second variant of tournament selection provides weaker selective pressure than the first variant. The tournament size is set to two, and the member that has the best objective value is chosen according to a probability that you specify. This best-player-wins probability can range from 0.5 to 1.0, where 1.0 implies that the best member is always chosen (equivalent to a conventional tournament of size two) and 0.5 implies an equal chance of either member being chosen, which is equivalent to pure random selection. You could set the best-player-wins probability close to 0.5 in the initial stages of the optimization and gradually increase it to strengthen the selective pressure as the optimization progresses, which results in an algorithm that is similar to the simulated annealing optimization technique.

Another important selection option 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 1 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

You can define modules for crossover and mutation operators, or you can choose from the built-in operators. The built-in operators are tied to the problem encoding options; the GA routines verify that a specified operator is appropriate for the problem encoding. You can turn off crossover, in which case the current population passes to the next generation subject only to mutation. Mutation can be turned off by setting the mutation probability to 0.

The valid crossover and mutation operators for each problem encoding are summarized in Table 22.1.

Table 22.1 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

The built-in genetic operators are described in the following sections.

Crossover Operators

The crossover operators are as follows:

Simple: This operator is defined for integer and real encodings. 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:k] || p2[k+1:n];
c2 = p2[1:k] || p1[k+1:n];
```

For real 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:k] || x2[k+1:n];

x1 = a * p1 + (1-a) * p2
c2 = p2[1:k] || x1[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 real and integer encodings. Let $n \geq 3$ be the length of the solution vector. 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:k1] || p2[k1+1:k2] || p1[k2+1:n];
c2 = p2[1:k1] || p1[k1+1:k2] || p2[k2+1:n];
```

For real 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:k1] || x2[k1+1:k2] || p1[k2+1:n];

x1 = a * p1 + (1-a) * p2;
c2 = p2[1:k1] || x1[k1+1:k2] || p2[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 vector encodings. 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 to the nearest integer. This operator has the advantage that it will always produce feasible offspring for a convex solution space. A disadvantage is that it will tend to produce offspring toward the interior of the search region, so that it might be less effective if the optimum lies on or near the search region boundary.

Heuristic: This operator is defined for real vector encodings. It computes the first offspring from the two parents $p1$ and $p2$ as

```
c1 = a * (p2 - p1) + p2;
```

where $p2$ is the parent that has 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. Any component outside its bound will be set equal to that bound. The heuristic operator performs best when the objective function is smooth, and might 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 that is consistent with the position and ordering in the other parent. Start with two parents and randomly chosen cutpoints as follows:

```
p1 = {1 2|3 4 5 6|7 8 9};
p2 = {8 7|9 3 4 1|2 5 6};
```

The first step is to cross the selected segments. A dot (.) 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 tends to maintain similarity of both the absolute position and relative ordering of the sequence elements and is useful for a wide range of sequencing problems.

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 order of the elements of *p*2 is as follows (cycling back to the beginning):

```
2 5 6 8 7 9 3 4 1
```

After removing 3, 4, 5 and 6, which have already been placed in *c1*, you have

```
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 (also called 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, start with parents *p*1 and *p*2:

```
p1 = {1 2 3 4 5 6 7 8 9};
p2 = {8 7 9 3 4 1 2 5 6};
```

For the first child, choose 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 *p*1, because the 8 position in *p*2 is already taken by the 1 in *c1*:

```
c1 = {1 . . . . . 8 .};
```

Now the position of 5 must come from *p*1, 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 *p*2:

```
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 the greatest importance to the objective value.

A user-defined module for a crossover operator must be a subroutine that has four parameters. The module should compute and return two new offspring solutions in the first two parameters, based on the two parent solutions, which are passed into the module in the last two parameters. The module should not modify the parent solutions. A global clause can be used to pass in any additional information that the module might need.

Mutation Operators

The mutation operators are as follows:

Uniform: This operator is defined for real or integer encodings that have 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 might prove especially useful in early stages of the optimization, since it tends to distribute solutions widely across the search space and avoid premature convergence to a local optimum. However, in later stages of an optimization, when the search needs to home in on an optimum, the uniform operator might hinder the optimization.

Delta: This operator is defined for real and integer encodings. It first chooses an element of the solution at random, and then perturbs that element by a fixed amount, which is 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 enables you to control the scope of the search by using 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 chooses 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 chooses two locations at random, and then reverses the order of elements between them. This operator is most often applied to the traveling salesman problem.

A user-defined module for a mutation operator must be a subroutine that has exactly one parameter, which contains the solution that is to be mutated. The module replaces the parameter with the new mutated value. As with crossover, a global clause can be used to pass in any additional information that the module needs.

Executing a Genetic Algorithm

The following sections describe how to set up a genetic algorithm and how to handle constraints. Several examples are presented.

Using the GA Modules to Implement an Optimization

After you formulate the GA optimization problem, executing the genetic algorithm is simple and straightforward. Table 22.2 summarizes the GA modules that you can use to set each of the optimization parameters. Some modules support default values for parameters that are not specified, and these default values are also listed. Parameters shown in italics are not required in all cases.

Table 22.2 Establishing Optimization Parameters

Type	Set By	Parameter	Value
Encoding	GASETUP	Encoding	0 (general)
			1 (real vector)
			2 (integer vector)
			3 (integer sequence)
		size	Size of vector
		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 problem)
		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)

Table 22.2 (continued)

Type	Set By	Parameter	Value
			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 can run the GA. First, use the GAINIT call to implement the *initialization* phase of the GA. The GAINIT call generates an initial solution population and evaluates the objective value of each member solution. In the GAINIT call, you specify the population size and any constant bounds for the solution domain.

Next implement a loop such as a DO loop or a DO WHILE loop. Put a GAREGEN call and a GAGETVAL call inside the loop. The GAREGEN call implements the *regeneration* phase of the GA, which generates a new solution population based on selection, crossover, and mutation of the current solution population. The GAREGEN call replaces the current population with the new population and computes the new objective function values. The GAGETVAL call retrieves the objective function values for the current population. This enables you to monitor the convergence of the GA. 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, because it will always be the first member retrieved.

Typically the loop ends when a stopping criterion is satisfied. After the loop, you can retrieve the members of the solution population by using the GAGETMEM call. To end the optimization, you should always use the GAEND call to free up memory resources that were allocated to the GA.

Incorporating Local Optimization

One commonly used technique is to combine the GA with a local optimization technique that is specific to the problem being solved. This can be done within the GA framework by incorporating a local optimization into the objective function evaluation. You can return a locally optimized objective value and optionally replace the original solution with the optimized solution.

If you always replace the original solution with the locally optimized solution, convergence occurs faster, but the algorithm is 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 to 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).

Handling Constraints

In practice, optimization problems often include constraints, which can 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 nonlinear optimization of n variables that have upper and lower bounds is easily formulated by using real vector 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 constrained optimization is to repair solutions only after genetic operators have been applied. This is what the GA routines do when you use the heuristic crossover operator or delta mutation operator with fixed bounds; any individual component that violates an upper or lower bound is adjusted. You can repair a solution inside a user-defined crossover or mutation module, or you can make repairs by modifying the solution in a user-defined objective function, 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 might converge prematurely to the first feasible solution found. Also, convergence might be poor to a solution that lies on or near a constraint boundary.

Examples

This section provides examples of using a genetic algorithm to solve optimization problems.

Example 22.1: The Traveling Salesman Problem

To illustrate how to set up and execute a genetic algorithm, the following example searches for a solution to the traveling salesman problem. In this problem, cities are located on a two-by-five grid. The cost of traveling from one city to the next is given by the "taxicab distance" between the cities, also known as the Manhattan distance or the L_1 distance. The cost coefficients (the "distances") are stored in the coeffs matrix. The optimal route has a total distance of 10.

```
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); /* allocate vector for 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 bestMember[f=3.0 L="Best Member"], value[L="Final Best Value"];

call gaend(id);
```

For this test case, there is no call to GASETSEL. Therefore, the algorithm uses the default selection parameters, namely an elite value of 1 and a conventional tournament of size 2. Also, since there is no GASETCRO or GASETMUT call, the algorithm uses the default genetic operators: the "order" operator for crossover and the "invert" operator for mutation. The default mutation probability is 0.05. The output is shown in Figure 22.1.1.

Output 22.1.1 Solution of a Traveling Salesman Problem

Iteration	BestValue
1	18
2	18
3	16
4	16
5	16
6	16
7	16
8	14
9	10
10	10
11	10
12	10
13	10
14	10
15	10
16	10
17	10
18	10
19	10
20	10

Best Member
4 5 6 7 8 9 10 1 2 3

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. Notice 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.

As an experiment, you can specify different genetic operators that override the default operators. The following statements use the GASETCRO and GASETMUT calls to set the crossover and mutation operators, respectively. The GASETSEL call is used to specify the selection parameters.

```
/* 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 */
```

The remainder of the program is the same:

```
/* initialization phase */
call gainit(id,
            100);        /* initial population size */

/* execute regeneration loop */
niter = 15;              /* number of iterations */
BestValue = j(niter,1); /* allocate vector for 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 bestMember[f=3.0 L="Best Member"], value[L="Final Best Value"];

call gaend(id);
```

The output of this program is shown in Figure 22.1.2.

Output 22.1.2 Alternate Solution of a Traveling Salesman Problem

Iteration	BestValue
1	18
2	18
3	16
4	16
5	16
6	14
7	14
8	14
9	12
10	10
11	10
12	10
13	10
14	10
15	10

Best Member
9 10 1 2 3 4 5 6 7 8

Final Best Value
10

Notice that the convergence was faster than for the previous case, reaching an optimum after 13 iterations. This illustrates that the convergence of a GA might be sensitive to the choice of genetic operators and selection parameters. In general, changing the optimization parameters affects the convergence.

Example 22.2: Genetic Algorithm with Local Optimization

For the symmetric traveling salesman problem, a simple local optimization is to check each pair of adjacent locations in the solution and swap their positions if that improves the objective function value. The following program repeats Example 22.1 but uses a modified objective function that implements this strategy. The optimized solution is not written back out to the solution population except to get the final solution at the end.

```
proc iml;
call randseed(1234);
/* 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 };
```

```
/* define objective function with local optimization */
start TSPObjectiveFunction(r) global(coeffs, p);
   s = r;
   nc = ncol(s);
   /* local optimization: assume 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 randfun(1, "uniform") <= p then
      r = s;
   return (cost);
finish;

/* problem setup */
id = gasetup(3,         /* 3 -> integer sequence encoding */
             10,        /* number of locations */
             123);      /* initial 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 */
BestValue = j(niter,1); /* allocate vector for 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;

/* write local optimization back to all solutions */
p = 1.;       /* set global probability to 1 */
call gareeval(id);

/* print final solution */
call gagetmem(bestMember, value, id, 1);
print bestMember[f=3.0 L="Best Member"], value[L="Final Best Value"];

call gaend(id);
```

The results of running this program are shown in Output 22.2.1.

Output 22.2.1 Solution with Custom Objective Function

Iteration	BestValue
1	12
2	12
3	10
4	10
5	10
6	10
7	10
8	10
9	10
10	10

Best Member
2 3 4 5 6 7 8 9 10 1

Final Best Value
10

Convergence is much improved by the local optimization, which reaches the optimum in just 5 iterations, as compared to 13 when there was 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 22.3: Real-Valued Objective Optimization with Constraints

The following example illustrates some of the strengths and weaknesses of the arithmetic and heuristic crossover operators. The objective function to be minimized has a minimum at $x = xopt$ but 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. The heuristic crossover operator is used.

```
proc iml;

/* objective function, has minimum value 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-> floating-point vector encoding */
             3,          /* 3-> length of solution vectors */
             1234);      /* 0-> initial random seed */
call gasetobj(id,
              0,         /* 0->minimize a user module */
              "sin_obj"); /* module name */
call gasetcro(id,
              0.9,       /* crossover probabilty */
              4);        /* 4-> heuristic crossover operator */
call gasetmut(id,
              0.05,      /* mutation probability */
              2,         /* 2-> delta mutation operator */
              0.01);     /* delta value  */
call gasetsel(id,
              5,         /* carry best 5 solutions forward */
              1,         /* dual tournament */
              0.95);     /* best-player wins probability */

bounds = {-1 -1 -1,
           1  1  1};
call gainit(id,
            200,         /* initial population size */
            bounds);     /* upper/lower bounds for random components */

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);
```

```
   summary[i,1] = value[1];
   summary[i,2] = value[:];
end;

print (t(1:20))[L="Iteration"] summary;
call gaend(id);
```

The output results are shown in Output 22.3.1.

Output 22.3.1 Solution of a Vector Encoding Problem

	summary	
Iteration	BestValue	AvgValue
1	0.1840732	0.8926763
2	0.14112	0.7445004
3	0.14112	0.590506
4	0.14112	0.4736059
5	0.14112	0.4167638
6	0.14112	0.3648824
7	0.1161997	0.3521076
8	0.0853626	0.3376295
9	0.0853626	0.3232832
10	0.0853626	0.3421653
11	0.0853626	0.3535389
12	0.0853626	0.3491909
13	0.0846332	0.322776
14	0.0846332	0.3172681
15	0.0846332	0.2877799
16	0.0811054	0.2990599
17	0.0811054	0.2643643
18	0.0622256	0.2759049
19	0.0600092	0.2714607
20	0.0600092	0.2112439

To show the convergence of the overall population, the average value of the objective function for the whole population is printed. 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.

You can change the crossover to the arithmetic operator by changing the following GASETCRO call:

```
call gasetcro(id,
              0.9,      /* crossover probabilty */
              3);       /* 3-> arithmetic crossover operator */
```

The new results are shown in Output 22.3.2. Choosing the arithmetic operator has resulted in improved convergence.

Output 22.3.2 Improved Convergence of a Vector Encoding Problem

summary		
Iteration	BestValue	AvgValue
1	0.1840732	0.8926763
2	0.1667349	0.802783
3	0.1237553	0.6296512
4	0.0799191	0.4124451
5	0.028965	0.2574221
6	0.0112894	0.1594558
7	0.0112894	0.1002809
8	0.0110598	0.0644709
9	0.0059744	0.0451367
10	0.005141	0.0302558
11	0.0028995	0.0212172
12	0.0028995	0.0151924
13	0.0028995	0.0112661
14	0.002648	0.008682
15	0.002648	0.0070962
16	0.0011485	0.0056683
17	0.0008818	0.0047837
18	0.0008818	0.0043693
19	0.0008345	0.0032666
20	0.0004998	0.0028028

Suppose you change the problem characteristics again by changing the constraints so that the optimum lies on a boundary. The following statement moves the optimum to a boundary:

```
bounds = {0 0 0,
          1 1 1};
```

If you use the arithmetic operator, the algorithm fails to converge to the true optimum. This is a characteristic of the arithmetic operator, which converges to interior points. However, if you revert to the heuristic crossover operator, the algorithm rapidly converges to the optimum.

This example illustrates that the results of a GA are operator-dependent. For complicated problems that have an unknown solution, you might need to try a number of different combinations of parameters in order to have confidence that the algorithm has converged to a true global optimum.

Example 22.4: Integer Programming Knapsack Problem

The following example uses the integer encoding and user-defined modules for crossover and mutation. It formulates the knapsack problem by using integer encoding. The integer 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-defined number of elements in the solution vector. For this problem, the value of the global optimum is 18.

```
proc iml;
call randseed(123);
/* weight: weight of each of 15 objects
   reward: value of each of 15 objects
   limit: maximum weight that can be put in the backpack
*/
weight = {2 3 4 4 1   1   1   1   1   1   1   1   1   1   1};
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};
limit = 9;                    /* weight limit */
nswitches = 3;                /* number of mutation sites */

start knapsack( x ) global( weight, reward, limit);
   wsum = sum(weight # x);
   rew = sum(reward # x);
   if wsum>limit then       /* penalty for exceeding weight */
      rew = rew - 5 * (wsum - limit);
   return(rew);
finish;

start switch_mut(s) global(nswitches);
   n = ncol(s);
   k = sample(1:n, nswitches, "NoReplace");/* choose random elements */
   /* if item is not in knapsack, put it in; if item in sack, take it out */
   s[k] = ^s[k];
finish;

start uniform_cross(child1, child2, parent1, parent2);
   child1 = parent1;
   child2 = parent2;
   b = j(ncol(parent1),1);
   call randgen(b, "Bernoulli", 0.5); /* 0/1 vector */
   idx = loc(b=1);                 /* locations to cross */
   if ncol(idx)>0 then do;
      child1[idx] = parent2[idx];
      child2[idx] = parent1[idx];
   end;
finish;

id = gasetup(2,          /* 2-> integer vector encoding */
             15,         /* size of vector */
             123);
call gasetobj(id, 1, "knapsack"); /* maximize objective module */
```

```
call gasetcro(id,
              1.0,       /* crossover probability */
              0, "uniform_cross"); /* user crossover module */
call gasetmut(id,
              0.20,                /* mutation probabilty */
              0, "switch_mut");    /* user mutation module */
call gasetsel(id,
              3,         /* carry 3 elites to next generation */
              1,         /* dual tournament */
              0.95);     /* best-player-wins probabilty */
call gainit(id, 100,     /* initial population size */
            {0 0 0 0 0 0 0 0 0 0 0 0 0 0 0,   /* lower bounds */
             1 1 1 1 1 1 1 1 1 1 1 1 1 1 1}); /* upper bounds */
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 mem[f=1.0 L="Best Member"], value[L="Final Best Value"];
print (t(1:niter))[L="Iteration"] summary;
call gaend(id);
```

The output results are shown in Output 22.4.1.

Output 22.4.1 Solution of a Knapsack Problem

Best Member
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0

Final Best Value
18

Output 22.4.1 *continued*

summary		
Iteration	BestValue	AvgValue
1	12.8	2.086
2	16.3	6.175
3	16.6	7.484
4	16.6	9.162
5	16.7	9.644
6	16.7	10.435
7	16.8	10.111
8	16.8	10.619
9	16.8	12.144
10	18	11.651
11	18	12.263
12	18	11.867
13	18	12.468
14	18	12.828
15	18	12.921
16	18	13.335
17	18	12.734
18	18	12.495
19	18	12.883
20	18	12.644

Note that for this problem, the mutation parameters are set higher than is often seen for GAs because higher values are necessary to prevent premature convergence.

Example 22.5: Optimization with Linear Constraints Using Repair Strategy

This problem seeks a minimum within a convex domain that is specified by a convex hull. All points in the search space are normalized linear combinations of the vertices of the convex hull. Each solution is represented by a set of weights w such that there is one w_i for each point in the convex hull, $0 \leq w_i \leq 1$, and $\Sigma w_i = 1$. In this example the feasible region is the rectangular region that is the convex hull defined by the vertices (–3 –2), (3 –2), (–3 2), and (3 2). The objective function is a six-hump camelback function (Michalewicz 1996, Appendix B) that has a known global minimum value of –1.0316 at two different points, (–0.0898,0.7126) and (0.0898,–0.7126). A user-defined mutation module is specified, and the simple crossover operator is used. Both the mutation operator and the crossover operator can produce solutions that violate the constraints, so in the objective function each solution is checked and renormalized to bring it back within the convex hull.

```
proc iml;
call randseed(12345);
/* Objective function has global minimum value of -1.0316
   at x = {-0.0898  0.7126} and x = { 0.0898 -0.7126}
*/
start sixhump(w) global(cvxhull);
   sum = w[+];
   /* guard against remote possibility of all-zero weights */
   if sum<=0 then do;
      nc = ncol(w);
      w = j(1, nc, 1/nc );
      sum = 1;
   end;

   w = w / sum;              /* normalize weights */
   x = (w * cvxhull)[+,];    /* convert to x-coordinate form */
   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);
   r = j(1, nrow(cvxhull));
   call randgen(r, "uniform");
   w = r / sum(r);
finish;

/* mutation module */
start cvxmut(w) global(cvxhull);
   row = sample(1:nrow(cvxhull), 1);       /* random row */
   call randgen(u, "uniform");             /* random weight */
   w[row] = u;
finish;

id = gasetup(1,          /* 1-> real encoding */
             nrow(cvxhull), /* size of weight vector */
             1234);
call gasetobj(id,
              0, "sixhump"); /* minimize objective function */
call gasetsel(id,
              5,         /* carry 5 elites to next 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,      /* mutation probabilty */
              0, "cvxmut");  /* user mutation module */

call gainit(id,
           100,          /* initial population size */
              ,          /* not using constant bounds */
           "cvxinit");   /* initialization module */

niter = 25; /* 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 bestX[L="Best X"], value[L="Best Value"];
print (t(1:nIter))[L="Iteration"] summary;
call gaend(id);
```

The output results are shown in Output 22.5.1.

Output 22.5.1 Solution of Linearly Constrained Problem

Best X	
0.0897764	-0.712784

Best Value
-1.031628

Output 22.5.1 *continued*

Iteration	summary BestValue	AvgValue
1	-0.961813	0.9531504
2	-1.013511	0.520233
3	-1.013511	0.224153
4	-1.013511	-0.012848
5	-1.013511	0.0838901
6	-1.013511	0.1262223
7	-1.023508	-0.085911
8	-1.023508	-0.124048
9	-1.023508	-0.103719
10	-1.026595	0.0236373
11	-1.026595	0.2859809
12	-1.031296	-0.046364
13	-1.031302	-0.003027
14	-1.031302	-0.2077
15	-1.03154	-0.094963
16	-1.03155	-0.078811
17	-1.03155	0.2706643
18	-1.03155	-0.201072
19	-1.03155	-0.463889
20	-1.031558	-0.647181
21	-1.031628	-0.801233
22	-1.031628	-0.824697
23	-1.031628	-0.838554
24	-1.031628	-0.856339
25	-1.031628	-0.947385

Any problem that has linear constraints could be formulated in this way, provided that you can specify the convex hull that corresponds to the constraints. The genetic operators and the repair strategy are straightforward to apply. Nevertheless, this example shows that the GA can converge to a global optimum.

References

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. New York: Springer-Verlag.

Miller, B. L., and Goldberg, D. E. (1995). *Genetic Algorithms, Tournament Selection, and the Effects of Noise*. Technical Report 95006, Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign.

Chapter 23
Sparse Matrix Algorithms

Contents

Overview **465**
Iterative Methods **466**
Input Data Description 467
Example: Conjugate Gradient Algorithm 467
Example: Minimum Residual Algorithm 469
Example: Biconjugate Gradient Algorithm 470
Symbolic LDL and Cholesky Factorizations **471**
References **471**

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 ITSOLVER call supports the following classes of iterative solvers:

- conjugate gradient for symmetric positive-definite systems
- conjugate gradient squared for general nonsingular systems
- minimum residual for symmetric indefinite systems
- biconjugate gradient for general nonsingular systems

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 467.

The SOLVELIN call supports the following direct sparse solvers for symmetric positive-definite systems:

- symbolic LDL
- Cholesky

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 subroutine.

Iterative Methods

The conjugate gradient algorithm can be interpreted as the following optimization problem: minimize $\phi(x)$ defined by

$$\phi(x) = 1/2x^T Ax - x^T b$$

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:

$$r_i \perp \mathcal{K}_i(A; r_0), \quad r_i = Ax_i - b$$

where $\mathcal{K}_i$ is a Krylov subspace:

$$\mathcal{K}_i\,(A; r) = \text{span}\{r,\ Ar,\ A^2r,\ \ldots,\ A^{i-1}r\}$$

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

The ITSOLVER call has the following syntax and arguments:

CALL ITSOLVER(*x, error, iter, method, A, b* < *, precon*> < *, tol*> < *, maxiter*> < *, start*> < *, history*> **);**

The conjugate gradient and minimum residual algorithms (*method* = 'CG' or *method* = 'MINRES') require A to be symmetric; hence you must specify only the lower triangular part of A, while the remaining algorithms require *all* nonzero coefficients to be listed. The following table lists valid values for the *precond* parameter for each class of algorithm.

Table 23.1 Subroutine Definitions and Valid Preconditioners

Method Value	Algorithm	Preconditioners
`"CG"`	conjugate gradient	`"NONE" "IC" "DIAG"`
`"MINRES"`	minimum residual	`"NONE" "IC" "DIAG"`
`"BICG"`	biconjugate gradient	`"NONE" "MILU"`
`"CGS"`	conjugate gradient squared	`"NONE"`

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 sides $b = (1\ 1\ 1\ 1)^T$. 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 program for this example 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 };

b = {1, 1, 1, 1};   /* right-hand sides */
tol = 1e-7;         /* desired solution tolerance (optional) */
maxit = 200;        /* maximum number of iterations (optional) */
hist = j(50, 1);    /* allocate iteration progress (optional) */
start = {2, 3, 4, 5};   /* provide an initial guess (optional) */

/* call conjugate gradient method */
call itsolver(
   x, st, it,          /* output parameters */
   'cg', A, b, 'ic',   /* input parameters */
   tol,                /* optional control parameters */
   maxit,
   start,
   hist    );

/* print solution, tolerance, and number of iterations */
print x, st, it;
```

Notice that the example used an incomplete Cholesky preconditioner, which is recommended. Figure 23.1 shows the results of the program:

Figure 23.1 Conjugate Gradient Solution of a Linear System

x
0.5882353
-0.764706
0.1764706
1.0980392

st
1.145E-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 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. The other data and options are unchanged:

```
/* 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 };

b = {1, 1, 1, 1};   /* right-hand sides */
tol = 1e-7;         /* desired solution tolerance (optional) */
maxit = 200;        /* maximum number of iterations (optional) */
hist = j(50, 1);    /* allocate iteration progress (optional) */
start = {2, 3, 4, 5};   /* provide an initial guess (optional) */

/* call minimum residual method */
call itsolver(
  x, st, it,            /* output parameters */
  'minres', a, b, 'ic', /* input parameters */
  tol,                  /* optional control parameters */
  maxit,
  start,
  hist    );

/* print solution, tolerance, and number of iterations */
print x, st, it;
```

Figure 23.2 Minimum Residual Solution of a Linear System

x
-0.27027
0.1891892
0.0810811
0.1441441

st
2.325E-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 program for this example follows. The output is shown in Figure 23.3. It is important to look at the actual tolerance in order to know how effective the solution is.

```
/* biconjugate gradient algorithm */
/*  value  row column */
A = { 10    1     1,
      3     2     2,
      4     3     3,
      0.1   2     1,
      0.2   1     3 };

b = {1, 1, 1};      /* right-hand sides */
tol = 1e-9;         /* desired solution tolerance (optional) */
maxit = 10000;      /* maximum number of iterations (optional) */
hist = j(50, 1);    /* allocate iteration progress (optional) */
start = {2, 3, 4};  /* provide an initial guess (optional) */

/* call biconjugate gradient subroutine */
call itsolver(
   x, st, it,            /* output parameters */
   'bicg', a, b, 'milu', /* input parameters */
   tol,                  /* optional control parameters */
   maxit,
   start,
   hist    );

/* print solution, tolerance, and number of iterations */
print x, st, it;
```

Figure 23.3 Biconjugate Gradient Solution of a Linear System

x
0.095
0.3301667
0.25

st
1.421E-16

it
3

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 SAS/IML function SOLVELIN provides an interface to both algorithms; the minimum degree ordering heuristic is invoked automatically as follows:

CALL 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 program for this example follows. The output is shown in Figure 23.4.

```
/* 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 };

b = {1, 1, 1, 1};   /* right-hand sides */
/* invoke LDL factorization */
call solvelin (x, status, a, b, "LDL");

print x;            /* print solution */
```

Figure 23.4 Symbolic LDL Solution of a Linear System

x
0.5882353
-0.764706
0.1764706
1.0980392

References

Golub, G. H., and Van Loan, C. F. (1996). *Matrix Computations*. 3rd ed. Baltimore: Johns Hopkins University Press.

Greenbaum, A. (1997). *Iterative Methods for Solving Linear Systems*. Philadelphia: SIAM.

Hestenes, M. R., and Stiefel, E. (1952). "Methods of Conjugate Gradients for Solving Linear Systems." *Journal of Research of the National Bureau of Standards* 46:409–436.

Paige, C. C., and Saunders, M. A. (1975). "Solution of Sparse Indefinite Systems of Linear Equations." *SIAM Journal on Numerical Analysis* 12:617–629.

Chapter 24
Further Notes

Contents

Memory and Workspace 473
Accuracy 474
Error Diagnostics 475
Efficiency 476
Missing Values 477
Principles of Operation 478
Operation-Level Execution 479

Memory and Workspace

SAS/IML matrices are kept in RAM. Memory is automatically allocated when needed. If you are interested in how memory is used and allocated in PROC IML, the following sections explain the details.

There are two logical areas of memory: the *symbol space* and the *workspace*. The symbol space contains memory that is associated with the symbol table and compiled statements. The workspace contains all values in SAS/IML matrices. The workspace itself is divided into one or more blocks of memory.

At the start of a PROC IML session, the symbol space and the first workspace block are allocated automatically. If matrix operations consume the available workspace, more memory is allocated. The SYMSIZE= and WORKSIZE= options in the PROC IML statement give you control over the initial size of symbol space 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 symbol space memory becomes full, more memory is allocated automatically. Symbol space is recycled whenever possible. For example, temporary symbols can be deleted after they are used in evaluating an expression. The symbol space formerly used by these temporary variables is added to a list of free symbol-table nodes. When allocating temporary variables to evaluate another expression, PROC IML looks for symbol-table nodes in this list first before consuming unused symbol space.

Workspace memory is allocated as needed to accommodate matrix contents. As matrices are assigned, the workspace memory fills up. As you free matrices (or as PROC IML frees temporary intermediate results), holes appear in the memory blocks. IML will first attempt to fill these holes when allocating new matrix

memory. When a memory block does not have room to store a new matrix, IML allocates a new block of memory. 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 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 is 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 maximum efficiency, use a high WORKSIZE= value and a low SYMSIZE= value for data-intensive applications that involve a few large matrices, and use a high SYMSIZE= value for symbol-intensive applications that involve many matrices, perhaps through the use of many IML modules.

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 symbol space and workspace, there might not be enough remaining to load these functions, resulting in the error message

`Unable to load module` *module-name.*

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 by using the FREE command.
- Store matrices you will need later in external library storage by using the STORE command, and then FREE their values. You can restore the matrices later by using the LOAD command. See Chapter 19.
- 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, a message is displayed in the SAS log. The message includes a description of the error, the operation being performed, and the line and column number of the statement in which the error occurred. The names of matrices that are involved in the operation are also displayed. Matrix names that begin with a number sign (#) or an asterisk (*) are temporary names that the IML procedure assigns.

Some errors can be caught at parse time. For example, syntax errors such as an incorrect number of arguments, unbalanced parentheses, and a missing semicolon are detected when the program is parsed.

When an error occurs, the operation is not completed and the result (if any) is not assigned a value. If an error occurs during execution of statements inside a module, a PAUSE statement is automatically issued. You can correct the error and resume execution of statements inside a module by submitting a RESUME statement.

The following list describes common errors and their associated error messages:

- Referencing a matrix that has not been set to a value. For example, the following statement references a matrix that has no value:

```
y = EmptyMatrix + 1;
```

```
ERROR: (execution) Matrix has not been set to a value.
```

- Making a subscripting error. For example, the following statements refer to an element that is not present in a matrix:

```
x = 1:3; x[4] = 1;
```

```
ERROR: (execution) Invalid subscript or subscript out of range.
```

- Performing an operation that involves nonconformable matrix arguments. When you add, subtract, or multiply matrices, the dimensions of the matrices must satisfy certain conditions. Otherwise, the matrix operation is not defined. Let **A** and **B** be matrices. Then the following conditions must hold:

  – The elementwise operations $\mathbf{A} + \mathbf{B}$, $\mathbf{A} - \mathbf{B}$, $\mathbf{A} \# \mathbf{B}$, and $\mathbf{A} / \mathbf{B}$ are defined if **A** and **B** are the same dimensions. If **A** is an $n \times p$ matrix, the operations are also defined if **B** is a $1 \times p$ row vector, an $n \times 1$ column vector, or a 1×1 scalar. Similarly, if **B** is a matrix, **A** can be a vector of appropriate dimensions or a scalar.

  – The matrix multiplication $\mathbf{A} * \mathbf{B}$ is defined if the number of columns of **A** equals the number of rows of **B**.

  – The horizontal concatenation operation **A** || **B** is defined if the number of rows of **A** equals the number of rows of **B**.

  – The vertical concatenation operation **A** // **B** is defined if the number of columns of **A** equals the number of columns of **B**.

  For example, the following statements generate an error:

```
x = 1:3; y = 1:4; m = x+y;
```

```
ERROR: (execution) Matrices do not conform to the operation.
```

- Passing a matrix of the wrong dimensions to function. For example, the following statement passes a vector to a function that is expecting a scalar argument:

```
d = do(1, 2, 1:3); /* third arg must be scalar */

ERROR: (execution) Argument should be a scalar.
```

- Passing a matrix that is not square to a function (such as INV, DET, or SOLVE) that requires a square matrix. For example:

```
v = eigval(1:3);

ERROR: (execution) Matrix should be square.
```

- Passing a matrix that is not symmetric to a function that requires a symmetric matrix. For example:

```
call geneig(M, E, 1 2, 3 4, 1 2, 2 1);

ERROR: (execution) Matrix should be symmetric.
```

- Passing a matrix that is singular to functions that require a nonsingular matrix. For example:

```
A = inv(1 1, 2 2);

ERROR: (execution) Matrix should be non-singular.
```

- Passing a matrix that is not positive definite or positive semidefinite to functions that require such matrices. For example:

```
G = root(1 2, 2 3);

ERROR: (execution) Matrix should be positive definite.
```

- Attempting to allocate a matrix for which there is not enough RAM. (See the section "Memory and Workspace" on page 473.) For example:

```
X = j(1e6, 1e6);

ERROR: (execution) Unable to allocate sufficient memory.
```

- Passing a numerical matrix to a function that expects a character matrix, or passing a character matrix to a function that expects a numerical matrix. For example:

```
v = eigval(A B, C D);

ERROR: (execution) Character argument should be numeric.
```

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 the following 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 for certain operations. The operators in the following list recognize missing values and propagate them. For example, matrix multiplication of a matrix with missing values is not supported. Most matrix operators and functions do not support missing values. Furthermore, many linear algebraic operations are not mathematically defined for a matrix with missing values. For example, the inverse of a matrix with missing values is meaningless.

Missing values are coded in the bit pattern of very large negative numbers, as an IEEE “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 following table.

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 about 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 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 25
Language Reference

Contents

Overview **491**
Statements, Functions, and Subroutines by Category **492**
Operators **506**
Addition Operator: + ... 506
Comparison Operators: <, <=, >, >=, =, ^= 507
Concatenation Operator, Horizontal: ǁ 508
Concatenation Operator, Vertical: // 509
Direct Product Operator: @ 510
Division Operator: / 511
Element Maximum Operator: <> 512
Element Minimum Operator: >< 513
Index Creation Operator: : 514
List Creation Operator: [ ] 516
List Item Operator: $ 516
List Sublist Operator: [ ] 517
Logical Operators: &, |, ^ 517
Multiplication Operator, Elementwise: # 518
Multiplication Operator, Matrix: * 520
Power Operator, Elementwise: ## 521
Power Operator, Matrix: ** 521
Sign Reversal Operator: – 523
Subscripts: [ ] 523
Subscript Reduction: [ ] 525
Subtraction Operator: – 526
Transpose Operator: ` 527
Statements, Functions, and Subroutines **528**
ABORT Statement 528
ABS Function 528
ALL Function 529
ALLCOMB Function 529
ALLPERM Function 531
ANY Function 532
APPCORT Call 532
APPEND Statement 534
APPLY Function 537

ARMACOV Call 539
ARMALIK Call 540
ARMASIM Function 542
BAR Call 543
BIN Function 547
BLANKSTR Function 549
BLOCK Function 549
BOX Call 550
BRANKS Function 553
BSPLINE Function 554
BTRAN Function 556
BYTE Function 557
CALL Statement 558
CCEPSTRUM Call 558
CHANGE Call 560
CHAR Function 561
CHOOSE Function 562
CLOSE Statement 563
CLOSEFILE Statement 564
COL Function 565
COLVEC Function 566
COMPORT Call 566
CONCAT Function 569
CONTENTS Function 570
CONTINUE Statement 571
CONVEXIT Function 571
CORR Function 572
CORR2COV Function 574
COUNTMISS Function 574
COUNTN Function 575
COUNTUNIQUE Function 576
COV Function 577
COV2CORR Function 578
COVLAG Function 578
CREATE Statement 579
CSHAPE Function 582
CUSUM Function 584
CUPROD Function 584
CV Function 585
CVEXHULL Function 585
DATASETS Function 586
DELETE Call 586
DELETE Statement 587
DESIGN Function 588

DESIGNF Function . . . 589
DET Function . . . 589
DFCONV Function . . . 590
DFDESIGN Call . . . 591
DFFILT Function . . . 593
DFFREQZ Function . . . 596
DFFREQZZPK Function . . . 597
DFMEDFILT Function . . . 598
DFORDER Call . . . 600
DFSOSFILT Function . . . 605
DFSOSFREQZ Function . . . 607
DFSOSFREQZZPK Function . . . 608
DIAG Function . . . 610
DIF Function . . . 611
DIMENSION Function . . . 612
DISTANCE Function . . . 612
DO Function . . . 615
DO Statement . . . 616
DO Statement, Iterative . . . 617
DO DATA Statement . . . 617
DO Statement with an UNTIL Clause . . . 618
DO Statement with a WHILE Clause . . . 619
DURATION Function . . . 620
ECHELON Function . . . 621
EDIT Statement . . . 621
EIGEN Call . . . 623
EIGVAL Function . . . 626
EIGVEC Function . . . 627
ELEMENT Function . . . 628
END Statement . . . 629
ENDSUBMIT Statement . . . 629
EXECUTE Call . . . 629
EXECUTEFILE Call . . . 630
EXP Function . . . 632
EXPMATRIX Function . . . 632
EXPANDGRID Function . . . 633
EXPORTDATASETTOR Call . . . 634
EXPORTMATRIXTOR Call . . . 635
EXPORTTABLETOR Call . . . 635
FARMACOV Call . . . 636
FARMAFIT Call . . . 638
FARMALIK Call . . . 639
FARMASIM Call . . . 641
FDIF Call . . . 643

FEVAL Function 643
FFT Function 644
FFTC Function 646
FILE Statement 647
FIND Statement 648
FINISH Statement 649
FORCE Statement 649
FORWARD Function 650
FREE Statement 650
FROOT Function 651
FULL Function 652
GAEND Call 654
GAGETMEM Call 654
GAGETVAL Call 655
GAINIT Call 655
GAREEVAL Call 656
GAREGEN Call 657
GASETCRO Call 657
GASETMUT Call 661
GASETOBJ Call 663
GASETSEL Call 664
GASETUP Function 664
GENEIG Call 666
GEOMEAN Function 668
GINV Function 668
GOTO Statement 670
GSCALE Call 671
GSORTH Call 672
HADAMARD Function 674
HALF Function 675
HANKEL Function 675
HARMEAN Function 677
HDIR Function 677
HEATMAPCONT Call 678
HEATMAPDISC Call 682
HERMITE Function 684
HISTOGRAM Call 685
HOMOGEN Function 687
I Function 688
ICCEPSTRUM Function 688
IF-THEN/ELSE Statement 690
IFFT Function 692
IFFTC function 693
IMPORTDATASETFROMR Call 694

IMPORTMATRIXFROMR Call 696
IMPORTTABLEFROMR Function 697
INDEX Statement 697
INFILE Statement 699
INPUT Statement 699
INSERT Function 701
INT Function 702
INV Function 703
INVUPDT Function 704
IPF Call 706
ISEMPTY Function 718
ISSKIPPED Function 718
ITSOLVER Call 719
J Function 722
JROOT Function 723
KALCVF Call 724
KALCVS Call 727
KALDFF Call 730
KALDFS Call 733
KPCATRAIN call 736
KPCASCORE function 740
KURTOSIS Function 744
LAG Function 745
LAMBERTW Function 746
LAV Call 747
LCP Call 751
LEAVE Statement 754
LENGTH Function 755
LINK Statement 755
LIST Statement 756
LISTADDITEM Call 757
LISTCREATE Function 757
LISTDELETEITEM Call 758
LISTDELETENAME Call 759
LISTGETALLNAMES Function 759
LISTGETITEM Function 760
LISTGETNAME Function 761
LISTGETSUBITEM Function 762
LISTINDEX Function 763
LISTINSERTITEM Call 764
LISTLEN Function 764
LISTSETITEM Call 765
LISTSETNAME Call 765
LISTSETSUBITEM Call 766

LMS Call 767
LOAD Statement 776
LOC Function 776
LOG Function 777
LOGABSDET Function 778
LP Call 778
LPSOLVE Call 779
LTS Call 781
LUPDT Call 788
MAD Function 789
MAGIC Function 791
MAHALANOBIS Function 792
MARG Call 793
MATTRIB Statement 795
MAX Function 797
MAXQFORM Call 797
MCD Call 800
MEAN Function 804
MEDIAN Function 806
MILPSOLVE Call 806
MIN Function 810
MOD Function 810
MODULEI Call 811
MODULEIC Function 812
MODULEIN Function 812
MODULESTACK Function 813
MVE Call 814
NAME Function 819
NCOL Function 820
NDX2SUB Function 820
NLENG Function 822
Nonlinear Optimization and Related Subroutines 822
NLPCG Call 825
NLPDD Call 826
NLPFDD Call 828
NLPFEA Call 832
NLPHQN Call 833
NLPLM Call 835
NLPNMS Call 836
NLPNRA Call 840
NLPNRR Call 843
NLPQN Call 845
NLPQUA Call 850
NLPTR Call 853

NORM Function 854
NROW Function 855
NUM Function 856
ODE Call 856
ODSGRAPH Call 862
OPSCAL Function 865
ORPOL Function 866
ORTVEC Call 872
PACKAGE Statement 876
PACKAGE HELP Statement 877
PACKAGE INFO Statement 878
PACKAGE INSTALL Statement 878
PACKAGE LIBNAME Statement 879
PACKAGE LIST Statement 880
PACKAGE LOAD Statement 881
PACKAGE UNINSTALL Statement 882
PARENTNAME Function 882
PALETTE Function 883
PAUSE Statement 888
POLYROOT Function 888
PRINT Statement 889
PROD Function 891
PRODUCT Function 891
PURGE Statement 892
PUSH Call 894
PUT Statement 895
PV Function 896
QNTL Call 898
QR Call 899
QUAD Call 904
QUARTILE Function 910
QUEUE Call 910
QUIT Statement 911
RANCOMB Function 912
RANDDIRICHLET Function 913
RANDFUN Function 914
RANDGEN Call 915
RANDMULTINOMIAL Function 928
RANDMVT Function 929
RANDNORMAL Function 930
RANDWISHART Function 931
RANPERK Function 933
RANPERM Function 934
RANDSEED Call 935

RANGE Function 936
RANK Function 937
RANKTIE Function 939
RATES Function 941
RATIO Function 942
RCEPSTRUM Function 943
RDODT and RUPDT Calls 945
READ Statement 948
REMOVE Function 950
REMOVE Statement 950
RENAME Call 951
REPEAT Function 951
REPLACE Statement 952
RESET Statement 953
RESUME Statement 955
RETURN Statement 955
ROOT Function 956
ROW Function 958
ROWCAT Function 958
ROWCATC Function 959
ROWVEC Function 960
RSUBSTR Function 960
RUN Statement 961
RUPDT Call 961
RZLIND Call 962
SAMPLE Function 974
SAVE Statement 975
SCATTER Call 976
SEQ, SEQSCALE, and SEQSHIFT Calls 979
SEQSCALE Call 990
SEQSHIFT Call 990
SERIES Call 991
SETDIF Function 993
SETIN Statement 994
SETOUT Statement 995
SHAPE Function 996
SHAPECOL Function 997
SHOW Statement 998
SKEWNESS Function 1000
SOLVE Function 1000
SOLVELIN Call 1001
SORT Call 1002
SORT Statement 1003
SORTNDX Call 1004

SOUND Call 1005
SPARSE Function 1006
SPECTROGRAM Call 1007
SPLINE and SPLINEC Calls 1013
SPLINEV Function 1022
SPOT Function 1022
SQRSYM Function 1023
SQRT Function 1024
SQRVECH Function 1024
SSQ Function 1025
STANDARD Function 1025
START Statement 1026
STD Function 1028
STOP Statement 1028
STORAGE Function 1028
STORE Statement 1029
SUB2NDX Function 1030
SUBMIT Statement 1031
SUBSTR Function 1033
SUM Function 1034
SUMMARY Statement 1034
SVD Call 1037
SWEEP Function 1038
SYMSQR Function 1040
T Function 1041
TABLEADDVAR Call 1041
TABLECREATE Function 1042
TABLECREATEFROMDATASET Function 1044
TABLEGETVARDATA Function 1044
TABLEGETVARFORMAT Function 1045
TABLEGETVARINDEX Function 1045
TABLEGETVARINFORMAT Function 1046
TABLEGETVARLABEL Function 1047
TABLEGETVARNAME Function 1047
TABLEGETVARTYPE Function 1048
TABLEISEXISTINGVAR Function 1048
TABLEISVARNUMERIC Function 1049
TABLEPRINT Call 1050
TABLERENAMEVAR Call 1053
TABLESETVARFORMAT Call 1053
TABLESETVARINFORMAT Call 1054
TABLESETVARLABEL Call 1054
TABLESORT Call 1055
TABLEWRITETODATASET Call 1056

TABULATE Call 1057
TFHILBERT Function 1057
TFPWV Function 1058
TFSTFT Function 1061
TFWINDOW Function 1063
TOEPLITZ Function 1067
TPSPLINE Call 1069
TPSPLNEV Call 1071
TRACE Function 1073
TRISOLV Function 1074
TSBAYSEA Call 1075
TSDECOMP Call 1075
TSMLOCAR Call 1075
TSMLOMAR Call 1075
TSMULMAR Call 1076
TSPEARS Call 1076
TSPRED Call 1076
TSROOT Call 1076
TSTVCAR Call 1076
TSUNIMAR Call 1077
TYPE Function 1077
UNION Function 1077
UNIQUE Function 1078
UNIQUEBY Function 1078
USE Statement 1080
VALSET Call 1081
VALUE Function 1081
VAR Function 1082
VARMACOV Call 1083
VARMALIK Call 1085
VARMASIM Call 1086
VECDIAG Function 1088
VECH Function 1089
VNORMAL Call 1089
VTSROOT Call 1091
WAVFT Call 1092
WAVGET Call 1095
WAVIFT Call 1097
WAVPRINT Call 1099
WAVTHRSH Call 1100
WIDETOLONG Call 1100
XMULT Function 1102
XSECT Function 1103
YIELD Function 1103

Base SAS Functions Accessible from SAS/IML Software **1104**
Bitwise Logical Operation Functions 1105
Character and Formatting Functions 1105
Character String Matching Functions and Subroutines 1109
Combinatorial Functions 1109
Date and Time Functions 1110
Descriptive Statistics Functions and Subroutines 1111
Double-Byte Character String Functions 1112
External Files Functions 1112
File I/O Functions 1113
Financial Functions 1114
Macro Functions and Subroutines 1115
Mathematical Functions and Subroutines 1115
Probability Functions 1116
Quantile Functions 1117
Random Number Functions and Subroutines 1117
State and Zip Code Functions 1118
Time Zone Functions 1118
Trigonometric and Hyperbolic Functions 1118
Truncation Functions 1119
Web Tools 1119
References **1119**

Overview

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

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

Statements, Functions, and Subroutines by Category

Mathematical Functions

ABS function	computes the absolute value
EXP function	applies the exponential function
EXPMATRIX function	returns the exponential of a matrix
INT function	truncates a value
LAMBERTW function	computes either branch of the Lambert W function
LOG function	computes the natural logarithm
LOGABSDET	computes the natural logarithm of the absolute value of the determinant
MOD function	computes the modulo (remainder)
SQRT function	computes the square root

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

- "Mathematical Functions and Subroutines" on page 1115
- "Probability Functions" on page 1116
- "Quantile Functions" on page 1117
- "Trigonometric and Hyperbolic Functions" on page 1118
- "Truncation Functions" on page 1119

Reduction Functions

MAX function	finds the maximum value of a matrix
MIN function	finds the smallest element of a matrix
PROD function	multiplies all elements
SSQ function	computes the sum of squares of all elements
SUM function	sums all elements

Matrix Inquiry Functions

ALL function	checks for all nonzero elements
ANY function	checks for any nonzero elements
COL function	returns a matrix, M, that is the same size as the input matrix and such that $M[i, j] = j$.
COUNTMISS function	returns the number of missing values
COUNTN function	returns the number of nonmissing values

COUNTUNIQUE function	returns the number of unique values
CHOOSE function	evaluates a logical matrix and returns values based on whether each element is true or false
DIMENSION function	returns the number of rows and columns of a matrix
ISEMPTY function	returns 1 if the argument is an empty matrix (zero rows and columns) and 1 otherwise
LOC function	finds indices for the nonzero elements of a matrix
NCOL function	finds the number of columns of a matrix
NLENG function	finds the size of an element
NROW function	finds the number of rows of a matrix
ROW function	returns a matrix, M, that is the same size as the input matrix and such that $M[i, j] = i$.
TYPE function	determines the type of a matrix

Matrix Sorting and BY-Group Processing Functions

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

Matrix Reshaping Functions

BLOCK function	forms block-diagonal matrices
BTRAN function	computes a block transpose
COLVEC function	converts a matrix into a column vector
DIAG function	creates a diagonal matrix
DO function	produces an arithmetic series
EXPANDGRID function	returns a matrix that contains all combinations of elements from specified vectors
FULL function	converts a matrix stored in a sparse format into a full (dense) matrix
I function	creates an identity matrix
INSERT function	inserts one matrix inside another
J function	creates a matrix of identical values
MAGIC function	returns a magic square of a given size
REMOVE function	discards elements from a matrix
REPEAT function	creates a new matrix of repeated values
ROWVEC function	converts a matrix into a row vector
SHAPE function	reshapes and repeats values
SHAPECOL function	reshapes and repeats values by columns

SPARSE function	converts a matrix that contains many zeros into a matrix stored in a sparse format
SQRSYM function	converts a symmetric matrix to a square matrix
SQRVECH function	converts a symmetric matrix which is stored columnwise to a square matrix
SYMSQR function	converts a square matrix to a symmetric matrix
T function	transposes a matrix
VECH function	creates a vector from the columns of the lower triangular elements of a matrix
VECDIAG function	creates a vector from a diagonal
WIDETOLONG call	creates long data from a multivariate time series

Combinatorial Functions

ALLCOMB function	generates all combinations of n elements taken k at a time
ALLPERM function	generates all permutations of n elements
RANCOMB function	returns random combinations of n elements taken k at a time
RANPERK function	returns generates a random permutation of k elements from a finite set of n elements, $k \leq n$
RANPERM function	returns random permutations of n elements

Character Manipulation Functions

BLANKSTR function	returns a blank string of a specified length.
BYTE function	translates numbers to ordinal characters
CHANGE call	replaces text
CHAR function	produces a character representation of a matrix
CONCAT function	concatenates elementwise strings
CSHAPE function	reshapes and repeats character values
LENGTH function	finds the lengths of character matrix elements
NAME function	lists the names of arguments
NUM function	produces a numeric representation of a character matrix
ROWCAT function	concatenates rows without using blank compression
ROWCATC function	concatenates rows by using blank compression
SUBSTR function	takes substrings of matrix elements

You can also call functions in Base SAS software such as those documented in the sections "Character and Formatting Functions" on page 1105 and "Character String Matching Functions and Subroutines" on page 1109.

Functions for Generating Random Numbers and Simulations

RANDFUN function	returns a matrix of random numbers from a specified distribution
RANDGEN call	generates pseudorandom numbers from specified distributions
RANDSEED call	initializes seed for subsequent RANDGEN calls
SAMPLE function	generates a random sample of a finite set

You can also call functions in Base SAS software such as those documented in the section "Random Number Functions and Subroutines" on page 1117.

For sampling from multivariate distributions, you can use the following functions:

RANDDIRICHLET	generates a random sample from a Dirichlet distribution
RANDMULTINOMIAL	generates a random sample from a multinomial distribution
RANDMVT	generates a random sample from a multivariate Student's *t* distribution
RANDNORMAL	generates a random sample from a multivariate normal distribution
RANDWISHART	generates a random sample from a Wishart distribution

Statistical Functions

BIN function	divides numeric values into a set of disjoint intervals
BRANKS function	computes bivariate ranks
CORR function	computes correlation statistics
CORR2COV function	scales a correlation matrix into a covariance matrix
COUNTMISS function	counts the number of missing values
COUNTN function	counts the number of nonmissing values
COUNTUNIQUE function	counts the number of unique values
COV function	computes the sample variance-covariance matrix
COV2CORR function	scales a covariance matrix into a correlation matrix
CUSUM function	computes cumulative sums
CUPROD function	computes cumulative products
CV function	computes the sample coefficient of variation
DESIGN function	creates a design matrix
DESIGNF function	creates a full-rank design matrix
DISTANCE function	computes pairwise distances between rows of a matrix
GEOMEAN function	computes geometric means
HADAMARD function	creates a Hadamard matrix
HARMEAN function	computes harmonic means
IPF call	performs an iterative proportional fit of a contingency table

KPCATRAIN call	perform kernel principal component analysis on training data
KPCASCORE function	computes the projections along kernel principal components for new data
KURTOSIS function	computes the sample kurtosis
LAV call	performs linear least absolute value regression by solving the L_1 norm minimization problem
LMS call	performs robust least median of squares (LMS) regression
LTS call	performs robust least trimmed squares (LTS) regression
MAD function	finds the univariate (scaled) median absolute deviation
Mahalanobis function	computes Mahalanobis distance
MARG call	evaluates marginal totals in a multiway contingency table
MAXQFORM call	computes the subsets of a matrix system that maximize the quadratic form
MCD call	finds the minimum covariance determinant estimator
MEAN function	computes sample means
MVE call	finds the minimum volume ellipsoid estimator
OPSCAL function	rescales qualitative data to be a least squared fit to qualitative data
QNTL call	computes sample quantiles (percentiles)
QUARTILE function	computes the five-number summary
RANGE function	returns the range of values for a set of matrices.
RANK function	ranks elements of a matrix, breaking ties arbitrarily
RANKTIE function	ranks elements of a matrix
SEQ call	performs discrete sequential tests
SEQSCALE call	performs estimates of scales associated with discrete sequential tests
SEQSHIFT call	performs estimates of means associated with discrete sequential tests
SKEWNESS function	computes the sample skewness
STD function	computes the sample standard deviation
TABULATE call	counts the number of unique values in a vector
SWEEP function	sweeps a matrix
VAR function	computes the sample variance

You can also call functions in Base SAS software such as those documented in the section "Descriptive Statistics Functions and Subroutines" on page 1111.

Time Series Functions

ARMACOV call	computes an autocovariance sequence for an autoregressive moving average (ARMA) model
ARMALIK call	computes the log likelihood and residuals for an ARMA model

ARMASIM function	simulates an ARMA series
CONVEXIT function	computes convexity of a noncontingent cash flow
COVLAG function	computes autocovariance estimates for a vector time series
DIF function	computes the difference between a value and a lagged value
DURATION function	computes modified duration of a noncontingent cash flow
FARMACOV call	computes the autocovariance function for an autoregressive fractionally integrated moving average (ARFIMA) model of the form ARFIMA(p, d, q)
FARMAFIT call	estimates the parameters of an ARFIMA(p, d, q) model
FARMALIK call	computes the log-likelihood function of an ARFIMA(p, d, q) model
FARMASIM call	generates an ARFIMA(p, d, q) process
FDIF call	computes a fractionally differenced process
FORWARD function	computes forward rates
KALCVF call	computes the one-step prediction $z_{t+1\|t}$ and the filtered estimate $z_{t\|t}$, in addition to their covariance matrices. The call uses forward recursions, and you can also use it to obtain k-step estimates.
KALCVS call	uses backward recursions to compute the smoothed estimate $z_{t\|T}$ and its covariance matrix, $P_{t\|T}$, where T is the number of observations in the complete data set
KALDFF call	computes the one-step forecast of state vectors in a state space model (SSM) by using the diffuse Kalman filter. The call estimates the conditional expectation of z_t, and it also estimates the initial random vector, δ, and its covariance matrix.
KALDFS call	computes the smoothed state vector and its mean squares error matrix from the one-step forecast and mean squares error matrix computed by the KALDFF subroutine.
LAG function	computes lagged values
PV function	computes the present value
RATES function	converts interest rates from one base to another
SPOT function	computes spot rates
SPECTROGRAM call	displays a spectrogram of a STFT
TFHILBERT function	compute the Hilbert transform
TFPWV function	compute the pseudo-Wigner-Ville distribution
TFSTFT function	compute the short-time Fourier transform
TFWINDOW function	compute a window function for a short-time Fourier transform
VARMACOV call	computes the theoretical cross-covariance matrices for a stationary vector autoregressive moving average (VARMA(p, q)) model
VARMALIK call	computes the log-likelihood function for a VARMA(p, q) model
VARMASIM call	generates VARMA(p, q) time series
VNORMAL call	(deprecated) generates multivariate normal random series

VTSROOT call	computes the characteristic roots for a VARMA(p, q) model
YIELD function	computes yield-to-maturity of a cash-flow stream

You can also call functions in Base SAS software such as those documented in the section "Financial Functions" on page 1114.

Digital Filtering Functions

CCEPSTRUM call	computes the complex cepstrum
DFCONV function	computes the convolution of two vectors
DFDESIGN call	returns the transfer function of a desired digital filter
DFFILT function	filters a time series by using a digital filter
DFFREQZ function	generates a digital filter's frequency response by using transfer function coefficients
DFFREQZZPK function	generates a digital filter's frequency response by using the transfer function's zeros, poles, and gain
DFMEDFILT function	filters a time series by using a median filter
DFORDER call	computes the minimum order that is required for a desired digital filter
DFSOSFILT function	filters a time series by using a digital filter's second-order section form
DFSOSFREQZ function	generates a digital filter's frequency response that corresponds to its second-order sections form by using transfer function coefficients
DFSOSFREQZZPK function	generates a digital filter's frequency response that corresponds to its second-order sections form by using the transfer function's zeros, poles, and gain
ICCEPSTRUM function	computes the inverse complex cepstrum
RCEPSTRUM function	computes the real cepstrum

Numerical Analysis Functions

BSPLINE function	computes a B-spline basis
FFT function	performs the finite Fourier transform
FFTC function	performs the finite Fourier transform for complex signals
FROOT function	finds zeros of a univariate function by using a numerical root-finding method
IFFT function	computes the inverse finite Fourier transform
IFFTC function	computes the inverse finite Fourier transform for complex signals
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
NORM function	computes a vector or matrix norm
ODE call	performs numerical integration of first-order vector differential equations with initial boundary conditions
ORPOL function	generates orthogonal polynomials on a discrete set of data

ORTVEC call	provides columnwise orthogonalization by the Gram-Schmidt process and step-wise QR decomposition by the Gram-Schmidt process
POLYROOT function	finds zeros of a real polynomial
PRODUCT function	multiplies matrices of polynomials
QUAD call	performs numerical integration of scalar functions in one dimension over infinite, connected semi-infinite, and connected finite intervals
RATIO function	divides matrix polynomials
SPLINE call	fits a cubic spline to data
SPLINEC call	fits a cubic spline to data and returns the spline coefficients
SPLINEV function	evaluates a cubic spline at new data points
TFHILBERT function	computes the Hilbert transform of a discrete time series
TFPWV function	computes the pseudo-Wigner-Ville transform
TFSTFT function	computes the short-time Fourier transform
TFWINDOW function	returns a window of a requested shape for smoothing spectra
TPSPLINE call	computes thin-plate smoothing splines
TPSPLNEV call	evaluates the thin-plate smoothing spline at new data points

Linear Algebra Functions

APPCORT call	computes a complete orthogonal decomposition
COMPORT call	computes a complete orthogonal decomposition by Householder transformations
CVEXHULL function	finds a convex hull of a set of planar points
DET function	computes the determinant of a square matrix
ECHELON function	reduces a matrix to row-echelon normal form
EIGEN call	computes eigenvalues and eigenvectors
EIGVAL function	computes eigenvalues
EIGVEC function	computes eigenvectors
GENEIG call	computes eigenvalues and eigenvectors of a generalized eigenvalue problem
GINV function	computes a generalized inverse
GSORTH call	computes the Gram-Schmidt orthonormalization
HALF function	computes the Cholesky decomposition
HANKEL function	generates a Hankel matrix
HDIR function	performs a horizontal direct product
HERMITE function	reduces a matrix to Hermite normal form
HOMOGEN function	solves homogeneous linear systems
INV function	computes the inverse
INVUPDT function	updates a matrix inverse

ITSOLVER call	solves a sparse general linear system by iteration
LUPDT call	provides updating and downdating for rank-deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses
QR call	computes the QR decomposition of a matrix by Householder transformations
RDODT call	downdates and updates QR and Cholesky decompositions
ROOT function	performs the Cholesky decomposition of a matrix
RUPDT call	updates QR and Cholesky decompositions
RZLIND call	updates QR and Cholesky decompositions
SOLVE function	solves a system of linear equations
SOLVELIN call	solves a sparse symmetric system of linear equations by direct decomposition
SVD call	computes the singular value decomposition
TOEPLITZ function	generates a Toeplitz or block-Toeplitz matrix
TRACE function	sums diagonal elements
TRISOLV function	solves linear systems with triangular matrices
XMULT function	performs extended-precision matrix multiplication

Optimization Subroutines

LCP call	solves the linear complementarity problem
LP call	solves the linear programming problem
LPSOLVE call	solves the linear programming problem
MILPSOLVE call	solves the mixed integer linear programming problem
NLPCG call	performs nonlinear optimization by conjugate gradient method
NLPDD call	performs nonlinear optimization by double-dogleg method
NLPFDD call	approximates derivatives by finite-differences method
NLPFEA call	computes feasible points subject to constraints
NLPHQN call	computes hybrid quasi-Newton least squares
NLPLM call	computes Levenberg-Marquardt least squares
NLPNMS call	performs nonlinear optimization by Nelder-Mead simplex method
NLPNRA call	performs nonlinear optimization by Newton-Raphson method
NLPNRR call	performs nonlinear optimization by Newton-Raphson ridge method
NLPQN call	performs nonlinear optimization by quasi-Newton method
NLPQUA call	performs nonlinear optimization by quadratic method
NLPTR call	performs nonlinear optimization by trust-region method
Nonlinear optimization and related subroutines	lists the nonlinear optimization and related subroutines in SAS/IML software

Set Functions

ELEMENT function	finds elements that are contained in a set
SETDIF function	compares elements of two matrices
UNION function	performs unions of sets
UNIQUE function	sorts and removes duplicates
XSECT function	intersects sets

Control Statements

CALL statement	calls a subroutine or function
CONTINUE statement	goes to the next iteration of a DO loop
DO statement	groups statements as a unit
DO statement, iterative	iteratively executes a DO group
DO UNTIL statement	iteratively executes statements until a condition is satisfied
DO WHILE statement	iteratively executes statements while a condition is satisfied
END statement	ends a DO loop or DO statement
FREE statement	deletes a matrix from memory
GOTO statement	jumps to a new statement
IF-THEN/ELSE statement	conditionally executes statement
LEAVE statement	exits a DO loop
LINK statement	jumps to another statement
MATTRIB statement	associates printing attributes with matrices
PRINT statement	prints matrix values
REMOVE statement	removes matrices from storage
RESET statement	sets processing options
RUN statement	executes statements in a module
SHOW statement	prints system information
SOUND call	produces a tone
VALSET call	performs indirect assignment
VALUE function	retrieves values by indirect reference

Data Set and File Statements

APPEND statement	adds observations to SAS data set
CLOSE statement	closes a SAS data set
CLOSEFILE statement	closes a file
CONTENTS function	returns the variables in a SAS data set

CREATE statement	creates a new SAS data set
DATASETS function	obtains the names of SAS data sets
DELETE call	deletes a SAS data set
DELETE statement	marks observations in a data set for deletion
DO DATA statement	repeats a loop until an end of file occurs
EDIT statement	opens a SAS data set for editing
FILE statement	opens or points to an external file
FIND statement	finds observations
FORCE statement	is an alias for the SAVE statement
INDEX statement	indexes a variable in a SAS data set
INFILE statement	opens a file for input
INPUT statement	inputs data
LIST statement	displays observations of a data set
PUT statement	writes data to an external file
READ statement	reads observations from a data set
RENAME call	renames a SAS data set
REPLACE statement	replaces values in observations and updates observations
SAVE statement	saves data
SETIN statement	makes a data set current for input
SETOUT statement	makes a data set current for output
SORT statement	sorts a SAS data set
SUMMARY statement	computes summary statistics for SAS data sets
PURGE statement	removes observations marked for deletion and renumbers records
USE statement	opens a SAS data set for reading

Defining, Storing, and Loading Modules

FINISH statement	denotes the end of a module
ISSKIPPED function	returns whether an optional argument to a user-defined module was skipped when the modules was called
LOAD statement	loads modules and matrices from library storage
MODULESTACK function	returns the calling sequence for a module
PARENTNAME function	returns the name of the matrix passed into a module
RETURN statement	returns to caller
START statement	defines a module
STORAGE function	lists names of matrices and modules in storage
STORE statement	stores matrices and modules in library storage

Mixed-Type Tables

TABLEADDVAR call	adds columns from a matrix to a table
TABLECREATE function	creates a table from a matrix
TABLECREATEFROMDATASET function	creates a table from a SAS data set
TABLEGETVARDATA function	creates a matrix from columns of a table
TABLEGETVARFORMAT function	returns the formats of the specified columns
TABLEGETVARINDEX function	returns the column indices for specified names
TABLEGETVARINFORMAT function	returns the informats of the specified columns
TABLEGETVARLABEL function	returns the labels of the specified columns
TABLEGETVARNAME function	returns the names of the specified columns
TABLEGETVARTYPE function	returns the types of the specified columns
TABLEISEXISTINGVAR function	indicates whether the specified column names exist
TABLEISVARNUMERIC function	returns a binary vector that indicates whether the specified columns are numeric
TABLERENAMEVAR call	changes the names of columns
TABLESETVARFORMAT call	sets the formats of the specified columns
TABLESETVARINFORMAT call	sets the informats of the specified columns
TABLESETVARLABEL call	sets the labels of the specified columns
TABLESORT call	sorts a table
TABLEWRITETODATASET call	creates a SAS data set from a table

Lists and Data Structures

LISTADDITEM call	adds a new item to the end of a list
LISTCREATE function	creates a new list
LISTDELETEITEM call	deletes an item from a list
LISTDELETENAME call	removes the name of an item
LISTGETALLNAMES function	gets names for all named elements
LISTGETITEM function	gets the value of an item
LISTGETNAME function	gets the names used in a list
LISTGETSUBITEM function	gets the value of an item in a nested sublist
LISTINDEX function	gets the numeric positions of items
LISTINSERTITEM call	inserts an item at a specified position
LISTLEN function	gets the number of items in a list
LISTSETITEM call	sets the value of an existing list item
LISTSETNAME call	sets the name of an item
LISTSETSUBITEM call	sets the value of an item in a nested sublist

Packages and Executing Statements

APPLY function	applies a module to arguments
EXECUTE call	executes statements at run time
EXECUTEFILE call	executes statements at run time
FEVAL function	evaluates a function indirectly
PACKAGE HELP statement	displays help for a package
PACKAGE INFO statement	displays information about a package
PACKAGE INSTALL statement	installs a package
PACKAGE LIBNAME statement	provides access to data in a package
PACKAGE LIST statement	lists the names of installed packages
PACKAGE LOAD statement	loads a package
PACKAGE UNINSTALL statement	uninstalls a package
PUSH call	pushes statements to the beginning of the command input stream
QUEUE call	queues statements at the end of the command input stream
RESUME statement	resumes execution

Statistical Graphics

BAR call	creates a bar chart
BOX call	creates a box plot
GSCALE call	computes round numbers for labeling axes
HEATMAPCONT call	creates a heat map with a continuous color ramp
HEATMAPDISC call	creates a heat map with a discrete color ramp
HISTOGRAM call	creates a histogram
ODSGRAPH call	renders a graph by using ODS statistical graphics
PALETTE function	returns a discrete color palette that is suitable for visualizing categorical data
SCATTER call	creates a scatter plot
SERIES call	creates a series plot
WIDETOLONG call	creates long data from a multivariate time series

Termination Statements

ABORT statement	ends PROC IML
PAUSE statement	interrupts module execution
QUIT statement	exits from PROC IML
STOP statement	stops execution of statements

Wavelet Analysis Functions

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

Genetic Algorithm Functions

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

Calling External Modules

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

Calling SAS Statements or R Functions

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

Operators

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

In addition to the matrix operators described in this section, SAS/IML supports subscript reduction operators that make it easy to compute basic descriptive statistics on rows and columns of a matrix.

Addition Operator: +

matrix1 + *matrix2* ;

matrix + *scalar* ;

matrix + *vector* ;

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

For example, the following statements add two matrices and store the result in the matrix **c**, shown in Figure 25.1:

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

Figure 25.1 Sum of Two Matrices

c
2 3
4 5

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

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

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

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

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

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

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

matrix1 < *matrix2* ;

matrix1 <= *matrix2* ;

matrix1 > *matrix2* ;

matrix1 >= *matrix2* ;

matrix1 = *matrix2* ;

matrix1 ^= *matrix2* ;

Comparison operators compare two matrices element by element and compute a new matrix that contains only zeros and ones. If an element comparison is true, the corresponding element of the new matrix is 1. If the comparison is not true, the corresponding element is 0. Unlike in the SAS DATA step, the SAS/IML language does not accept the English equivalents GT and LT for the greater than and less than operators.

For example, the following statements assign the matrix **c**, shown in Figure 25.2:

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

Figure 25.2 Results of a Matrix Comparison

c		
1	0	1
1	1	1

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

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

- If the comparison is with an $n \times 1$ column vector, each row of the vector is compared to each row of the matrix.
- If the comparison is with a $1 \times p$ row vector, each column of the vector is compared to each column of the matrix.

For example, the following statements assign the matrix **d**, shown in Figure 25.3:

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

Figure 25.3 Results of a Comparison with a Scalar

d
0 1 0
1 0 1

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

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

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

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

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

Concatenation Operator, Horizontal: ||

matrix1 || *matrix2* **;**

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

For example, the following statements produce the matrix **c**, shown in Figure 25.4:

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

Figure 25.4 Result of Horizontal Concatenation

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

For character operands, the element size in the result matrix is the larger of the two operands. For example, the following statements produce a matrix **f** which has elements of size 2, which are shown in Figure 25.5:

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

Figure 25.5 Result of Horizontal Concatenation of Character Matrices

f
A B C GH IJ
D E F KL MN

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

```
x = {};       /* define empty matrix */
y = 1:3;
z = x || y;
print z;
```

Figure 25.6 Concatenation of an Empty Matrix

z
1 2 3

You can also use the concatenation operator to concatenate lists. For example, if **S** and **T** are two lists, then **L = S || T** is the list that contains all items of **S** followed by all items of **T**.

Concatenation Operator, Vertical: //

matrix1 // *matrix2* ;

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

For example, the following statements produce the matrix **c**, shown in Figure 25.7:

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

Figure 25.7 Result of Vertical Concatenation

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

For character matrices, the element size of the result matrix is the larger of the element sizes of the two operands, as shown in Figure 25.8:

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

Figure 25.8 Result of Vertical Concatenation

f	
AB	CD
EF	GH
I	J
K	L
M	N

You can use the vertical concatenation operator when one of the arguments has not been assigned a value. For an example, see the horizontal concatenation operator.

Direct Product Operator: @

matrix1 @ *matrix2* ;

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

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

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

For example, the following statements compute the matrices **c** and **d**, which are shown in Figure 25.9:

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

Figure 25.9 Results of Direct Product Computation

c			
0	2	0	4
0	6	0	8

d			
0	0	2	4
0	0	6	8

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

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

Division Operator: /

matrix1 / *matrix2* ;

matrix / *scalar* ;

matrix / *vector* ;

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

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

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

– If you divide by a $1 \times p$ row vector, each column of the matrix is divided by the corresponding column of the vector.

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

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

The following statements compute the matrices **c** and **d**, shown in Figure 25.10:

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

Figure 25.10 Results of Division

c	
0.2	0.3333333
0.4285714	0.5

d	
0.25	0.5
0.75	1

Element Maximum Operator: <>

matrix1 <> *matrix2* ;

matrix <> *scalar* ;

matrix <> *vector* ;

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

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

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

The element maximum operator can take as operands two character matrices or a character matrix and a character string. Character values are compared in ASCII order. In ASCII order, numerals precede uppercase letters, which precede lowercase letters. If the element lengths of character operands are different, the shorter elements are padded on the right with blanks. The element length of the result is the longer of the two operand element lengths.

For example, the following statements compute the matrix `c`, shown in Figure 25.11:

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

Figure 25.11 Maximum Elements

c		
2	9	6
20	11	40

Element Minimum Operator: ><

matrix1 >< *matrix2* ;

matrix1 >< *scalar* ;

matrix1 >< *vector* ;

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

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

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

The element minimum operator can take as operands two character matrices or a character matrix and a character string. Character values are compared in ASCII order. In ASCII order, numerals precede uppercase letters, which precede lowercase letters. If the element lengths of character operands are different, the shorter elements are padded on the right with blanks. The element length of the result is the longer of the two operand element lengths.

For example, the following statements compute the matrix **c**, shown in Figure 25.11:

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

Figure 25.12 Minimum Elements

c		
1	4	2
10	10	12

Index Creation Operator: :

value1 : value2 **;**

The index creation operator (:) creates a row vector with a first element that is *value1*. The second element is *value1*+1, and so on, until the last element which is less than or equal to *value2*.

For example, the following statement creates the vector **s** which contains consecutive integers, shown in Figure 25.13:

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

Figure 25.13 Increasing Sequence

s			
7	8	9	10

If *value1* is greater than *value2*, a reverse-order index is created. For example, the following statement creates the vector **r** which contains a decreasing sequence of integers, shown in Figure 25.14:

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

Figure 25.14 Decreasing Sequence

r				
10	9	8	7	6

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

The index creation operator also works on character arguments with a numeric suffix. For example, the following statements create a sequence of values that begin with the prefix "var", shown in Figure 25.15:

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

Figure 25.15 Sequence of Character Values

varList				
var1	var2	var3	var4	var5

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

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

Figure 25.16 Sequence of Variable Names

x			
X1	X2	X3	X4
1	2	3	4
5	6	7	8
7	6	5	4

Lastly, you can use the index operator to create a sequence of English letters, in either increasing or descending order, as follows:

```
a = "a":"h";
b = "P":"L";
print a, b;
```

Figure 25.17 Sequence of Letters

a							
a	b	c	d	e	f	g	h

b				
P	O	N	M	L

List Creation Operator: [ ]

[*item1, item2, ..., item_n* **] ;**

[*#name1=item1, #name2=item2, ..., #name_n=item_n* **] ;**

The list creation operator creates a list. The items, which must be separated by commas, can be any valid SAS/IML data type, such as matrices, tables, and other lists. You can also use expressions or function calls inside the brackets. Some or all of the items can be empty.

Optionally, you can assign names to the items in the list by specifying a name-value pair. The name must be prefixed by the **#** symbol. The name must be a character literal, either quoted or unquoted. The following statements are valid ways to specify lists:

```
numMat = 1:4;
charMat = {"X" "Y" "Z"};
list = [100, "Q"];                    /* simple list */
L1 = [1, "a", numMat, charMat, list]; /* multiple types in list  */
L2 = [1, , 1:3, ];                    /* 2nd and 4th items empty */
L3 = [#N=numMat, #C=charMat];         /* named items */
L4 = [#N=numMat, #E= , charMat];      /* some named items */
L5 = [#"Numeric Matrix"=numMat, #"Char Matrix"=charMat]; /* quoted names */
```

List Item Operator: $

*list***$***index* **;**

*list***$***index1***$***index2***$**... **;**

If **L** is a list, then **L$1** is the first item in the list, **L$2** is the second item, and so forth. When used on the right side of an assignment operator, the list item operator (**$**) returns an item from a list. When used on the left side of an assignment operator, the list item operator assigns an item in a list.

The following statements demonstrate the list item operator:

```
numMat = 1:4;
charMat = {"X" "Y" "Z"};
list = [#N=100, #C="Q"];                /* list */
L = [#N=numMat, #C=charMat, #A=list];  /* multiple types in list  */
M = L$1;                                /* get first item */
L2 = L$"A";                             /* get item named "A" */
if L$"A"$2 = "Q" then print "Q found";
else print "Q not found";
```

The list item operator always returns one item, although the *index* argument can take several forms. Usually the *index* argument is a scalar value, similar to the syntax for the ListGetItem function. However, if a list has nested sublists, the *index* argument can be a vector or a list that specifies the items, similar to the syntax for the ListGetSubItem function. The following list summarizes the valid values for the item operator:

- An integer. For example, **L$3** is the third item in the list **L**.
- A literal character string. For example, **L$"A"** is the item in the list that has the name “A”.

- A vector of integers. For example, suppose that **L** is a list and the third item is also a list. Then the expression **L${3 1}** is the first item of the third item of **L**.
- A vector of character strings. For example, suppose that **L** is a list and the item named "A" is also a named list. Then the expression **L${"A" "C"}** is the item named "C" in the sublist **L$"A"**.
- A list that contains scalar items. The list must be enclosed in parentheses. For example, if **L** is a list that contains an item named "A", then the expression **L$(["A", 2])** is the third item in the sublist **L$"A"**.
- An expression enclosed in parentheses that resolves to one of the previous situations. For example, **L$(1+2)** or **L$(3 || 2)**

List Sublist Operator: [ ]

list[*index*] ;

If **L** is a list, then **L[1]** is the list that contains the first item in **L**, **L[{1 2}]** is the list that contains the first and second items, and so on. When used on the right side of an assignment operator, the list sublist operator returns a sublist. When used on the left side of an assignment operator, the list sublist operator assigns a sublist.

The following statements demonstrate the list sublist operator:

```
numMat = 1:4;
charMat = {"X" "Y" "Z"};
list = [100, "Q"];                      /* list */
L = [#N=numMat, #C=charMat, #A=list];   /* multiple types in list  */
L2 = L[1:2];                            /* sublist contains 1st and 2nd items */
L3 = L[{3 1}];                          /* sublist contains 3rd and 1st items */
L4 = L[{"A" "N"}];                      /* sublist contains two named items */
```

The *index* argument can be a numeric vector, a character vector, or a list that contains scalar items. The sublist operator always returns a list. If you want to copy an item, use the item operator (**$**). For example, the syntax **M=L$1** copies the first item of the list **L**, whereas the **L[1]** is a list that contains the first element of **L**.

Logical Operators: &, |, ^

matrix1 & matrix2 ;

matrix & scalar ;

matrix & vector ;

matrix1 | matrix2 ;

matrix | scalar ;

matrix | vector ;

^matrix ;

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

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

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

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

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

The following statements illustrate the use of these logical operators. The results are shown in Figure 25.18:

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

Figure 25.18 Results of Logical Comparisons

u
1 1 0 1 1 0

v
0 1 0 0 0 0

w
1 0 1 0 1 1

Multiplication Operator, Elementwise:

matrix1 # matrix2 ;

matrix # scalar ;

matrix # vector ;

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

For example, the following statements compute the matrix **ab**, shown in Figure 25.19:

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

Figure 25.19 Results of Elementwise Multiplication

ab	
4	16
0	20

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

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

For example, a 2×3 matrix can be multiplied on either side by a 2×3, 1×3, 2×1, or 1×1 matrix. The following statements multiply the 2×2 matrix **a** by a column vector and a row vector. The results are shown in Figure 25.20.

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

Figure 25.20 Elementwise Multiplication with Vectors

ac	
10	20
300	400

ar	
10	200
30	400

Elementwise multiplication is mathematically equivalent to multiplying by a diagonal matrix. However, the elementwise operation is more efficient, as shown by the following statements:

```
A = j(5,5,1);
v = 2:6;            /* 1x5 row vector */
D = diag(v);        /* 5x5 diagonal matrix */
/* multiply columns by constants */
B = A*D;            /* less efficient */
B = v # A;          /* more efficient */
/* multiply rows by constants */
C = D*A;            /* less efficient */
C = A # v`;         /* more efficient */
```

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

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

The # symbol also has a special meaning that is unrelated to multiplication. When creating lists, you can create named items by using name-value pairs. Names must be prefixed with the # symbol. For example, the syntax `L = [#A=2, #"B"=3, #"A * B"=2*3]` creates a list that contains three named items.

Multiplication Operator, Matrix: *

matrix1 * *matrix2* ;

The matrix multiplication operator (*) computes a new matrix by performing matrix multiplication. The first matrix must have the same number of columns as the second matrix has rows. The new matrix has the same number of rows as the first matrix and the same number of columns as the second matrix. That is, if **A** is an $n \times p$ matrix and **B** is a $p \times m$ matrix, then the product $\mathbf{A} * \mathbf{B}$ is an $n \times m$ matrix. The (i, j) element of the product is the sum $\sum_{k=1}^{p} A_{ik} B_{kj}$.

The matrix multiplication operator does not support missing values.

The following statements multiply matrices. The results are shown in Figure 25.21.

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

Figure 25.21 Result of Matrix Multiplication

c
7 10

d
5
11

Power Operator, Elementwise:

matrix1 ## matrix2 ;

matrix ## scalar ;

matrix ## vector ;

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

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

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

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

For example, the following statements raise each element of a matrix to a power, as shown in Figure 25.22:

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

Figure 25.22 Result of Raising Each Element to a Power

b
1 8 27

c
1 1.4142136 1.7320508

Power Operator, Matrix: **

*matrix ** scalar* ;

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

For example, the following statements compute a matrix that is the result of multiplying a matrix by itself. The result is shown in Figure 25.23.

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

Figure 25.23 Result of Raising a Matrix to a Power

c	
3	4
2	3

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

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

Figure 25.24 Matrix Inversion by Using the Power Operator

inv	
-1	2
1	-1

ident	
1	0
0	1

The matrix power operator does not support missing values.

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

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

Figure 25.25 Matrix Powers by Using Eigenvalues

a20	
165580141	102334155
102334155	63245986

Sign Reversal Operator: –

–matrix ;

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

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

The following statements reverse the signs of each element of a matrix, as shown in Figure 25.26:

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

Figure 25.26 The Result of a Sign Reversal Operator

b		
1	-7	-6
-2	0	8

Subscripts: [ ]

matrix[rows, columns] ;

matrix[elements] ;

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

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

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

Figure 25.27 Submatrices Formed by Specifying Indices

y
6

z		
4	5	6

Figure 25.27 *continued*

w
4
6

The output is shown in Figure 25.27. The matrix **y** contains the element of **x** from the second row and the third column. The matrix **z** contains the entire second row of **x**. The matrix **w** contains the fourth and sixth elements of **x**. Because SAS/IML software store matrices in row-major order, **w** contains the first and third elements from the second row of **x**.

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

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

Figure 25.28 Submatrices Formed by Specifying Column Names

m
4 6

A subscripted matrix can appear on the left side of the equal sign. The dimensions of the target submatrix must conform to the dimensions of the source matrix, as shown in the following statements:

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

Figure 25.29 Result of Assigning Submatrices of an Existing Matrix

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

See the section "Using Matrix Expressions" on page 46 for further information about matrix subscripts.

Subscript Reduction: [ ]

matrix[rowOp, colOp] ;

matrix[Op] ;

SAS/IML supports subscript reduction operators, which make it easy to compute basic descriptive statistics on rows and columns of a matrix.

A reduction operator is a statistical operation (for example, a sum or a mean) that returns a matrix of a smaller dimension. Reduction operators are often encountered in frequency tables; the marginal frequencies represent the sum of the frequencies across rows or down columns.

You can use reduction operators in place of values for subscripts to get reductions across all rows or columns. Table 25.1 lists operators for subscript reduction.

Table 25.1 Subscript Reduction Operators

Operator	Description
+	Addition
#	Multiplication
<>	Maximum
><	Minimum
<:>	Index of maximum
>:<	Index of minimum
:	Mean
##	Sum of squares

For example, the following statements use reduction operators to compute statistics for each column in a matrix. The operators are placed in the "row index" location. Because the column index is blank, the results are computed for all columns. Consequently, the results are row vectors, as shown in Figure 25.30.

```
X = { 1   2  3,
     -1   1  2,
      1  -1 -2,
      3   6  9};
/* create row vectors that contain statistics for each column */
sum  = X[ +, ];
mean = X[ :, ];
min  = X[><, ];
max  = X[<>, ];
ssq  = X[##, ];

cName = "Col1":"Col3";
print sum[c=cName], mean[c=cName],
      min[c=cName], max[c=cName], ssq[c=cName];
```

Figure 25.30 Statistics for Each Column

sum		
Col1	Col2	Col3
4	8	12

mean		
Col1	Col2	Col3
1	2	3

min		
Col1	Col2	Col3
-1	-1	-2

max		
Col1	Col2	Col3
3	6	9

ssq		
Col1	Col2	Col3
12	42	98

In a similar way, you can compute statistics for each row in a matrix by leaving the row index blank and placing an operator in the "column index" location. The following statements compute statistics for each row:

```
/* create column vectors that contain statistics for each row */
sum  = X[ , +];
mean = X[ , :];
min  = X[ , ><];
max  = X[ , <>];
ssq  = X[ , ##];

rName = T("Row1":"Row4");
print rName sum mean min max ssq;
```

Subtraction Operator: –

matrix1 – *matrix2* ;

matrix – *scalar* ;

matrix – *vector* ;

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

In addition to subtracting conformable matrices, you can also use the subtraction operator to subtract a scalar from a matrix or subtract a vector from a matrix.

- When either argument is a scalar, the subtraction is performed between the scalar and each element of the matrix argument. For example, when you use the *matrix* – *scalar* form, the scalar value is

subtracted from each element of the matrix.

- When you use the *matrix* – *vector* form, the vector is subtracted from each row or column of the $n \times p$ matrix.
  - If you subtract an $n \times 1$ column vector, each row of the vector is subtracted from each row of the matrix.
  - If you subtract a $1 \times p$ row vector, each column of the vector is subtracted from each column of the matrix.

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

For example, the following statements subtract two matrices and store the result in the matrix **c**, shown in Figure 25.31:

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

Figure 25.31 Difference of Two Matrices

c
0 1
2 3

Transpose Operator: `

matrix `;

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

For example, the following statements transpose the matrix **a**, shown in Figure 25.32:

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

Figure 25.32 Transpose of a Matrix

b
1 3 5
2 4 6

You can also transpose a matrix with the T function.

Statements, Functions, and Subroutines

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

ABORT Statement

ABORT < *error-message* > **;**

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

If you specify the optional *error-message*, the message is written to the SAS Log.

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

```
proc iml;
do i = 1 to 10;
   if i>2 then
      abort;
   print i;
end;
quit;
```

Figure 25.33 Result of Aborting a Computation

i
1

i
2

ABS Function

ABS(*matrix***);**

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

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

Figure 25.34 Absolute Values

a
2 1 0 1 2

ALL Function

ALL(*matrix***);**

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

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

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

Figure 25.35 Result of Comparing All Elements

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

In the preceding statements, the comparison operation **a>b** creates a matrix of zeros and ones. The ALL function returns a value of 1 because every element of **a** is greater than the corresponding element of **b**.

The ALL function is implicitly applied to the evaluation of all conditional expressions, so in fact the previous IF-THEN statement is equivalent to the following:

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

ALLCOMB Function

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

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

The ALLCOMB function generates all combinations of *k* elements taken from a set of *n* numerical indices. The combinations are produced in the same order and using the same algorithm (Nijenhuis and Wilf 1978) as

the ALLCOMBI function in Base SAS software. In particular, the function returns indices in the range 1–*n*, and each combination is in sorted order.

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

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

Figure 25.36 All Pairwise Combinations of Four Items

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

The second argument can be a scalar or a vector. If it is a vector, it must contain a valid combination of the set $\{1, 2, \ldots, n\}$. (To be valid, the *comb* elements must be in increasing order.) The number of elements in the vector determines the value of *k*. For example, the following statements generate all combinations of length two from a set with four elements, beginning with the third combination that is shown in Figure 25.36:

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

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

The optional third argument, *idx*, controls the number of rows in the output of the function. If you specify *idx*, then the sequence is initialized with the *comb* argument and the first row of the output is the combination that occurs *after* the *comb* argument. For example, the following statements generate five pairwise combinations, beginning *after* the third combination shown in Figure 25.36:

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

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

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

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

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

ALLPERM Function

ALLPERM(*n*);

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

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

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

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

Figure 25.37 All Permutations of Three Items

p		
1	2	3
1	3	2
3	1	2
3	2	1
2	3	1
2	1	3

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

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

Figure 25.38 All Permutations of a Character Vector

a		
C	B	A
C	A	B
A	C	B
A	B	C
B	A	C
B	C	A

The syntax ALLPERM(n) is valid when $n \leq 11$ or when n is a set that contains 11 or fewer elements.

The optional second argument, *idx*, can be used to control the number of rows in the output of the function.

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

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

The syntax ALLPERM(*set*, *idx*) is valid when the first argument contains 18 or fewer elements.

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

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

ANY Function

ANY(*matrix***);**

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

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

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

Figure 25.39 Result of Comparing Elements

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

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

APPCORT Call

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

If $\mathbf{A}$ is rank-deficient, then the least squares problem $\min_x \|\mathbf{Ax} - \mathbf{b}\|_2^2$ has infinitely many solutions (Golub and Van Loan 1989, p. 241). However, there is a unique solution which has the smallest Euclidean norm. The

APPCORT subroutine computes the minimum Euclidean-norm solution of the (rank-deficient) least squares problem by applying a complete orthogonal decomposition by Householder transformations to the vector **b**.

The input arguments to the APPCORT subroutine are as follows:

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

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

b is a $m \times p$ matrix, **B**.

sing is an optional scalar that specifies a singularity criterion.

The APPCORT subroutine returns the following values:

prqb is an $n \times p$ matrix product

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

which is the minimum Euclidean-norm solution of the rank-deficient least squares problem $\|\mathbf{Ax} - \mathbf{b}|_2^2$.

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

See the section "COMPORT Call" on page 566 for information about complete orthogonal decomposition.

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

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

Figure 25.40 Solution to a Rank-Deficient Least Squares Problem

x
0.3
0.6

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

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

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

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

Figure 25.41 Generalized Inverse

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

msg
Pseudoinverse conditions satisfied

APPEND Statement

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

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

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

The arguments to the APPEND statement are as follows:

operand specifies a set of variables. You can specify variables by using any of the methods described in the section "Select Variables with the VAR Clause" on page 105.

matrices is one or more matrix names that contain data to append. Each column of each matrix becomes a variable in the data set.

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

You can use the APPEND statement to add data to the end of the current output data set. The appended observations are from either the variables specified in the VAR clause or variables created from the columns of *matrices*. You cannot use the FROM clause and the VAR clause in the same statement.

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

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

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

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

The values in the Temp1 data set are shown in Figure 25.42. Notice that the 2×2 matrix **y** is written to the data set in row-major order.

Figure 25.42 Data Set Created from Matrices

X	Y
1	4
2	3
3	2
4	1

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

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

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

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

The values in the Temp2 data set are shown in Figure 25.43. The data set contains four observations because that is the number of elements in the matrix with the greatest number of elements. Elements are appended in row-major order. Notice that the variable z contains a missing value at the end because the variable was created from a SAS/IML matrix that contained fewer than four elements.

Figure 25.43 Data Set Created from All Matrices

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

As shown in the previous example, the default variables for the APPEND statement are all matrices that match variables in the current data set with respect to name and type.

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

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

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

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

The values in the Temp3 data set are shown in Figure 25.44. The matrix **cov** contains the data that are saved to the Temp3 data set. The character vector **VarName** contains the names of the variables for the Temp3 data set. (If you use the FROM clause in the CREATE statement, but do not specify the COLNAME= option, then the variables are named COL1, COL2, and so on.) The ROWNAME= option enables you to specify a single character variable when you are creating a data set from a numerical matrix. This is useful for specifying variable names in a correlation or covariance matrix, but can also be used more generally to specify a row label for each observation.

Figure 25.44 Data Set That Contains Row Labels

VarName	x	y
x	0.91667	-4.1667
y	-4.16667	19.6667

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

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

You can specify more than one FROM matrix to append data from multiple matrices, as shown in the following example:

```
a = {1 2 3,
     4 5 6 };
b = {"A" "B",
     "C" "D"};
varnames= {"a1" "a2" "a3" "b1" "b2"};
create Temp4 from a b [colname=varnames];
append from a b;
close Temp4;
```

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

APPLY Function

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

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

The arguments to the APPLY statement are as follows:

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

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

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

If the function module takes n scalar arguments, *argument1* through *argumentn* should be passed to APPLY where $1 \leq n \leq 14$. The APPLY function calls the module one time for each element in its input arguments. The result has the same dimension as the input arguments, and each element of the result corresponds to the module applied to the corresponding elements of the argument matrices. The APPLY function accepts either numeric or character arguments. For example, the following statements define module ABC and then call the APPLY function, with matrix **a** as an argument:

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

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

The result is shown in Figure 25.45.

Figure 25.45 Result of a Module Applied to Each Argument in a Matrix

s		
106	107	108
109	110	111

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

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

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

Notice that the third element of the **a** and **b** arguments are exchanged, as shown in Figure 25.46.

Figure 25.46 Result of a Module Applied to Each Argument in a Matrix

a		
-1	0	2

b		
-2	0	1

r		
1	0	1

The APPLY function is provided as a convenience, but it is usually unnecessary. It is usually more efficient to write your functions to take vector, rather than scalar, arguments.

ARMACOV Call

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

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

phi	refers to a $1 \times (p+1)$ matrix that contains the autoregressive parameters. The first element is assumed to have the value 1.
theta	refers to a $1 \times (q+1)$ matrix that contains the moving average parameters. The first element is assumed to have the value 1.
num	refers to a scalar that contains n, the number of autocovariances to be computed, which must be a positive number.

The ARMACOV subroutine returns the following values:

auto	specifies a variable to contain the returned $1 \times n$ matrix that contains the autocovariances of the specified ARMA model, assuming unit variance for the innovation sequence.
cross	specifies a variable to contain the returned $1 \times (q+1)$ matrix that contains the covariances of the moving-average term with lagged values of the process.
convol	specifies a variable to contain the returned $1 \times (q+1)$ matrix that contains the autocovariance sequence of the moving-average term.

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

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

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

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

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

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

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

for $k = 1, 2, \ldots, q+1$, where ψ_i is the ith impulse response value. The ψ_i sequence, if desired, can be computed with the RATIO function. It can be shown that ψ_k is the same as $E(Y_{t-k}\epsilon_t^2)/\sigma$, which is used by

Box and Jenkins (1976) in their formulation of the autocovariances. The kth autocovariance, denoted γ_k and returned as the $k+1$ element of the *auto* argument ($k = 0, 1, \ldots, n-1$), is defined implicitly for $k > 0$ by

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

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

Consider the model

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

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

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

The result is show in Figure 25.47.

Figure 25.47 Result of the ARMACOV Subroutine

auto				
3.2533333	2.4266667	1.2133333	0.6066667	0.3033333

cross		convol	
2.04	0.8	1.64	0.8

ARMALIK Call

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

The ARMALIK subroutine computes the log likelihood and residuals for an autoregressive moving average (ARMA) model. The input arguments to the ARMALIK subroutine are as follows:

x is an $n \times 1$ or $1 \times n$ matrix that contains values of the time series (assuming mean zero).

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

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

The ARMALIK subroutine returns the following values:

lnl specifies a 3×1 matrix that contains the log likelihood concentrated with respect to the innovation variance; the estimate of the innovation variance (the unconditional sum of squares divided by n); and the log of the determinant of the variance matrix, which is standardized to unit variance for the innovations.

resid specifies an $n \times 1$ matrix that contains the standardized residuals. These values are uncorrelated with a constant variance if the specified ARMA model is the correct one.

std specifies an $n \times 1$ matrix that contains the scale factors used to standardize the residuals. The actual residuals from the one-step-ahead predictions that use the past values can be computed as **`std # resid`**.

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

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

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

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

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

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

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

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

Figure 25.48 Results from an ARMALIK Call

lnl	resid	std
-0.822608	0.4057513	2.4645637
0.8721154	0.9198158	1.2330147
2.3293833	0.8417343	1.0419028
	1.0854175	1.0098042
	1.2096421	1.0024125

ARMASIM Function

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

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

The arguments to the ARMASIM function are as follows:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, consider the following model:

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

To generate a time series of length 10 from this model, use the following statements to produce the result shown in Figure 25.49:

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

Figure 25.49 Simulated Time Series

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

BAR Call

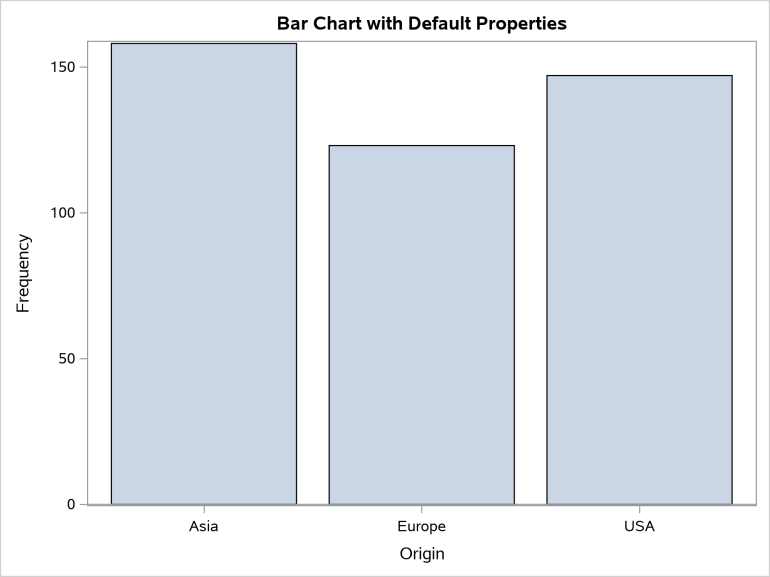
CALL BAR(*x***) TYPE="VBar" | "HBar"**
GROUP=*GroupVector*
GROUPOPT=*GroupOption*
FREQ=*FreqVector*
ORDER="DATA" | "UNFORMATTED"
GRID={"X" <,"Y" >}
LABEL={*XLabel* **<,** *YLabel***>}**
XVALUES=*xValues*
YVALUES=*yValues*
PROCOPT=*ProcOption*
OTHER=*Stmts* **;**

The BAR subroutine displays a bar chart by calling the SGPLOT procedure. The required argument *x* is a vector that contains character or (discrete) numeric data to plot. The remaining arguments are optional and can be specified in any order. The BAR subroutine is not a comprehensive interface to the SGPLOT procedure. It is intended for creating simple bar charts for exploratory data analysis. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

A simple example follows:

```
use Sashelp.Cars;
read all var {origin};
close Sashelp.Cars;

title "Bar Chart with Default Properties";
call Bar(origin);
```

Figure 25.50 A Bar Chart



Specify the *x* vector inside parentheses and specify all optional options outside the parentheses. Use the global TITLE and FOOTNOTE statements to specify titles and footnotes. Each option corresponds to a statement or option in the SGPLOT procedure.

Valid values for the TYPE= option are "VBar" and "HBar." The "VBar" value creates a vertical bar chart and corresponds to the VBAR statement in PROC SGPLOT. The "HBar" value creates a horizontal bar chart and corresponds to the HBAR statement.

The following options correspond to options in the VBAR and HBAR statements in the SGPLOT procedure:

GROUP= specifies a vector of values that determine groups in the plot. You can use a numeric or character vector. This option corresponds to the GROUP= option in the VBAR and HBAR statements.

GROUPOPT= specifies a character vector of values that determine how groups are displayed. This option is ignored if the GROUP= option is not specified. You can specify the following values:

- "Stack" or "Cluster" specifies how to display grouped bars. These values correspond to the GROUPDISPLAY= option in the VBAR and HBAR statements in PROC SGPLOT. The default value is "Stack."
- "Ascending," "Descending," or "Data" specifies how to display grouped bars. These values corresponds to the GROUPORDER= option in the VBAR and HBAR statements in PROC SGPLOT. The default value is "Ascending."

For example, a valid call is `GROUPOPT={"Cluster" "Data"};`

FREQ= specifies a vector of numerical values that are used as frequencies for each corresponding value of the *x* variable. This option corresponds to the FREQ= option in the VBAR and HBAR statements in PROC SGPLOT.

Some options are common to all of the ODS graphics routines. The following common options specify options in the XAXIS and YAXIS statements in the SGPLOT procedure:

ORDER= specifies the order in which discrete tick values are to be placed on the categorical axis. Valid options are "DATA" and "UNFORMATTED." This option corresponds to the DISCRETEORDER= option in the XAXIS and YAXIS statements.

GRID= specifies whether to display grid lines for the X or Y axis. This option corresponds to the GRID option in the XAXIS and YAXIS statements. Valid values follow:

- **GRID={X}** displays grid lines for the X axis.
- **GRID={Y}** displays grid lines for the Y axis.
- **GRID={X, Y}** displays grid lines for both axes.

LABEL= specifies axis labels for the X or Y axis. If the argument is a scalar, the value of the argument is used for the X axis label. If the argument has two elements, the first is used for the X axis label and the second for the Y axis label. If this option is not specified, the labels "X" and "Y" are used for labels.

XVALUES= specifies a vector of values for ticks for the X axis.

YVALUES= specifies a vector of values for ticks for the Y axis.

In addition, the following common options specify additional options and statements in the SGPLOT procedure:

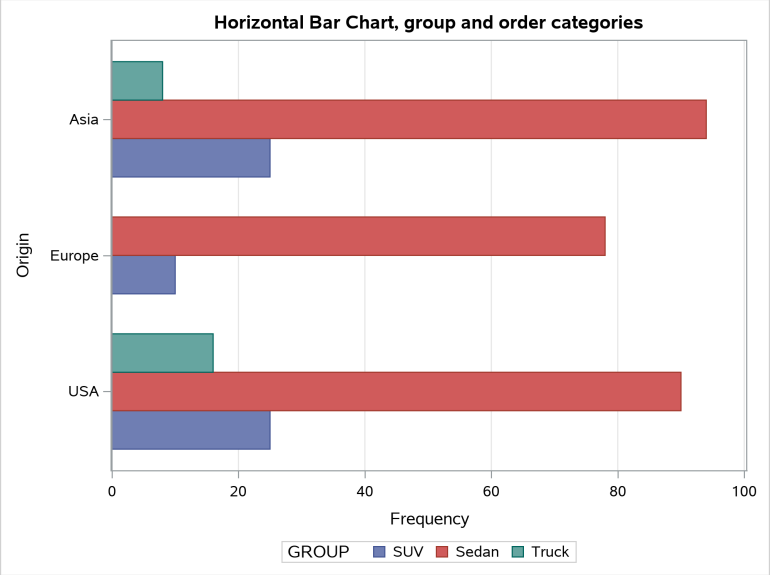
PROCOPT= specifies a character matrix or string literal. The value is used verbatim to specify options in the PROC SGPLOT statement.

OTHER= specifies a character matrix or string literal. You can use this option to specify one or more complete statements in the SGPLOT procedure. For example, you can specify multiple REFLINE statements and an INSET statement.

The following example shows how to create a bar chart that uses the GROUP=, GROUPOPT=, GRID=, and LABEL= options:

```
use Sashelp.Cars where(type ? {"SUV" "Truck" "Sedan"});
read all var {origin type};
close Sashelp.Cars;

title "Horizontal Bar Chart, group and order categories";
/* 1. Use the GROUP= option to assign a group to each observation
 * 2. Use the GROUPOPT= option to specify the grouping options
 * 3. Use the GRID= and LABEL= options to improve the appearance
 */
call Bar(origin) type="HBar" group=type groupopt="Cluster"
         grid="X" label="Origin";
```

Figure 25.51 Clustered Bar Chart



Notice that the TYPE="HBar" option results in the bars being drawn horizontally, as shown in Figure 25.51. Also, because the categories are displayed on the vertical axis, the LABEL= option changes the label on the vertical axis.

The next example shows how to create a bar chart from tabulated data. The frequencies for each category are precomputed. The FREQ= option specifies the vector of frequencies. The ORDER= option requests that the categories be displayed in the same order as they appear in the data.

```
y    = { 3  2  1 .};
freq = {30 20 10 5};
title "Freq and Missing Category";
call Bar(y) freq=freq order="Data";
```

Figure 25.52 Bar Chart from Summarized Data



BIN Function

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

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

x specifies a numerical vector or matrix.

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

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

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

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

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

Figure 25.53 Bins for Each Observation

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

You can use the special missing values `.M` and `.I` to specify unbounded intervals. A missing value of `.M` in the first element is interpreted as $-\infty$, and a missing value of `.I` in the last element is interpreted as $+\infty$.

For example, the following statements are all valid specifications of the *cutpoints* argument:

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

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

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

Figure 25.54 Bins That Are Associated with Each Value

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

Notice in Figure 25.54 that the value 3 is placed into the third interval because the last interval is closed on the right.

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

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

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

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

Figure 25.55 Bins Counts for Evenly Spaced Intervals

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

BLANKSTR Function

BLANKSTR(*n*);

The BLANKSTR function returns a blank character string of a specified length. You can use the BLANKSTR function in conjunction with the J function to allocate character arrays, as follows:

```
/* combine colors and objects */
color  = {"Red" "Green" "Blue"};            /* nleng(color) =5 */
object = {"Balloon" "Leaf" "Marble"};       /* nleng(object)=7 */

/* compute maximum length of a color/object combination */
len = nleng(color) + nleng(object) + 1;
items = j(3, 3, BlankStr(len));             /* allocate char vector */
do i = 1 to ncol(color);
   do j = 1 to ncol(object);
      items[i,j] = color[i] + " " + object[j]; /* concatenate strings */
   end;
end;
print items;
```

Figure 25.56 Filling an Allocated Character Matrix

items					
Red	Balloon	Red	Leaf	Red	Marble
Green	Balloon	Green	Leaf	Green	Marble
Blue	Balloon	Blue	Leaf	Blue	Marble

BLOCK Function

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

The BLOCK function forms a block-diagonal matrix. The blocks are defined by the arguments to the function. Up to 15 matrices can be specified. Empty matrices are supported, but have no effect. The matrices are combined diagonally to form a new matrix.

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

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

The following statements produce a block-diagonal matrix composed of three blocks, shown in Figure 25.57:

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

```
d = block(a, b, c);
print d;
```

Figure 25.57 Block Matrix

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

BOX Call

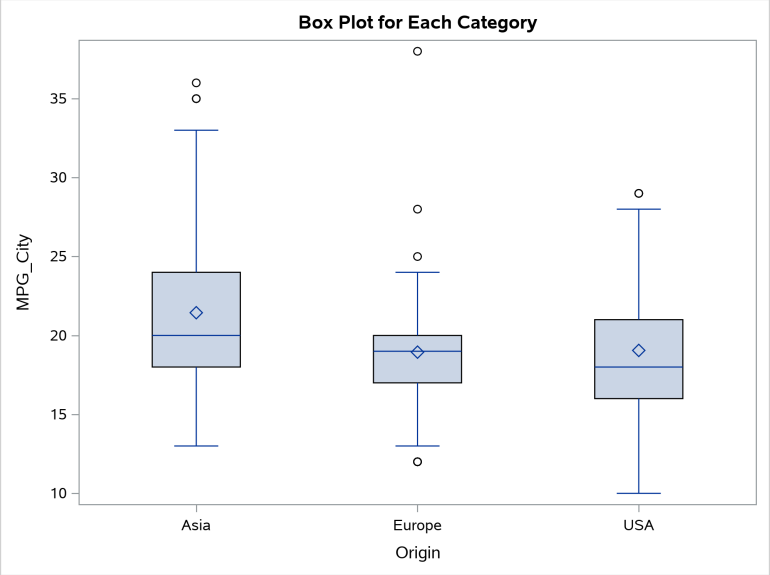
CALL BOX(*x***) TYPE="VBox" | "HBox"**
CATEGORY=*CategoryVector*
GROUP=*GroupVector*
GROUPOPT=*GroupOption*
DATALABEL=*DataLabelVector*
OPTION=*BoxOption*
ORDER="DATA" | "UNFORMATTED"
GRID={"X" <,"Y">}
LABEL={*XLabel* **<,** *YLabel***>}**
XVALUES=*xValues*
YVALUES=*yValues*
PROCOPT=*ProcOption*
OTHER=*Stmts* **;**

The BOX subroutine displays a bar chart by calling the SGPLOT procedure. The required argument *x* is a vector that contains character or (discrete) numeric data to plot. The remaining arguments are optional and can be specified in any order. The BOX subroutine is not a comprehensive interface to the SGPLOT procedure. It is intended for creating simple bar charts for exploratory data analysis. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

A simple example follows:

```
use Sashelp.Cars where(type ? {"SUV" "Truck" "Sedan"});
read all var {MPG_City Origin Type Make Model};
close Sashelp.Cars;

title "Box Plot for Each Category";
call Box(MPG_City) Category=Origin;
```

Figure 25.58 A Box Plot



Specify the *x* vector inside parentheses and specify all optional arguments outside the parentheses. Use the global TITLE and FOOTNOTE statements to specify titles and footnotes. Each option corresponds to a statement or option in the SGPLOT procedure.

Valid values for the TYPE= option are "VBox" and "HBox." The "VBox" value creates a vertical box plot and corresponds to the VBOX statement in PROC SGPLOT. The "HBox" value creates a horizontal box plot and corresponds to the HBOX statement.

The following options correspond to options in the VBOX and HBOX statements in the SGPLOT procedure:

CATEGORY= specifies a vector of values that define a category variable for the plot. A box plot is created for each distinct value of the category variable.

GROUP= specifies a vector of values that determine groups in the plot. You can use a numeric or character vector. This option corresponds to the GROUP= option in the VBOX and HBOX statements.

GROUPOPT= specifies a character vector of values that determine how groups are displayed. This option is ignored if the GROUP= option is not specified. You can specify one or both of the following values:

- "Cluster" or "Overlay" specifies how to display grouped boxes. This option corresponds to the GROUPDISPLAY= option in the VBOX and HBOX statements in PROC SGPLOT. The default value is "Cluster."
- "Ascending," "Descending," or "Data" specifies how to display grouped boxes. This option corresponds to the GROUPORDER= option in the VBOX and HBOX statements in PROC SGPLOT. The default value is "Ascending."

For example, a valid call is `GROUPOPT={"Cluster" "Data"};`

DATALABEL= specifies a vector of values that are used to label outliers.

OPTION= specifies a character matrix or string literal. This option is used verbatim to specify options in the HBOX or VBOX statement.

The BOX subroutine also supports the following options. The BAR subroutine documents these options and gives an example of their usage.

ORDER= specifies the order in which discrete tick values are to be placed on the categorical axis.

GRID= specifies whether to display grid lines for the X or Y axis.

LABEL= specifies axis labels for the X or Y axis.

XVALUES= specifies a vector of values for ticks for the X axis.

YVALUES= specifies a vector of values for ticks for the Y axis.

PROCOPT= specifies options in the PROC SGPLOT statement.

OTHER= specifies statements in the SGPLOT procedure.

If you use the LABEL= option to specify a single label, that label is used to label the interval axis that shows the distribution of data values. If you specify two labels, the first labels the categorical variable (if you use the CATEGORY= option) and the second labels the data axis.

The following statements provide additional examples of creating box plots:

```
title "Category and Group Variables";
call Box(MPG_City) Type="HBox" Category=Origin group=Type grid="x"
                   label={"Country of Origin" "MPG City"}
                   other="refline 22 / axis=x;";

title "Data Labels and Jittering";
call Box(MPG_City) Category=Type label={"Vehicle Type" "MPG City"}
                   datalabel=putc(Model,"$10.") option="spread";
```

Figure 25.59 Horizontal Box Plot with Categorical and Group Variables

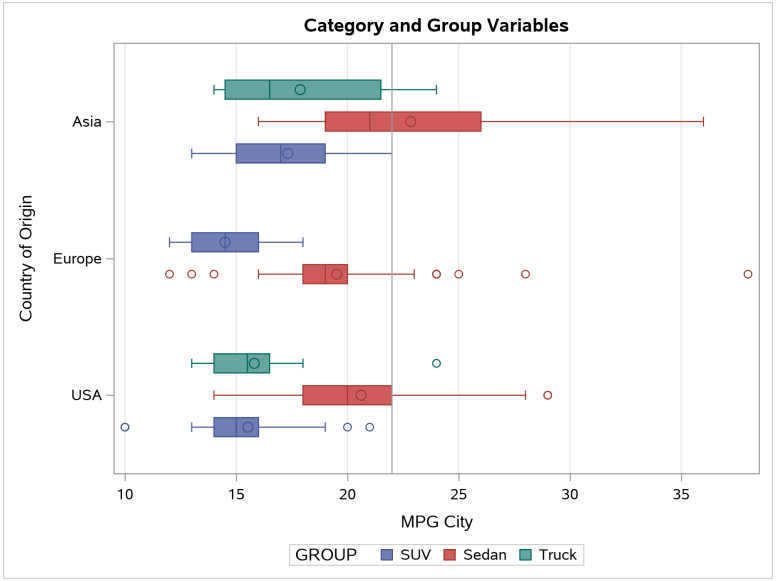
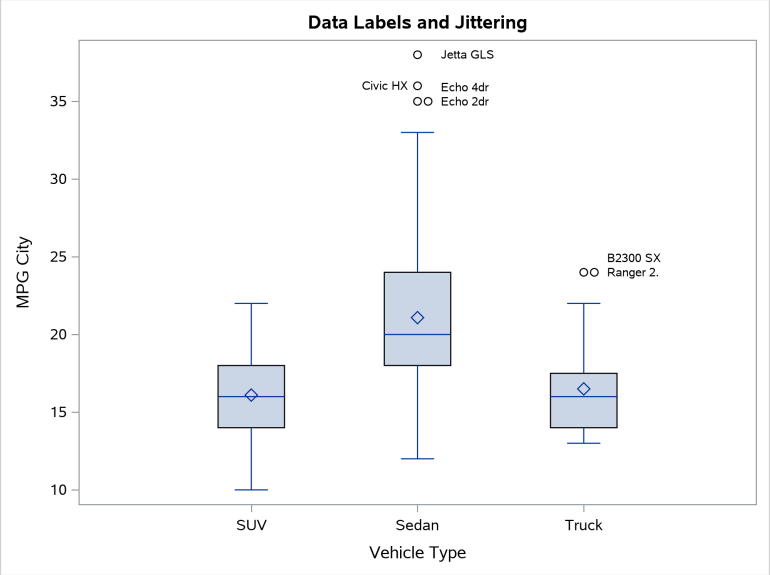


Figure 25.60 Box Plot with Data Labels and Jittered Observations



BRANKS Function

BRANKS(*matrix*);

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

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

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

where

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

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

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

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

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

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

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

Figure 25.61 Tied Ranks and Bivariate Ranks

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

BSPLINE Function

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

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

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

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

k is a numeric vector of size *n* that contains the B-spline knots. If *k* is a scalar that contains a missing value, the knot positions are determined by the *i* argument. Otherwise, the elements of the knot vector must satisfy the following requirements:

- The elements of the knot vector must be nondecreasing: $k_{j-1} \leq k_j$ for $j = 2, \ldots, n$.
- At least *d* knots must be less than or equal to the smallest data value.
- At least max(*d*,1) knots must be greater less than or equal to the largest data value.

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

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

In addition, the BSPLINE function places d exterior knots that are less than $x_{(1)}$ and max(d,1) exterior knots that are greater than $x_{(m)}$. The exterior knots are evenly spaced and start at $x_{(1)} - 10^{-12}$ and $x_{(m)} + 10^{-12}$, respectively. In this case the BSPLINE function returns a matrix with m rows and $i + d + 1$ columns.

The BSPLINE function computes B-splines of degree d. Suppose that $B_j^d(x)$ denotes the jth B-spline of degree d in the knot sequence $k_1, \ldots, k_n$. De Boor (1978) defines the splines based on the following relationships:

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

and for $d > 0$

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

Note that De Boor (1978) expresses B-splines in terms of order rather than degree; in his notation $B_{j,d} = B_j^{d-1}$. B-splines have many interesting properties, including the following:

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

See De Boor (1978) for more details. The BSPLINE function defines B-splines of degree 0 as nonzero when $k_j < x \le k_{j+1}$.

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

For example, the following statements creates a B-spline basis with three interior knots. The BSPLINE function returns a matrix with $3 + d + 1 = 7$ columns, shown in Figure 25.63.

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

Figure 25.62 B-Spline Basis

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

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

```
bsp2 = bspline(x, 2, ., 4);
print bsp2[format=best5.];
```

The resulting matrix is shown in Figure 25.63.

Figure 25.63 B-Spline Basis with Four Interior Knots

bsp2						
0.5	0.5	2E-24	0	0	0	0
74E-5	0.537	0.462	0	0	0	0
0	0	0	0.012	0.63	0.358	0
0	0	0	0	2E-24	0.5	0.5

BTRAN Function

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

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

x is an $(in) \times (jm)$ numeric matrix.

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

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

The argument *x* is a partitioned matrix formed from submatrices of dimension $n \times n$. If the *i*th, *j*th submatrix of the argument *x* is denoted $\mathbf{A}_{ij}$, then the *i*th, *j*th submatrix of the result is $\mathbf{A}_{ji}$.

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

For example, the following statements compute the block transpose of a matrix:

```
a11 = {1 1,                    /* a 3 x 2 matrix */
       1 1,
       1 1};
a12 = 1 + a11;
a13 = 2 + a11;
```

```
a21 = 3 + a11;
a22 = 4 + a11;
a23 = 5 + a11;

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

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

Figure 25.64 Block Transpose of a Partitioned Matrix

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

BYTE Function

BYTE(*matrix***);**

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

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

```
a1 = byte(47);
a2 = "/";       /* the slash character */
print a1 a2;
```

Figure 25.65 Specifying the Slash Character

a1	a2
/	/

The lowercase English letters can be generated with the following statement, shown in Figure 25.66:

```
y = byte(97:122);    /* or use "a":"z" */
print y;
```

Figure 25.66 Lowercase English Letters

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

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

CALL Statement

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

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

The arguments to the CALL statement are as follows:

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

arguments are arguments to the module or subroutine.

The CALL statement executes a subroutine. If you define a module that has the same name as a built-in subroutine, the CALL statement can be used to call the built-in subroutine.

See also the section on the RUN statement.

CCEPSTRUM Call

CALL CCEPSTRUM(*y, nd, x***);**

The CCEPSTRUM subroutine computes the complex cepstrum of a time series. The complex cepstrum of a time series x is defined as

$$y = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log[X(e^{i\omega})] e^{i\omega n} d\omega$$

where $X(e^{i\omega})$ is the Fourier transform of the input time series and $i = \sqrt{-1}$.

The input argument to the CCEPSTRUM subroutine is as follows:

x specifies the input data vector. It can be either a column vector or a row vector. Any missing value in this vector is replaced with 0.

The CCEPSTRUM subroutine returns the following values:

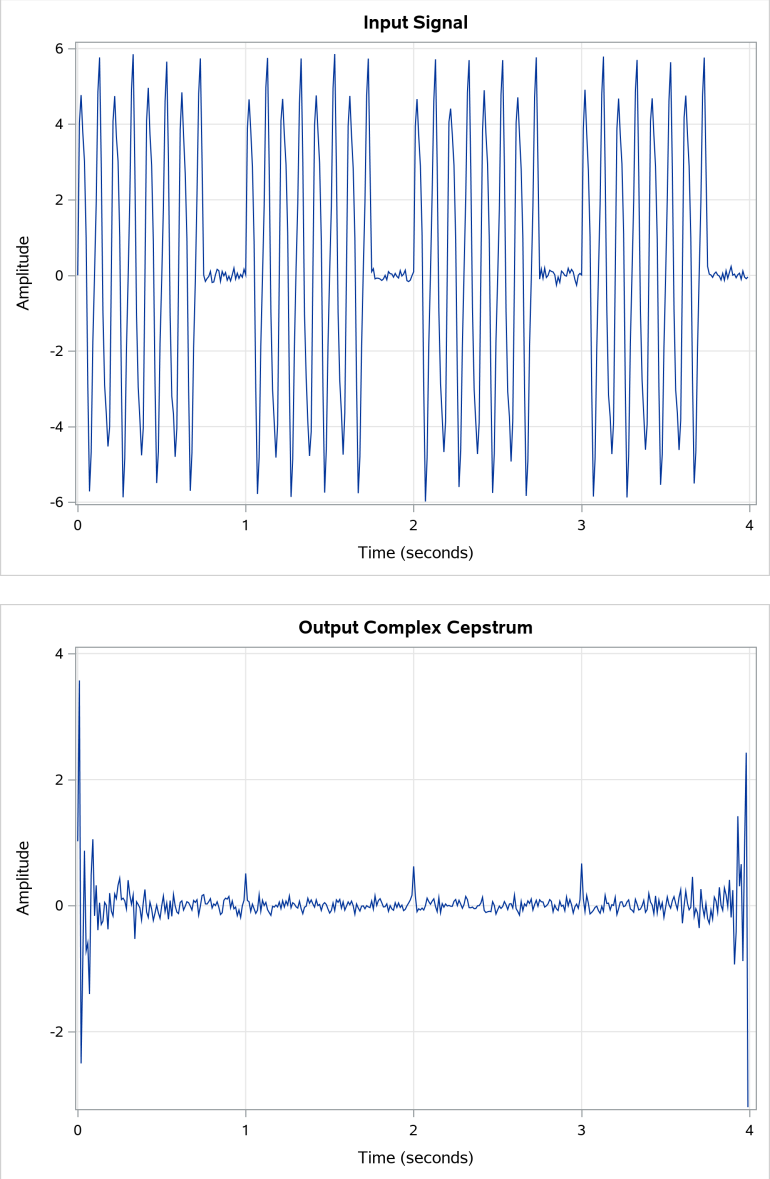
y is a column vector that contains the complex cepstrum of the input data. The output vector has the same length as the input data vector.

nd is a nonnegative integer that is the number of samples of (circular) delay that are added to the input data before the complex cepstrum is computed.

The following example shows how to use the CCEPSTRUM call to estimate the period of a periodic signal. First an input signal is generated that consists of four segments. Each segment is 1 second long; the first 0.75 seconds contain the sum of two sinusoidal signals, and the next 0.25 seconds contain all zeros. This segment is repeated four times, and Gaussian random noise is added to form a pseudoperiodic signal. The output complex spectrum in Figure 25.67 shows a spike around $t = 1$ second, which is a good estimation of the period.

```
sf  = 100;    /* sampling frequency */
T = 1.0/sf;   /* sampling period */
N = 400;
t = do(0, (N-1)*T, T);
t1 = t[, 1:N*3/16];

f1 = 10;
f2 = 25;
pi = constant("pi");
x1 = 5*sin(2*pi*f1*t1);
x2 = sin(2*pi*f2*t1);
x3 = x1 + x2;        /* sum of two sinusoidal signals */
x4 = j(1, N/16, 0);
x5 = repeat(x3 || x4, 1, 4);

call randseed(1);
noise_mean = 0;
noise_std  = 0.1;
xn = randfun(N, "Normal", noise_mean, noise_std);
x = x5` + xn;  /* input signal */

call ccepstrum(y, nd, x);

title "Input Signal";
call series(t,x) grid= {X Y}
                 label={"Time (seconds)" "Amplitude"};

title "Output Complex Cepstrum";
call series(t,y) grid= {X Y}
                 label={"Time (seconds)" "Amplitude"};
```

Figure 25.67 Complex Cepstrum Example



CHANGE Call

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

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

matrix	is a character matrix.
old	is the string to be changed.
new	is the string to replace the *old* string.

numchange is the number of times to make the change.

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

For example, consider the following statements:

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

The result of these statements is shown in Figure 25.68.

Figure 25.68 New String

a
It was a dark and stormy day.

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

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

CHAR Function

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

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

The arguments to the CHAR function are as follows:

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

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

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

For example, the following statements produce the output shown in Figure 25.69:

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

Figure 25.69 Character Matrix

m	1 row	4 cols	(character, size 4)

-1.1	0.0	3.1	4.0

CHOOSE Function

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

The CHOOSE function examines each element of the first argument for being true (nonzero and not missing) or false (zero or missing). For each true element, it returns the corresponding element in the second argument. For each false element, it returns the corresponding element in the third argument.

The arguments to the CHOOSE function are as follows:

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

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

For example, suppose that you want to choose between x and y according to whether $x \# y$ is odd or even, respectively. You can use the following statements to execute this task, as shown in Figure 25.70:

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

Figure 25.70 Result of the CHOOSE Function

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

As another example, the following statements replace all missing values in the matrix **z** with zeros, as shown in Figure 25.71:

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

Figure 25.71 Replacement of Missing Values

z		
1	2	.
100	.	-90
.	5	8

newZ		
1	2	0
100	0	-90
0	5	8

CLOSE Statement

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

CLOSE *(matrix)* **;**

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

The optional argument specifies the name of one or more SAS data sets. The data sets can be specified with a literal value or with an expression that resolves to the name of a SAS data set. You can specify a one-level name (for example, A) or a two-level name (for example, Sasuser.A). For example, the following statements are valid:

```
use Sashelp.Class;
close Sashelp.Class;/* literal value */
f = "Sashelp.Class";
use (f);
close (f);          /* expression */
```

If you do not specify a data set name, the current data set is closed. For more information about specifying SAS data sets, refer to Chapter 7, "Working with SAS Data Sets."

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

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

The following statements provide examples of using the CLOSE statement:

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

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

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

Figure 25.72 Open Data Sets

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

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

CLOSEFILE Statement

CLOSEFILE *files* **;**

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

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

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

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

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

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

Figure 25.73 Open External File

```
FILE NAME                 MODE            EOF    OPTIONS
---------                 ------          -----  --------
FILENAME:MYFILE           Current Input , no eof, lrecl=32767 STOPOVER
```

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

```
*src = "C:\My Data\MyData.txt";  /* Windows */
src = "U:/temp/MyData.txt";      /* Linux   */
infile (src);
show files;
closefile (src);
```

COL Function

COL(*x*);

The COL function is part of the IMLMLIB library. The COL function returns a matrix that has the same dimensions as the *x* matrix and whose *j*th column has the value *j*. You can use the COL and ROW function to extract elements of a matrix. For example, the following statements fill the subdiagonal, superdiagonal, and main diagonal of a matrix with a sequence of numbers:

```
x = j(5, 5, 0);              /* allocate 5 x 5 matrix of zeros */
r = row(x);                  /* create helper matrices */
c = col(x);
idx = loc(abs(r-c)<= 1);     /* indices of sub-, super-, and main diagonal */
x[idx] = 1:ncol(idx);        /* fill with 1,2,3,... */
print x[format=Best3.];
```

Figure 25.74 A Tridiagonal Matrix

x				
1	2	0	0	0
3	4	5	0	0
0	6	7	8	0
0	0	9	10	11
0	0	0	12	13

If `r = row(m)` and `c = col(m)` are two matrices, then you can use logical comparisons of `r` and `c` to describe certain submatrices, such as in Table 25.2:

Table 25.2 Some Common Submatrices

Submatrix	Index by LOC of
Diagonal	`r = c`
Upper triangular	`r < c`
Lower triangular	`r > c`
Banded with radius *d*	`abs(r-c) <= d`
Antidiagonal	`r + c -1 = ncol(r)`

You can also use the COL function to generate an ID variable when you convert data from a wide format to a long format. For example, the following statements show how to generate a column vector with values $\{1, 2, 3, 1, 2, 3, \ldots, 1, 2, 3\}$:

```
NumSubjects = 5;         /* number of subjects */
NumRepeated = 3;         /* number of repeated obs per subject */
Y = col(j(NumSubjects, NumRepeated));
Repl = shape(Y, 0, 1);  /* {1, 2, 3, 1, 2, 3, ..., 1, 2, 3}   */
```

COLVEC Function

COLVEC(*matrix* **);**

The COLVEC function is part of the IMLMLIB library. The COLVEC function converts a matrix into a column vector. If *matrix* is any $n \times m$ matrix, the COLVEC function returns an $nm \times 1$ vector that contains the elements of *matrix* in row-major order. The first m elements in the vector correspond to the first row of the input matrix, the next m elements correspond to the second row, and so on, as shown in the following example.

```
x = {1 2 3,
     4 5 6};
y = colvec(x);
print y;
```

Figure 25.75 A Column Vector

y
1
2
3
4
5
6

See the ROWVEC function for converting a matrix into a row vector.

COMPORT Call

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

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

The subroutine returns the following values:

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

r is the $n \times n$ upper triangular matrix **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 **A**. The full $m \times n$ upper triangular matrix **R** of the orthogonal decomposition of matrix **A** can be obtained by vertical concatenation of the $(m - n) \times n$ zero matrix to the result *r*.

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

piv is an $n \times 1$ vector of permutations of the columns of **A**. That is, the QR decomposition is computed, not of **A**, but of the matrix with columns $[\mathbf{A}_{\text{piv}[1]} \ldots \mathbf{A}_{\text{piv}[n]}]$. The vector *piv*

corresponds to an $n \times n$ permutation matrix, $\mathbf{\Pi}$, of the pivoted QR decomposition in the first step of orthogonal decomposition.

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

The input arguments to the COMPORT subroutine are as follows:

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

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

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 that specifies a singularity criterion.

The complete orthogonal decomposition of the singular matrix $\mathbf{A}$ can be used to compute the Moore-Penrose inverse $\mathbf{A}^-$, $r = \mathrm{rank}(\mathbf{A}) < n$, or to compute the minimum Euclidean-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,

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

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

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

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

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

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

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

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

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

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

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

   (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 by using backward substitution, which yields an $r \times t$ intermediate matrix.

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

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

The following example demonstrates some uses of the COMPORT subroutine:

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

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

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

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

/* verify generalized inverse */
eps = 1e-12;
if any(A*Ainv*A-A > eps) |
```

```
   any(Ainv*A*Ainv-Ainv > eps) |
   any((A*Ainv)`-A*Ainv > eps) |
   any((Ainv*A)`-Ainv*A > eps) then
   msg = "Pseudoinverse conditions not satisfied";
else
   msg = "Pseudoinverse conditions satisfied";
print msg;
```

Figure 25.76 Results from a Complete Orthogonal Factorization

A2
1 0 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 1 0 0
0 1 0 1 0
0 1 0 0 1

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

msg
Pseudoinverse conditions satisfied

CONCAT Function

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

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

All nonscalar arguments must have the same dimensions. Any scalar arguments are used repeatedly to concatenate to all elements of the other arguments. The element length of the result equals the sum of the element lengths of the arguments. Trailing blanks of one matrix argument appear before elements of the next matrix argument in the result matrix.

For example, suppose you specify the following matrices:

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

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

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

Figure 25.77 Elementwise Concatenation of Strings

a
ABH C IJ
DE K FGLM

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

CONTENTS Function

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

The CONTENTS function returns a column vector that contains the variable names for a SAS data set. The vector contains *n* rows, where *n* is the number of variables in the data set. The variable list is returned in the order in which the variables occur in the data set.

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

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

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

classVars = contents("Sashelp", "Class"); /* contents of data set in */
                                          /* Sashelp library         */
print tempVars classVars;
```

Figure 25.78 Names of Variables in SAS Data Sets

tempVars	classVars
COL1	Name
COL2	Sex
COL3	Age
COL4	Height
COL5	Weight

See also the description of the SHOW CONTENTS statement.

CONTINUE Statement

CONTINUE ;

The CONTINUE statement stops the processing of the current iteration of a DO loop and resumes processing at the next iteration of the DO loop. The CONTINUE statement can be used only inside a DO loop.

You can use the CONTINUE statement to skip a computation within a DO loop when a specified condition occurs. For example, the following program skips computations that involve a missing value:

```
x = 1:10;
x[{3 7 8}] = .;

sum = 0;
do i = 1 to ncol(x);
   if x[i] = . then continue; /* go to the next iteration */
   sum = sum + x[i];
end;
print sum;
```

Figure 25.79 Sum of Nonmissing Values

sum
37

The CONTINUE statement can be used to skip an iteration without exiting a DO loop. To exit a DO loop, you can use the DO-WHILE statement, the DO-UNTIL statement, or the LEAVE statement.

CONVEXIT Function

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

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

times	is an n-dimensional column vector of times. Elements should be nonnegative.
flows	is an n-dimensional column vector of cash flows.
ytm	is the per-period yield-to-maturity of the cash-flow stream. This is a scalar and should be positive.

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

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

Under certain assumptions, the convexity of cash flows that are not yield-sensitive is given by

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

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

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

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

Figure 25.80 Convexity of a Noncontingent Cash Flow

convexit
199.26229

CORR Function

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

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

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

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

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

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

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

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

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

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

Figure 25.81 Correlation Matrices

corr		
1	0.1091089	0.265165
0.1091089	1	-0.289319
0.265165	-0.289319	1

spearman		
1	0.3441236	0.2236068
0.3441236	1	-0.410391
0.2236068	-0.410391	1

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

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

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

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

Figure 25.82 Correlation Matrices with Named Rows and Columns

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

spearman			
	X1	**X2**	**X3**
X1	1	0.3441236	0.2236068
X2	0.3441236	1	-0.410391
X3	0.2236068	-0.410391	1

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

CORR2COV Function

CORR2COV(*R, sd***);**

The CORR2COV function is part of the IMLMLIB library. The CORR2COV function converts a correlation matrix into a covariance matrix. The first argument, *R*, is the correlation matrix, and the second argument, *sd*, is a vector such that **sd[j]** is the standard deviation of the *j*th column. An example follows:

```
R = {1.00 0.25 0.90,
     0.25 1.00 0.50,
     0.90 0.50 1.00};
sd = {1 4 9};              /* std devs of the vars */
S = Corr2Cov(R, sd);       /* convert correlation to covariance */
print S;
```

Figure 25.83 Covariance Matrix

S		
1	1	8.1
1	16	18
8.1	18	81

The function scales the correlation matrix so that $S = DRD$, where $D = \text{diag}(\text{sd})$ is the diagonal matrix of standard deviations.

To convert from a covariance matrix to a correlation matrix, use the COV2CORR function.

COUNTMISS Function

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

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

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

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

"all" specifies that all missing values are counted. This is the default value. The function returns a 1×1 matrix.

"row" specifies that the function return an $n \times 1$ matrix whose *i*th element is the number of missing values in the *i*th row of *x*.

"col" specifies that the function return a $1 \times p$ matrix whose *j*th element is the number of missing values in the *j*th row of *x*.

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

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

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

Figure 25.84 Counts of Missing Values

totalMiss
4

rowMiss
0
1
2
1

colMiss		
1	1	2

COUNTN Function

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

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

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

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

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

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

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

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

Figure 25.85 Counts of Nonmissing Values

totalN
8

rowN
3
2
1
2

colN
3 3 2

COUNTUNIQUE Function

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

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

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

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

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

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

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

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

Figure 25.86 Counts of Unique Values

allUnique
4

rowUnique
3
2
1
2

colUnique		
1	3	4

COV Function

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

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

x specifies an $n \times p$ numerical matrix of data. The COV function computes a $p \times p$ variance-covariance matrix of the data.

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

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

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

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

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

```
print cov;
```

Figure 25.87 Variance-Covariance Matrix

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

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

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

COV2CORR Function

COV2CORR(*S*);

The COV2CORR function is part of the IMLMLIB library. A correlation matrix estimates the correlations of centered and standardized variables, where each variable has been scaled by its standard deviation. The COV2CORR function converts a covariance matrix into a correlation matrix, as in the following example:

```
S = {1.0  1.0  8.1,
     1.0 16.0 18.0,
     8.1 18.0 81.0 };
R = Cov2Corr(S);
print R;
```

Figure 25.88 Correlation Matrix

R		
1	0.25	0.9
0.25	1	0.5
0.9	0.5	1

The variances of the three variables are found on the diagonal of S. Equivalently, the square roots of the diagonal elements are the standard deviations. The COV2CORR function scales S so that $R = D^{-1}SD^{-1}$, where $D = \text{diag}(\text{sd})$ is the diagonal matrix of standard deviations.

To convert from a correlation matrix to a covariance matrix, use the CORR2COV function.

COVLAG Function

COVLAG(*x*, *k*);

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

The arguments to the COVLAG function are as follows:

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

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

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

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

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

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

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

For example, the following statements produce a lagged crossproduct matrix:

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

Figure 25.89 Lagged Crossproduct Matrix

cov			
33	23.1	13.6	4.9

CREATE Statement

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

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

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

When you write to a SAS data set, the variable types and lengths correspond to the attributes of the vectors specified in the VAR clause or the matrices in the FROM clause.

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

The arguments to the CREATE statement are as follows:

SAS-data-set is the name of a SAS data set. It can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). You can also specify an expression (enclosed in parentheses) that resolves to the name of a SAS data set. See the example for the CLOSE statement.

operand specifies a set of existing SAS/IML matrices that contain data. The names of the matrices become the names of the data set variables. If you do not specify the name of a variable, all variables in scope are assumed. You can specify variables by using any of the methods described in the section "Select Variables with the VAR Clause" on page 105.

matrix-names specifies one or more matrices that contain the data. Each column of the matrix produces a variable in the data set.

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

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

Writing Data from Vectors

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

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

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

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

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

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

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

```
append;
close Population;
```

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

Writing Data from a Matrix

The following example uses the FROM clause with the COLNAME= option to create a SAS data set named MyData. The new data set has variables named with the COLNAME= operand. The data are in the FROM matrix **x**, and there are two observations because **x** has two rows of data. The COLNAME= operand gives descriptive names to the data set variables, as shown in the following statements:

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

As shown in the example, you can specify a COLNAME= and a ROWNAME= matrix after 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 sum of the number of columns in each FROM matrix. The CREATE statement uses the first *ncol* elements of the COLNAME= matrix in row-major order.

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

The following modifies the previous example to show how to create a data set from two or more matrices. The data are in the numerical matrix **x** and the character matrix **y**.

```
x = {1 2 3, 4 5 6};
y = {A B, C D};
varNames = ("X1":"X3") || ("Y1":"Y2") ;
rowNames = {"row1" "row2"};
/* create data set MYDATA with variables rowNames, X1, X2, X3, Y1, Y2 */
create MyData from x y [colname = varNames rowname = rowNames];
append from x y [rowname = rowNames];
close MyData;
```

Writing Data That Contain Formats

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

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

/* time of equinox, GMT (Greenwich Mean Time) */
```

```
time = { '17:32't,      '23:21't,      '05:14't,
         '11:02't,      '16:57't,      '22:45't };
mattrib time format=TIMEAMPM.;

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

proc print data=MarchEquinox;
run;
```

Figure 25.90 Data Set That Contains Formats

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

CSHAPE Function

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

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

The arguments to the CSHAPE function are as follows:

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

The dimension of the matrix created by the CSHAPE function is specified by *nrow* (the number of rows), *ncol* (the number of columns), and *size* (the element length). A padding character is specified by *padchar*.

The CSHAPE function works by looking at the source matrix as if the characters of the source elements had been concatenated in row-major order. The source characters are then regrouped into elements of length *size*. These elements are assigned to the result matrix, once again in row-major order.

If there are not enough characters for the result matrix, the source of the remaining characters depends on whether padding was specified with *padchar*. If no padding was specified, the characters in the source matrix are cycled through again. If a padding character was specified, the remaining characters are all the padding character.

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

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

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

Figure 25.91 Reshaped Character Matrix

a
a b
c d

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

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

Figure 25.92 Reshaped Character Matrix

b
abc def
ghi jkl

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

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

Figure 25.93 Reusing Characters

c
abc dea
bcd eab

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

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

Figure 25.94 Using a Pad Character

d
abc de*
*** ***

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

CUSUM Function

CUSUM(*matrix***);**

The CUSUM function computes cumulative sums. The argument to this function is a numeric matrix or literal.

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

For example, the following statements compute cumulative sums:

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

Figure 25.95 Cumulative Sums

a
1 3 7 12

b
5 11
14 18

CUPROD Function

CUPROD(*matrix***);**

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

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

For example, the following statements compute cumulative products:

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

Figure 25.96 Output from the CUPROD Function

a
1 2 8 40

b
5 30
30 120

CV Function

CV(*x*);

The CV function is part of the IMLMLIB library. The CV function returns the sample coefficient of variation for each column of a matrix.

The coefficient of variation (CV) is the ratio of the standard deviation to the arithmetic mean. Conceptually, it is a measure of the variability; it is expressed in units of the mean. For univariate data, the CV is the quantity $100s/\bar{x}$, where s is the sample standard deviation and $\bar{x}$ is the sample mean.

The following example computes the CV for each column of a matrix:

```
x = {1 0,
     2 1,
     4 2,
     8 3,
    16 . };
cv = cv(x);
print cv;
```

Figure 25.97 Sample Coefficient of Variation of Two Columns

cv	
98.373875	86.066297

CVEXHULL Function

CVEXHULL(*matrix*);

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

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

The CVEXHULL function returns an $n \times 1$ matrix of indices. The indices of points in the convex hull in counterclockwise order are returned as the first part of the result matrix, and the negative of the indices of the internal points are returned as the remaining elements of the result matrix. Any points that lie on the convex hull but lie on a line segment joining two other points on the convex hull are not included as part of the convex hull.

The result matrix can be split into positive and negative parts by using the LOC function. For example, the following statements find the index vector for the convex hull and print the associated points:

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

Figure 25.98 Convex Hull of a Planar Set of Points

convexHull	
0	2
0	0
2	-1
4	-1
6	0
5	2

DATASETS Function

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

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

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

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

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

Figure 25.99 Several Data Sets in the Sashelp Library

First5
AACOMP
AARFM
ADSMSG
AFMSG
AIR

DELETE Call

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

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

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

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

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

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

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

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

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

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

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

DELETE Statement

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

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

The arguments to the DELETE statement are as follows:

range specifies a range of observations. You can specify a range of observations by using the ALL, CURRENT, NEXT, AFTER, and POINT keywords, as described in the section "Process a Range of Observations" on page 104.

expression specifies a criterion by which certain observations are selected. The optional WHERE clause conditionally selects observations that are contained within the *range* specification. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

The following statements show examples of using the DELETE statement:

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

/* delete observations in data set */
edit MyData;
```

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

proc print data=MyData;
run;
```

Figure 25.100 Observations That Remain after Deletion

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

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

DESIGN Function

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

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

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

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

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

Figure 25.101 Design Matrix for a Vector with Three Unique Values

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

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

DESIGNF Function

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

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

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

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

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

Figure 25.102 Design Matrix for Vector with Three Unique Values

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

The matrix that is returned by the DESIGNF function can be used to produce full-rank designs. The matrix corresponds to the EFFECT parameterization of classification variables as documented in the section "Parameterization of Model Effects" in the *SAS/STAT User's Guide*.

DET Function

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

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

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

The following statements compute the determinant of a matrix:

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

Figure 25.103 Determinant of a Matrix

d
2

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

DFCONV Function

DFCONV(*u, v,* < *type*>**);**

The DFCONV function computes the convolution of two input vectors, u and v. Assuming that the length of vector u is M and the length of vector v is N, the convolution between these two vectors is defined as

$$y(k) = \sum_{j=\max(1,k+1-N)}^{\min(k,M)} u(j)v(k-j+1)$$

where $k = 1, 2, \ldots, M + N - 1$.

In digital signal processing, convolution plays an important role. The output of a linear time-invariant digital filter, $y(n)$, is the full convolution of its impulse response, $h(n)$, with the input signal, $x(n)$.

The input arguments to the DFCONV function are as follows:

u specifies the first input data vector. Any missing value in this vector is replaced with 0.

v specifies the second input data vector. Any missing value in this vector is replaced with 0.

type is an optional string that specifies the type of convolution to be computed. It is one of the following three values: “Full”, “Same”, or “Valid”. The three types of convolution are defined as follows:

- Full — is the full convolution, which outputs the results with length $M + N - 1$. This is the default type.
- Same — is the same-size convolution, which outputs the central part of the full results with the same length as the first input vector, u.
- Valid — is the valid convolution, which outputs only the results that are computed without the zero-padded samples. If the length of u is less than the length of v, then the output is empty.

The input *type* string is not case-sensitive. When it is not specified, a full convolution is computed.

The DFCONV function returns the convolution results, the length of which depends on the type of convolution specified.

The following example uses the DFCONV function to convolute two vectors:

```
u = {1, 0, 3, 5, 2};
v = {6, 3, 5};
conv_type = "full";

y = dfconv(u, v, conv_type);
print y;
```

Figure 25.104 Convolution Results of Two Vectors

y
6
3
23
39
42
31
10

DFDESIGN Call

CALL DFDESIGN(*b, a, z, p, k, filterName, filterType, n, Wc***);**

The DFDESIGN subroutine returns the transfer function of a digital filter that meets the design specification. The specification includes a digital filter's name, type, order, and low cutoff frequency (or low and high cutoff frequencies). The transfer function's numerator polynomial coefficients, denominator polynomial coefficients, zeros, poles, and gain are returned.

A digital filter's transfer function *H(z)* is defined as

$$H(z) = \frac{Y(z)}{X(z)}$$

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{a_0 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}}$$

$$H(z) = k\frac{(1 - z_1 z^{-1})(1 - z_2 z^{-1}) \cdots (1 - z_N z^{-1})}{(1 - p_1 z^{-1})(1 - p_2 z^{-1}) \cdots (1 - p_M z^{-1})}$$

where the order of the filter is the greater of N or M; $b_i (i = 0, 1, 2, \ldots, N)$ are the transfer function's numerator polynomial coefficients; $a_i (i = 0, 1, 2, \ldots, M)$ are the transfer function's denominator polynomial coefficients; and $z_i (i = 1, 2, \ldots, N)$, $p_i (i = 1, 2, \ldots, M)$, and k are the zeros, poles, and gain of the transfer function, respectively.

The input arguments to the DFDESIGN subroutine are as follows:

filterName is a string that specifies the name of the desired digital filter. Currently only the Butterworth filter is supported. The input string can be "BUTTER" or "BUTTERWORTH" and is not case-sensitive.

filterType is a string that specifies the type of the desired digital filter. It is one of the following four values: "LOWPASS", "HIGHPASS", "BANDPASS", or "BANDSTOP". The input string is not case-sensitive.

n is a positive integer scalar that specifies the order of the digital filter. The maximum supported filter order is 200.

Wc specifies the digital filter's cutoff frequencies. The mathematical notation for the cutoff frequency is ω_c. For a lowpass or highpass filter, ω_c is a scalar value. For a bandpass or bandstop filter, ω_c is a 1×2 or 2×1 vector. The values of the cutoff frequencies are between 0 and 1, where 1 corresponds to the normalized Nyquist frequency (π rad/sample). See the DFORDER call for how to specify the cutoff frequencies for each type of filter.

The DFDESIGN subroutine returns the following values:

b is a column vector that contains the numerator polynomial coefficients of the digital filter's transfer function. For a lowpass or highpass filter, b is an $(n + 1) \times 1$ vector; for a bandpass or bandstop filter, b is a $(2n + 1) \times 1$ vector, where n is the input filter order.

a is a column vector that contains the denominator polynomial coefficients of the digital filter's transfer function. For a lowpass or highpass filter, a is an $(n + 1) \times 1$ vector; for a bandpass or bandstop filter, a is a $(2n + 1) \times 1$ vector, where n is the input filter order.

z is a two-column matrix that contains the zeros of the digital filter's transfer function. Each row of the matrix represents one zero value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. For a lowpass or highpass filter, z is an $n \times 2$ matrix; for a bandpass or bandstop filter, z is a $2n \times 2$ matrix, where n is the input filter order.

p is a two-column matrix that contains the poles of the digital filter's transfer function. Each row of the matrix represents one pole value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. For a lowpass or highpass filter, p is an $n \times 2$ matrix; for a bandpass or bandstop filter, p is a $2n \times 2$ matrix, where n is the input filter order.

k is a scalar that contains the gain of the digital filter's transfer function.

The following example uses the DFDESIGN call to design a lowpass filter:

```
filter_name = "butter";
filter_type = "lowpass";
n = 7;
Wc = 0.3;

call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);
print b, a, z, p, k;
```

Figure 25.105 Output from the DFDESIGN Call for a Lowpass Filter

b
0.0009629
0.0067403
0.0202208
0.0337013
0.0337013
0.0202208
0.0067403
0.0009629

a
1
-2.782514
3.9668079
-3.40515
1.8758627
-0.650946
0.1308525
-0.011663

z	
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0

p	
0.4981133	0.6684049
0.4981133	-0.668405
0.3907072	0.4204395
0.3907072	-0.420439
0.3399766	0.2030305
0.3399766	-0.203031
0.3249197	0

k
0.0009629

DFFILT Function

DFFILT(*x, b, a***);**

DFFILT(*x, z, p, k***);**

The DFFILT function filters a time series by using a digital filter's transfer function. You can use the DFDESIGN subroutine to compute the numerator and denominator coefficients, zeros, poles, and gain of a filter's transfer function. You can choose to use either the transfer function's numerator and denominator coefficients or its zeros, poles, and gain to filter the data. The output time series has the same length as the input time series.

The input arguments to the DFFILT function are as follows:

x specifies the input data vector. It can be either a column vector or a row vector. Any missing value in this vector is replaced with 0.

b is a vector that specifies the numerator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

a is a vector that specifies the denominator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

z is a matrix that contains the zeros of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When z is a two-row matrix, each column represents one zero value whose real part is in the first row and whose imaginary part is in the second row. When z is a two-column matrix, each row represents one zero value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If z is a 2×2 matrix, each column represents one zero value.

p is a matrix that contains the poles of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When p is a two-row matrix, each column represents one pole value whose real part is in the first row and whose imaginary part is in the second row. When p is a two-column matrix, each row represents one pole value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If p is a 2×2 matrix, each column represents one pole value.

k is a scalar that specifies the gain of the digital filter's transfer function.

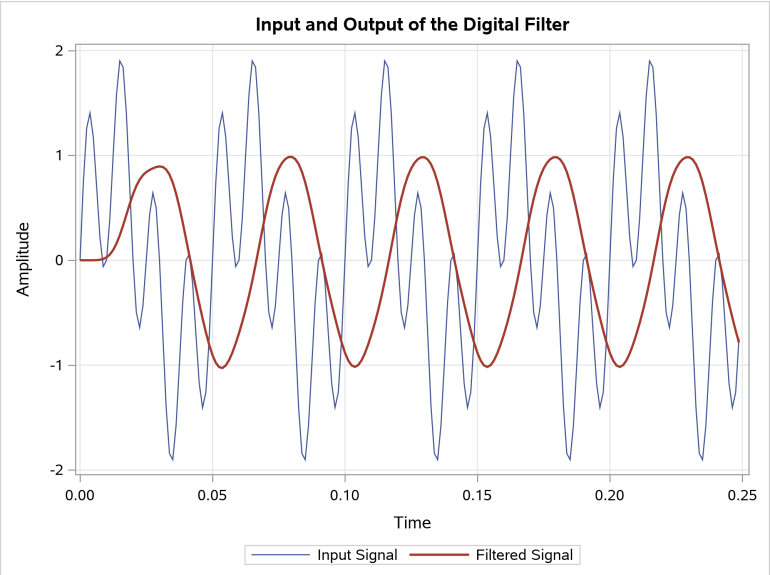
The DFFILT function returns the filtered data that are column vectors of the same length as the input data.

The following example shows how to use a DFDESIGN call to get the desired lowpass filter transfer function and then use the DFFILT function to filter an input signal. First an input signal (which is the sum of two sinusoidal signals, with frequencies *f1* = 20 Hz, *f2* = 80 Hz and sampling frequency *samp_freq* = 800 Hz) is generated. The Nyquist frequency of this signal is 400 Hz, and the normalized frequencies of the two sinusoids are $\frac{20}{400} = 0.05$ and $\frac{80}{400} = 0.2$. Then you use the DFDESIGN call to design a lowpass filter whose normalized cutoff frequency is 0.125. Finally, you use the DFFILT function to apply a lowpass filter to the input signal. Only the sinusoidal signal whose normalized frequency is 0.05 is in the output; the other sinusoidal signal, whose normalized frequency is 0.2, is filtered out. The results, shown in Figure 25.106, use the filter transfer function coefficients *b* and *a*. You can also call the DFFILT function by using the filter transfer function's zeros, poles, and gain, as shown in the code comments.

```
proc iml;
sf  = 800;   /* sampling frequency */
sf2 = sf/2;
T = 1.0/sf;  /* sampling period */
N = 200;
t = do(0, (N-1)*T, T);
```

```
f1 = 20;
f2 = 80;
pi = constant("pi");
x1 = sin(2*pi*f1*t);
x2 = sin(2*pi*f2*t);
x = x1 + x2; /* input signal */

filter_name = "butter";
filter_type = "lowpass";
n = 8;
Wc = 0.125;
call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);

/* use b, a for filtering */
y = dffilt(x, b, a);

/* use z, p, k for filtering */
/* y = dffilt(x, z, p, k);  */

create results var {"t", "x", "y"};
append;
close results;
quit;

title "Input and Output of the Digital Filter";
proc sgplot data=results;
   series x=t y=x / legendlabel="Input Signal";
   series x=t y=y / legendlabel="Filtered Signal" lineattrs=(thickness=2);
   xaxis grid label="Time";
   yaxis grid label="Amplitude";
run;
```

Figure 25.106 Lowpass Filtering Example



DFFREQZ Function

DFFREQZ(*b, a* < *, N*>**);**

The DFFREQZ function generates the frequency response of a digital filter by using the filter transfer function's numerator and denominator coefficients.

The input arguments to the DFFREQZ function are as follows:

b is a vector that specifies the numerator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

a is a vector that specifies the denominator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

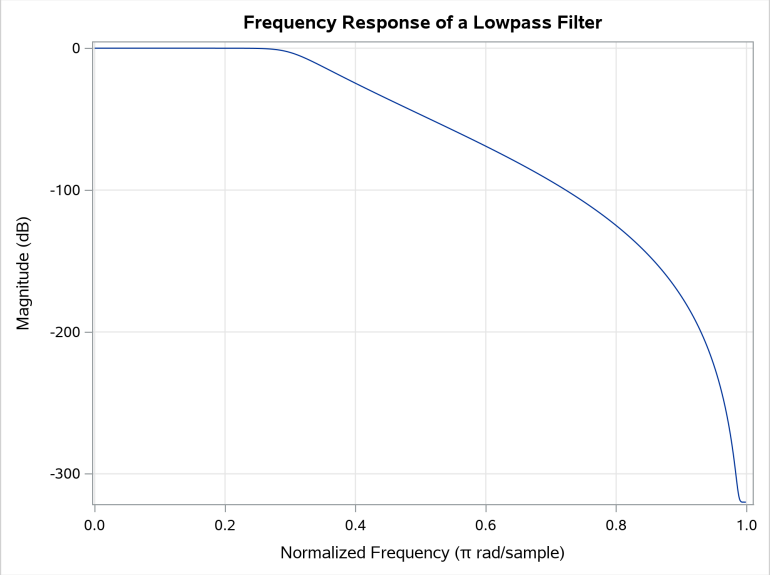
N is a positive integer that specifies the number of evaluation points for the frequency response. *N* must be greater than 1 and greater than the filter's order. *N* is an optional input argument. If it is not specified, a value of 512 is used.

The DFFREQZ function returns a three-column matrix that has *N* rows. The first two columns are the complex frequency response, whose real part is in the first column and whose imaginary part is in the second column. The third column contains the angular frequencies at which the frequency response is evaluated. The range of the angular frequencies is from 0 to π.

The following example uses the DFFREQZ function to generate the frequency response of a lowpass filter by using the filter transfer function's numerator and denominator coefficients, *b* and *a*:

```
filter_name = "butter";
filter_type = "lowpass";
n = 8;
Wc = 0.3;
call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);

N = 1024;
h = dffreqz(b, a, N);

w = h[,3] / constant('pi');
logh = 20*log10(sqrt(h[,1]##2+h[,2]##2) + 1e-16);
/* a small constant is added to avoid the input to log10() is 0 */

title "Frequency Response of a Lowpass Filter";
call series(w,logh) grid= {X Y}
     label={"Normalized Frequency ((*ESC*){unicode pi} rad/sample)" "Magnitude (dB)"};
```

Figure 25.107 Frequency Response Using Filter Transfer Function Coefficients



DFFREQZZPK Function

DFFREQZZPK(*z, p, k* **<** *, N***>);**

The DFFREQZZPK function generates the frequency response of a digital filter by using the filter transfer function's zeros, poles, and gain. It is easier to read the subroutine name when it is written in mixed case: DFFreqzZPK.

The input arguments to the DFFreqzZPK function are as follows:

z is a matrix that contains the zeros of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When *z* is a two-row matrix, each column represents one zero value whose real part is in the first row and whose imaginary part is in the second row. When *z* is a two-column matrix, each row represents one zero value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If *z* is a 2×2 matrix, each column represents one zero value.

p is a matrix that contains the poles of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When *p* is a two-row matrix, each column represents one pole value whose real part is in the first row and whose imaginary part is in the second row. When *p* is a two-column matrix, each row represents one pole value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If *p* is a 2×2 matrix, each column represents one pole value.

k is a scalar that specifies the gain of the digital filter's transfer function.

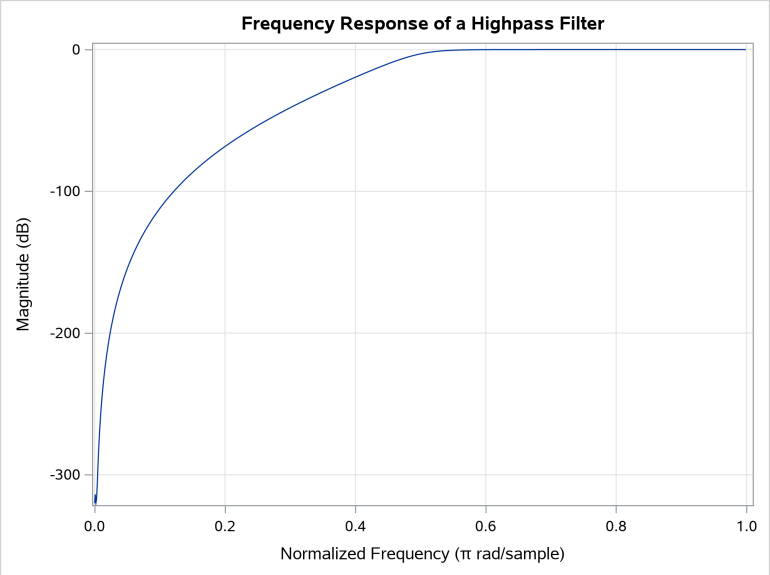
N is a positive integer that specifies the number of evaluation points for the frequency response. *N* must be greater than 1 and greater than the filter's order. *N* is an optional input argument. If it is not specified, a value of 512 is used.

The DFFreqzZPK function returns a three-column matrix. The number of rows of this matrix is the input parameter *N*. The first two columns are the complex frequency response, whose real part is in the first column and whose imaginary part is in the second column. The third column contains the angular frequencies at which the frequency response is evaluated. The range of the angular frequencies is from 0 to π.

The following example uses the DFFreqzZPK function to generate the frequency response of a highpass filter by using the filter transfer function's zeros, poles, and gain:

```
filter_name = "butter";
filter_type = "highpass";
n = 7;
Wc = 0.5;
call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);

N = 1024;
h = DFFreqzZPK(z, p, k, N);

w = h[,3] / constant('pi');
logh = 20*log10(sqrt(h[,1]##2+h[,2]##2) + 1e-16);
/* a small constant is added to avoid the input to log10() is 0 */

title "Frequency Response of a Highpass Filter";
call series(w,logh) grid= {X Y}
     label={"Normalized Frequency ((*ESC*){unicode pi} rad/sample)" "Magnitude (dB)"};
```

Figure 25.108 Frequency Response Using Filter Transfer Function's Zeros, Poles, and Gain



DFMEDFILT Function

DFMEDFILT(*x, wlen***);**

The DFMEDFILT function applies the one-dimensional median filter to the input data. The output *y* is the median value of the input *x* within a window,

$$y(k) = \text{median}(x[k : k + \textit{wlen} - 1])$$

where *wlen* is the length of the window; $k = 1, 2, \ldots, n - \textit{wlen} + 1$; and n is the length of the input data. For windows that have missing data, the median for each window is computed using the nonmissing values in that window. The implementation of the median filter uses a running median algorithm that incrementally updates the median of each sliding window, instead of recalculating the median of the whole window. The complexity of the algorithm is O($n$log(*wlen*)).

The input arguments to the DFMEDFILT function are as follows:

x	specifies the input data vector. It can be either a column vector or a row vector.
wlen	is a positive integer that specifies the window length. It must be less than or equal to the input data length.

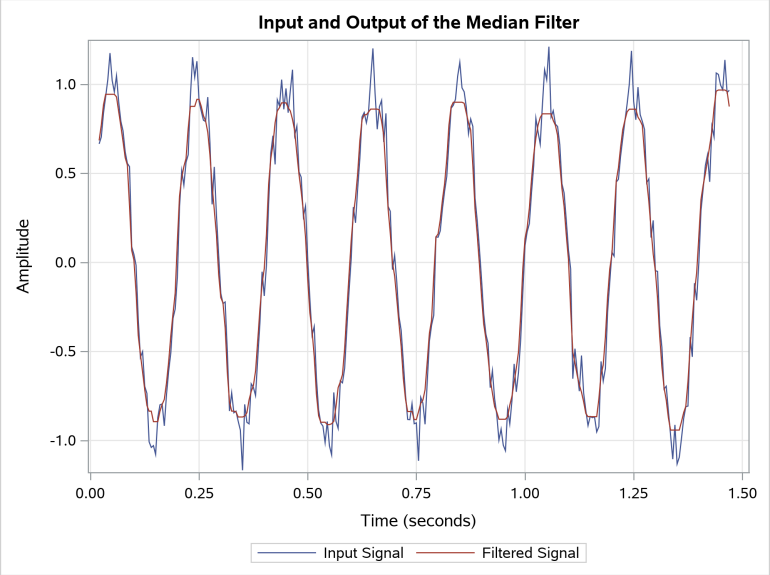
The DFMEDFILT function returns a column vector that contains the filter output. The output length is $n - \textit{wlen} + 1$, where n is the length of the input data.

The following example shows how to use the DFMEDFILT function for noise suppression by using median filtering. The input signal is the sum of a sinusoidal signal and random noise. The DFMEDFILT function filters the input signal. The output shows that the noise is mostly filtered out.

```
proc iml;
sf  = 200;   /* sampling frequency */
T = 1.0/sf;  /* sampling period    */
N = 300;
t = do(0, (N-1)*T, T);
f = 5;       /* frequency of the signal */
pi = constant("pi");
x1 = sin(2*pi*f*t);

call randseed(1);
noise_mean = 0;
noise_std  = 0.1;
x2 = randfun(N, "Normal", noise_mean, noise_std);

x = x1` + x2; /* input signal */

wlen = 10;
y = dfmedfilt(x, wlen);  /* length of y is N-wlen+1 */

shift = wlen/2;          /* if wlen is odd, use wlen/2+1 */
td = t[shift:N-shift];
xd = x[shift:N-shift];

create resultsmed var {"td", "xd", "y"};
append;
close resultsmed;
quit;

title "Input and Output of the Median Filter";
proc sgplot data=resultsmed;
   series x=td y=xd / legendlabel="Input Signal";
   series x=td y=y  / legendlabel="Filtered Signal";
```

```
   xaxis grid label="Time (seconds)";
   yaxis grid label="Amplitude";
run;
```

Figure 25.109 Median Filtering Example



DFORDER Call

CALL DFORDER(*n, Wc, filterName, filterType, Wp, Ws, Rp, Rs***);**

The DFORDER subroutine computes the minimum required order for a digital filter that satisfies a requirement that you specify. Digital filters are designed by specifying the frequency response of the filters in the frequency domain. The specification includes a digital filter's passband and stopband edge frequencies, passband ripple, and stopband attenuation. Figure 25.110 through Figure 25.113 illustrate the filter specification for each type of digital filter (Oppenheim and Schafer 2010; Parks and Burrus 1987).

Figure 25.110 Filter Specification for Lowpass Filter

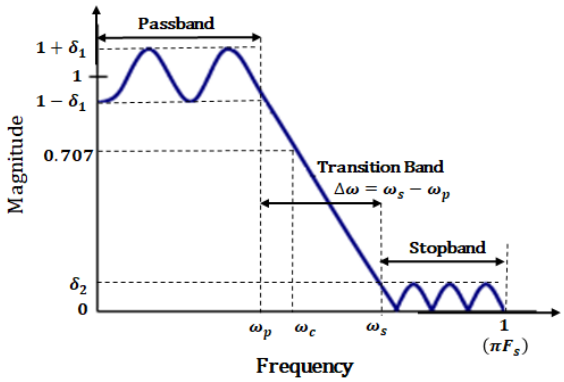


Figure 25.111 Filter Specification for Highpass Filter

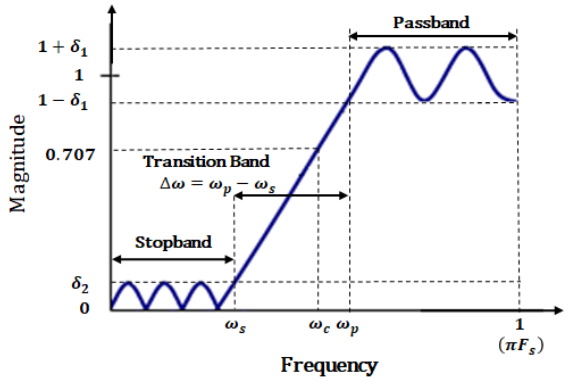


Figure 25.112 Filter Specification for Bandpass Filter

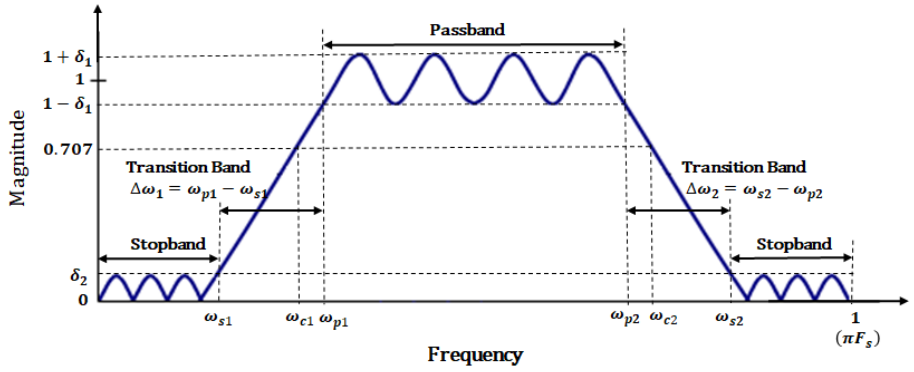
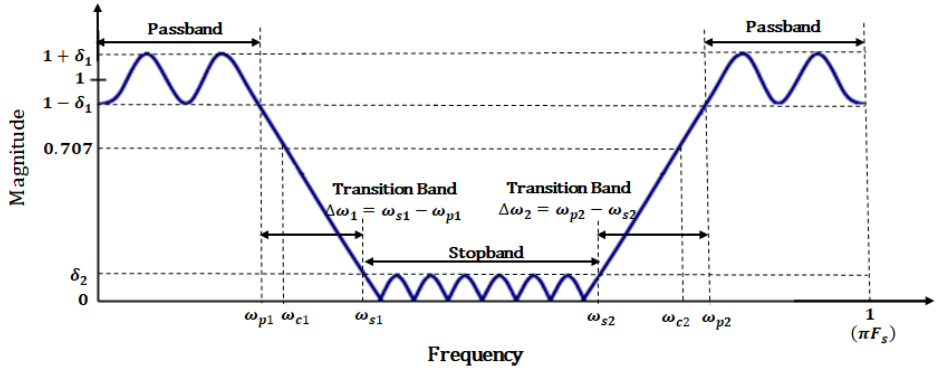


Figure 25.113 Filter Specification for Bandstop Filter



The digital filter passband frequency and stopband edge frequencies are ω_p and ω_s, respectively. For lowpass and highpass filters, you need to specify only one value of each edge frequency. However, for bandpass and bandstop filters, you need to specify two values of the passband edge frequencies (ω_{p1} and ω_{p2}) and two values of the stopband edge frequencies (ω_{s1} and ω_{s2}). The frequency range from the passband edge frequency to the stopband edge frequency is the transition band. The transition band has a frequency response that is unspecified (Oppenheim and Schafer 2010; Parks and Burrus 1987; Roy 2005; Constantinides 1970).

The digital filter passband and stopband can contain oscillations known as ripples. The symbol δ_1 represents the magnitude of the passband ripple, which equals the maximum deviation from the unity magnitude. The symbol δ_2 represents the magnitude response of the stopband attenuation, which equals the maximum deviation from zero. The passband ripple (*Rp*) and stopband attenuation (*Rs*) are usually measured in decibels (dB). They are represented by the symbols R_s and R_s and are defined as

$$R_p = -20\log_{10}(1-\delta_1)\ \text{(dB)}$$
$$R_s = -20\log_{10}(\delta_2)\ \text{(dB)}$$

where the absolute value of the passband ripple, R_p, must be less than the absolute value of the stopband attenuation, R_s; that is, $|R_p| < |R_s|$.

The values of the passband and stopband edge frequencies are normalized values between 0 and 1, where 1 corresponds to the normalized Nyquist frequency (π rad/sample). When the passband and stopband edge frequencies are specified, the following expressions must be satisfied:

Lowpass filter: $\omega_p < \omega_s$

Highpass filter: $\omega_s < \omega_p$

Bandpass filter: $\omega_{s1} < \omega_{p1} < \omega_{p2} < \omega_{s2}$

Bandstop filter: $\omega_{p1} < \omega_{s1} < \omega_{s2} < \omega_{p2}$

In addition to the minimum required order for a digital filter that satisfies the input requirement, the DFORDER subroutine also outputs the digital filter's cutoff frequencies, which the DFDESIGN subroutine can use to design the filter's transfer function. A digital filter's cutoff frequency is defined as the frequency at which the power of the frequency response reaches half the unity power, or equivalently $\sqrt{1/2} \approx 0.707$ of the unity magnitude, which is approximately $-20\log_{10}(0.707) = 3$ in dB. Because half power is about 3dB away from unity power, this frequency is often called the 3dB cutoff frequency.

The input arguments to the DFORDER subroutine are as follows:

filterName is a string that specifies the name of the desired digital filter. Currently only the Butterworth filter is supported. The input string can be "BUTTER" or "BUTTERWORTH" and is not case-sensitive.

filterType is a string that specifies the type of the desired digital filter. It is one of the following four values: "LOWPASS", "HIGHPASS", "BANDPASS", or "BANDSTOP". The input string is not case-sensitive.

Wp specifies the passband edge frequencies, which are denoted by the symbol ω_p. For a lowpass or highpass filter, ω_p is a scalar value. For a bandpass or bandstop filter, ω_p is a 1×2 or 2×1 vector. The values of the passband edge frequencies are between 0 and 1, where 1 corresponds to the normalized Nyquist frequency (π rad/sample).

Ws specifies the stopband edge frequencies, which are denoted by the symbol ω_s. For a lowpass or highpass filter, ω_s is a scalar value. For a bandpass or bandstop filter, ω_s is a 1×2 or 2×1 vector. The values of the stopband edge frequencies are between 0 and 1, where 1 corresponds to the normalized Nyquist frequency (π rad/sample).

Rp is a scalar value that specifies the passband ripple, R_p, in dB.

Rs is a scalar value that specifies the stopband attenuation, R_s, in dB. The absolute value of the passband ripple, R_p, must be less than the absolute value of the stopband attenuation, R_s; that is, $|R_p| < |R_s|$.

The DFORDER subroutine returns the following values:

n is the minimum required order for a digital filter that satisfies the input requirement.

Wc is the digital filter's cutoff frequencies, ω_c, which the DFDESIGN subroutine can use to design the filter's transfer function. For a lowpass or highpass filter, ω_c is a scalar value. For a bandpass or bandstop filter, ω_c is a 2×1 vector. The values of the cutoff frequencies are between 0 and 1, where 1 corresponds to the normalized Nyquist frequency (π rad/sample).

The following statements use the DFORDER call to compute the minimum required order and cutoff frequencies for a Butterworth lowpass filter:

```
filter_name = "butter";
filter_type = "lowpass";
Wp = 0.125;
Ws = 0.25;
Rp = 3;
Rs = 50;

call dforder(n, Wc, filter_name, filter_type, Wp, Ws, Rp, Rs);
print n, Wc;
```

Figure 25.114 Output from the DFORDER Call for a Lowpass Filter

n
8

Wc
0.1267115

The following statements use the DFORDER call to compute the minimum required order and cutoff frequencies for a Butterworth highpass filter:

```
filter_name = "butter";
filter_type = "highpass";
Wp = 0.5;
Ws = 0.25;
Rp = 3;
Rs = 40;

call dforder(n, Wc, filter_name, filter_type, Wp, Ws, Rp, Rs);
print n, Wc;
```

Figure 25.115 Output from the DFORDER Call for a Highpass Filter

n
6

Wc
0.4638373

The following statements use the DFORDER call to compute the minimum required order and cutoff frequencies for a Butterworth bandpass filter:

```
filter_name = "butter";
filter_type = "bandpass";
Wp = 0.25 || 0.5;
Ws = 0.125|| 0.625;
Rp = 3;
Rs = 40;

call dforder(n, Wc, filter_name, filter_type, Wp, Ws, Rp, Rs);
print n, Wc;
```

Figure 25.116 Output from the DFORDER Call for a Bandpass Filter

n
7

Wc
0.2428073
0.5102878

The following statements use the DFORDER call to compute the minimum required order and cutoff frequencies for a Butterworth bandstop filter:

```
filter_name = "butter";
filter_type = "bandstop";
Wp = 0.125 || 0.625;
Ws = 0.25  || 0.5;
Rp = 3;
Rs = 40;

call dforder(n, Wc, filter_name, filter_type, Wp, Ws, Rp, Rs);
print n, Wc;
```

Figure 25.117 Output from the DFORDER Call for a Bandstop Filter

n
7

Wc
0.1804219
0.6098648

DFSOSFILT Function

DFSOSFILT(*x, b, a***);**

DFSOSFILT(*x, z, p, k***);**

The DFSOSFILT function filters a time series by using the second-order section form of a digital filter's transfer function. You can use the DFDESIGN subroutine to compute the numerator and denominator coefficients, zeros, poles, and gain of a filter's transfer function. You can use either the transfer function's numerator and denominator coefficients or its zeros, poles, and gain to filter the data. The output time series has the same length as the input time series.

When filtering by using the second-order section form, the filter's transfer function is a concatenation of a series of second-order section digital filters and has following form:

$$H(z) = \prod_{k=1}^{N} H_k(z)$$
$$H(z) = \prod_{k=1}^{N} \frac{b_{0k} + b_{1k}z^{-1} + b_{2k}z^{-2}}{1 + a_{1k}z^{-1} + a_{2k}z^{-2}}$$

Filtering by using the second-order section form with the DFSOSFILT function is numerically more stable than direct filtering with the DFFILT function, especially when the filter order is high. So it is recommended that you use the DFSOSFILT function to filter a time series.

The input arguments to the DFSOSFILT function are as follows:

x specifies the input data vector. It can be either a column vector or a row vector. Any missing value in this vector is replaced with 0.

b is a vector that specifies the numerator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

a is a vector that specifies the denominator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

z is a matrix that contains the zeros of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When z is a two-row matrix, each column represents one zero value whose real part is in the first row and whose imaginary part is in the second row. When z is a two-column matrix, each row represents one zero value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If z is a 2×2 matrix, each column represents one zero value.

p is a matrix that contains the poles of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When p is a two-row matrix, each column represents one pole value whose real part is in the first row and whose imaginary part is in the second row. When p is a two-column matrix, each row represents one pole value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If p is a 2×2 matrix, each column represents one pole value.

k is a scalar that specifies the gain of the digital filter's transfer function.

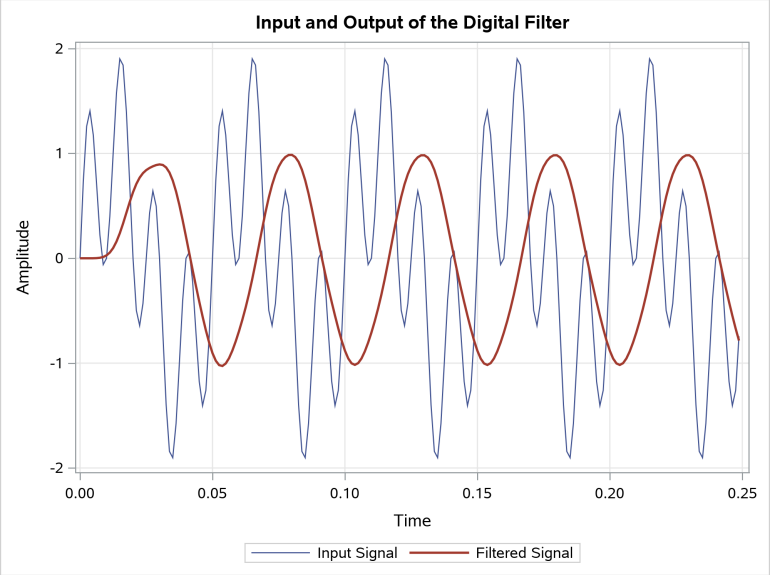
The DFSOSFILT function returns the filtered data that is a column vector of the same length as the input data.

The following example shows how to use a DFDESIGN call to get the desired lowpass filter transfer function and then use the DFSOSFILT function to filter an input signal. First an input signal (which is the sum of two sinusoidal signals, with frequencies *f1* = 20 Hz, *f2* = 80 Hz and sampling frequency *samp_freq* = 800 Hz) is generated. The Nyquist frequency of this signal is 400 Hz, and the normalized frequencies of the two sinusoids are $\frac{20}{400} = 0.05$ and $\frac{80}{400} = 0.2$. Then you use the DFDESIGN call to design a lowpass filter whose normalized cutoff frequency is 0.125. Finally, you use the DFSOSFILT function to apply a lowpass filter to the input signal. Only the sinusoidal signal whose normalized frequency is 0.05 is in the output; the other sinusoidal signal, whose normalized frequency is 0.2, is filtered out. The results, shown in Figure 25.118, are using the filter transfer function's zeros, poles, and gain. Although you can also call the DFSOSFILT function with the filter transfer function coefficients *b* and *a*, it is recommended that you use zeros, poles, and gain if they are available.

The same example can use DFFILT function for digital filtering. But using a filter's second-order section form (DFSOSFILT) is numerically more stable, so it is recommended.

```
sf  = 800;   /* sampling frequency */
sf2 = sf/2;
T = 1.0/sf;  /* sampling period */
N = 200;
t = do(0, (N-1)*T, T);

f1 = 20;
f2 = 80;
pi = constant("pi");
x1 = sin(2*pi*f1*t);
x2 = sin(2*pi*f2*t);
x = x1 + x2; /* input signal */

filter_name = "butter";
filter_type = "lowpass";
n = 8;
Wc = 0.125;
call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);

/* use z, p, k for filtering */
y = dfsosfilt(x, z, p, k);

create resultssos var {"t", "x", "y"};
append;
close resultssos;

title "Input and Output of the Digital Filter";
proc sgplot data=resultssos;
   series x=t y=x / legendlabel="Input Signal";
   series x=t y=y / legendlabel="Filtered Signal" lineattrs=(thickness=2);
   xaxis grid label="Time";
   yaxis grid label="Amplitude";
run;
```

Figure 25.118 Lowpass Filtering Example



DFSOSFREQZ Function

DFSOSFREQZ(*b, a* **<** *, N***>);**

The DFSOSFREQZ function generates the frequency response of a digital filter that corresponds to its second-order sections form by using the filter transfer function's numerator and denominator coefficients. It is easier to read the subroutine name when it is written in mixed case: DFSOSFreqz.

The input arguments to the DFSOSFreqz function are as follows:

b is a vector that specifies the numerator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

a is a vector that specifies the denominator polynomial coefficients of the digital filter's transfer function. It can be a column vector or a row vector.

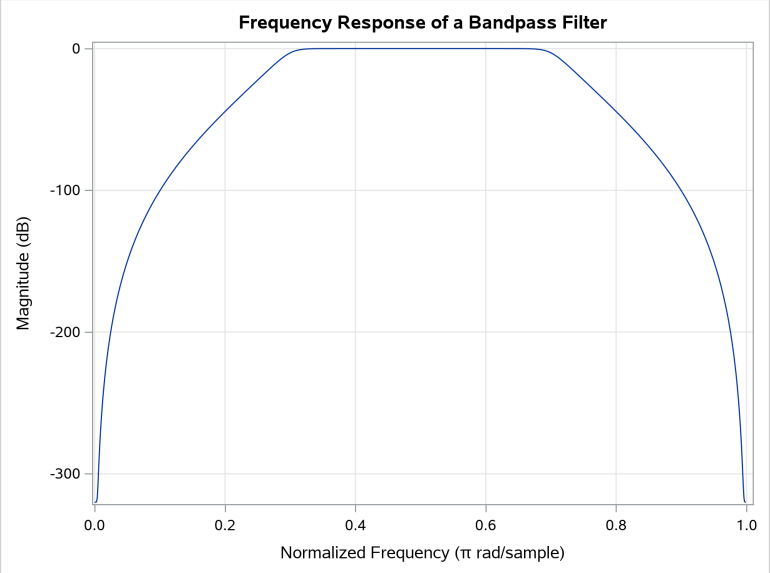
N is a positive integer that specifies the number of evaluation points for the frequency response. *N* must be greater than 1 and greater than the filter's order. *N* is an optional input argument. If it is not specified, a value of 512 is used.

The DFSOSFreqz function returns a three-column matrix. The number of rows of this matrix is the input parameter *N*. The first two columns are the complex frequency response, whose real part is in the first column and whose imaginary part is in the second column. The third column contains the angular frequencies at which the frequency response is evaluated. The range of the angular frequencies is from 0 to π.

The following example uses the DFSOSFreqz function to generate the frequency response of a bandpass filter that corresponds to its second-order sections form by using the filter transfer function's numerator and denominator coefficients, *b* and *a*.

```
filter_name = "butter";
filter_type = "bandpass";
n = 8;
Wc = 0.3 || 0.7;
call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);

N = 1024;
h = DFSOSFreqz(b, a, N);

w = h[,3] / constant('pi');
logh = 20*log10(sqrt(h[,1]#h[,1]+h[,2]#h[,2]) + 1e-16);
/* a small constant is added to avoid the input to log10() is 0 */

w = h[,3] / constant('pi');
logh = 20*log10(sqrt(h[,1]##2+h[,2]##2) + 1e-16);
/* a small constant is added to avoid the input to log10() is 0 */

title "Frequency Response of a Bandpass Filter";
call series(w,logh) grid= {X Y}
     label={"Normalized Frequency ((*ESC*){unicode pi} rad/sample)" "Magnitude (dB)"};
```

Figure 25.119 Frequency Response Using Filter Transfer Function Coefficients



DFSOSFREQZZPK Function

DFSOSFREQZZPK(*z, p, k, < , N>***);**

The DFSOSFREQZZPK function generates the frequency response of a digital filter that corresponds to its second-order sections form by using the filter transfer function's zeros, poles, and gain. It is easier to read the subroutine name when it is written in mixed case: DFSOSFreqzZPK.

The input arguments to the DFSOSFreqzZPK function are as follows:

z is a matrix that contains the zeros of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When *z* is a two-row matrix, each column represents one zero value whose real part is in the first row and whose imaginary part is in the second row. When *z* is a two-column matrix, each row represents one zero value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If *z* is a 2×2 matrix, each column represents one zero value.

p is a matrix that contains the poles of the digital filter's transfer function. It can be a two-column matrix or a two-row matrix. When *p* is a two-row matrix, each column represents one pole value whose real part is in the first row and whose imaginary part is in the second row. When *p* is a two-column matrix, each row represents one pole value, which is a complex number whose real part is in the first column and whose imaginary part is in the second column. If *p* is a 2×2 matrix, each column represents one pole value.

k is a scalar that specifies the gain of the digital filter's transfer function.

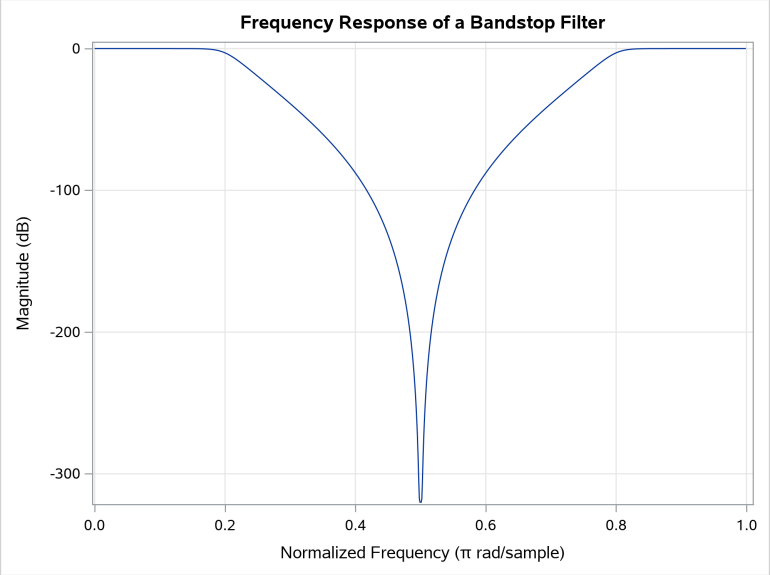
N is a positive integer that specifies the number of evaluation points for the frequency response. *N* must be greater than 1 and greater than the filter's order. *N* is an optional input argument. If it is not specified, a value of 512 is used.

The DFSOSFreqzZPK function returns a three-column matrix. The number of rows of this matrix is the input parameter *N*. The first two columns are the complex frequency response, whose real part is in the first column and whose imaginary part is in the second column. The third column contains the angular frequencies at which the frequency response is evaluated. The range of the angular frequencies is from 0 to π.

The following example uses the DFSOSFreqzZPK function to generate the frequency response of a bandstop filter that corresponds to its second-order sections form by using the filter transfer function's zeros, poles, and gain:

```
filter_name = "butter";
filter_type = "bandstop";
n = 7;
Wc = 0.2 || 0.8;
call dfdesign(b, a, z, p, k, filter_name, filter_type, n, Wc);

N = 1024;
h = DFSOSFreqzZPK(z, p, k, N);

w = h[,3] / constant('pi');
logh = 20*log10(sqrt(h[,1]#h[,1]+h[,2]#h[,2]) + 1e-16);
/* a small constant is added to avoid the input to log10() is 0 */

w = h[,3] / constant('pi');
logh = 20*log10(sqrt(h[,1]##2+h[,2]##2) + 1e-16);
/* a small constant is added to avoid the input to log10() is 0 */

title "Frequency Response of a Bandstop Filter";
call series(w,logh) grid= {X Y}
     label={"Normalized Frequency ((*ESC*){unicode pi} rad/sample)" "Magnitude (dB)"};
```

Figure 25.120 Frequency Response Using Filter Transfer Function's Zeros, Poles, and Gain



DIAG Function

DIAG(*matrix***);**

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

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

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

For example, the following statements produce a diagonal matrix by extracting the diagonal elements of a square matrix:

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

Figure 25.121 Diagonal Matrix Obtained from a Full Matrix

c
4 0
0 1

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

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

Figure 25.122 Diagonal Matrix Obtained from a Vector

d		
1	0	0
0	2	0
0	0	3

The DIAG function is useful, but is not always necessary. Most multiplication operations with diagonal matrices can be accomplished by using the elementwise multiplication operator. To add or subtract from the diagonal of a matrix, you can directly reference the matrix elements, as shown in the following example:

```
A = j(5,5,1);
v = T(1:5);
D = diag(v);

/* overwrite A with A + diag(v) */
/* method 1: less efficient */
A = j(5,5,1);
A = A + diag(v);          /* explicitly form 5x5 diagonal matrix */

/* method 2: more efficient */
A = j(5,5,1);
j = do(1, ncol(A)*nrow(A), ncol(A)+1);      /* {1 7 13 19 25} */
A[j] = vecdiag(A) + v;   /* access diagonal elements directly */
```

DIF Function

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

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

x specifies a $n \times 1$ numerical matrix of time series data.

lags specifies integer lags. The *lags* argument can be an integer matrix with d elements. If so, the DIF function returns an $n \times d$ matrix where the ith column represents the difference between the time series and the lagged data for the ith lag. If the *lags* argument is not specified, a value of 1 is used.

delRows is an binary flag variable that specifies whether to delete certain rows in the result matrix. By default, *delRows* is 0 and the output matrix contains the same number of rows as the input vector. If *delRows* is nonzero, then the first max(*lags*,0) rows and the last min(*lags*,0) rows of the result matrix are deleted. For example, if *x* has n elements and `y=DIF(x,3)`, then `y` is an n-element vector whose first three rows contain missing values. In contrast, `z=DIF(x,3,1)` returns a vector that has $n - 3$ elements and that equals `y[4:`n`]`.

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

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

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

Figure 25.123 Differences between Data and Lagged Data

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

DIMENSION Function

DIMENSION(*x* **);**

The DIMENSION function returns the dimensions of the *x* matrix. The total number of elements in a matrix is `prod(dimension(x))`.

The returned vector is a 1×2 vector. The first element is the number of rows in *x*, and the second element is the number of columns, as shown in the following example:

```
x = {1 2, 3 4, 5 6};
d = dimension(x);
print d;
```

Figure 25.124 The Dimensions of a Matrix

d	
3	2

DISTANCE Function

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

DISTANCE(*x, y, < , method>* **);**

The DISTANCE function computes the pairwise distances in one of two ways:

- When called with one numerical matrix, *x*, the DISTANCE function computes the pairwise distances between the rows of *x*.

- When called with two numerical matrices, *x* and *y*, the DISTANCE function computes the pairwise distances between rows of *x* and the rows of *y*.

The arguments are as follows:

x specifies an $n \times d$ numerical matrix that contains n points in d-dimensional space.

y specifies an $m \times d$ numerical matrix that contains m points in d-dimensional space.

method is an optional argument that specifies which method to use to specify the distance between pairs of points. The *method* argument is either a scalar numeric value or a case-insensitive character value. Only the first four characters are used to distinguish one method from the others. You can specify the following values for *method*:

"L2" computes the Euclidean (L_2) distance between two points. This is the default value. An equivalent alias is "Euclidean".

"L1" computes the Manhattan (L_1) distance between two points. An equivalent alias is "CityBlock" or "Manhattan".

"LInf" computes the Chebyshev (L_∞) distance between two points. An equivalent alias is "Chebyshev".

p computes the distance by using the L_p-norm, where $p \geq 1$.

Distance between Rows of a Matrix

When you specify a single $n \times d$ numerical matrix *x*, the DISTANCE function returns an $n \times n$ symmetric matrix. The (i, j) element is the distance between the ith and jth rows of *x*.

If u and v are two d-dimensional points, then the following formulas are used to compute the distance between u and v:

- Euclidean distance: $\|u - v\|_2 = (\Sigma_k |u_k - v_k|^2)^{1/2}$
- L_1 distance: $\|u - v\|_1 = \Sigma_k |u_k - v_k|$
- L_∞ distance: $\|u - v\|_\infty = \max(|u_1 - v_1|, |u_2 - v_2|, \ldots, |u_p - v_p|)$
- L_p distance: $\|u - v\|_p = (\Sigma_k |u_k - v_k|^p)^{1/p}$

The following statements illustrate the DISTANCE function:

```
x = {1 0,
     0 1,
    -1 0,
     0 -1};
d2 = distance(x, "L2");
print d2[format=best5.];
```

Figure 25.125 Euclidean Distance between Pairs of Points

d2			
0	1.414	2	1.414
1.414	0	1.414	2
2	1.414	0	1.414
1.414	2	1.414	0

The *i*th column of **d2** contains the distances between the *i*th row of **x** and the other rows. Notice that the **d2** matrix has zeros along the diagonal.

You can also compute non-Euclidean distances, as follows:

```
d1 = distance(x, "L1");
dInf = distance(x, "LInfinity");
print d1, dInf;
```

Figure 25.126 Distance between Pairs of Points

d1			
0	2	2	2
2	0	2	2
2	2	0	2
2	2	2	0

dInf			
0	1	2	1
1	0	1	2
2	1	0	1
1	2	1	0

If a row contains a missing value, all distances that involve that row are assigned a missing value.

Distance between Rows of Two Matrices

You can also specify two matrices as arguments. If *x* is an $n \times d$ numerical matrix and *y* is an $m \times d$ numerical matrix, the DISTANCE function returns an $n \times m$ matrix. The (i, j) element is the distance between the *i*th row of *x* and the *j*th row of *y*. The following example computes the pairwise distances between four points in *x* and two points in *y*. The distances are shown in Figure 25.127.

```
x = {1  0,
     0  1,
    -1  0,
     0 -1};
y = {0  0,
     1  1};
dxy = distance(x, y);
print dxy[format=best5.];
```

Figure 25.127 Pairwise Euclidean Distance between Rows of Matrices

dxy	
1	1
1	1
1	2.236
1	2.236

The *j*th column of **dxy** contains the distances between the *j*th row of **y** and the rows of **x**. For example, the first column in Figure 25.127 shows that the first row of **y** (which is (0, 0)) is one unit away from every ordered pair in the rows of **x**. The second column shows that the second row of **y** (which is (1, 1)) is one unit away from the coordinates in the first two rows of **x**, but is 2.236 units away from the coordinates in the last two rows of **x**.

DO Function

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

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

The arguments to the DO function are as follows:

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

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

The following statements show examples of using the DO function:

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

Figure 25.128 Arithmetic Sequences

i					
3	6	9	12	15	18

k			
3	2	1	0

DO Statement

DO ;

statements **;**

END ;

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

For example, consider the following statements:

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

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

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

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

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

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

DO Statement, Iterative

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

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

The arguments to the DO statement are as follows:

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

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

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

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

For example, the following statements print the value of `i` three times, as shown in Figure 25.129:

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

Figure 25.129 Arithmetic Sequences

i
The value of i is: 1

i
The value of i is: 3

i
The value of i is: 5

DO DATA Statement

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

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

The arguments to the DO DATA statement are as follows:

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

The DO DATA statement is used for repetitive DO loops that need to be exited upon the occurrence of an end of file for an INPUT, READ, or other I/O statement. This form is common for loops that read data from either a sequential file or a SAS data set.

When an end of file is reached inside the DO DATA group, the program immediately jumps from the group and starts executing the statement that follows the END statement. DO DATA groups can be nested, where each end of file causes a jump from the most local DO DATA group. The DO DATA loop simulates the end-of-file behavior of the SAS DATA step. You should not use GOTO statement and the LINK statement to jump out of a DO DATA group.

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

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

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

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

Be aware that the ALL keyword does not return an end-of-file condition. Consequently, the READ ALL statement is usually not used inside a DO DATA loop. If you use the READ ALL statement inside a DO DATA loop, you should use a looping variable or a statement that returns an end-of-file condition (such as READ NEXT) in order to avoid creating an infinite loop.

DO Statement with an UNTIL Clause

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

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

The DO UNTIL statement conditionally executes statements iteratively.

The arguments to the DO UNTIL statement are as follows:

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

Using an UNTIL expression enables you to conditionally execute a set of statements iteratively. The UNTIL expression is evaluated at the bottom of the loop, and the statements inside the loop are executed repeatedly

as long as the expression yields a zero or missing value. In the following example, the body of the loop executes until the value of X exceeds 100:

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

Figure 25.130 Result of a DO-UNTIL Statement

x
101

DO Statement with a WHILE Clause

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

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

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

The arguments to the DO WHILE statement are as follows:

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

Using a WHILE expression enables you to conditionally execute a set of statements iteratively. The WHILE expression is evaluated at the top of the loop, and the statements inside the loop are executed repeatedly as long as the expression yields a nonzero or nonmissing value.

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

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

Figure 25.131 Result of a DO-WHILE Statement

x
100

The next example increments the starting value by 2:

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

Figure 25.132 Result of an Iterative DO-WHILE Statement

i	y
11	945

DURATION Function

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

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

times	is an *n*-dimensional column vector of times. The *i*th time corresponds to the time (often in years) until the *i*th cash flow occurs. Elements should be nonnegative.
flows	is an *n*-dimensional column vector of cash flows.
ytm	is the per-period yield-to-maturity of the cash-flow stream. This is a scalar and should be positive.

Duration of a security is generally defined as

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

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

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

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

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

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

The following statements call the DURATION function:

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

Figure 25.133 Duration of a Cash Flow

duration
0.9090909

ECHELON Function

ECHELON(*matrix***);**

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

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

Figure 25.134 Result of the ECHELON Function

e		
1	2	0
0	0	1
0	0	0

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

EDIT Statement

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

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

The EDIT statement can define a set of variables and the selection criteria that are used to control access to data set observations.

The VAR, WHERE, and NOBS clauses are optional. and can be specified in any order.

The arguments to the EDIT statement are as follows:

SAS-data-set specifies a SAS data set. You can specify a one-level name (for example, A) or a two-level name (for example, Sasuser.A). You can also specify an expression (enclosed in parentheses) that resolves to the name of a SAS data set. See the example for the CLOSE statement.

operand specifies a set of variables to edit. You can specify variables by using any of the methods described in the section "Select Variables with the VAR Clause" on page 105.

expression specifies observations to edit. If you omit the WHERE clause, all observations are selected. For more details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

name specifies a variable to contain the number of observations. The NOBS clause returns the total number of observations in the data set in the variable *name*.

For example, to read and update observations for which the Age variable is greater than 30, use the following statements:

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

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

Figure 25.135 Result of the LIST Statement

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

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

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

Figure 25.136 Number of Observations in a Data Set

NumObs
6

See Chapter 7 for more information about editing SAS data sets. For additional examples of using the EDIT statement, see the DELETE statement and the REPLACE statement.

EIGEN Call

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

The EIGEN subroutine computes eigenvalues and eigenvectors of an arbitrary square numeric matrix. The EIGEN subroutine will use vendor-supplied eigenvalue routines if they are available on your system. (An example is the Intel Math Kernel Library (MKL), which is tuned to provide optimal performance for a given Intel processor.) Because eigenvectors are not unique, the results of eigenvector computations that use vendor-supplied routines are not necessarily identical to the results from earlier releases. Use the RESET EIGEN93 statement to prevent SAS/IML from using vendor-supplied routines.

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

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

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

If *A* is symmetric, the eigenvalues are arranged in descending order. Otherwise, the eigenvalues are sorted first by their real parts, then by the magnitude of their imaginary parts. Complex conjugate eigenvalues, $\mathrm{Re}(\lambda) \pm i\,\mathrm{Im}(\lambda)$, are stored in standard order; that is, the eigenvalue of the pair with a positive imaginary part is followed by the eigenvalue of the pair with the negative imaginary part.

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

The following paragraphs present some properties of eigenvalues and eigenvectors. Let $\mathbf{A}$ be a general $n \times n$ matrix. The eigenvalues of $\mathbf{A}$ are the roots of the characteristic polynomial, which is defined as $p(z) = \det(z\mathbf{I} - \mathbf{A})$. The spectrum, denoted by $\lambda(A)$, is the set of eigenvalues of the matrix A. If $\lambda(\mathbf{A}) = \{\lambda_1, \ldots, \lambda_n\}$, then $\det(\mathbf{A}) = \lambda_1\lambda_2\cdots\lambda_n$.

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

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

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

An eigenvector is a nonzero vector, $\mathbf{x}$, 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 $\mathbf{x}$. 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 $\mathbf{A}$ is square but not necessarily symmetric:

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

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

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

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

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

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

If $\mathbf{A}$ is a *symmetric* matrix and $\mathbf{M}$ and $\mathbf{E}$ are the eigenvalues and eigenvectors, respectively, of $\mathbf{A}$, then the matrices have the following properties:

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

These properties imply the following:

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

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

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

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

```
F = root(einv);
A = F*H*F`;
call eigen(L, W, A);
V = F`*W;
```

The computation can be checked by forming the residuals, **r**, as shown in the following statement:

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

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

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

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

The sorted eigenvalues of the **A** matrix are shown in Figure 25.137.

Figure 25.137 Complex Eigenvalues of a Nonsymmetric Matrix

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

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

```
/* verify that the right eigenvectors are correct */
vec = rvec;
do j = 1 to ncol(vec);
  /* if eigenvalue is real */
  if val[j,2] = 0. then do;
    v = A * vec[,j] - val[j,1] * vec[,j];
    if any( abs(v) > 1e-12 ) then
      badVectors = badVectors || j;
    end;
  /* if eigenvalue is complex with positive imaginary part */
  else if val[j,2] > 0. then do;
    /* the real part */
```

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

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

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

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

EIGVAL Function

EIGVAL(*A*);

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

The EIGVAL function uses vendor-supplied eigenvalue routines if they are available on your system. Use the RESET EIGEN93 statement to prevent SAS/IML from using vendor-supplied routines.

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

```
A = {  67.00  177.60  -63.20 ,
      -20.40   95.88  -87.16 ,
       22.80   67.84   12.12 };
val = eigval(A);
print val;
```

Figure 25.138 Eigenvalues

val	
75	100
75	-100
25	0

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

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

```
A = {4 10, 10 30};      /* A is a symmetric matrix */
rval = eigval(A);
print rval;
```

Figure 25.139 Real Eigenvalues of a Symmetric Matrix

rval
33.401219
0.5987805

EIGVEC Function

EIGVEC(*A*);

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

The EIGVEC function uses vendor-supplied eigenvalue routines if they are available on your system. Because eigenvectors are not unique, the results of eigenvector computations that use vendor-supplied routines are not necessarily identical to the results from earlier releases. Use the RESET EIGEN93 statement to prevent SAS/IML from using vendor-supplied routines.

The following example calculates the eigenvectors of a symmetric matrix:

```
A = {4 10, 10 30};      /* A is a symmetric matrix */
evec = eigvec(A);
print evec;
```

Figure 25.140 Eigenvectors of a Symmetric Matrix

evec	
0.3220062	-0.946738
0.9467376	0.3220062

ELEMENT Function

ELEMENT(*x, y*);

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

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

The arguments are as follows:

x	specifies a matrix of elements to test for membership.
y	specifies a set.

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

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

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

Figure 25.141 Elements That Belong to a Set

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

END Statement

END ;

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

ENDSUBMIT Statement

ENDSUBMIT ;

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

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

The ENDSUBMIT statement must appear on a line by itself. There cannot be any space between the statement and the trailing semicolon.

See Chapter 14, "Calling Functions in the R Language," and the documentation for the SUBMIT statement for details and examples.

EXECUTE Call

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

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

The EXECUTE subroutine pushes character arguments to the input command stream, executes them, and then returns to the calling module. The subroutine should be called from a module rather than from the immediate environment because it uses the RESUME statement that works only from modules. The strings you push do not appear in the log.

Following are examples of valid EXECUTE subroutines:

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

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

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

Figure 25.142 Results of Executing SAS/IML Statements

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

For more details about the EXECUTE subroutine, see Chapter 20, "Using SAS/IML Software to Generate SAS/IML Statements."

EXECUTEFILE Call

CALL EXECUTEFILE(*filename* < , *encoding* >**);**

The EXECUTEFILE subroutine executes SAS/IML statements that are contained in a text file, which is called the *source file*. The statements in the source file can be SAS/IML statements or global SAS statements such as the TITLE statement. The statements in the source file do not appear in the SAS log.

The first argument to the EXECUTEFILE subroutine is a string literal, a character matrix that contains a valid file name, or a SAS fileref that points to a valid file. The argument can refer to an absolute path such as **`C:\Temp\commands.sas`** or a relative path such as **`.\commands2.sas`**. Relative paths depend on the current working directory for the SAS session.

The EXECUTEFILE subroutine accepts an optional second argument that specifies the encoding of the source file. You can use this argument when the encoding of the source file is different from the SAS session encoding. If you omit the second argument, an attempt is made to deduce the encoding of the source file automatically. SAS encoding strings and their usage are described in the chapter "Encoding Values in SAS Language Elements" in the *SAS National Language Support (NLS): Reference Guide*. Examples of valid values for the second argument include "UTF-8," "us-ascii," "latin2," and "ms-949."

The following DATA step creates a file called *commands.sas* in the current working directory:

```
filename ExeFile "./commands.sas";
data _null_;
   file ExeFile;
   put 'start MySqr(t);    ';
   put '   return( t##2 );';
   put 'finish;            ';
   put 'x = {1 2, 3 4};    ';
run;
```

The file *commands.sas* contains SAS/IML statements. It is created in the current working directory, assuming that you have permission to write to that directory. The statements define a matrix (**`x`**) and a module named MYSQR. After you use the EXECUTEFILE subroutine to execute the statements in the source file, you can refer to the matrix and call the module, as shown in the following example:

```
proc iml;
call executefile("commands.sas");
y = MySqr(x);
print x y;
```

Figure 25.143 Results of Executing SAS/IML Statements from a Source File

x		y	
1	2	1	4
3	4	9	16

Figure 25.143 shows that the statements in the source file were executed. The matrices and the module are available for use in subsequent SAS/IML statements.

In the preceding example, the statements that reference the **x** matrix are in the same calling environment (scope) as the EXECUTEFILE statement. In the following program, the EXECUTEFILE subroutine is called from inside a module:

```
proc iml;
start MyMod(a);
   call executefile("commands.sas");
   /* x is defined inside the module, but not outside */
   print x[label="x inside module"];
finish;
call MyMod(1);

show modules names;
```

Figure 25.144 Modules and Names Known at Main Scope

x inside module	
1	2
3	4

```
Modules:
  MYMOD                                 MYSQR


 SYMBOL   ROWS   COLS TYPE   SIZE
 ------ ------ ------ ---- ------
  Number of symbols = 0  (includes those without values)
```

Notice in Figure 25.144 that the matrix **x** is not known outside the module. However, the MYSQR module is known because all modules are defined at the main scope, as discussed in the section "Nesting Module Definitions" on page 75.

The EXECUTEFILE subroutine is similar to the %INCLUDE macro statement for including source code into a SAS program. However, the EXECUTEFILE subroutine parses and runs the source file at run time, whereas the %INCLUDE statement inserts the contents of a source file at parse time. One advantage of the EXECUTEFILE statement is that the name of the file does not need to be known until run time. In fact, the

file itself does not need to exist until run time, as demonstrated by the DATA step in this section that creates the source file.

Some SAS/IML statements are not supported by the EXECUTEFILE subroutine. The file should not contain the PROC IML statement, the QUIT statement, or other statements that terminate the IML procedure.

In SAS/IML 14.1, there are differences between the implementation of the EXECUTEFILE statement in PROC IML and in the IMLPlus language:

- In PROC IML, the source file should not contain the SUBMIT and ENDSUBMIT statements. An alternative is to use the %INCLUDE statement to include SUBMIT blocks.
- In IMLPlus, filerefs are not supported.
- In IMLPlus, modules that are defined inside a source file are not recognized by the parser until after the EXECUTEFILE subroutine is run.

If a statement in the source file contains an error, an error message appears in the SAS log. However, PROC IML concatenates the source file into a single text string, so the error is always reported as being on "Line 1." For some kinds of errors, the SAS log displays the message **`NOTE: Paused in module _EXECUTEFILE`**. If you see this note, submit the RESUME statement to restore program control to the main scope.

EXP Function

EXP(*matrix***);**

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

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

Figure 25.145 Exponential of Several Numbers

a			
2.7182818	7.3890561	20.085537	54.59815

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

EXPMATRIX Function

EXPMATRIX(*matrix***);**

The EXPMATRIX function is part of the IMLMLIB library. Given an $n \times n$ 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).

Note that this module does not exponentiate each element 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 $\begin{pmatrix} e^t & te^t \\ 0 & e^t \end{pmatrix}$. You can compute the exponential matrix as follows:

```
A = { 1 1, 0 1 };
t = 3;
X = ExpMatrix( t*A );
ExactAnswer = ( exp(t) || t*exp(t) ) //
( 0      ||   exp(t) );
print X, ExactAnswer;
```

Figure 25.146 Matrix Exponential

X	
20.085537	60.256611
0	20.085537

ExactAnswer	
20.085537	60.256611
0	20.085537

EXPANDGRID Function

EXPANDGRID(*x1, x2* **<,** *x3***> . . . <,** *x15***>);**

The EXPANDGRID function is part of the IMLMLIB library. The arguments to the EXPANDGRID function are k vectors, $2 \leq k \leq 15$. The EXPANDGRID function returns a matrix that contains the Cartesian product of elements from the specified vectors. If the ith argument has n_i elements, the return matrix has $\Pi_{i \leq k} n_i$ rows and k columns.

Each row of the result contains a combination of elements of the input vectors. The first row contains the elements (x1[1], x2[1],. . . , xk[1]). The second row contains the elements (x1[1], x2[1],. . . , xk[2]). The first column varies the slowest, and the last column varies the fastest.

The following statement create a matrix of 0s and 1s. Each row is a vertex of the three-dimensional unit cube.

```
g = ExpandGrid(0:1, 0:1, 0:1);
print g;
```

Figure 25.147 A Cartesian Product of Three Vectors

g		
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

You can use the EXPANDGRID function to generate a complete factorial design from the set of factors. You can also use it to evaluate a multivariate function on a dense grid of points. For example, the following statements evaluate the bivariate cubic polynomial $f(x, y) = x^3 - y^2 - 2x + 1$ on a grid of points in the region $[-2, 2] \times [-2, 2]$:

```
vx = do(-2, 2, 0.1);
vy = do(-2, 2, 0.1);
g = ExpandGrid(vx, vy);  /* grid on [-2,2] x [-2,2] */
x = g[,1]; y = g[,2];
z = x##3 - y##2 - 2#x + 1;
```

EXPORTDATASETTOR Call

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

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

The arguments to the subroutine are as follows:

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

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

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

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See the section "The RLANG System Option" on page 244.)

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

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

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

To demonstrate that the data were successfully transferred, you can use the **names** function in the R language to print the names of the variables in the R data frame. The output is shown in Figure 25.148.

Figure 25.148 Output from R

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

You can transfer data from an R data frame into a SAS data set by using the IMPORTDATASETFROMR call. For more information about transferring data between R and SAS software, see Chapter 14, "Calling Functions in the R Language."

EXPORTMATRIXTOR Call

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

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

The arguments to the subroutine are as follows:

IMLMatrix	is a SAS/IML matrix that contains the data you want to transfer.
RMatrix	is a literal string or a character matrix that specifies the name of an R matrix to contain a copy of the data.

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

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See the section "The RLANG System Option" on page 244.)

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

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

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

To demonstrate that the data were successfully transferred, the **print** function in the R language is used to print the values of the **m** matrix. The output is shown in Figure 25.149. Note that the SAS missing value in the SAS/IML matrix was automatically converted to the R missing value (**NA**).

Figure 25.149 Output from R

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

You can transfer data from an R matrix into a SAS/IML matrix by using the IMPORTMATRIXFROMR call. See Chapter 14, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

EXPORTTABLETOR Call

CALL EXPORTTABLETOR(*table, RDataFrame***);**

You can use the EXPORTTABLETOR subroutine to transfer data from a SAS/IML table to an R data frame. It is easier to read the subroutine name when it is written in mixed case: ExportTableToR.

The arguments to the subroutine are as follows:

table	is a SAS/IML table that contains the data you want to transfer.
RDataFrame	is a literal string or a character matrix that specifies the name of an R data frame.

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

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See the section "The RLANG System Option" on page 244.)

The following statements copy data from the **tbl** table into an R data frame called **RTbl**:

```
proc iml;
C = {"Alonzo" "M",  "Juanita" "F",  "Wei" "M"};
tbl = TableCreate({"Name" "Sex"}, C);       /* create table */
X = { 14 71,  16 78,  15 66};
call TableAddVar(tbl, {"Age" "Height"}, X); /* add columns */
call ExportTableToR(tbl, "RTbl");

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

To demonstrate that the data were successfully transferred, you can use the **names** function in the R language to print the names of the variables in the R data frame. The output is shown in Figure 25.150.

Figure 25.150 Output from R

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

You can transfer data from an R data frame into a SAS/IML table by using the IMPORTTABLEFROMR function. For more information about transferring data between R and SAS software, see Chapter 14, "Calling Functions in the R Language."

FARMACOV Call

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

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

The input arguments to the FARMACOV subroutine are as follows:

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

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

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

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

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

For example, consider *phi*=0.5.

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

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

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

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

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

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

The FARMACOV subroutine returns the following value:

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

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

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

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

```
d = 0.3;
phi = 0.5;
theta = -0.1;
sigma = 1.2;
call farmacov(cov, d, phi, theta, sigma) lag=5;
print cov;
```

Figure 25.151 Autocovariance of an ARFIMA Process

cov
4.2493033
3.5806774
2.9152846
2.4381017
2.1068697
1.8743199

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

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

and the autocorrelation function

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

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

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

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

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

Let $x_t = \theta(B)^{-1}\phi(B) y_t$, so that x_t follows an ARFIMA$(0, d, 0)$ process; let $z_t = (1-B)^d y_t$, so that z_t follows an ARMA(p, q) process; let γ_k^x be the autocovariance function of $\{x_t\}$ and γ_k^z be the autocovariance function of $\{z_t\}$.

Then the autocovariance function of $\{y_t\}$ is as follows:

$$\gamma_k = \sum_{j=-\infty}^{j=\infty} \gamma_j^z \gamma_{k-j}^x$$

The explicit form of the autocovariance function of $\{y_t\}$ is given by Sowell (1992).

FARMAFIT Call

CALL FARMAFIT(*d, phi, theta, sigma, series* **<** *, p* **>** **<** *, q* **>** **<** *, opt* **>** **);**

The FARMAFIT subroutine estimates the parameters of an ARFIMA(p, d, q) model.

The input arguments to the FARMAFIT subroutine are as follows:

series specifies a time series (assuming mean zero).

p specifies the set or subset of the AR order. If you do not specify *p*, the default is *p*=0.

If you specify *p*=3, the FARMAFIT subroutine estimates the coefficient of the lagged variable y_{t-3}.

If you specify *p*={1, 2, 3}, the FARMAFIT subroutine estimates the coefficients of lagged variables y_{t-1}, y_{t-2}, and y_{t-3}.

q specifies the subset of the MA order. If you do not specify *q*, the default value is 0.

If you specify *q*=2, the FARMAFIT subroutine estimates the coefficient of the lagged variable ϵ_{t-2}.

If you specify *q*={1, 2}, the FARMAFIT subroutine estimates the coefficients of lagged variables ϵ_{t-1} and ϵ_{t-2}.

opt specifies the method of computing the log-likelihood function.

0 requests the conditional sum of squares function. This is the default.

1 requests the exact log-likelihood function. This option requires that the time series be stationary and invertible.

The FARMAFIT subroutine returns the following values:

d is a scalar that contains a fractional differencing order.

phi is a vector that contains the autoregressive coefficients.

theta is a vector that contains the moving average coefficients.

sigma is a scalar that contains a variance of the innovation series.

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

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

In this model, $\epsilon_t \sim \text{NID}(0, 1)$. The following statements estimate the parameters of this model:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
call farmasim(yt, d, phi, theta) seed=1234;
call farmafit(d, ar, ma, sigma, yt) p=1 q=1;
print d ar ma sigma;
```

Figure 25.152 Parameter Estimates for a ARFIMA Model

d	ar	ma	sigma
0.3950157	0.5676217	-0.012339	1.2992989

The FARMAFIT subroutine estimates the parameters d, $\phi(B)$, $\theta(B)$, and σ_ϵ^2 of an ARFIMA(p, d, q) model. The log-likelihood function is solved by iterative numerical procedures such as the quasi-Newton optimization. The starting value d is obtained by the approach of Geweke and Porter-Hudak (1983); the starting values of the AR and MA parameters are obtained from the least squares estimates.

FARMALIK Call

CALL FARMALIK(*lnl, series, d* **<** *, phi* **>** **<** *, theta* **>** **<** *, sigma* **>** **<** *, p* **>** **<** *, q* **>** **<** *, opt* **>** **);**

The FARMALIK subroutine evaluates the log-likelihood function of an ARFIMA(p, d, q) model for a given time series.

The input arguments to the FARMALIK subroutine are as follows:

series specifies a time series (assuming mean zero).

d specifies a fractional differencing order. This argument is required; the value of d should be in the open interval $(-1, 1)$ excluding zero.

phi specifies an m_p-dimensional vector that contains the autoregressive coefficients, where m_p is the number of the elements in the subset of the AR order. The default is zero.

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

sigma specifies a variance of the innovation series. The default is one.

p specifies the subset of the AR order. See the FARMACOV subroutine for additional details.

q specifies the subset of the MA order. See the FARMACOV subroutine for additional details.

opt specifies the method of computing the log-likelihood function. The following are valid values:

- 0 requests the conditional sum of squares function. This is the default.
- 1 requests the exact log-likelihood function. This option requires that the time series be stationary and invertible.

The FARMALIK subroutine returns the following value:

lnl is a three-dimensional vector. If *opt*=0 is specified, the conditional sum of squares function is evaluated and the result returns in `lnl[1]`. Otherwise, `lnl[1]` contains the log-likelihood function of the model; `lnl[2]` contains the sum of the log determinant of the innovation variance; and `lnl[3]` contains the weighted sum of squares of residuals. The log-likelihood function is computed as $-0.5\times$ (`lnl[2]+lnl[3]`).

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

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

In this model, $\epsilon_t \sim \text{NID}(0, 1.2)$. The following statements compute the log-likelihood function of this model:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
sigma = 1.2;
call farmasim(yt, d, phi, theta, sigma) seed=1234;
call farmalik(lnl, yt, d, phi, theta, sigma);
print (lnl[1])[label="Conditional Sum of Squares"];
```

Figure 25.153 Log-Likelihood for an ARFIMA Model

Conditional Sum of Squares
-16.67587

The FARMALIK subroutine computes a log-likelihood function of the ARFIMA(p, d, q) model. The exact log-likelihood function was proposed by Sowell (1992); the conditional sum of squares function was proposed by Chung (1996).

The exact log-likelihood function only considers a stationary and invertible ARFIMA(p, d, q) process with $d \in (-0.5, 0.5)\backslash\{0\}$ represented as

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

where $\epsilon_t \sim \text{NID}(0, \sigma^2)$.

Let $Y_T = [y_1, y_2, \ldots, y_T]'$ and the log-likelihood function is as follows without a constant term:

$$\ell = -\frac{1}{2}(\log|\Sigma| + Y_T'\Sigma^{-1}Y_T)$$

where $\Sigma = \left[\gamma_{i-j}\right]$ for $i, j = 1, 2, \ldots, T$.

The conditional sum of squares function does not require the normality assumption. The initial observations $y_0, y_{-1}, \ldots$ and $\epsilon_0, \epsilon_{-1}, \ldots$ are set to zero.

Let y_t be an ARFIMA(p, d, q) process represented as

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

Then the conditional sum of squares function is

$$\ell = -\frac{T}{2}\log\left(\frac{1}{T}\sum_{t=1}^{T}\epsilon_t^2\right)$$

FARMASIM Call

CALL FARMASIM(*series, d* **<,** *phi***> <,** *theta***> <,** *mu***> <,** *sigma***> <,** *n***> <,** *p***> <,** *q***> <,** *initial***> <,** *seed***>);**

The FARMASIM subroutine generates an ARFIMA(p, d, q) process. The input arguments to the FARMASIM subroutine are as follows:

d specifies a fractional differencing order. This argument is required; the value of d should be in the open interval $(-1, 1)$ excluding zero.

phi specifies an m_p-dimensional vector that contains the autoregressive coefficients, where m_p is the number of the elements in the subset of the AR order. The default is zero.

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

mu specifies a mean value. The default is zero.

sigma specifies a variance of the innovation series. The default is one.

n specifies the length of the series. The value of n should be greater than or equal to the AR order. The default is $n = 100$ is used.

p specifies the subset of the AR order. See the FARMACOV subroutine for additional details.

q specifies the subset of the MA order. See the FARMACOV subroutine for additional details.

initial specifies the initial values of random variables. The initial value is used for the nonstationary process. If *initial*$= a_0$, then $y_{-p+1}, \ldots, y_0$ take the same value a_0. If the *initial* option is not specified, the initial values are set to zero.

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

If *seed* > 0, the input seed is used for generating the series.

If *seed* = 0, the system clock is used to generate the seed.

If *seed* < 0, the value $(-1)\times(\textit{seed})$ is used for generating the series.

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

On subsequent calls of the subroutine in the DO-loop-like environment, the seed variable is used as follows: If seed > 0, the seed remains unchanged. In other cases, after each execution of the subroutine, the current seed is updated internally.

The FARMASIM subroutine returns the following value:

series is an n vector that contains the generated ARFIMA(p, d, q) process.

As an example, consider the following ARFIMA$(1, 0.3, 1)$ process:

$$(1 - 0.5B)(1 - B)^{0.3}(y_t - 10) = (1 + 0.1B)\epsilon_t$$

In this process, $\epsilon_t \sim \text{NID}(0, 1.2)$. The following statements generate this process:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
mu    = 10;
sigma = 1.2;
call farmasim(yt, d, phi, theta, mu, sigma, 10) seed=1234;
print yt;
```

Figure 25.154 Data Simulated from a ARFIMA Process

yt
12.17358
13.954495
15.817231
15.94882
12.25926
13.641022
13.399623
11.930759
10.049435
9.1445036

The FARMASIM subroutine generates a time series of length n from an ARFIMA(p, d, q) model. If the process is stationary and invertible, the initial values $y_{-p+1}, \ldots, y_0$ are produced by using covariance matrices obtained from FARMACOV. If the process is nonstationary, the time series is recursively generated by using the user-defined initial value or the zero initial value.

To generate an ARFIMA(p, d, q) process with $d \in [0.5, 1)$, x_t is first generated for $d' \in (-0.5, 0)$, where $d' = d - 1$ and then y_t is generated by $y_t = y_{t-1} + x_t$.

To generate an ARFIMA(p, d, q) process with $d \in (-1, -0.5]$, a two-step approximation based on a truncation of the expansion $(1 - B)^d$ is used; the first step is to generate an ARFIMA$(0, d, 0)$ process $x_t = (1 - B)^{-d}\epsilon_t$, with truncated moving average weights; the second step is to generate $y_t = \phi(B)^{-1}\theta(B)x_t$.

FDIF Call

CALL FDIF(*out, series, d***);**

The FDIF subroutine computes a fractionally differenced process. The input arguments to the FDIF subroutine are as follows:

series specifies a time series with n length.

d specifies a fractional differencing order. This argument is required; the value of d should be in the open interval $(-1, 1)$ excluding zero.

The FDIF subroutine returns the following value:

out is an n vector that contains the fractionally differenced process.

As an example, consider an ARFIMA$(1, 0.3, 1)$ process

$$(1 - 0.5B)(1 - B)^{0.3} y_t = (1 + 0.1B)\epsilon_t$$

Let $z_t = (1 - B)^{0.3} y_t$; that is, z_t follows an ARMA(1,1). The following statements compute the filtered series z_t:

```
d = 0.3;
phi = 0.5;
theta = -0.1;
call farmasim(yt, d, phi, theta) n=10 seed=1234;
call fdif(zt, yt, d);
print zt;
```

Figure 25.155 A Fractionally Differenced Process

zt
.
3.0146839
4.0190575
3.3402864
-0.41881
1.6149336
1.1998534
-0.137789
-1.475051
-1.670366

FEVAL Function

FEVAL(*funcname, < argument1, . . . , argument14 >* **);**

The FEVAL function enables you to evaluate a function indirectly. You specify the name of the function and its arguments. The FEVAL function returns the result of calling the function with the specified arguments.

The arguments to the FEVAL function are as follows:

funcname specifies the name of an existing built-in function or user-defined function. You can specify the function name as a literal string or as a matrix that contains the function name.

argument specifies an argument that is passed to the function. You can specify up to 14 arguments. These arguments can be of any dimension and any type (character, numeric, lists, and so on).

For example, the following program uses the FEVAL function to evaluate the MEAN function on a column vector of data:

```
y = {0,1,2,4,4,5,6,6,17};
m = FEval("mean", y);  /* same as MEAN(y) */
```

For this example, calling the FEVAL function offers little advantage over calling the MEAN function directly. However, the next example uses a DO loop to evaluate six different functions that compute descriptive statistics. The names of the functions are contained in a character vector. Each function has the syntax *funcname*(*y*). The six descriptive statistics are shown in Figure 25.156.

```
stats = {"mean" "median" "std" "var" "skewness" "kurtosis"};
results = j(1, ncol(stats), .);
do i = 1 to ncol(stats);
   results[i] = FEval(stats[i], y);
end;

print results[colname=stats format=best5.];
```

Figure 25.156 Descriptive Statistics from Calling Six Functions

results					
mean	median	std	var	skewness	kurtosis
5	4	4.975	24.75	1.974	4.918

FFT Function

FFT(*x*);

The FFT function computes the finite Fourier transform. The argument *x* is a $1 \times n$ or $n \times 1$ numeric vector. The FFT function returns the cosine and sine coefficients for the expansion of a vector into a sum of cosine and sine functions. This is an $np \times 2$ matrix, where

$$np = \text{floor}\left(\frac{n}{2} + 1\right)$$

The elements of the first column of the returned matrix are the cosine coefficients; that is, the *i*th 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 1977).

The FFT function can be used to compute the periodogram of a time series. In conjunction with the inverse finite Fourier transform routine IFFT, the FFT function can be used to efficiently compute convolutions of large vectors (Gentleman and Sande 1966; Nussbaumer 1982).

As an example, suppose you measure a signal at constant time intervals. You believe the signal consists of a small number of Fourier components (that is, sines and cosines) corrupted by noise. The following examples uses the FFT function to transform the signal into the frequency domain. The example then prints the frequencies with the largest amplitudes in the signal. According to this analysis, the signal is primarily composed of a constant signal, a signal with frequency 4 (for example, $A\cos(4t) + B\sin(4t)$), a signal with frequency 1, and a signal with frequency 3. The amplitudes of the remaining Fourier components, are all substantially smaller.

```
Signal = {
 1.96  1.45  0.86  0.46  0.39  0.54 -1.65  0.60  0.43  0.20
-1.15  1.10  0.42  3.22  2.02  3.41  3.46  3.51  4.33  4.38
 3.92  4.35  2.60  3.95  4.72  4.84  1.62  0.97  0.96  1.10
 2.53  1.09  2.84  2.51  2.38  2.40  2.76  3.42  3.78  4.08
 3.84  5.62  4.33  6.66  5.27  3.14  3.82  5.74  3.45  1.07
 0.31  2.07  0.49 -1.85  0.61  0.35 -0.89 -0.92  0.33  2.31
 1.13  2.28  3.73  3.78  2.63  4.15  5.27  3.62  5.99  3.79
 4.00  3.18  3.03  3.52  2.08  1.70 -1.50 -1.35 -0.34 -1.52
-2.37 -2.84 -1.68 -2.22 -2.49 -3.28 -2.12 -0.81  0.84  1.91
 2.10  2.24  1.24  3.24  2.89  3.14  4.21  2.65  4.67  3.87
 }`;

z = fft(Signal);
Amplitude = z[,1]##2 + z[,2]##2;

/* find index into Amplitude so that idx[1] is the largest
   value, idx[2] is the second largest value, etc. */
call sortndx(idx,Amplitude,1,1);

/* print the 10 most dominant frequencies */
Amplitude = Amplitude[idx[1:10],];
print (idx[1:10]-1)[label="Freqs"] Amplitude[format=10.2];
```

Figure 25.157 Frequencies and Amplitudes of a Signal

Freqs	Amplitude
0	38757.80
4	13678.28
1	4077.99
3	2726.76
26	324.23
44	269.48
12	224.09
20	217.35
11	202.30
23	201.05

Based on these results, you might choose to filter the signal to keep only the most dominant Fourier components. One way to accomplish this is to eliminate any frequencies with small amplitudes. When the truncated frequencies are transformed back by using IFFT, you obtain a filtered version of the original signal. The following statements perform these tasks:

```
/* based on amplitudes, keep only first few dominant frequencies */
NumFreqs = 4;
FreqsToDrop = idx[(NumFreqs+1):nrow(idx)];
z[FreqsToDrop,] = 0;

FilteredSignal = ifft(z) / nrow(Signal);
```

FFTC Function

FFTC(*x*);

The FFTC function computes the finite Fourier transform of a complex vector, where *x* is an $n \times 2$ (or $2 \times n$) numeric matrix that represents a complex vector. The first column (or row) of *x* specifies the real components, and the second column (or row) specifies the imaginary components. (If *x* is a 2×2 matrix, the first row contains the real components, and the second row contains the imaginary components.)

The FFTC function returns an $n \times 2$ matrix that contains the complex Fourier coefficients that correspond to *x*. The first column of the matrix contains the real component of the Fourier coefficients, and the second column contains the imaginary components.

The Fourier coefficients can be expressed in terms of a complex matrix multiplication. If the input *x* represents a complex column vector z of dimension n and if $\omega = \exp(2\pi \mathbf{i}/n)$, where $\mathbf{i} = \sqrt{-1}$, then define an $n \times n$ matrix W as

$$W = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

Then the complex vector of Fourier coefficients is simply the product Wz.

The following example demonstrates the FFTC function:

```
/* sample every millisecond from 0 to 10 seconds */
t = do(0,10,0.001);
pi = constant("pi");
/* complex signal takes values on the unit circle and makes
   three full rotations per second in the *clockwise* direction */
freq = 3;
x = cos(2*pi*freq*t); y = sin(-2*pi*freq*t);
Signal = t(x) || t(y);

f = fftc(Signal);

/* find the three most dominant frequencies */
power = f[,##];
call sortndx(idx,power,1,1);

/* convert idx to hertz; expect the dominant frequencies to be near 3 */
freq = (idx[1:3]-1)/10;
print freq[l="Freq(Hertz)"] (power[idx[1:3]])[l="Power"];
```

Figure 25.158 Dominant Frequencies in a Complex Signal

Freq(Hertz)	Power
3	100017040
3.1	905.57871
2.9	894.77667

FILE Statement

FILE *filename* <**RECFM**=*N*> <**LRECL**=*operand*> **;**

The FILE statement opens an external file for output.

The arguments to the FILE statement are as follows:

filename	is a name (for defined file names), a quoted literal, or an expression in parentheses (for path names).
RECFM=N	specifies that the file be written as a pure binary file without record-separator characters.
LRECL=*operand*	specifies the record length of the output file. The default record length is 512.

You can use the FILE statement to open a file for output, or if the file is already open, to make it the current output file so that subsequent PUT statements write to it. The FILE statement is similar in syntax and operation to the INFILE statement. The FILE statement is described in detail in Chapter 8.

The *filename* argument is either a predefined file name or a quoted string or character expression in parentheses referring to the path name. You can refer to an input or output file two ways: by a path name or by a file name. The path name is the name as known to the operating system. The file name 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 file name as the operand, just give the

name. The name must be one already connected to a path name by a previously issued FILENAME statement. However, two special file names are recognized by the SAS/IML language: LOG and PRINT. These refer to the standard output streams for all SAS sessions. To specify a path name, enclose it in quotes or specify an expression in parentheses that yields the path name.

When the path name is specified, the operand is limited to 64 characters.

Note that RECFM=U is equivalent to RECFM=N. If an output file is subsequently read by a SAS DATA step, RECFM=N must be specified in the DATA step to guarantee that the file is read properly.

Following are several valid uses of FILE statement:

```
file "student.dat";           /* by literal path name   */

filename out "student.dat";   /* specify file name OUT  */
file out;                     /* refer to by file name  */

file print;                   /* standard print output */
file log;                     /* output to log         */

file "student.dat" recfm=n;   /* for a binary file     */
```

FIND Statement

FIND <*range*> <**WHERE***(expression)*> **INTO** *matrix-name* **;**

The FIND statement finds the observation numbers in *range* that satisfy the conditions of the WHERE clause. The FIND statement places these observation numbers in the numeric matrix whose name follows the INTO keyword.

The arguments to the FIND statement are as follows:

range specifies a range of observations. You can specify a range of observations by using the ALL, CURRENT, NEXT, AFTER, and POINT keywords, as described in the section "Process a Range of Observations" on page 104.

expression specifies a criterion by which certain observations are selected. The optional WHERE clause conditionally selects observations that are contained within the *range* specification. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

matrix-name names a matrix to contain the observation numbers.

The following statements are valid examples of the FIND statement:

```
use Sashelp.Class;
find all  where(name=:"J") into p;
find point (10:18) where(age>14) into p2;
print p, p2;
close Sashelp.Class;
```

The column vectors **p** and **p2** contain the observation numbers that satisfy the WHERE clause in the given range, as shown in Figure 25.159. The default range is all observations.

Figure 25.159 Finding Observations

p
6
7
8
9
10
11
12

p2
14
15
17

FINISH Statement

FINISH < *module-name* > **;**

The FINISH statement signals the end of a module and the end of module definition mode. Optionally, the FINISH statement can take the module name as its argument. See the description of the START statement and consult Chapter 6 for further information about defining modules.

Some examples follow:

```
start myAdd(a,b);
   return (a+b);
finish;

start mySubtract(a,b);
   return (a-b);
finish mySubtract;

r = myAdd(5, 3);
s = mySubtract(5, 3);
print r s;
```

Figure 25.160 Results of Calling Modules

r	s
8	2

FORCE Statement

The FORCE statement is an alias for the SAVE statement.

FORWARD Function

FORWARD(*times, spot_rates***);**

The FORWARD function computes a column vector of (per-period) forward rates, given vectors of spot rates and times. The arguments to the function are as follows:

times	is an $n \times 1$ column vector of times in consistent units. Elements should be nonnegative.
spot_rates	is an $n \times 1$ column vector of corresponding (per-period) spot rates. Elements should be positive.

The FORWARD function transforms the given spot rates as

$$f_1 = s_1$$

$$f_i = \left(\frac{(1+s_i)^{t_i}}{(1+s_{t_{i-1}})^{t_{i-1}}} \right)^{\frac{1}{t_i - t_{i-1}}} - 1; \quad i = 2, \ldots, n$$

For example, the following statements compute forward rates:

```
time = T(do(1, 5, 1));
spot = T(do(0.05, 0.09, 0.01));
forward = forward(time, spot);
print forward;
```

Figure 25.161 Forward Rates

forward
0.05
0.0700952
0.0902839
0.1105642
0.1309345

FREE Statement

FREE *matrices* **;**

FREE */* < *keep-matrices* > **;**

The FREE statement releases memory associated with matrices. The matrices specified in the FREE statement lose their values; the memory becomes available for other uses. After the FREE statement executes, the matrix is empty. The NROW function and the NCOL function return 0. However, any printing attributes (assigned by the MATTRIB statement) are not released.

The FREE statement is used mostly in large applications or under tight memory constraints to make room for more data (matrices) in the workspace.

For example, the following statement frees the matrices **a**, **b**, and **c**:

```
free a b c;
```

If you want to free all matrices, specify a slash (/) after the keyword FREE. If you want to free all matrices except a few, then list the ones you do not want to free after the slash. For example, the following statement frees all matrices except **d** and **e**:

```
free / d e;
```

For more information, see the discussion of workspace storage in Chapter 24.

FROOT Function

FROOT(*"fun", bounds* < *, opt* > **);**

The FROOT function finds zeros of the univariate function *"fun"* by using Brent's numerical root-finding method (Brent 1973; Moler 2004). Brent's method uses a combination of bisection, linear interpolation, and quadratic interpolation to converge to a root when given an interval in which the function changes signs.

The arguments are as follows:

"fun" is the name of a SAS/IML function module. The module defines the function whose roots you want to compute. The module takes one argument and returns a scalar value. You can use a GLOBAL statement to pass parameters to *"fun"*.

bounds is an $n \times 2$ matrix. Each row of *bounds* specifies an interval in which the function changes sign. This implies that there is a root inside the interval. The return value of FROOT is an $n \times 1$ vector, where the ith element contains the root in the interval *bounds*[i,].

opt is an optional vector that contains three elements. Each element specifies a parameter that controls the convergence of Brent's algorithm. A missing value specifies that the algorithm should use the default parameter value. The parameters are as follows:

opt[1] specifies the maximum number of iterations used to search for a root. The default value is 100.

opt[2] specifies a tolerance that determines how close the computed root is to the true root. The default value is machine epsilon, which on many computers is approximately 2.2×10^{-16}.

opt[3] specifies a tolerance that determines how close the function at the computed root is to zero. The default value is machine epsilon.

Brent's algorithm starts with an interval in which the function changes signs. At each step, the algorithm computes a smaller interval in which the function also changes signs. (If each interval is half the sign of the previous, this is the bisection method.) The algorithm stops when one of the following conditions is met:

- The algorithm has performed *opt[1]* iterations.
- The bounding interval $[a, b]$ is sufficiently small. If ϵ is the value of *opt[2]*, then the algorithm stops when $\|b - a\| \leq 4\epsilon \max(\|b\|, 1)$.

- The function that is evaluated on the interval is sufficiently small. If δ is the value of *opt[3]*, then the algorithm stops when $\| f(b) \| \leq \delta$.

The following program defines a cubic function that has three roots. The roots are contained in the intervals $[-2, 0]$, $[0, 1]$, and $[1, 2]$, as shown by computing the function at the endpoints of these intervals and noticing that the function changes signs on each interval.

```
start Func(x);
   return( 2 - 3#x - 1#x##2 + x##3 );
finish;

bounds = {-2 0,
           0 1,
           1 2 };
fBounds = Func(bounds[,1]) || Func(bounds[,2]);
print bounds fBounds;
roots = froot( "Func", bounds);
print roots;
```

Figure 25.162 Bounding Intervals and Roots

bounds		fBounds	
-2	0	-4	2
0	1	2	-1
1	2	-1	0

roots
-1.618034
0.618034
2

The FROOT function returns a missing value if the search fails to return a root. Usually this indicates that the function does not change signs at the endpoints of the specified interval. For example, the following statement returns a missing value:

```
r = froot("Func", {3 4});
```

FULL Function

FULL(*x* < , *nrow*> < , *ncol*> **);**

The FULL function converts a matrix stored in a sparse format into a matrix stored in a dense format. See the SPARSE function for a description of how sparse matrices are stored.

The arguments are as follows:

x	specifies a $k \times 3$ numerical matrix that contains a sparse representation of an $n \times p$ matrix.
nrow	specifies the number of rows in the dense matrix. If this argument is not specified, then the number of rows is determined by the maximum value of the second column of *x*.

ncol specifies the number of columns in the dense matrix. If this argument is not specified, then the number of columns is determined by the maximum value of the third column of *x*.

The matrix returned by the FULL function is an $n \times p$ matrix with k nonzero values determined by the *x* matrix, as shown in the following example:

```
s = {3   1  1,
     1.1 2  1,
     4   2  2,
     1   3  2,
     10  3  3,
     3.2 4  2,
     3   4  4 };
x = full(s);
print x;
```

Figure 25.163 Matrix Converted from Sparse to Dense Storage

x			
3	0	0	0
1.1	4	0	0
0	1	10	0
0	3.2	0	3

In the previous example, the **s** matrix specifies a lower triangular matrix. However, the **s** matrix might represent a symmetric matrix rather than a lower triangular matrix, but only the lower triangular entries were stored. (For example, the **s** matrix might have been created by the SPARSE function by using the "SYM" option; see the SPARSE function documentation.) If that is the case, you can use the following statement to recover the symmetric matrix representation of **s**:

```
xSym = (x+x`)- diag(x);
print xSym;
```

Figure 25.164 Symmetric Matrix Converted from Sparse Symmetric Storage

xSym			
3	1.1	0	0
1.1	4	1	3.2
0	1	10	0
0	3.2	0	3

By default, the size of the matrix returned by the FULL function is determined by the maximum row and column entry in the first argument. You can override this behavior by specifying values for the number of rows and columns returned by the FULL function, as shown in the following statements:

```
z = full(s, 5, 6);
print z;
```

Figure 25.165 Matrix with Zeros in Last Row or Column

z					
3	0	0	0	0	0
1.1	4	0	0	0	0
0	1	10	0	0	0
0	3.2	0	3	0	0
0	0	0	0	0	0

GAEND Call

CALL GAEND(*id*);

The GAEND subroutine ends a genetic algorithm optimization and frees memory resources. The arguments to the GAEND call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

The GAEND call ends the genetic algorithm calculations associated with *id* and frees up all associated memory.

See the GASETUP function for an example.

GAGETMEM Call

CALL GAGETMEM(*members, values, id< , index >***);**

The GAGETMEM subroutine gets members of the current solution population for a genetic algorithm optimization.

The GAGETMEM call returns the following values as output arguments:

members names a matrix that contains the members of the current solution population specified by the *index* parameter.

values names a matrix that contains objective function values, with a value at each row that corresponds to the solution in *members*.

The input arguments to the GAGETMEM call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

index is a matrix of indices of the requested solution population members. If *index* is not specified, the entire population is returned.

The GAGETMEM call is used to retrieve members of the solution population and their objective function values. If the *elite* parameter of the GASETSEL call is nonzero, then the first *elite* members of the population have the most optimal objective function values of the population, and those *elite* members are sorted in ascending order of objective function value for a minimization problem and in descending order for a maximization problem.

If a single member is requested, it is returned in *members*. If more than one member is requested in a GAGETMEM call, each row of *members* has one solution, shaped into a row vector. If solutions are not of fixed length, then the number of columns of *members* equals the number of elements of the largest solution and rows that represent solutions with fewer elements have the extra elements filled in with missing values.

See the GASETUP function for an example.

GAGETVAL Call

CALL GAGETVAL(*values, id*< *, index*>**);**

The GAGETVAL subroutine gets objective function values for members of the population in a genetic algorithm optimization. The GAGETVAL call returns the following output argument:

values names a matrix that contains objective function values for solutions in the current population that are specified by *index*. If *index* is not present, then values for all solutions in the population are returned. Each row in *values* corresponds to one solution.

The input arguments to the GAGETVAL call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

index is a matrix of indices of the requested objective function values. If *index* is not specified, then all objective function values are returned.

The GAGETVAL call is used to retrieve objective function values of the current solution population. If the *elite* parameter of the GASETSEL call is nonzero, then the first *elite* members of the population have the most optimal objective function values of the population, and those *elite* members are sorted in ascending order of objective function value for a minimization problem or in descending order for a maximization problem.

See the GASETUP function for an example.

GAINIT Call

CALL GAINIT(*id, popsize* < *, bounds*> < *, modname*> **);**

The GAINIT subroutine creates and initializes a solution population for a genetic algorithm optimization. The input arguments to the GAINIT call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

popsize is the number of solution matrices to create and initialize.

bounds is an optional parameter matrix that specifies the lower and upper bounds for each element of a solution matrix. It is used only for integer and real fixed-length vector problem encoding.

modname is the name of a user-written module to be called from GAINIT when it generates the initial members of the solution population.

The GAINIT call creates the members and computes the objective values for an initial solution population for a genetic algorithm optimization. If the problem encoding is specified as a sequence in the corresponding GASETUP function call and no *modname* parameter is specified, then GAINIT creates an initial population of vectors of randomly ordered integer values ranging from 1 to the *size* parameter of the GASETUP function call. Otherwise, you control how the population is created and initialized with the *bounds* and *modname* parameters.

If real or integer fixed-length vector encoding is specified in the corresponding GASETUP function call, then the *bounds* parameter can be supplied as a $2 \times n$ matrix, where the dimension n equals the *size* parameter of the GASETUP function call: the first row specifies the lower bounds of the corresponding vector elements and the second row specifies the upper bounds. The solutions that result from all crossover and mutation operators are checked to ensure they are within the upper and lower bounds, and any solution components that violate the bounds are reset to the bound. However, if user-written modules are provided for these operators, the modules are expected to do the bounds checking internally. If no *modname* parameter is specified, the initial population is generated by random variation of the solution components between the lower and upper bounds.

For all problem encodings, if the *modname* parameter is specified, it is expected to be the name of a user-written subroutine module with one parameter. The module should generate and return an individual solution in that parameter. The GAINIT call invokes that module *popsize* times, once for each member of the initial solution population. The *modname* parameter is required if the *encoding* parameter of the corresponding GASETUP function call was 0 or if the *bounds* parameter is not specified for real or integer fixed-length vector encoding.

See the GASETUP function for an example.

GAREEVAL Call

CALL GAREEVAL(*id*);

The GAREEVAL subroutine reevaluates the objective function values for a solution population of a genetic algorithm optimization. The input arguments to the GAREEVAL call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

The GAREEVAL call computes the objective values for a solution population of a genetic algorithm optimization. Since the GAINIT call and the GAREGEN call also evaluate the objective function values, it is usually not necessary to call GAREEVAL. It is provided to handle the situation of a user modifying an objective function independently—for example, adjusting a global variable to relax or tighten a penalty constraint. In such a case, GAREEVAL should be called before the next GAREGEN call.

GAREGEN Call

CALL GAREGEN(*id*);

The GAREGEN subroutine replaces the current solution population by applying selection, crossover, and mutation for a genetic algorithm optimization problem. The input arguments to the GAREGEN call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

The GAREGEN call applies the genetic algorithm to create a new solution population from the current population. As the first step, if the *elite* parameter of the corresponding GASETSEL call is nonzero, the best *elite* members of the current population are copied into the new population, sorted by objective value with the best objective value first. If a crossover operator has been specified in a corresponding GASETCRO call or a default crossover operator is in effect, the remaining members of the population are determined by selecting members of the current population, applying the crossover operator to generate offspring, and mutating the offspring according to the mutation probability and mutation operator. Either the mutation probability and operator are specified in the corresponding GASETMUT call or, if no such call is made, a default value of 0.05 is assigned to the mutation probability and a default mutation operator is assigned based on the problem encoding (see the GASETMUT call). The offspring are then transferred to the new population. If the no-crossover option is specified in the GASETCRO call, then only mutation is applied to the non-elite members of the current population to form the new population. After the new population is formed, it becomes the current solution population, and the objective function specified in the GASETOBJ call is evaluated for each member.

See the GASETUP function for an example.

GASETCRO Call

CALL GASETCRO(*id, crossprob, type* < *, parm* > **);

The GASETCRO subroutine sets the crossover operator for a genetic algorithm optimization. The input arguments to the GASETCRO call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

crossprob is the crossover probability, which has a range from zero to one. It specifies the probability that selected members of the current generation undergo crossover to produce new offspring for the next generation.

type specifies the kind of crossover operator to be used. *type* is used in conjunction with *parm* to specify either a user-written module for the crossover operator or one of several other operators, as explained in the following list.

parm is a matrix whose interpretation depends on the value of *type*, as described in the following list.

The following list specifies the valid values of the *type* parameter and the corresponding crossover operators:

–1 specifies that no crossover operator be applied and the new population be generated by applying the mutation operator to the old population, according to the mutation probability.

0 specifies that a user-written module, whose name is passed in the *parm* parameter, be used as the crossover operator. This module should be a subroutine with four parameters. The module should return the new offspring solutions in the first two parameters based on the input parent solutions, which are selected by the genetic algorithm and passed into the module in the last two parameters. The module is called once for each crossover operation within the GAREGEN call to create a new generation of solutions.

1 specifies the simple operator, defined for fixed-length integer and real vector encoding. To apply this operator, a position k within the vector of length n is chosen at random, such that $1 \leq k < n$. Then for parents **p1** and **p2** the offspring are as follows:

```
c1= p1[1,1:k] || p2[1,k+1:n];
c2= p2[1,1:k] || p1[1,k+1:n];
```

For real fixed-length vector encoding, you can specify an additional parameter, a, with the *parm* parameter, where a is a scalar and $0 < a \leq 1$. It modifies the offspring as follows:

```
x2 = a * p2 + (1-a) * p1;
c1 = p1[1,1:k] || x2[1,k+1:n];

x1 = a * p1 + (1-a) * p2
c2 = p2[1,1:k] || x1[1,k+1:n];
```

Note that for $a = 1$, which is the default value, **x2** and **x1** are the same as **p2** and **p1**. Small values of a reduce the difference between the offspring and parents. For integer encoding, the *parm* parameter is ignored and a is always 1.

2 specifies the two-point operator, defined for fixed-length integer and real vector encoding with length $n \geq 3$. To apply this operator, two positions k_1 and k_2 within the vector are chosen at random, such that $1 \leq k_1 < k_2 < n$. Element values between those positions are swapped between parents. For parents **p1** and **p2** the offspring are as follows:

```
c1 = p1[1,1:k1] || p2[1,k1+1:k2] || p1[1,k2+1:n];
c2 = p2[1,1:k1] || p1[1,k1+1:k2] || p2[1,k2+1:n];
```

For real vector encoding, you can specify an additional parameter, a, in the *parm* field, where $0 < a \leq 1$. It modifies the offspring as follows:

```
x2 = a * p2 + (1-a) * p1;
c1 = p1[1,1:k1] || x2[1,k1+1:k2] || p1[1,k2+1:n];

x1 = a * p1 + (1-a) * p2;
c2 = p2[1,1:k1] || x1[1,k1+1:k2] || p2[1,k2+1:n];
```

Note that for $a = 1$, which is the default value, **x2** and **x1** are the same as **p2** and **p1**. Small values of a reduce the difference between the offspring and parents. For integer encoding, the *parm* parameter is ignored if present and a is always 1.

3 specifies the arithmetic operator, defined for real and integer fixed-length vector encoding. This operator computes offspring of parents **p1** and **p2** as follows:

```
c1 = a * p1 + (1-a) * p2;
c2 = a * p2 + (1-a) * p1;
```

where *a* is a random number between 0 and 1. For integer encoding, each component is rounded off to the nearest integer. An advantage of this operator is that it always produces feasible offspring for a convex solution space. A disadvantage is that it tends to produce offspring toward the interior of the search region, so that it can be less effective if the optimum lies on or near the search region boundary.

4 specifies the heuristic operator, defined for real fixed-length vector encoding. This operator computes the first offspring from the two parents **p1** and **p2** as follows:

```
c1 = a * (p2 - p1) + p2;
```

where **p2** is the parent with the better objective value and *a* is a random number between 0 and 1. The second offspring is computed as in the arithmetic operator, as follows:

```
c2 = (1 - a) * p1 + a * p2;
```

This operator is unusual in that it uses the objective value. It has the advantage of directing the search in a promising direction and automatically fine-tuning the search in an area where solutions are clustered. If upper and lower bound constraints are specified in the GAINIT call, the offspring are checked against the bounds and any component outside its bound is set equal to that bound.

5 specifies the partial match operator, defined for sequence encoding. This operator produces offspring by transferring a subsequence from one parent and filling the remaining positions in a way consistent with the position and ordering in the other parent. Start with two parents and randomly chosen cut-points as follows:

```
p1 = {1 2|3 4 5 6|7 8 9};
p2 = {8 7|9 3 4 1|2 5 6};
```

The first step is to cross the selected segments; a missing value (.) indicates a position that is not determined):

```
c1 = {. . 9 3 4 1 . . .};
c2 = {. . 3 4 5 6 . . .};
```

Next, define a mapping according to the two selected segments, as follows:

$$9 \leftrightarrow 3, 3 \leftrightarrow 4, 4 \leftrightarrow 5, 1 \leftrightarrow 6$$

Next, fill in the positions where there is no conflict from the corresponding parent:

```
c1 = {. 2 9 3 4 1 7 8 .};
c2 = {8 7 3 4 5 6 2 . .};
```

Last, fill in the remaining positions from the subsequence mapping. In this case, for the first child $1 \rightarrow 6$ and $9 \rightarrow 3$, and for the second child $5 \rightarrow 4$, $3 \rightarrow 9$, and $6 \rightarrow 1$:

```
c1 = {6 2 9 3 4 1 7 8 5};
c2 = {8 7 3 4 5 6 2 9 1};
```

This operator tends to maintain similarity of both the absolute position and relative ordering of the sequence elements, and is useful for a wide range of sequencing problems.

6 specifies the order operator, defined for sequence encoding. This operator produces offspring by transferring a subsequence of random length and position from one parent and filling the remaining positions according to the order from the other parent. For parents **p1** and **p2**, first choose a subsequence, as follows:

```
p1 = {1 2|3 4 5 6|7 8 9};
p2 = {8 7|9 3 4 1|2 5 6};
c1 = {. . 3 4 5 6 . . .};
c2 = {. . 9 3 4 1 . . .};
```

Starting at the second cut-point, the elements of **p2** are in the following order (cycling back to the beginning):

```
2 5 6 8 7 9 3 4 1
```

After removing 3, 4, 5, and 6, which have already been placed in **c1**, you have the following:

```
2 8 7 9 1
```

Placing these back in order, starting at the second cut-point, yields the following:

```
c1 = {9 1 3 4 5 6 2 8 7};
```

Applying this logic to **c2** yields the following:

```
c2 = {5 6 9 3 4 1 7 8 2};
```

This operator maintains the similarity of the relative order (also called the adjacency) of the sequence elements of the parents. It is especially effective for circular path-oriented optimizations, such as the traveling salesman problem.

7 specifies the cycle operator, defined for sequence encoding. This operator produces offspring such that the position of each element value in the offspring comes from one of the parents. For example, consider the following parents **p1** and **p2**:

```
p1 = {1 2 3 4 5 6 7 8 9};
p2 = {8 7 9 3 4 1 2 5 6};
```

For the first child, pick the first element from the first parent, as follows:

```
c1 = {1 . . . . . . . .};
```

To maintain the condition that the position of each element value must come from one of the parents, the position of the ‘8’ value must come from **p1**, because the ‘8’ position in **p2** is already taken by the ‘1’ in **c1**:

```
c1 = {1 . . . . . 8 .};
```

Now the position of '5' must come from **p1** and so on until the process returns to the first position:

```
c1 = {1 . 3 4 5 6 . 8 9};
```

At this point, choose the remaining element positions from **p2**:

```
c1 = {1 7 3 4 5 6 2 8 9};
```

For the second child, starting with the first element from the second parent, similar logic produces the following:

```
c2 = {8 2 9 3 4 1 7 5 6};
```

This operator is most useful when the absolute position of the elements is of most importance to the objective value.

A GASETCRO call is required when 0 is specified for the *encoding* parameter in the GASETUP function. But for fixed-length vector and sequence encoding, a default crossover operator is used in the GAREGEN call when no GASETCRO call is made. For sequence encoding, the default is the partial match operator, unless the traveling salesman option was specified in the GASETOBJ call, in which case the order operator is the default. For integer fixed-length vector encoding, the default is the simple operator. For real fixed-length vector encoding, the default is the heuristic operator.

See the GASETUP function for an example.

GASETMUT Call

CALL GASETMUT(*id, mutprob* < *, type* > < *, parm* > **);**

The GASETMUT subroutine sets the mutation operator for a genetic algorithm optimization. The input arguments to the GASETMUT call are as follows:

id	is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.
mutprob	is the probability for a given solution to undergo mutation, a number between 0 and 1.
type	specifies the kind of mutation operator to be used. *type* is used in conjunction with *parm* to specify either a user-written module for the mutation operator or one of several other operators, as explained in the following list.
parm	is a matrix whose interpretation depends on the value of *type*, as described in the following list.

The GASETMUT call enables you to specify the frequency of mutation and the mutation operator to be used in the genetic algorithm optimization problem. If the *type* parameter is not specified, then the GASETMUT call only alters the mutation probability, without resetting the mutation operator, and any operator set by a previous GASETMUT call remains in effect. You can specify the following mutation operators with the *type* parameter:

0 specifies that a user-written module, whose name is passed in the *parm* parameter, be used as the mutation operator. This module should be a subroutine with one parameter, which receives the solution to be mutated. The module is called once for each mutation operation and is expected to modify the input solution according to the desired mutation operation. Any checking of bounds specified in the GAINIT call should be done inside the module; in this case they are not checked by the SAS/IML language.

1 specifies the uniform mutation operator, defined for fixed-length real or integer encoding, with upper and lower bounds specified in the GAINIT call. The *parm* parameter is not used with this option. To apply this operator, a position k is randomly chosen within the solution vector v and $v[k]$ is modified to a random value between the upper and lower bounds for element k. This operator can prove especially useful in early stages of the optimization, since it tends to distribute solutions widely across the search space and avoid premature convergence to a local optimum. However, in later stages of an optimization with real vector encoding when the search needs to be fine-tuned to home in on an optimum, the uniform operator can hinder the optimization.

2 specifies the delta mutation operator, defined for integer and real fixed-length vector encoding. This operator first chooses an element of the solution at random, and then perturbs that element by a fixed amount, *delta*, which is set with the *parm* parameter. *delta* has the same dimension as the solution vectors, and each element *delta*[k] is set to *parm*[k], unless *parm* is a scalar, in which case all elements are set equal to *parm*. For integer encoding, all *delta*[k] are truncated to integers if they are not integers in *parm*. To apply the mutation, a randomly chosen element k of the solution vector v is modified such that one of the following statements is true:

```
v[k] = v[k] + delta[k]; /* with probability 0.5 */
       or
v[k] = v[k] - delta[k];
```

If bounds are specified for the problem in the GAINIT call, then $v[k]$ is adjusted as necessary to fit within the bounds. This operator enables you to control the scope of the search with the *parm* matrix. One possible strategy is to start with a larger *delta* value and then reduce it with subsequent GASETMUT calls as the search progresses and begins to converge to an optimum. This operator is also useful if the optimum is known to be on or near a boundary, in which case *delta* can be set large enough to always perturb the solution element to a boundary.

3 specifies the swap operator, which is defined for sequence problem encoding. This operator picks two random locations in the solution vector and swaps their values. It is the default mutation operator for sequence encoding, except for when the traveling salesman option is specified in the GASETOBJ call. You can also specify that multiple swaps be made for each mutation with the *parm* parameter. The number of swaps defaults to 1 if *parm* is not specified, and is equal to *parm* otherwise.

4 specifies the invert operator, defined for sequence encoding. This operator picks two locations at random and then reverses the order of elements between them. This operator is most often applied to the traveling salesman problem. The *parm* parameter is not used with this operator.

Mutation is generally useful in the application of the genetic algorithm to ensure that a diverse population of solutions is sampled to avoid premature convergence to a local optimum. More than one GASETMUT call can be made at any time in the progress of the algorithm. This enables flexible adaptation of the mutation process, either changing the mutation probability or changing the operator itself. You can do this to ensure a wide search at the beginning of the optimization, and then reduce the variation later to narrow the search close to an optimum.

A GASETMUT call is required when an *encoding* parameter of 0 is specified in the GASETUP function. But when no GASETMUT call is made for fixed-length vector and sequence encoding, a default value of 0.05 is set for *mutprob* and a default mutation operator is used in the GAREGEN call. The mutation operator defaults to the uniform operator for fixed-length vector encoding with bounds specified in the GAINIT call, the delta operator with a *parm* value of 1 for fixed-length vector encoding with no bounds specified, the invert operator for sequence encoding when the traveling salesman option is chosen in the GASETOBJ call, and the swap operator for all other sequence encoded problems.

See the GASETUP function for an example.

GASETOBJ Call

CALL GASETOBJ(*id, type* < *, parm* > **);**

The GASETOBJ subroutine sets the objective function for a genetic algorithm optimization. The input arguments to the GASETOBJ call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

type specifies the type of objective function to be used.

parm is a matrix whose interpretation depends on the value of *type*, as described in the following list.

You can specify that a user-written module be used to compute the value of the objective function, or you can specify a standard preset function. This is specified with the *type* and *parm* parameters. The following list specifies the valid values of the *type* parameter:

0 specifies that a user-written function module is to be minimized. The name of the module is supplied in the *parm* parameter. The specified module should take a single parameter that represents a given solution, and return a scalar numeric value for the objective function.

1 specifies that a user-written function module be maximized. The name of the module is supplied in the *parm* parameter. The specified module should take a single parameter that represents a given solution, and return a scalar numeric value for the objective function.

2 specifies an objective function from the traveling salesman problem, which is minimized. This option is valid only if three conditions are met: sequence encoding was specified in the GASETUP function call, the solution vector is to be interpreted as a circular route, and each element represents a location. The *parm* parameter should be a square cost matrix, such that *parm*[*i*, *j*] is the cost of going from location *i* to location *j*. The dimension of the matrix should be the same as the *size* parameter of the corresponding GASETUP function call.

The specified objective function is called once for each solution to evaluate the objective values for the GAREGEN call, GAINIT call, and GAREEVAL call. Also, the objective values for the current solution population are reevaluated if GASETOBJ is called after a GAINIT call.

See the GASETUP function for an example.

GASETSEL Call

CALL GASETSEL(*id, elite, type, parm* **);**

The GASETSEL subroutine sets the selection parameters for a genetic algorithm optimization.

The input arguments to the GASETSEL call are as follows:

id is the identifier for the genetic algorithm optimization problem, which was returned by the GASETUP function.

elite specifies the number of solution population members to carry over unaltered to the next generation in the GAREGEN call. If nonzero, then *elite* members with the best objective function values are carried over without crossover or mutation.

type specifies the selection method to use.

parm is a parameter used to control the selection pressure.

This module sets the selection parameters that are used in the GAREGEN call to select solutions for the crossover operation. You can choose between two variants of the "tournament" selection method in which a group of different solutions is picked at random from the current solution population and the solution from that group with the best objective value is selected. In the first variation, chosen by setting *type* to 0, the most optimal solution is always selected, and the *parm* parameter is used to specify the size of the group, always two or greater. The larger the group size, the greater the selective pressure. In the second variation, chosen by setting *type* to 1, the group size is set to 2 and the best solution is chosen with probability specified by *parm*. If *parm* is 1, the best solution is always picked; a *parm* value of 0.5 is equivalent to pure random selection. The *parm* value must be between 0.5 and 1. When *type* is 0, the selective pressure is greater than when *type* is 1. Higher selective pressure leads to faster convergence of the genetic algorithm, but is more likely to give premature convergence to a local optimum.

In order to ensure that the best solution of the current solution population is always carried over to the next generation, an *elite* value of 1 should be specified. Higher values of *elite* generally lead to faster convergence of the algorithm, but they increase the chances of premature convergence to a local optimum. If GASETSEL is not called, the optimization uses the default values of 1 for *elite*, 1 for *type*, and 2 for *parm*.

See the GASETUP function for an example.

GASETUP Function

GASETUP(*encoding, size* **<** *, seed* **>** **);**

The GASETUP function sets up the problem encoding for a genetic algorithm optimization problem. The GASETUP function returns a scalar number that identifies the genetic algorithm optimization problem. This number is used in subsequent calls to carry out the optimization.

The arguments to the GASETUP function are as follows:

encoding is a scalar number used to specify the form or structure of the problem solutions to be optimized. A value of 0 indicates a numeric matrix of arbitrary dimensions, 1 indicates a fixed-length floating-point row vector, 2 indicates a fixed-length integer row vector, and 3

indicates a fixed-length sequence of integers, with alternate solutions distinguished by different sequence ordering.

size is a numeric scalar, whose value is the vector or sequence length, if a fixed-length *encoding* is specified. For arbitrary matrix encoding (*encoding* value of 0), *size* is not used.

seed is an optional initial random number seed to be used for the initialization and the selection process. If *seed* is not specified or its value is 0, an initial seed is derived from the current system time.

GASETUP is the first call that must be made to set up a genetic algorithm optimization problem. It specifies the problem encoding, the size of a population member, and an optional seed that initializes the random number generator used in the selection process. GASETUP returns an identifying number that must be passed to the other modules that specify genetic operators and control the execution of the genetic algorithm. More than one optimization can be active concurrently, and optimization problems with different problem identifiers are completely independent. When a satisfactory solution has been determined, the optimization problem should be terminated with a GAEND call to free up resources associated with the genetic algorithm.

The following example demonstrates the use of several genetic algorithm subroutines:

```
/* Use a genetic algorithm to explore the solution space for the
   "traveling salesman" problem. First, define the objective
   function to minimize:
   Compute the sum of distances between sequence of cities */
start EvalFitness( pop ) global ( dist );
   fitness = j( nrow(pop),1 );
   do i = 1 to nrow(pop);
      city1 = pop[i,1];
      city2 = pop[i,ncol(pop)];
      fitness[i] = dist[ city1, city2 ];
      do j = 1 to ncol(pop)-1;
         city1 = pop[i,j];
         city2 = pop[i,j+1];
         fitness[i] = fitness[i] + dist[city1,city2];
      end;
   end;
   return ( fitness );
finish;

/* Set up parameters for the genetic algorithm */

mutationProb = 0.15;   /* prob that a child will be mutated */
numElite = 2;          /* copy this many to next generation */
numCities = 15;        /* number of cities to visit */
numGenerations = 100;  /* number of generations to evolve */
seed = 12345;            /* random number seed */
call randseed(seed);

/* fix population size; generate random locations for cities */
popSize = max(30,2*numCities);
locations = j(numCities,2);
call randgen(locations, "uniform");

/* compute distances between cities one time */
```

```
dist = distance(locations);

/* run the genetic algorithm */
id = gasetup( 3, numCities, seed);
call gasetobj(id, 0, "EvalFitness" );
call gasetcro(id, 1.0, 6);
call gasetmut(id, mutationProb, 3);
call gasetsel(id, numElite, 1, 0.95);
call gainit(id, popSize );

do i = 1 to numGenerations;
   if mod(i,20)=0 then do;
      call gagetval( value, id, 1 );
      print "Iteration:" i "Top value:" value;
   end;
   call garegen(id);
end;

/* report final sequence for cities */
call gagetmem(mem, value, id, 1);
print mem, value;
call gaend(id);
```

Figure 25.166 Result of a Genetic Algorithm Optimization

i	value
Iteration: 20	Top value: 3.8362386

i	value
Iteration: 40	Top value: 3.5099897

i	value
Iteration: 60	Top value: 3.3084172

i	value
Iteration: 80	Top value: 3.3084172

i	value
Iteration: 100	Top value: 3.3084172

	mem														
	COL1	COL2	COL3	COL4	COL5	COL6	COL7	COL8	COL9	COL10	COL11	COL12	COL13	COL14	COL15
ROW1	15	4	14	8	12	9	10	7	5	6	2	1	11	13	3

value
3.3084172

GENEIG Call

CALL GENEIG(*eval, evecs, sym-matrix1, sym-matrix2***);**

The GENEIG subroutine computes eigenvalues and eigenvectors of a generalized eigenvalue problem.

The input arguments to the GENEIG subroutine are as follows:

sym-matrix1 is a symmetric numeric matrix.

sym-matrix2 is a positive definite symmetric matrix.

The subroutine returns the following output arguments:

evals names a vector in which the eigenvalues are returned.

evecs names a matrix in which the corresponding eigenvectors are returned.

The GENEIG subroutine computes eigenvalues and eigenvectors of the generalized eigenvalue problem. If **A** and **B** are symmetric and **B** is positive definite, then the vector **M** and the matrix **E** solve the generalized eigenvalue problem provided that

$$\mathbf{A} * \mathbf{E} = \mathbf{B} * \mathbf{E} * \mathrm{diag}(\mathbf{M})$$

The vector **M** contains the eigenvalues arranged in descending order, and the matrix **E** contains the corresponding eigenvectors in the columns.

The following example is from Wilkinson and Reinsch (1971):

```
A = {10   2   3   1   1,
      2  12   1   2   1,
      3   1  11   1  -1,
      1   2   1   9   1,
      1   1  -1   1  15};

B = {12   1  -1   2    1,
      1  14   1  -1    1,
     -1   1  16  -1    1,
      2  -1  -1  12   -1,
      1   1   1  -1   11};

call geneig(M, E, A, B);
print M, E;
```

Figure 25.167 Solution of a Generalized Eigenproblem

M
1.4923532
1.1092845
0.943859
0.6636627
0.4327872

E				
-0.076387	0.142012	0.19171	-0.08292	-0.134591
0.017098	0.14242	-0.158991	-0.153148	0.0612947
-0.066665	0.1209976	0.0748391	0.1186037	0.1579026
0.086048	0.125531	-0.137469	0.182813	-0.109466
0.2894334	0.0076922	0.0889779	-0.003562	0.041473

GEOMEAN Function

GEOMEAN(*matrix***);**

The GEOMEAN function returns a scalar that contains the geometric mean of the elements of the input matrix. The geometric mean of a set of nonnegative numbers $a_1, a_2, \ldots, a_n$ is the nth root of the product $a_1 \cdot a_2 \cdots a_n$.

The geometric mean is zero if any of the a_i are zero. The geometric mean is not defined for negative numbers. If any of the a_i are missing, they are excluded from the computation.

The geometric mean can be used to compute the average return on an investment. For example, the following data are the annual returns on U.S. Treasury bonds from 1994 to 2004. The following statements compute the average rate of return during this time. The output, shown in Figure 25.168, shows that the average rate of return was 6.43%.

```
/*          year  return% */
TBonds = { 1994  -8.04,
           1995  23.48,
           1996   1.43,
           1997   9.94,
           1998  14.92,
           1999  -8.25,
           2000  16.66,
           2001   5.57,
           2002  15.12,
           2003   0.38,
           2004   4.49 };

proportion = 1 + TBonds[,2]/100; /* convert to proportion */
aveReturn = geomean( proportion );
print aveReturn;
```

Figure 25.168 Average Rate of Return for an Investment

aveReturn
1.0643334

GINV Function

GINV(*matrix***);**

The GINV function computes the Moore-Penrose generalized inverse of *matrix*. This inverse, known as the four-condition inverse, has these properties:

If $\mathbf{G} = \text{GINV}(\mathbf{A})$ then

$$\mathbf{AGA} = \mathbf{A} \quad \mathbf{GAG} = \mathbf{G} \quad (\mathbf{AG})' = \mathbf{AG} \quad (\mathbf{GA})' = \mathbf{GA}$$

The generalized inverse is also known as the *pseudoinverse*, usually denoted by $\mathbf{A}^-$. It is computed by using the singular value decomposition (Wilkinson and Reinsch 1971).

See Rao and Mitra (1971) for a discussion of properties of this function.

As an example, consider the following model:

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

Least squares regression for this model can be performed by using the quantity **ginv(x)*y** as the estimate of $\boldsymbol{\beta}$. This solution has minimum $\mathbf{b}'\mathbf{b}$ among all solutions that minimize $\boldsymbol{\epsilon}'\boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} = \mathbf{Y} - \mathbf{X}\mathbf{b}$.

Projection matrices can be formed by specifying GINV($\mathbf{X}$) $*$ $\mathbf{X}$ (*row space*) or $\mathbf{X}*$GINV($\mathbf{X}$) (*column space*).

The following program demonstrates some common uses of the GINV function:

```
A = {1 0 1 0 0,
     1 0 0 1 0,
     1 0 0 0 1,
     0 1 1 0 0,
     0 1 0 1 0,
     0 1 0 0 1 };

/* find generalized inverse */
Ainv = ginv(A);

/* find LS solution: min  |Ax-b|^2 */
b = { 3, 2, 4, 2, 1, 3 };
x = Ainv*b;

/* form projection matrix onto row space.
   Note P = P` and P*P = P */
P = Ainv*A;

/* find numerical rank of A */
rankA = round(trace(P));

reset fuzz;
print Ainv, rankA, x, P;
```

Figure 25.169 Common Uses of the Generalized Inverse

Ainv					
0.2666667	0.2666667	0.2666667	-0.066667	-0.066667	-0.066667
-0.066667	-0.066667	-0.066667	0.2666667	0.2666667	0.2666667
0.4	-0.1	-0.1	0.4	-0.1	-0.1
-0.1	0.4	-0.1	-0.1	0.4	-0.1
-0.1	-0.1	0.4	-0.1	-0.1	0.4

rankA
4

x
2
1
1
0
2

Figure 25.169 *continued*

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.

GOTO Statement

GOTO *label* **;**

The GOTO statement causes a program to jump to a new statement in the program. When the GOTO statement is executed, the program jumps immediately to the statement with the given *label* and begin executing statements from that point. A label is a name followed by a colon that precedes an executable statement.

GOTO statements are often clauses of IF-THEN statements. For example, the following statements use a GOTO statement to iterate until a condition is satisfied:

```
start Iterate;
   x = 1;
   TheStart:
   if x > 10 then
      goto TheEnd;
   x = x + 1;
   goto TheStart;

   TheEnd: print x;
finish;

run Iterate;
```

Figure 25.170 Iteration by Using the GOTO Statement

x
11

The function of GOTO statements is usually better performed by DO groups. For example, the preceding statements could be better written as follows:

```
x = 1;
do until(x > 10);
   x = x + 1;
end;
```

```
print x;
```

Figure 25.171 Avoiding the GOTO Statement

x
11

As good programming practice, you should avoid using a GOTO statement that refers to a label that precedes the GOTO statement; otherwise, an infinite loop is possible. You cannot use a GOTO statement to jump out of a module; use the RETURN statement instead.

GSCALE Call

CALL GSCALE(*scale, x, nincr* < *, nicenum* > < *, fixed-end* > **);**

The GSCALE subroutine computes a suitable scale and tick values for labeling axes.

The required arguments to the GSCALE subroutine are as follows:

scale	is a returned vector that contains the scaled minimum data value, the scaled maximum data value, and a grid increment.
x	is a numeric matrix or literal.
nincr	is the number of intervals desired.

The optional arguments to the GSCALE subroutine are as follows:

nicenum	is numeric and provides up to 10 numbers to use for scaling. By default, *nicenum* is the vector {1,2,2.5,5}.
fixed-end	is a character argument that specifies which end of the scale is held fixed. The default is 'X'.

The GSCALE subroutine obtains simple (round) numbers with uniform grid interval sizes to use in scaling a linear axis. The GSCALE subroutine implements Algorithm 463 (Lewart 1973) of the *Collected Algorithms* from the Association for Computing Machinery (ACM). The scale values are integer multiples of the interval size. They are returned in the first argument, a vector with three elements. The first element is the scaled minimum data value. The second element is the scaled maximum data value. The third element is the grid increment.

The required input parameters are *x*, a matrix of data values, and *nincr*, the number of intervals desired. If *nincr* is positive, the scaled range includes approximately *nincr* intervals. If *nincr* is negative, the scaled range includes exactly ABS(*nincr*) intervals. The *nincr* parameter cannot be zero.

The *nicenum* and *fixed-end* arguments are optional. The *nicenum* argument provides up to 10 numbers, all between 1 and 10 (inclusive of the endpoints), to be used for scaling. The default for *nicenum* is 1, 2, 2.5, and 5. The linear scale with this set of numbers is a scale with an interval size that is the product of an integer power of 10 and 1, 2, 2.5, or 5. Changing these numbers alters the rounding of the scaled values.

For *fixed-end*, 'U' fixes the upper end; 'L' fixes the lower end; 'X' allows both ends to vary from the data values. The default is 'X'. An example that uses the GSCALE subroutine follows:

```
call randseed(123);
x = randfun(100, "Normal");   /* generate standard normal data */
call gscale(scale, x, 5); /* ask for about 5 intervals */
ticks = do(scale[1], scale[2], scale[3]);
print scale, ticks;
```

Figure 25.172 Tick Marks for Standard Normal Data

scale
-3
3
1

ticks
-3 -2 -1 0 1 2 3

GSORTH Call

CALL GSORTH(*P, T, lindep, A***);**

The GSORTH subroutine computes the Gram-Schmidt orthonormal factorization of the $m \times n$ matrix $\mathbf{A}$, where m is greater than or equal to n. The GSORTH subroutine implements an algorithm described by Golub (1969).

The GSORTH subroutine has a single input argument:

A is an input $m \times n$ matrix.

The output arguments to the GSORTH subroutine are as follows:

P is an $m \times n$ column-orthonormal output matrix.

T is an upper triangular $n \times n$ output matrix.

lindep is a flag with a value of 0 if columns of *A* are independent and a value of 1 if they are dependent. The *lindep* argument is an output scalar.

Specifically, the GSORTH subroutine computes the column-orthonormal $m \times n$ matrix $\mathbf{P}$ and the upper triangular $n \times n$ matrix $\mathbf{T}$ such that

$$\mathbf{A} = \mathbf{P} * \mathbf{T}$$

If the columns of $\mathbf{A}$ are linearly independent (that is, $\text{rank}(\mathbf{A}) = n$), then $\mathbf{P}$ is full-rank column-orthonormal: $\mathbf{P}'\mathbf{P} = \mathbf{I}_w$, $\mathbf{T}$ is nonsingular, and the value of *lindep* (a scalar) is set to 0. If the columns of $\mathbf{A}$ are linearly dependent (say, $\text{rank}(\mathbf{A}) = k < n$) then $n - k$ columns of $\mathbf{P}$ are set to 0, the corresponding rows of $\mathbf{T}$ are set to 0 ($\mathbf{T}$ is singular), and *lindep* is set to 1. The pattern of zero columns in $\mathbf{P}$ corresponds to the pattern of linear dependencies of the columns of $\mathbf{A}$ when columns are considered in left-to-right order.

The following statements call the GSORTH subroutine and print the output parameters to the call:

```
x = {1 1 3 1 2,
     1 0 1 2 3,
     1 1 3 3 4,
     1 0 1 4 5,
     1 1 3 5 6,
     1 0 1 6 7};
call gsorth(P, T, lindep, x);
reset fuzz;
print P, T, lindep;
```

Figure 25.173 Results of a Gram-Schmidt Orthonormalization

P				
0.4082483	0.4082483	0	-0.5	0
0.4082483	-0.408248	0	-0.5	0
0.4082483	0.4082483	0	0	0
0.4082483	-0.408248	0	0	0
0.4082483	0.4082483	0	0.5	0
0.4082483	-0.408248	0	0.5	0

T				
2.4494897	1.2247449	4.8989795	8.5732141	11.022704
0	1.2247449	2.4494897	-1.224745	-1.224745
0	0	0	0	0
0	0	0	4	4
0	0	0	0	0

lindep
1

If *lindep* is 1, you can permute the columns of $\mathbf{P}$ and rows of $\mathbf{T}$ so that the zero columns of $\mathbf{P}$ are rightmost—that is, $\mathbf{P} = (\mathbf{P}_1, \ldots, \mathbf{P}, k, 0, \ldots, 0)$, where k is the column rank of $\mathbf{A}$ and the equality $\mathbf{A} = \mathbf{P} * \mathbf{T}$ is preserved. The following statements show a permutation of columns:

```
d = loc(vecdiag(T)^=0) || loc(vecdiag(T)=0);
temp = P;
P[,d] = temp;
temp = T;
T[,d] = temp;
print d, P, T;
```

Figure 25.174 Rearranging Columns

d
1 2 4 3 5

Figure 25.174 *continued*

P				
0.4082483	0.4082483	-0.5	0	0
0.4082483	-0.408248	-0.5	0	0
0.4082483	0.4082483	0	0	0
0.4082483	-0.408248	0	0	0
0.4082483	0.4082483	0.5	0	0
0.4082483	-0.408248	0.5	0	0

T				
2.4494897	1.2247449	8.5732141	4.8989795	11.022704
0	1.2247449	-1.224745	2.4494897	-1.224745
0	0	0	0	0
0	0	4	0	4
0	0	0	0	0

The GSORTH subroutine is not recommended for the construction of matrices of values of orthogonal polynomials; the ORPOL function should be used for that purpose.

HADAMARD Function

HADAMARD(*n, < , i>* **);**

The HADAMARD function returns a Hadamard matrix. The arguments to the HADAMARD function are as follows:

n specifies the order of the Hadamard matrix. You can specify that *n* is 1, 2, or a multiple of 4. Furthermore, *n* must satisfy at least one of the following conditions:

- $n \leq 448$ or *n* equals 596, 604, 612, 732, or 756
- $n - 1$ is prime
- $(n/2) - 1$ is prime and $n/2 = 2 \bmod 4$
- $n = 2^p h$ for some positives integers p and h, and h satisfies one of the preceding conditions

When any other *n* is specified, the HADAMARD function returns a zero.

i specifies the row number to return. When *i* is not specified or *i* is negative, the full Hadamard matrix is returned.

The HADAMARD function returns a Hadamard matrix, which is an $n \times n$ matrix that consists entirely of the values 1 and –1. The columns of a Hadamard matrix are all orthogonal. Hadamard matrices are frequently used to make orthogonal array experimental designs for two-level factors. For example, the following statements create a 12×12 Hadamard matrix:

```
h = hadamard(12);
print h[format=2.];
```

The output is shown in Figure 25.175. The first column is an intercept and the next 11 columns form an orthogonal array experimental design for 11 two-level factors in 12 runs, 2^{11}.

Figure 25.175 A Hadamard Matrix

h											
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	1	-1	1	-1	-1	-1	1	1	1	-1	1
1	1	1	-1	1	-1	-1	-1	1	1	1	-1
1	-1	1	1	-1	1	-1	-1	-1	1	1	1
1	1	-1	1	1	-1	1	-1	-1	-1	1	1
1	1	1	-1	1	1	-1	1	-1	-1	-1	1
1	1	1	1	-1	1	1	-1	1	-1	-1	-1
1	-1	1	1	1	-1	1	1	-1	1	-1	-1
1	-1	-1	1	1	1	-1	1	1	-1	1	-1
1	-1	-1	-1	1	1	1	-1	1	1	-1	1
1	1	-1	-1	-1	1	1	1	-1	1	1	-1
1	-1	1	-1	-1	-1	1	1	1	-1	1	1

To request the seventeenth row of a Hadamard matrix of order 448, use the following statement:

```
h17 = hadamard(448, 17);
```

HALF Function

HALF(*matrix*);

The HALF function is an alias for the ROOT function, which computes the Cholesky decomposition of a symmetric positive definite matrix.

HANKEL Function

HANKEL(*matrix*);

The HANKEL function generates a Hankel matrix from a vector or a block Hankel matrix from a matrix. A block Hankel matrix has the property that all matrices on the reverse diagonals are the same. The argument matrix is an $(np) \times p$ or $p \times (np)$ matrix; the value returned is the $(np) \times (np)$ result.

The Hankel function uses the first $p \times p$ submatrix $\mathbf{A}_1$ of the argument matrix as the blocks of the first reverse diagonal. The second $p \times p$ submatrix $\mathbf{A}_2$ of the argument matrix forms the second reverse diagonal. The remaining reverse diagonals are formed accordingly. After the values in the argument matrix have all been placed, the rest of the matrix is filled in with 0. If $\mathbf{A}$ is $(np) \times p$, then the first p columns of the returned matrix, $\mathbf{R}$, are the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, then the first p rows of $\mathbf{R}$ are the same as $\mathbf{A}$.

The HANKEL function is especially useful in time series applications that involve a set of variables that represent the present and past and a set of variables that represent the present and future. In this situation, the

covariance matrix between the sets of variables is often assumed to be a block Hankel matrix. If

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2|\mathbf{A}_3|\cdots|\mathbf{A}_n]$$

and if **R** is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 & \cdots & \mathbf{0} \\ \mathbf{A}_3 & \mathbf{A}_4 & \mathbf{A}_5 & \cdots & \mathbf{0} \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

If

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

and if **R** is the matrix formed by the HANKEL function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 & \cdots & \mathbf{0} \\ \vdots & & & & \\ \mathbf{A}_n & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix}$$

For example, the following statements produce Hankel matrices, as shown in Figure 25.176:

```
r1 = hankel({1 2 3 4 5});
r2 = hankel({1 2 ,
             3 4 ,
             5 6 ,
             7 8});
r3 = hankel({1 2 3 4 ,
             5 6 7 8});
print r1, r2, r3;
```

Figure 25.176 Hankel Matrices

r1				
1	2	3	4	5
2	3	4	5	0
3	4	5	0	0
4	5	0	0	0
5	0	0	0	0

r2			
1	2	5	6
3	4	7	8
5	6	0	0
7	8	0	0

Figure 25.176 *continued*

r3
1 2 3 4
5 6 7 8
3 4 0 0
7 8 0 0

HARMEAN Function

HARMEAN(*matrix***);**

The HARMEAN function returns a scalar that contains the harmonic mean of the elements of the input matrix. The input matrix must contain only nonnegative numbers. The harmonic mean of a set of positive numbers $a_1, a_2, \ldots, a_n$ is n divided by the sum of the reciprocals of a_i. That is, $n / \sum a_i^{-1}$.

The harmonic mean is zero if any of the a_i are zero. The harmonic mean is not defined for negative numbers. If any of the a_i are missing, they are excluded from the computation.

The harmonic mean is sometimes used to compute an average sample size in an unbalanced experimental design. For example, the following statements compute an average sample size for five samples:

```
sizes = { 8, 12, 23, 10, 8 }; /* sample sizes */
aveSize = harmean( sizes );
print aveSize;
```

Figure 25.177 Harmonic Mean

aveSize
10.486322

HDIR Function

HDIR(*matrix1, matrix2***);**

The HDIR function computes the horizontal direct product of two numeric matrices. This operation is useful in constructing design matrices of interaction effects.

Specifically, the HDIR function performs a direct product on all rows of *matrix1* and *matrix2* and creates a new matrix by stacking these row vectors into a matrix. The *matrix1* and *matrix2* arguments must have the same number of rows, which is also the same number of rows in the result matrix. The number of columns in the result matrix is equal to the product of the number of columns in *matrix1* and *matrix2*.

For example, the following statements produce the matrix `c`, shown in Figure 25.178:

```
a = {1 2,
     2 4,
     3 6};
b = {0  2,
     1  1,
```

```
     0 -1};
c = hdir(a, b);
print c;
```

Figure 25.178 Horizontal Direct Product

c			
0	2	0	4
2	2	4	4
0	-3	0	-6

The HDIR function is useful for constructing crossed and nested effects from main-effect design matrices in ANOVA models.

HEATMAPCONT Call

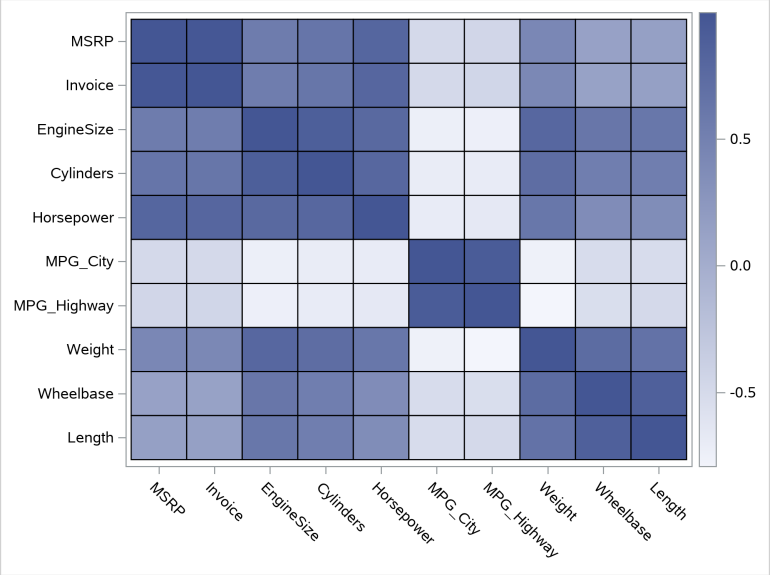
CALL HEATMAPCONT(*x*) **COLORRAMP=***ColorRamp*
SCALE=*scale*
XVALUES=*xValues*
YVALUES=*yValues*
XAXISTOP=*top*
DISPLAYOUTLINES=*outlines*
TITLE=*plotTitle*
LEGENDTITLE=*legendTitle*
LEGENDLOC=*loc*
SHOWLEGEND=*show*
RANGE=*range*
OTHER=*other* **;**

The HEATMAPCONT subroutine is part of the IMLMLIB library. The HEATMAPCONT subroutine displays a heat map of a numeric matrix, *x*, whose values are assumed to vary continuously. The heat map is produced by calling the SGRENDER procedure to render a template, which is created at run time. The argument *x* is a matrix that contains numeric data. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

A simple example follows. The numeric variables from the Sashelp.Cars data set are read into a matrix and the CORR function is used to compute the correlation matrix for those variables. The HEATMAPCONT subroutine creates the image in Figure 25.179, which visualizes the correlations. The correlation matrix has high values (1) on the main diagonal. There are large negative correlations between horsepower and the fuel efficiency variables, MPG_City and MPG_Highway.

```
use Sashelp.Cars;
read all var _NUM_ into Y[c=varNames];
close Sashelp.Cars;
corr = corr(Y);
call HeatmapCont(corr) xvalues=varNames yvalues=varNames;
```

Figure 25.179 A Heat Map of a Correlation Matrix



Specify the required *x* vector inside parentheses and specify all optional arguments outside the parentheses. Titles are specified by using the TITLE= option. Each option corresponds to a statement or option in the graph template language (GTL).

The following list documents the options to the HEATMAPCONT routine:

COLORRAMP= specifies a color ramp that assigns colors to cells in the heat map. You can specify the color ramp in the following ways:

- A character string that matches a predefined color ramp. The "TwoColor" and "ThreeColor" ramps are defined by the current ODS style. Other predefined color ramps are as follows. The first color corresponds to low values; the last color corresponds to high values. Intermediate values are linearly interpolated.
  - "Gray" is a three-color ramp composed of white, gray, and black
  - "BlueRed" is a two-color ramp composed of blue and red
  - "BlueGreenRed" is a four-color ramp composed of blue, cyan, yellow, and red
  - "Rainbow" is a four-color ramp composed of magenta, cyan, yellow, and red
  - "Temperature" is a five-color ramp composed of white, cyan, yellow, red, and black
- A character vector with n color names that are valid in the GTL. For example, the expression {lightblue blue black red lightred} defines a five-color ramp.
- An $n \times 3$ matrix that defines a user-defined color ramp with n colors. Each row specifies an RGB color for the ramp. For example, the expression {255 0 0, 0 255 136, 136 00 255} defines a three-color ramp.
- A character vector with n hexadecimal color values that are valid in the GTL. For example, the expression {CXA6611A CXDFC27D CXF5F5F5 CX80CDC1 CX018571} defines a five-color ramp.

SCALE= specifies how the input matrix should be scaled. Valid values are "None" (the default), "Row", or "Column". For data matrices, variables often have different scales. The "Column" option

standardizes each column to have zero mean and unit standard deviation. The "Row" option standardizes each row to have zero mean and unit standard deviation.

XVALUES= specifies a vector of values for ticks for the X axis. If no values are specified, the column numbers are used. If the vector is a strictly decreasing numerical sequence, then the X axis is reversed.

YVALUES= specifies a vector of values for ticks for the Y axis. If no values are specified, the row numbers are used. If the vector is a strictly decreasing numerical sequence, then the Y axis is reversed.

XAXISTOP= specifies the location of the X axis. The value 0 (the default) specifies that the X axis be displayed at the bottom of the heat map. A nonzero value specifies that the X axis be displayed at the top of the heat map.

DISPLAYOUTLINES= specifies whether to display grid lines for the heat map cells. The value 0 specifies that the no grid lines be displayed. A nonzero value (the default) specifies that grid lines be displayed.

TITLE= specifies a title for the heat map. By default, no title is displayed.

LEGENDTITLE= specifies a title for the legend, which shows the color ramp. By default, no title is displayed.

LEGENDLOC= specifies a location for the legend. Valid values are "Right" (the default), "Left", "Top", and "Bottom".

SHOWLEGEND= specifies whether to display the continuous legend. The default value is 1, which shows the legend. To suppress the legend, specify 0.

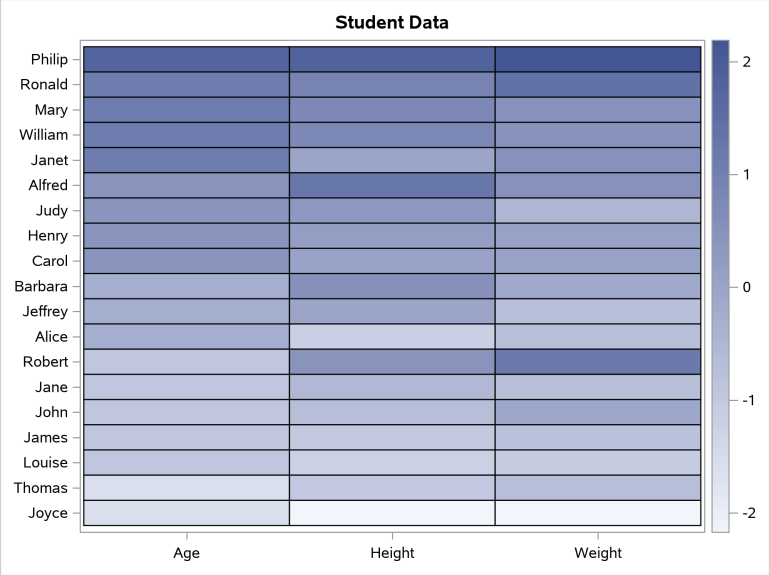
RANGE= specifies the range of the color ramp. By default, the range of the data is used. You can specify a two-element array to change the range. For example, RANGE={−1, 1} specifies that the color ramp colors values on the interval [−1, 1]. You can use missing values to specify the minimum and maximum values. Thus RANGE={−1, .} specifies that −1 is the lower endpoint of the range and that the maximum data value should be used for the upper endpoint.

OTHER= specifies any valid GTL plot statement that is compatible with the HEATMAPPARM statement. For example, you can specify a vertical reference line by specifying "REFERENCELINE x=1;". You must include the ending semicolon.

The following example shows how to create a heat map that uses the SCALE=, XVALUES=, YVALUES=, and TITLE= options.

```
use Sashelp.Class;
read all var _NUM_ into Students[c=varNames r=Name];
close Sashelp.Class;

/* sort data in descending order according to Age and Height */
call sortndx(idx, Students, 1:2, 1:2);
Students = Students[idx,];
Name = Name[idx];

/* standardize each column */
call HeatmapCont(Students) scale="Col"
        xvalues=varNames yvalues=Name title="Student Data";
```

Figure 25.180 Heat Map of a Data Matrix

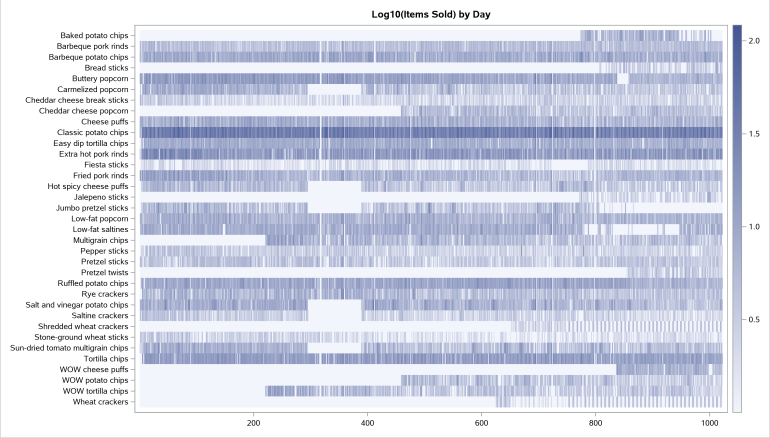


In Figure 25.180, you can see that Philip is the biggest student, Joyce is the smallest, Robert is heavy for his age, and Alfred is tall for his age.

For a more complicated visualization of a data matrix, the following statements visualize the number of snack items sold at a fictitious store over the course of 1,022 days. The heat map that uses the YVALUES=, DISPLAYOUTLINES=, and TITLE= options. Because the quantity of items sold range over two orders of magnitude (from 0 to 121), a logarithmic transformation is used to transform the data.

```
use Sashelp.Snacks;
read all var {QtySold Date Product};
close Sashelp.Snacks;

QtySold = choose(QtySold>=0, QtySold, .); /* remove invalid quantities */
Names = unique(Product);
x = shape(QtySold, ncol(Names));

ods graphics / height=800 width=1400;
call HeatmapCont(log10(x+1)) yvalues=Names displayoutlines=0
                             title="Log10(Items Sold) by Day";
```

Figure 25.181 Time Series Visualization for 35 Snack Items



In Figure 25.181, horizontal white bands indicate periods of time for which a particular snack item was not sold. Vertical white bands indicate days for which the store was closed. Dark shades, such as for "classic potato chips" and "tortilla chips," indicate items for which the average number of units sold each day was about $10^2 = 100$. Lighter shades, such as for "fiesta sticks" and "stone-ground wheat sticks," indicate less popular items.

HEATMAPDISC Call

CALL HEATMAPDISC(*x***) COLORRAMP=***ColorRamp***
XVALUES=***xValues***
YVALUES=***yValues***
XAXISTOP=***top***
DISPLAYOUTLINES=***outlines***
TITLE=***plotTitle***
LEGENDTITLE=***legendTitle***
LEGENDLOC=***loc***
SHOWLEGEND=***show***
OTHER=***other*** ;**

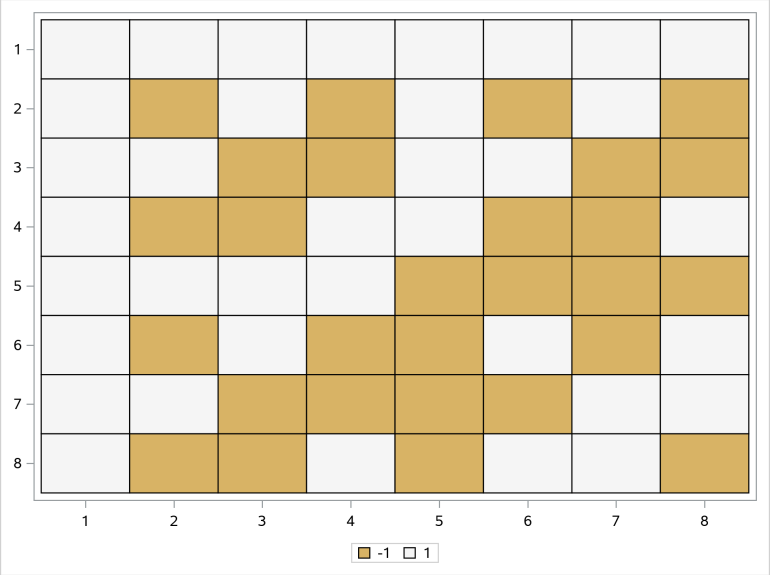
The HEATMAPDISC subroutine is part of the IMLMLIB library. The HEATMAPDISC subroutine displays a heat map of a numeric or character matrix, *x*, whose values are assumed to have a small number of discrete values. The heat map is produced by calling the SGRENDER procedure to render a template, which is created at run time. The argument *x* is a matrix that contains numeric or character data. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

In addition to visualizing matrices with discrete values, you can use the HEATMAPDISC subroutine to visualize quantiles of a continuous variable.

A simple example follows. The HADAMARD function generates an 8×8 matrix, each element is either 1 or -1. The HEATMAPDISC subroutine creates the image in Figure 25.182, which uses two colors to visualize the matrix.

```
h = hadamard(8);
run HeatmapDisc(h);
```

Figure 25.182 A Heat Map of a Matrix of Two Values



Specify the required *x* vector inside parentheses and specify all optional arguments outside the parentheses. Titles are specified by using the TITLE= option. Each option corresponds to a statement or option in the graph template language (GTL).

Except for the SCALE= option, the options for the HEATMAPDISC subroutine are the same as for the HEATMAPCONT subroutine.

You can use the PALETTE function to obtain colors from a wide variety of discrete color palettes.

By default, the row and column numbers of a matrix are used to label the heat map that is created by the HEATMAPDISC or HEATMAPCONT subroutine. The heat map is displayed in the same way that a matrix is printed: row 1 is at the top, row 2 is next, and so on. Columns increase from left to right. That means that the Y axis points down by default; the X axis points to the right.

Just as displaying a heat map is similar to printing a matrix, the XVALUES= and YVALUES= options are similar to the COLNAME= and ROWNAME= options (respectively) on the PRINT statement. You can use the XVALUES= and YVALUES= options to change the labels for the axes ticks, but these options do not change the values of the data. The following program shows a PRINT statement followed by a call to the HEATMAPDISC subroutine. The output is not shown, but in each case the printed output matches the graphical output.

```
m = {1 2 3, 4 5 6, 1 2 3, 4 5 6};
XLbl = char(1:ncol(m));
YLbl = char(1:nrow(m));

/* standard printing: row labels increase down the page */
print m[colname=XLbl rowname=YLbl];
run HeatmapDisc(m);  /* by default, axes labels are row/col numbers */
```

```
/* reverse Y labels; matrix is unchanged! */
Yrev = char(nrow(m):1);
print m[colname=XLbl rowname=Yrev];
run HeatmapDisc(m)   yvalues=Yrev;

/* to make the Y  axis point up, flip the rows and the labels */
r = m[nrow(m):1, ];
print r[colname=XLbl rowname=Yrev];
run HeatmapDisc(r)   yvalues=Yrev;
```

HERMITE Function

HERMITE(*matrix***);**

The HERMITE function uses elementary row operations to compute the Hermite normal form of a matrix. For square matrices this normal form is upper triangular and idempotent.

If the argument is square and nonsingular, the result is the identity matrix. In general the result satisfies the following four conditions (Graybill 1969):

- It is upper triangular.
- It has only values of 0 and 1 on the diagonal.
- If a row has a 0 on the diagonal, then every element in that row is 0.
- If a row has a 1 on the diagonal, then every off-diagonal element is 0 in the column in which the 1 appears.

The following statements compute an example from Graybill (1969):

```
a = {3  6  9,
     1  2  5,
     2  4 10};
h = hermite(a);
print h;
```

Figure 25.183 Hermite Matrix

h		
1	2	0
0	0	0
0	0	1

If the argument is a square matrix, then the Hermite normal form can be transformed into the row-echelon form by rearranging rows in which all values are 0.

HISTOGRAM Call

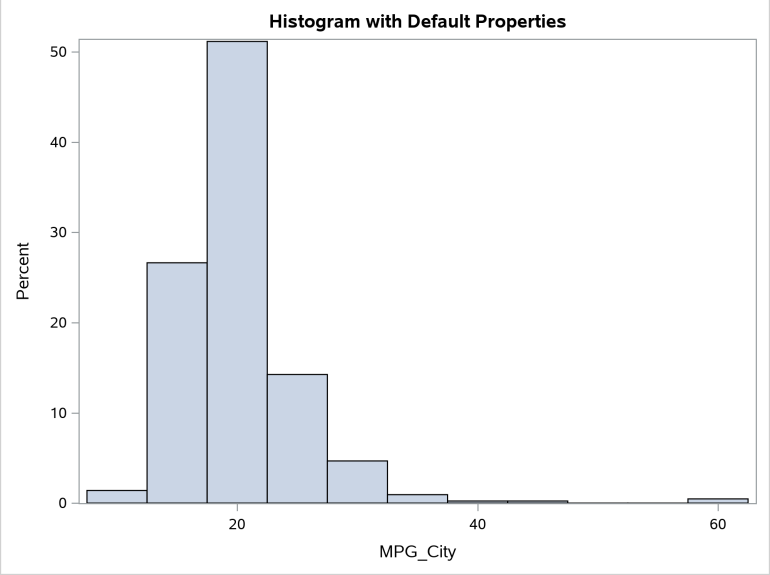
CALL HISTOGRAM(*x***) SCALE="Count" | "Percent" | "Proportion"**
DENSITY="Normal" | "Kernel"
REBIN=*{BinStart, BinWidth}*
GRID={"X" <,"Y" >}
LABEL={*XLabel* <,*YLabel*>**}**
XVALUES=*xValues*
YVALUES=*yValues*
PROCOPT=*ProcOption*
OTHER=*Stmts* **;**

The HISTOGRAM subroutine displays a histogram by calling the SGPLOT procedure. The required argument *x* is a numeric vector that contains the data to plot. The remaining arguments are optional and can be specified in any order. The HISTOGRAM subroutine is not a comprehensive interface to the SGPLOT procedure. It is intended for creating simple histogram for exploratory data analysis. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

A simple example follows:

```
use Sashelp.Cars;
read all var {MPG_City};
close Sashelp.Cars;

title "Histogram with Default Properties";
call Histogram(MPG_City);
```

Figure 25.184 A Histogram



Specify the required *x* vector inside parentheses and specify all optional arguments outside the parentheses. Use the global TITLE and FOOTNOTE statements to specify titles and footnotes. Each option corresponds to a statement or option in the SGPLOT procedure.

The following options correspond to options in the HISTOGRAM or DENSITY statement in the SGPLOT procedure:

SCALE= specifies the scaling to apply to the vertical axis of the histogram. Valid options are "Count" (the default), "Percent," and "Proportion."

DENSITY= specifies whether to overlay the density estimate on the histogram. The valid values are as follows:

- **DENSITY={"Normal"}** overlays a normal density estimate.
- **DENSITY={"Kernel"}** overlays a kernel density estimate.
- **DENSITY={"Normal", "Kernel"}** overlays a normal and a kernel density estimate.

REBIN= specifies two numerical values that set the location of the first bins and the width of bins. An option of the form **REBIN={*x0*, *h*}** corresponds to the BINSTART=*x0* and BINWIDTH=*h* options in the HISTOGRAM statement in PROC SGPLOT.

The HISTOGRAM subroutine also supports the following options. The BAR subroutine documents these options and gives an example of their usage.

GRID= specifies whether to display grid lines for the X or Y axis.

LABEL= specifies axis labels for the X or Y axis.

XVALUES= specifies a vector of values for ticks for the X axis.

YVALUES= specifies a vector of values for ticks for the Y axis.

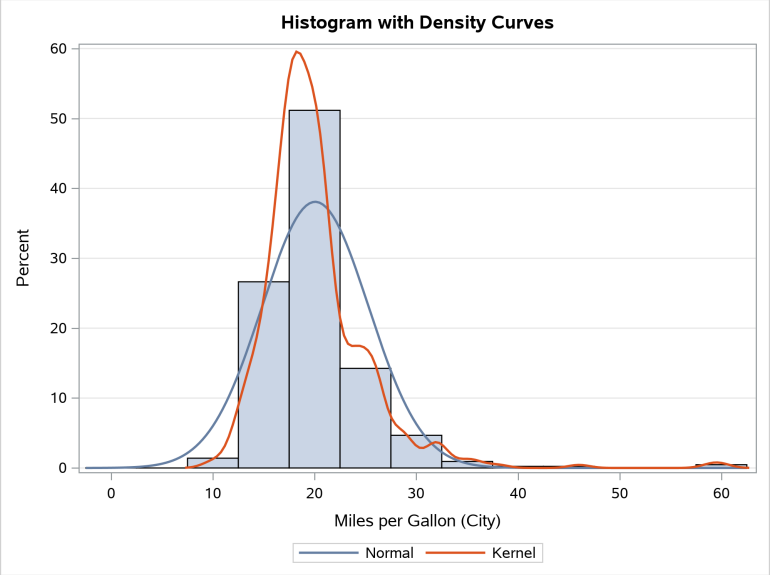
PROCOPT= specifies options in the PROC SGPLOT statement.

OTHER= specifies statements in the SGPLOT procedure.

The following statements create a histogram, overlay density estimates, and specify several options:

```
use Sashelp.Cars;
read all var {MPG_City};
close Sashelp.Cars;

title "Histogram with Density Curves";
call Histogram(MPG_City)
          scale = "Percent"
          density={"Normal" "Kernel"}
          rebin={0 5}
          grid="y"
          label="Miles per Gallon (City)"
          xvalues = do(0, 60, 10);
```

Figure 25.185 A Histogram with Overlaid Densities



HOMOGEN Function

HOMOGEN(*matrix*);

The HOMOGEN function solves the homogeneous system of linear equations $\mathbf{A} * \mathbf{X} = \mathbf{0}$ for $\mathbf{X}$. For at least one solution vector $\mathbf{X}$ to exist, the $m \times n$ matrix $\mathbf{A}$, $m \geq n$, has to be of rank $r < n$. The HOMOGEN function computes an $n \times (n - r)$ column orthonormal matrix $\mathbf{X}$ with the properties that $\mathbf{A} * \mathbf{X} = \mathbf{0}$ and $\mathbf{X}'\mathbf{X} = \mathbf{I}$. In other words, the columns of $\mathbf{X}$ form an orthonormal basis for the nullspace of A.

If $\mathbf{A}'\mathbf{A}$ is ill-conditioned, rounding-error problems can occur in determining the correct rank of $\mathbf{A}$ and in determining the correct number of solutions $\mathbf{X}$.

The following statements compute an example from Wilkinson and Reinsch (1971):

```
a = {22  10   2   3   7,
     14   7  10   0   8,
     -1  13  -1 -11   3,
     -3  -2  13  -2   4,
      9   8   1  -2   4,
      9   1  -7   5  -1,
      2  -6   6   5   1,
      4   5   0  -2   2};
x = homogen(a);
print x;
```

Figure 25.186 Solutions to a Homogeneous System

x	
-0.419095	0
0.4405091	0.4185481
-0.052005	0.3487901
0.6760591	0.244153
0.4129773	-0.802217

In addition, you can use the HOMOGEN function to determine the rank of an $m \times n$ matrix $\mathbf{A}$ where $m \geq n$ by counting the number of columns in the matrix $\mathbf{X}$.

If A is an $n \times m$ matrix, then, in addition to the memory allocated for the return matrix, the HOMOGEN function temporarily allocates an $n^2 + nm$ array for performing its computation.

I Function

I(*dim***);**

The I function creates an identity matrix with *dim* rows and columns. The diagonal elements of an identity matrix are ones; all other elements are zeros. The value of *dim* must be an integer greater than or equal to 1. Noninteger operands are truncated to their integer part.

For example, the following statements compute a 3×3 identity matrix:

```
a = I(3);
print a;
```

Figure 25.187 An Identity Matrix

a		
1	0	0
0	1	0
0	0	1

ICCEPSTRUM Function

ICCEPSTRUM(*x* **<** **,** *nd***>);**

The ICCEPSTRUM function computes the inverse complex cepstrum of a time series. The inverse complex cepstrum of a time series x is defined as

$$y = \frac{1}{2\pi} \int_{-\pi}^{\pi} \exp[X(e^{i\omega})] e^{i\omega n} d\omega$$

where $X(e^{i\omega})$ is the Fourier transform of the input time series and $i = \sqrt{-1}$.

The input arguments to the ICCEPSTRUM function are as follows:

x specifies the input data vector. It can be either a column vector or a row vector. Any missing value in this vector is replaced with 0.

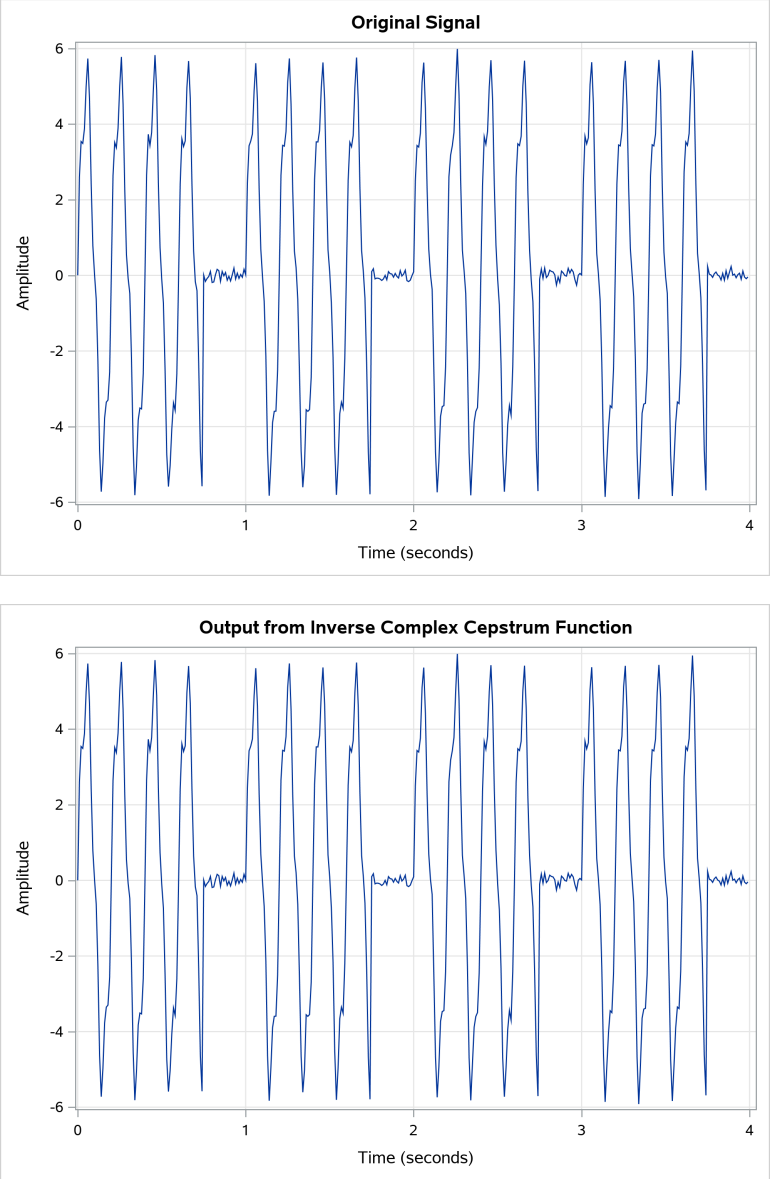
nd is a nonnegative integer that specifies the number of samples of (circular) delay to be removed when the inverse complex cepstrum is computed. This value is the output when the CCEPSTRUM call is used to compute the complex cepstrum.

The ICCEPSTRUM function returns a column vector that contains the inverse complex cepstrum of the input data. The output has the same length as the input data.

The following example shows how to use the ICCEPSTRUM function to invert the complex cesptrum exactly. The output from the ICCEPSTRUM function is the same as the input to the CCEPSTRUM call.

```
sf  = 100;    /* sampling frequency */
T = 1.0/sf;   /* sampling period */
N = 400;
t = do(0, (N-1)*T, T);
t1 = t[, 1:N*3/16];

f1 = 5;
f2 = 20;
pi = constant("pi");
x1 = 5*sin(2*pi*f1*t1);
x2 = sin(2*pi*f2*t1);
x3 = x1 + x2;        /* sum of two sinusoidal signals */
x4 = j(1, N/16, 0);
x5 = repeat(x3 || x4, 1, 4);

call randseed(1);
noise_mean = 0;
noise_std  = 0.1;
xn = randfun(N, "Normal", noise_mean, noise_std);
x0 = x5` + xn;  /* input signal */

call ccepstrum(y, nd, x0);

x = iccepstrum(y, nd);

title "Original Signal";
call series(t,x0) grid= {X Y}
                  label={"Time (seconds)" "Amplitude"};

title "Output from Inverse Complex Cepstrum Function";
call series(t,x) grid= {X Y}
                 label={"Time (seconds)" "Amplitude"};
```

Figure 25.188 Inverse Complex Cepstrum Example



IF-THEN/ELSE Statement

IF *expression* **THEN** *statement1* **;**

ELSE *statement2* **;**

The IF-THEN/ELSE statement conditionally executes statements. The ELSE statement is optional.

The arguments to the IF-THEN/ELSE statement are as follows:

expression is an expression that is evaluated for being true or false.

statement1 is a statement executed when *expression* is true.

statement2 Is a statement executed when *expression* is false.

The IF statement contains an expression to be evaluated, the keyword THEN, and an action to be taken when the result of the evaluation is true.

The ELSE statement optionally follows the IF statement and specifies an action to be taken when the IF expression is false. The expression to be evaluated is often a comparison. For example:

```
a = {0, 5, 1, 10};
if max(a)<20 then
   p = 0;
else
   p = 1;
```

The IF statement results in the evaluation of the condition **max(a)<20**. If the largest value found in the matrix **a** is less than 20, the scalar value **p** is set to 0. Otherwise, **p** is set to 1. See the description of the MAX function for details.

When the condition to be evaluated is a matrix expression, the result of the evaluation is another matrix. If all values of the result matrix are nonzero and nonmissing, the condition is true; if any element in the result matrix is 0 or missing, the condition is false. This evaluation is equivalent to using the ALL function.

For example, consider the following statements:

```
a = { 1 2, 3 4};
b = {-1 0, 0 1};
if a>b then do;
   /* statements */
end;
```

This code produces the same result as the following statements:

```
if all(a>b) then do;
   /* statements */
end;
```

IF statements can be nested within the clauses of other IF or ELSE statements. There is no limit on the number of nesting levels. Consider the following example:

```
if a>b then
   if a>abs(b) then do;
      /* statements */
   end;
```

Consider the following statements:

```
if a^=b then do;
   /* statements */
end;
if ^(a=b) then do;
   /* statements */
end;
```

The two IF statements are equivalent. In each case, the THEN clause is executed only when all corresponding elements of **a** and **b** are unequal.

Evaluation of the following statement requires only one element of **a** and **b** to be unequal in order for the expression to be true:

```
if any(a^=b) then do;
   /* statements */
end;
```

If a symbol is empty (undefined), the IF statement evaluates to false.

IFFT Function

IFFT(*f*);

The IFFT function computes the inverse finite Fourier transform of a matrix *f*, where *f* is an $np \times 2$ numeric matrix.

The IFFT function expands a set of sine and cosine coefficients into a sequence equal to the sum of the coefficients times the sine and cosine functions. The argument *f* is an $np \times 2$ matrix; the value returned is an $n \times 1$ vector.

If the element in the last row and second column of *f* is exactly 0, then n is $2np - 2$; otherwise, n is $2np - 1$.

The inverse finite Fourier transform of a two column matrix $\mathbf{F}$, denoted by the vector $\mathbf{x}$, is

$$x_i = F_{1,1} + 2\sum_{j=2}^{np}\left(F_{j,1}\cos\left(\frac{2\pi}{n}(j-1)(i-1)\right) + F_{j,2}\sin\left(\frac{2\pi}{n}(j-1)(i-1)\right)\right) + q_i$$

for $i = 1, \ldots, n$, where $q_i = (-1)^i \mathbf{F}_{np,1}$ if n is even, or $q = 0$ if n is odd.

For the most efficient use of the IFFT function, n should be a power of 2. If n is a power of 2, a fast Fourier transform is used (Singleton 1969); otherwise, a Chirp-Z algorithm is used (Monro and Branch 1977).

The expression IFFT(FFT(X)) returns n times $\mathbf{x}$, where n is the dimension of $\mathbf{x}$. If f is not the Fourier transform of a real sequence, then the vector generated by the IFFT function is not a true inverse Fourier transform. However, applications exist in which the FFT function and the IFFT function can be used for operations on multidimensional or complex data (Gentleman and Sande 1966; Nussbaumer 1982).

As an example, the convolution of two vectors $\mathbf{x}$ ($n \times 1$) and $\mathbf{y}$ ($m \times 1$) can be accomplished by using the following module:

```
start conv(u,v);
/* w = conv(u,v) convolves vectors u and v.
 * Algebraically, convolution is the same operation as
 * multiplying the polynomials whose coefficients are the
 * elements of u and v. Straight convolution is too slow,
 * so use the FFT.
 *
 * Both of u and v are column vectors.
 */
   m = nrow(u);
   n = nrow(v);

   wn = m + n - 1;
```

```
   /* find p so that 2##(p-1) < wn <= 2##p */
   p = ceil( log(wn)/ log(2) );
   nice = 2##p;

   a = fft( u // j(nice-m,1,0) );
   b = fft( v // j(nice-n,1,0) );
   /* complex multiplication of a and b */
   wReal = a[,1]#b[,1] - a[,2]#b[,2];
   wImag = a[,1]#b[,2] + a[,2]#b[,1];
   w = wReal || wImag;
   z=ifft(w);
   z = z[1:wn,1] / nice;  /* take real part and first wn elements */
   return (z);
finish;

/* example of convolution of two waveforms */
TimeStep = 0.01;
t = T( do(0,8,TimeStep) );

Signal = j(nrow(t),1,5);
Signal[ loc(t>4) ] = -5;

ImpulseResponse = j(nrow(t),1,0);
ImpulseResponse[ loc(t<=2) ] = 3;

/* The time domain for this convolution is [0,16]
   with the same time step.
   For waveforms, rescale amplitude by the time step. */
y = conv(Signal,ImpulseResponse) * TimeStep;
```

Other applications of the FFT and IFFT functions include windowed spectral estimates and the inverse autocorrelation function.

IFFTC function

IFFTC(*x*);

The IFFTC function computes the inverse finite Fourier transform of a complex-valued vector, where *x* is an $n \times 2$ (or $2 \times n$) numeric matrix that represents a complex vector. The first column (or row) of *x* specifies the real components, and the second column (or row) specifies the imaginary components. (If *x* is a 2×2 matrix, the first row contains the real components, and the second row contains the imaginary components.)

The IFFTC function returns an $n \times 2$ matrix that contains the complex Fourier coefficients that correspond to *x*. The first column of the matrix contains the real components of the Fourier coefficients, and the second column contains the imaginary components.

The inverse Fourier transform is related to the Fourier transform. The expression **`IFFTC(FFTC(x))`** is equal to *nx* for every vector *x*. Similarly, **`FFTC(IFFTC(x))`** equals *nx*.

The Fourier coefficients can be expressed in terms of a complex matrix multiplication. If the input *x* represents a complex column vector z of dimension n and if $\omega = \exp(-2\pi \mathbf{i}/n)$, where $\mathbf{i} = \sqrt{-1}$, then define a $n \times n$

matrix W as

$$W = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

Then the complex vector of Fourier coefficients is simply the product Wz.

The following example demonstrates the IFFTC function:

```
/* sample every millisecond from 0 to 10 seconds */
t = do(0,10,0.001);
pi = constant("pi");
/* complex signal takes values on the unit circle and makes
   three full rotations per second in the *clockwise* direction */
freq = 3;
x = cos(2*pi*freq*t); y = sin(-2*pi*freq*t);
Signal = t(x) || t(y);
f = fftc(Signal);

/* reconstruct the original signal from the Fourier coefficients */
z = ifftc(f) / nrow(f);
maxDiff = max(abs(Signal - z));  /* this should be essentially zero */
print maxDiff;
```

Figure 25.189 Reconstructing a Complex Signal

Output from Inverse Complex Cepstrum Function

maxDiff
4.875E-12

IMPORTDATASETFROMR Call

CALL IMPORTDATASETFROMR(*SAS-data-set, RExpr***);**

You can use the IMPORTDATASETFROMR subroutine to transfer data from an R data frame to a SAS data set. It is easier to read the subroutine name when it is written in mixed case: ImportDataSetFromR.

The arguments for the subroutine are as follows:

SAS-data-set is a literal string or a character matrix that specifies the two-level name of a SAS data set (for example, Work.MyData).

RExpr is a literal string or a character matrix that specifies the name of an R data frame or, in general, an R expression that can be coerced to an R data frame.

You can call the subroutine provided that the following statements are true:

1. The R statistical software is installed on the SAS workspace server.

2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See the section "The RLANG System Option" on page 244.)

The following statements create a data frame in R named RData and copy the data into Work.MyData. The SHOW CONTENTS statement is then used to display attributes of the Work.MyData data, which demonstrates that the data were successfully transferred.

```
proc iml;
submit / R;
z = c('a','b','c','d','e')
RData <- data.frame(x=1:5, y=(1:5)^2, z=z)
endsubmit;

call ImportDataSetFromR("Work.MyData", "RData");

use Work.MyData;
show contents;
close Work.MyData;
```

Figure 25.190 Contents of a SAS Data Set Created from R Data

Output from Inverse Complex Cepstrum Function

```
DATASET : WORK.MYDATA.DATA

 VARIABLE                         TYPE  SIZE
 -------------------------------  ----  ----
 X                                num      8
 Y                                num      8
 Z                                char     1

Number of Variables   : 3
Number of Observations: 5
```

You can transfer data from a SAS data set into an R data frame by using the EXPORTDATASETTOR call. See Chapter 14, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

The names of the variables in the SAS data set are derived from the names of the variables in the R data frame. The following rules are used to convert an R variable name to a valid SAS variable name:

1. If the name is longer than 32 characters, it is truncated to 32 characters.
2. A SAS variable name must begin with one of the following characters: 'A'–'Z', 'a'–'z', or the underscore (_). Therefore, if the first character is not a valid beginning character, it is replaced by an underscore (_).
3. A SAS variable name can contain only the following characters: 'A'–'Z', 'a'–'z', '0'–'9', or the underscore (_). Therefore, if any of the remaining characters is not valid in a SAS variable name, it is replaced by an underscore.

4. If the resulting name duplicates an existing name in the data set, a number is appended to the name to make it unique. If appending the number causes the length of the name to exceed 32 characters, the name is truncated to make room for the number.

IMPORTMATRIXFROMR Call

CALL IMPORTMATRIXFROMR(*IMLMatrix, RExpr***);**

You can use the IMPORTMATRIXFROMR subroutine to transfer data from an R data frame to a SAS/IML matrix. It is easier to read the subroutine name when it is written in mixed case: ImportMatrixFromR.

The arguments to the subroutine are as follows:

IMLMatrix	is a SAS/IML matrix to contain the data you want to transfer.
RExpr	is a literal string or a character matrix that specifies the name of an R matrix, data frame, or an R expression that can be coerced to an R data frame.

If the *RExpr* argument is a data frame, then the resulting SAS/IML matrix has columns that correspond to variables from the data frame. If the first variable in the data frame is a numeric variable, a numeric matrix is created from all numeric variables in the data frame. If the first variable in the data frame is a character variable, a character matrix is created from all character variables in the data frame.

You can call the subroutine provided that the following statements are true:

1. The R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See the section "The RLANG System Option" on page 244.)

The following statements define an R matrix and copy the data from the matrix to a SAS/IML matrix:

```
proc iml;
submit / R;
m <- matrix( c(1,2,3,4,NA,6), nrow=2, byrow=TRUE)
endsubmit;

call ImportMatrixFromR(a, "m");
print a;
```

To demonstrate that the data were successfully transferred, the PRINT statement is used to print the values of the **a** matrix. The output is shown in Figure 25.191. Note that the R missing value (**NA**) in the R matrix **m** was automatically converted to the SAS missing value in the SAS/IML matrix, **a**.

Figure 25.191 Data from R

Output from Inverse Complex Cepstrum Function

a		
1	2	3
4	.	6

You can transfer data from a SAS/IML matrix into an R matrix by using the EXPORTMATRIXTOR call. See Chapter 14, "Calling Functions in the R Language," for details about transferring data between R and SAS software.

IMPORTTABLEFROMR Function

IMPORTTABLEFROMR(*RExpr***);**

You can use the IMPORTTABLEFROMR function to transfer data from an R data frame to a SAS/IML table. It is easier to read the function name when it is written in mixed case: ImportTableFromR.

The argument for the function follows:

RExpr is a literal string or a character matrix that specifies the name of an R data frame or, in general, an R expression that can be coerced to an R data frame.

The following statements create a data frame in R named RData and copy the data into the SAS/IML table named **tbl**.

```
proc iml;
submit / R;
z = c('a','b','c','d','e')
RData <- data.frame(x=1:5, y=(1:5)^2, z=z)
endsubmit;

tbl = ImportTableFromR("RData");
call TablePrint(tbl);
```

Figure 25.192 Contents of a SAS Data Set Created from R Data

Output from Inverse Complex Cepstrum Function

	tbl		
Obs	X	Y	Z
1	1	1	a
2	2	4	b
3	3	9	c
4	4	16	d
5	5	25	e

For more information about transferring data between R data frames and SAS/IML tables, see the ImportDataSetFromR subroutine.

INDEX Statement

INDEX *variables* | **NONE ;**

The INDEX statement creates an index for the named variables in the current input SAS data set. An index is created for each variable listed, provided that the variable does not already have an index. Current retrieval is

set to the last variable indexed. Subsequent I/O operations such as LIST, READ, FIND, and DELETE can use this index to retrieve observations from the data. The indices are automatically updated when a data set is edited with the APPEND, DELETE, or REPLACE statements. Only one index is in effect at any given time. The SHOW CONTENTS command indicates which index is in use.

For example, the following statements copy the Sasuser.Class data set and create indexes for the Name and Sex variables. Current retrieval is set to use the Sex variable, as shown in Figure 25.193.

```
data class;
   set Sashelp.Class;
run;

proc iml;
use class;
index name sex;
list all;
close class;
```

Figure 25.193 Result of Listing Observations of an Indexed Data Set

Output from Inverse Complex Cepstrum Function

OBS	Name	Sex	Age	Height	Weight
2	Alice	F	13.0000	56.5000	84.0000
3	Barbara	F	13.0000	65.3000	98.0000
4	Carol	F	14.0000	62.8000	102.5000
7	Jane	F	12.0000	59.8000	84.5000
8	Janet	F	15.0000	62.5000	112.5000
11	Joyce	F	11.0000	51.3000	50.5000
12	Judy	F	14.0000	64.3000	90.0000
13	Louise	F	12.0000	56.3000	77.0000
14	Mary	F	15.0000	66.5000	112.0000
1	Alfred	M	14.0000	69.0000	112.5000
5	Henry	M	14.0000	63.5000	102.5000
6	James	M	12.0000	57.3000	83.0000
9	Jeffrey	M	13.0000	62.5000	84.0000
10	John	M	12.0000	59.0000	99.5000
15	Philip	M	16.0000	72.0000	150.0000
16	Robert	M	12.0000	64.8000	128.0000
17	Ronald	M	15.0000	67.0000	133.0000
18	Thomas	M	11.0000	57.5000	85.0000
19	William	M	15.0000	66.5000	112.0000

The INDEX NONE statement can be used to set retrieval back to physical order.

When a WHERE clause is being processed, the SAS/IML language automatically determines which index to use, if any. The decision is based on the variables and operators involved in the WHERE clause, and the decision criterion is based on the efficiency of retrieval.

INFILE Statement

INFILE *operand* < *options* > **;**

The INFILE statement opens an external file for input or, if the file is already open, makes it the current input file. A subsequent INPUT statement reads from the specified file.

The arguments to the INFILE statement are as follows:

operand	is either a predefined file name or a quoted string that contains in parentheses the file name or character expression that refers to the path name.
options	are explained in the following list.

The valid values for the *options* argument are as follows:

LENGTH=variable
specifies a variable into which the length of a record is stored.

RECFM=N
specifies that the file be read in as a pure binary file rather than as a file with record separator characters. To do this, you must use the byte operand (<) in the INPUT statement to get new records rather than use separate input statements or the new line (/) operator.

The following keywords control how a program behaves when an INPUT statement tries to read past the end of a record. The default behavior is STOPOVER.

FLOWOVER
enables the INPUT statement to go to the next record to obtain values for the variables.

MISSOVER
tolerates attempted reading past the end of the record by assigning missing values to variables read past the end of the record.

STOPOVER
treats going past the end of a record as an error condition, which triggers an end-of-file condition.

Several examples of INFILE statements follow:

```
filename in1 "student.dat";        /* specify file name IN1   */
infile in1;                        /* infile path name        */

infile "student.dat";              /* path by quoted literal */

infile "student.dat" missover;     /* use missover option    */
```

See Chapter 8 for further information.

INPUT Statement

INPUT < *variables* > < *informats* > < *record-directives* > < *positionals* > **;**

The INPUT statement reads records from the current input file, placing the values into matrices. The INFILE statement sets up the current input file. See Chapter 8 for details.

The INPUT statement supports the following arguments:

variables specify the variable or variables you want to read from the current position in the record. Each variable can be followed immediately by an input format specification.

informats specify an input format. These are of the form *w.d* or *$w.* for standard numeric and character informats, respectively, where *w* is the width of the field and *d* is the decimal parameter, if any. You can also use a named SAS format such as BEST*w.d*. Also, you can use a single $ or & for list input applications. If the width is unspecified, the informat uses list-input rules to determine the length by searching for a blank (or comma) delimiter. The special format $RECORD. is used for reading the rest of the record into one variable. For more information about formats, see *SAS Language Reference: Dictionary*.

Record holding is always implied for RECFM=N binary files, as if the INPUT statement has a trailing @ sign. For more information, see Chapter 8.

Examples of valid INPUT statements follow:

```
input x y;
input @1 name $ @20 sex $ @(20+2) age 3.;

eight=8;
input >9 <eight  number2 ib8.;
```

The following example uses binary input:

```
file "out2.dat" recfm=n ;
number=499; at=1;
do i = 1 to 5;
   number=number+1;
   put >at number ib8.; at=at+8;
end;
closefile "out2.dat";

infile "out2.dat" recfm=n;
size=8;     /* 8 bytes */
do pos=1 to 33 by size;
   input >pos number ib8.;
   print number;
end;
```

record-directives are used to advance to a new record. *Record-directives* are the following:

holding @ sign is used at the end of an INPUT statement to hold the current record so that you can continue to read from the record with later INPUT statements. Otherwise, the next record is used for the next INPUT statement.

/ advances to the next record.

> *operand* specifies that the next record to be read start at the indicated byte position in the file (for RECFM= N files only). The *operand* is a literal number, a variable name, or an expression in parentheses.

< *operand* specifies that the indicated number of bytes are read as the next record. The record directive must be specified for binary files (RECFM=N). The *operand* is a literal number, a variable name, or an expression in parentheses.

positionals specifies a specific column on the record. The *positionals* are the following:

@ *operand* specifies a column, where *operand* is a literal number, a variable name, or an expression in parentheses. For example, @30 means to go to column 30. The operand can also be a character operand when pattern searching is needed. For more information, see Chapter 8.

+ *operand* skips the indicated number of columns. The *operand* is a literal number, a variable name, or an expression in parentheses.

INSERT Function

INSERT(*x, y, row*< *, column*>**);**

The INSERT function inserts one matrix inside another.

The arguments to the INSERT function are as follows:

x is the target matrix. It can be either numeric or character.

y is the matrix to be inserted into the target. It can be either numeric or character, depending on the type of the target matrix.

row is the row where the insertion is to be made.

column is the column where the insertion is to be made.

The INSERT function returns the result of inserting the matrix *y* inside the matrix *x* at the place specified by the *row* and *column* arguments. This is done by splitting *x* either horizontally or vertically before the row or column specified and concatenating *y* between the two pieces. Thus, if *x* has m rows and n columns, *row* can range from 0 to $m + 1$ and *column* can range from 0 to $n + 1$.

It is not possible to insert in both dimensions simultaneously, so either *row* or *column* must be 0, but not both. The *column* argument is optional and defaults to 0. Also, the matrices must conform in the dimension in which they are joined.

The following statements show two examples of the INSERT function. Figure 25.194 shows that the matrix `c` is the result of inserting matrix `b` prior to the second row of matrix `a`. The matrix `d` is the result of inserting matrix `b` after the second column of matrix `a`.

```
a = {1 2, 3 4};
b = {5 6, 7 8};
c = insert(a, b, 2, 0);
d = insert(a, b, 0, 3);
print c, d;
```

Figure 25.194 Inserted Matrices

Output from Inverse Complex Cepstrum Function

c
1 2
5 6
7 8
3 4

d
1 2 5 6
3 4 7 8

INT Function

INT(*matrix***);**

The INT function truncates the decimal portion of the value of the argument. The integer portion of the value of the argument remains. The INT function takes the integer value of each element of the argument matrix, as shown in the following statements:

```
y = 2.8;
b = int(y);
x={12.95  10.9999999999999,
   -30.5  1e-6};
c = int(x);
print b, c;
```

Figure 25.195 Truncated Values

Output from Inverse Complex Cepstrum Function

b
2

c
12 11
-30 0

In Figure 25.195, notice that the value 11 is returned as the second element of `c`. If a value is within 10^{-12} of an integer, the INT function rounds up.

INV Function

INV(*matrix***);**

The INV function computes the inverse of a square and nonsingular matrix.

For $\mathbf{G} = \text{INV}(\mathbf{A})$ the inverse has the properties

$$\mathbf{GA} = \mathbf{AG} = \text{identity}$$

To solve a system of linear equations $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$, you can use the expression `x = inv(a)*b`. However, the SOLVE function is more accurate and efficient for this task.

The following statements compute a matrix inverse and solve a linear system:

```
A = {0 0 1 0 1,
     1 0 0 1 0,
     0 1 1 0 1,
     1 0 0 0 1,
     0 1 0 1 0};

 b = {9, 4, 10, 8, 2};

 /* find inverse and solve linear system */
 Ainv = inv(A);
 x1 = Ainv*b;

 /* solve by using a more efficient algorithm */
 x2 = solve(A,b);
 print x1 x2;
```

Figure 25.196 Solving a Linear System

Output from Inverse Complex Cepstrum Function

x1	x2
3	3
1	1
4	4
1	1
5	5

The INV function uses an LU decomposition followed by back substitution to solve for the inverse, as described in Forsythe, Malcom, and Moler (1967).

The INV function (in addition to the DET function and SOLVE function) uses the following criterion to decide whether the input matrix, $\mathbf{A} = [a_{ij}]_{i,j=1,\ldots,n}$, is singular:

$$\text{sing} = 100 \times \text{MACHEPS} \times \max_{1 \le i,j \le n} |a_{ij}|$$

where MACHEPS is the relative machine precision.

All matrix elements less than or equal to *sing* are considered rounding errors of the largest matrix elements, so they are taken to be zero in subsequent computations. For example, if a diagonal or triangular coefficient

matrix has a diagonal value that is less than or equal to *sing*, the matrix is considered singular by the DET, INV, and SOLVE functions.

The criterion is used by some functions to detect a singular matrix and to abort a computation that cannot be performed on a singular matrix. The typical error message is as follows:

```
ERROR: (execution) Matrix should be non-singular.
```

If you are getting this error message but believe that your matrix is actually nonsingular, you can try one of the following:

- Center and scale the data.
- Use the GINV function to compute the generalized inverse.
- Examine the size of the singular values returned by the SVD call. The SVD call can be used to compute a generalized inverse with a user-specified singularity criterion.

If $\mathbf{A}$ is an $n \times n$ matrix, the INV function allocates an $n \times n$ matrix in order to return the inverse. It also temporarily allocates an n^2 array in order to compute the inverse.

INVUPDT Function

INVUPDT(*matrix, vector< , scalar>***);**

The INVUPDT function updates a matrix inverse.

The arguments to the INVUPDT function are as follows:

matrix	is an $n \times n$ nonsingular matrix. In most applications *matrix* is symmetric positive definite.
vector	is an $n \times 1$ or $1 \times n$ vector.
scalar	is a numeric scalar.

The Sherman-Morrison-Woodbury formula is

$$(\mathbf{A} + \mathbf{U}\mathbf{V}')^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}'\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}'\mathbf{A}^{-1}$$

where $\mathbf{A}$ is an $n \times n$ nonsingular matrix and $\mathbf{U}$ and $\mathbf{V}$ are $n \times k$. The formula shows that a rank k update to $\mathbf{A}$ corresponds to a rank k update of $\mathbf{A}^{-1}$.

The INVUPDT function implements the Sherman-Morrison-Woodbury formula for rank-one updates with $\mathbf{U} = w\mathbf{X}$ and $\mathbf{V} = \mathbf{X}$, where $\mathbf{X}$ is an $n \times 1$ vector and w is a scalar.

If $\mathbf{M} = \mathbf{A}^{-1}$, then you can call the INVUPDT function as follows:

```
R = invupdt(M, X, w);
```

This statement computes the following matrix:

$$\mathbf{R} = \mathbf{M} - w\mathbf{M}\mathbf{X}(\mathbf{I} + w\mathbf{X}'\mathbf{M}\mathbf{X})^{-1}\mathbf{X}'\mathbf{M}$$

The matrix **R** is equivalent to $(\mathbf{A} + w\mathbf{X}\mathbf{X}')^{-1}$. If **A** is symmetric positive definite, then so is **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 **Z** is a design matrix, $\mathbf{M} = (\mathbf{Z}'\mathbf{Z})^{-1}$ is the associated inverse crossproduct matrix, and **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 **M** by using the following statement:

```
M2 = invupdt(M, v);
```

If *w* is 1, the function adds an observation to the inverse; if *w* is −1, the function removes an observation from the inverse. If weighting is used, *w* is the weight.

To perform the computation, the INVUPDT function uses about $2n^2$ multiplications and additions, where *n* is the row dimension of the positive definite argument matrix.

The following program demonstrates adding or removing observations from a linear fit and updating the inverse crossproduct matrix:

```
X = {0, 1, 1, 1, 2, 2, 3, 4, 4};
Y = {1, 1, 2, 6, 2, 3, 3, 3, 4};

/* find linear fit */
Z = j(nrow(X), 1, 1) || X;           /* design matrix */
M = inv(Z`*Z);

b = M*Z`*Y;                          /* LS estimate */
resid = Y - Z*b;                     /* residuals */
print "Original Fit", b resid;

/* residual for observation (1,6) seems too large.
   Take obs number 4 out of data set and refit. */
v = z[4,];
M = invupdt(M, v, -1);               /* update inverse crossprod */

keepObs = (1:3) || (5:nrow(X));
Z = Z[keepObs, ];
Y = Y[keepObs, ];
b = M*Z`*Y;                          /* new LS estimate */
print "After deleting observation 4", b;

/* Add a new obs (x,y) = (0,2) and refit. */
obs = {0 2};
v = 1 || obs[1];                     /* new row in design matrix */
M = invupdt(M, v);

Z = Z // v;
Y = Y // obs[2];
b = M*Z`*Y;                          /* new LS estimate */
print "After adding observation (0,2)", b;
```

Figure 25.197 Refitting Linear Regression Models

Output from Inverse Complex Cepstrum Function

Original Fit

b	resid
2.0277778	-1.027778
0.375	-1.402778
	-0.402778
	3.5972222
	-0.777778
	0.2222222
	-0.152778
	-0.527778
	0.4722222

After deleting observation 4

b
1
0.6470588

After adding observation (0,2)

b
1.3
0.5470588

IPF Call

CALL IPF(*fit, status, dim, table, config* < , *initab*> < , *mod*> **);**

The IPF subroutine performs an iterative proportional fit of a contingency table. This is a standard statistical technique to obtain maximum likelihood estimates for cells under any hierarchical log-linear model. The algorithm is described in Bishop, Fienberg, and Holland (1975).

The arguments to the IPF subroutine are as follows:

fit is a returned matrix. The matrix *fit* contains an array of the estimates of the expected number in each cell under the model specified in *config*. This matrix conforms to *table*, meaning that it has the same dimensions and order of variables.

status is a returned matrix. The *status* argument is a row vector of length 3. *status[1]* is 0 if there is convergence to the desired accuracy, otherwise it is nonzero. *status[2]* is the maximum difference between estimates of the last two iterations of the IPF algorithm. *status[3]* is the number of iterations performed.

dim is an input matrix. If the problem contains v variables, then *dim* is $1 \times v$ row vector. The value *dim[i]* is the number of possible levels for variable i in a contingency table.

table is an input matrix that specifies an array of the number of observations at each level of each variable. Variables are nested across columns and then across rows.

config is an input matrix that specifies which marginal totals to fit. Each column of *config* specifies a distinct marginal in the model under consideration. Because the model is hierarchical, all subsets of specified marginals are included in fitting.

initab is an input matrix that specifies initial values for the iterative procedure. If you do not specify values, ones are used. For incomplete tables, *initab* is set to 1 if the cell is included in the design, and 0 if it is not.

mod is a two-element vector that specifies the stopping criteria. If *mod*= {*MaxDev*, *MaxIter*}, then the procedure iterates either until the maximum difference between estimates of the last two iterations is less than *MaxDev* or until *MaxIter* iterations are completed. Default values are *MaxDev*=0.25 and *MaxIter*=15.

The matrix *table* must conform in size to the contingency table as specified in *dim*. In particular, if *table* is $n \times m$, the product of the entries in *dim* must equal nm. Furthermore, there must be some integer k such that the product of the first k entries in *dim* equals m. If you specify *initab*, then it must be the same size as *table*.

Adjusting a Table from Marginals

A common use of the IPF subroutine is to adjust the entries of a table in order to fit a new set of marginals while retaining the interaction between cell entries.

Example 1: Adjusting Marital Status by Age Bishop, Fienberg, and Holland (1975) present data from D. Friedlander that shows the distribution of women in England and Wales according to their marital status in 1957. One year later, new official marginal estimates were announced. The problem is to adjust the entries in the 1957 table so as to fit the new marginals while retaining the interaction between cells. This problem can arise when you have internal cells that are known from sampling a population and then get margins based on a complete census.

When you want to adjust an observed table of cell frequencies to a new set of margins, you must set the *initab* parameter to be the table of observed values. The new marginals are specified through the *table* argument. The particular cell values for *table* are not important, since only the marginals are used (the proportionality between cells is determined by *initab*).

There are two easy ways to create a table that contains given margins. Recall that a table of independent variables has an expected cell value $A_{ij} =$ (sum of row i)(sum of col j)/(sum of all cells). Thus you could form a table with these cell entries. Another possibility is to use a "greedy algorithm" to assign as many of the marginals as possible to the first cell, then assign as many of the remaining marginals as possible to the second cell, and so on until all of the marginals have been distributed. Both of these approaches are encapsulated into modules in the following program:

```
/* Return a table such that cell (i,j) has value
   (sum of row i)(sum of col j)/(sum of all cells) */
start GetIndepTableFromMargins( bottom, side );
   if bottom[+] ^= side[+] then do;
      print "Marginal totals are not equal";
      abort;
   end;
   table = side*bottom/side[+];
   return (table);
```

```
finish;

/* Use a "greedy" algorithm to create a table whose
   marginal totals match given marginal totals.
   Margin1 is the vector of frequencies totaled down
           each column. Margin1 means that
           Variable 1 has NOT been summed over.
   Margin2 is the vector of frequencies totaled across
           each row. Margin2 means that Variable 2
           has NOT been summed over.
   After calling, use SHAPE to change the shape of
           the returned argument. */
start GetGreedyTableFromMargins( Margin1, Margin2 );
   /* copy arguments so they are not corrupted */
   m1 = colvec(Margin1); /* colvec is in IMLMLIB */
   m2 = colvec(Margin2);
   if m1[+] ^= m2[+] then do;
      print "Marginal totals are not equal";
      abort;
   end;
   dim1 = nrow(m1);
   dim2 = nrow(m2);
   table = j(1,dim1*dim2,0);
   /* give as much to cell (1,1) as possible,
      then as much as remains to cell (1,2), etc,
      until all the margins have been distributed */
   idx = 1;
   do i2 = 1 to dim2;
      do i1 = 1 to dim1;
         t = min(m1[i1],m2[i2]);
         table[idx] = t;
         idx = idx + 1;
         m1[i1] = m1[i1]-t;
         m2[i2] = m2[i2]-t;
      end;
   end;
   return (table);
finish;

Mod = {0.01 15}; /* tighten stopping criterion */

Columns = {"  Single" " Married" "Widow/Divorced"};
Rows    = {"15 - 19" "20 - 24" "25 - 29" "30 - 34"
           "35 - 39" "40 - 44" "45 - 49" "50 Or Over"};

/* Marital status has 3 levels. Age has 8 levels */
Dim = {3 8};

/* Use known distribution for start-up values */
IniTab = { 1306  83    0 ,
           619  765    3 ,
           263 1194    9 ,
           173 1372   28 ,
           171 1393   51 ,
```

```
           159 1372   81 ,
           208 1350  108 ,
          1116 4100 2329 };

/* New marginal totals for age by marital status */
NewMarital = { 3988 11702 2634 };
NewAge = {1412,1402,1450,1541,1681,1532,1662,7644};

/* Create any table with these marginals */
Table = GetGreedyTableFromMargins(NewMarital, NewAge);
Table = shape(Table, nrow(IniTab), ncol(IniTab));

/* Consider all main effects */
Config = {1 2};

call ipf(Fit, Status, Dim, Table, Config, IniTab, Mod);

if Status[1] = 0 then
   print "Known Distribution (1957)",
      IniTab [colname=Columns rowname=Rows format=8.0],,
      "Adjusted Estimates of Distribution (1958)",
      Fit [colname=Columns rowname=Rows format=8.2];
else
   print "IPF did not converge in "
         (Status[3]) " iterations";
```

The results of this program are shown in Figure 25.198. The same results are obtained if the *table* parameter is formed by using the "independent algorithm."

Figure 25.198 Iterative Proportional Fitting

Output from Inverse Complex Cepstrum Function

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)

Figure 25.198 *continued*

	Fit		
	Single	**Married**	**Widow/Divorced**
15 - 19	1325.27	86.73	0.00
20 - 24	615.56	783.39	3.05
25 - 29	253.94	1187.18	8.88
30 - 34	165.13	1348.55	27.32
35 - 39	173.41	1454.71	52.87
40 - 44	147.21	1308.12	76.67
45 - 49	202.33	1352.28	107.40
50 Or Over	1105.16	4181.04	2357.81

Example 2: Adjusting Votes by Region A similar technique can be used to standardize data from raw counts into percentages. For example, consider data from a 1836 vote in the U.S. House of Representatives on a resolution that the House should adopt a policy of tabling all petitions for the abolition of slavery. Attitudes toward abolition were different among slaveholding states that would later secede from the Union ("the South"), slaveholding states that refused to secede ("the Border States"), and nonslaveholding states ("the North").

The raw votes for the resolution are defined in the following statements. The data are hard to interpret because the margins are not homogeneous.

```
/*          Yea Abstain Nay */
IniTab = {  61    12    60,   /* North */
            17     6     1,   /* Border */
            39    22     7 }; /* South */
```

Standardizing the data by specifying homogeneous margins reveals interactions and symmetry that were not apparent in the raw data. Suppose the margins are specified as follows:

```
NewVotes   = {100 100 100};
NewSection = {100,100,100};
```

In this case, the program for marital status by age can be easily rewritten to adjust the votes into a standardized form. The resulting output is shown in Figure 25.199:

Figure 25.199 Standardizing Counts into Percentages

Output from Inverse Complex Cepstrum Function

	Fit		
	Yea	**Abstain**	**Nay**
North	20.1	10.2	69.7
Border	47.4	42.8	9.8
South	32.5	47.0	20.5

Generating a Table with Given Marginals The "greedy algorithm" presented in the Marital-Status-By-Age example can be extended in a natural way to the case where you have n one-way marginals and want to form an n-dimensional table. For example, a three-dimensional "greedy algorithm" would allocate the vector *table* as `table=j(dim1*dim2*dim3,1,0);` and have three nested loops as indicated in the following statements. Afterwards, the *table* parameter can be reshaped by using the SHAPE function.

```
do i3 = 1 to dim3;
   do i2 = 1 to dim2;
      do i1 = 1 to dim1;
         t = min(m1[i1],m2[i2],m3[i3]);
         table[idx] = t;
         idx = idx + 1;
         m1[i1] = m1[i1]-t;
         m2[i2] = m2[i2]-t;
         m3[i3] = m3[i3]-t;
      end;
   end;
end;
```

The idea of the "greedy algorithm" can be extended to marginals that are not one-way. For example, the following three-dimensional table is similar to one that appears in Christensen (1997) based on data from M. Rosenberg. The table presents data on a person's self-esteem for people classified according to their religion and their father's educational level.

		Father's Educational Level				
Religion	Self-Esteem	Not HS Grad	HS Grad	Some Coll	Coll Grad	Post Coll
Catholic	High	575	388	100	77	51
	Low	267	153	40	37	19
Jewish	High	117	102	67	87	62
	Low	48	35	18	12	13
Protestant	High	359	233	109	197	90
	Low	159	173	47	82	32

Since the father's education level is nested across columns, it is Variable 1 with levels that correspond to not finishing high school, graduating from high school, attending college, graduating from college, and attending graduate courses. The variable that varies the quickest across rows is Self-Esteem, so Self-Esteem is Variable 2 with values "High" and "Low." The Religion variable is Variable 3 with values "Catholic," "Jewish," and "Protestant."

The following program encodes this table by using the MARG call to compute a two-way marginal table by summing over the third variable, and a one-way marginal by summing over the first two variables. Then a new table (**NewTable**) is created by applying the greedy algorithm to the two marginals. Finally, the marginals of **NewTable** are computed and compared with those of **table**.

```
dim={5 2 3};
table={
/* Father's Education:
           NotHSGrad HSGrad Col ColGrad PostCol
          Self-
   Relig  Esteem                          */
/* Cath-   Hi */ 575   388  100    77   51,
/* olic    Lo */ 267   153   40    37   19,
```

```
/* Jew-    Hi */ 117   102   67    87   62,
/*  ish    Lo */  48    35   18    12   13,

/* Prot-   Hi */ 359   233  109   197   90,
/* estant  Lo */ 159   173   47    82   32
        };

config = { 1 3,
           2 0 };
call marg(locmar, marginal, dim, table, config);
print locmar, marginal, table;

/* Examine marginals: The name indicates the
   variable(s) that are NOT summed over.
   The locmar variable tells where to index
   into the marginal variable. */
Var12_Marg = marginal[1:(locmar[2]-1)];
Var12_Marg = shape(Var12_Marg,dim[2],dim[1]);
Var3_Marg  = marginal[locMar[2]:ncol(marginal)];

NewTable = j(nrow(table),ncol(table),0);
/* give as much to cell (1,1,1) as possible,
   then as much as remains to cell (1,1,2), etc,
   until all the margins have been distributed. */
idx = 1;
do i3 = 1 to dim[3];        /* over Var3 */
   do i2 = 1 to dim[2];     /* over Var2 */
      do i1 = 1 to dim[1];  /* over Var1 */
         /* Note Var12_Marg has Var1 varying across
            the columns */
         t = min(Var12_Marg[i2,i1],Var3_Marg[i3]);
         NewTable[idx] = t;
         idx = idx + 1;
         Var12_Marg[i2,i1] = Var12_Marg[i2,i1]-t;
         Var3_Marg[i3] = Var3_Marg[i3]-t;
      end;
   end;
end;

call marg(locmar, NewMarginal, dim, table, config);
maxDiff = abs(marginal-NewMarginal)[<>];
if maxDiff=0 then
   print "Marginals are unchanged";
print NewTable;
```

Figure 25.200 Table with Given Marginals

Output from Inverse Complex Cepstrum Function

locmar
1 11

Figure 25.200 *continued*

marginal													
	COL1	COL2	COL3	COL4	COL5	COL6	COL7	COL8	COL9	COL10	COL11	COL12	COL13
ROW1	1051	723	276	361	203	474	361	105	131	64	1707	561	1481

table				
575	388	100	77	51
267	153	40	37	19
117	102	67	87	62
48	35	18	12	13
359	233	109	197	90
159	173	47	82	32

Marginals are unchanged

NewTable				
1051	656	0	0	0
0	0	0	0	0
0	67	276	218	0
0	0	0	0	0
0	0	0	143	203
474	361	105	131	64

Fitting a Log-Linear Model to a Table

A second common usage of the IPF algorithm is to hypothesize that the table of observations can be fitted by a model with known effects and to ask whether the observed values indicate that the model hypothesis can be accepted or should be rejected. In this usage, you normally do not specify the *initab* argument to the IPF subroutine (but see the comment on structural zeros in the section "Additional Details" on page 716).

Example 3: Food Illness Korff, Taback, and Beard (1952) reported statistics related to the outbreak of food poisoning at a company picnic. A total of 304 people at the picnic were surveyed to determine who had eaten either of two suspect foods: potato salad and crabmeat. The predictor variables are whether the individual ate potato salad (Variable 1: "Yes" or "No") and whether the person ate crabmeat (Variable 2: "Yes" or "No"). The response variable is whether the person was ill (Variable 3: "Ill" or "Not Ill"). The order of the variables is determined by the *dim* and *table* arguments to the IPF subroutine. The variables are nested across columns, then across rows.

Crabmeat:	YES		NO	
Potato salad:	Yes	No	Yes	No
Ill	120	4	22	0
Not Ill	80	31	24	23

The following program defines the variables and observations, and then fits three separate models. How well each model fits the data is determined by computing a Pearson chi-square statistic $\chi^2 = \sum(O-E)^2/E$, where the sum is over all cells, O stands for the observed cell count, and E stands for the fitted estimate. Other statistics, such as the likelihood-ratio chi-square statistic $G^2 = -2\sum O\log(E/O)$, could also be used.

The program first fits a model that excludes the three-way interaction. The model fits well, so you can conclude that an association between illness and potato salad does not depend on whether an individual ate crabmeat. The next model excludes the interaction between potato salad and illness. This model is rejected with a large chi-square value, so the data support an association between potato salad and illness. The last model excludes the interaction between the crabmeat and the illness. This model fits moderately well.

```
/* Compute a chi-square score for a table of observed
   values, given a table of expected values. Compare
   this score to a chi-square value with given degrees
   of freedom at 95% confidence level. */
start ChiSqTest( obs, model, degFreedom );
   diff = (obs - model)##2 / model;
   chiSq = diff[+];
   chiSqCutoff = cinv(0.95, degFreedom);
   print chiSq chiSqCutoff;
   if chiSq > chiSqCutoff then
      print "Reject hypothesis";
   else
      print "No evidence to reject hypothesis";
finish;

dim={2 2 2};

/* Crab meat:   Y E S        N O
   Potato:   Yes   No      Yes  No   */
table={      120    4      22   0,   /* Ill */
              80   31      24  23 }; /* Not Ill */

crabmeat = "          C R A B      N O  C R A B";
potato  = {"YesPot" "NoPot" "YesPot" "NoPot"};
illness = {"Ill", "Not Ill"};

hypoth = "Hypothesis: no three-factor interaction";
config={1  1  2,
        2  3  3};
call ipf(fit,status,dim,table,config);

print hypoth, "Fitted Model:",
      fit[label=crabmeat colname=potato
          rowname=illness format=6.2];
run ChiSqTest(table, fit, 1); /* 1 deg of freedom */

/* Test for interaction between Var 3 (Illness) and
   Var 1 (Potato Salad) */
hypoth = "Hypothesis: no Illness-Potato Interaction";
config={1 2,
        2 3};
call ipf(fit,status,dim,table,config);

print hypoth, "Fitted Model:",
      fit[label=crabmeat colname=potato
          rowname=illness format=6.2];
run ChiSqTest(table, fit, 2); /* 2 deg of freedom */
```

```
/* Test for interaction between Var 3 (Illness) and
   Var 2 (Crab meat) */
hypoth = "Hypothesis: no Illness-Crab Interaction";
config={1 1,
        2 3};
call ipf(fit,status,dim,table,config);

print hypoth, "Fitted Model:",
      fit[label=crabmeat colname=potato
          rowname=illness format=6.2];
run ChiSqTest(table, fit, 2); /* 2 deg of freedom */
```

Figure 25.201 Fitting Log-Linear Models

Output from Inverse Complex Cepstrum Function

hypoth
Hypothesis: no three-factor interaction

Fitted Model:

	CRAB		NO CRAB	
	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 Illness-Potato Interaction

Fitted Model:

	CRAB		NO CRAB	
	YesPot	NoPot	YesPot	NoPot
Ill	105.53	18.47	14.67	7.33
Not Ill	94.47	16.53	31.33	15.67

chiSq	chiSqCutoff
44.344643	5.9914645

Reject hypothesis

hypoth
Hypothesis: no Illness-Crab Interaction

Fitted Model:

Figure 25.201 *continued*

	CRAB		NO CRAB	
	YesPot	NoPot	YesPot	NoPot
Ill	115.45	2.41	26.55	1.59
Not Ill	84.55	32.59	19.45	21.41

chiSq	chiSqCutoff
5.0945132	5.9914645

No evidence to reject hypothesis

Additional Details

Structural versus Random Zeros In the Marital-Status-By-Age example, the *initab* argument contained a zero for the "15–19 and Widowed/Divorced" category. Because the *initab* parameter determines the proportionality between cells, the fitted model retains a zero in that category. By contrast, in the Food-Illness example, the *table* parameter contained a zero for number of illnesses observed among those who did not eat either crabmeat or potato salad. This is a sampling (*random*) zero. Some models preserve that zero; others do not. If your table has a *structural zero* (for example, the number of ovarian cancers observed among male patients), then you can use the *initab* parameter to preserve that zero. see Bishop, Fienberg, and Holland (1975) or the documentation for the CATMOD procedure in the *SAS/STAT User's Guide* for more information about structural zeros and incomplete tables.

The *config* Parameter The columns of this matrix specify which interaction effects should be included in the model. The following table specifies the model and the configuration parameter for common interactions for an $I \times J \times K$ table in three dimensions. The so-called *noncomprehensive* models that do not include all variables (for example, *config* = {1}) are not listed in the table, but can be used. You can also specify combinations of main and interaction effects. For example, *config* = {1 3, 2 0}) specifies all main effects and the 1-2 interaction. Bishop, Fienberg, and Holland (1975) and Christensen (1997) explain how to compute the degrees of freedom associated with any model. For models with structural zeros, computing the degrees of freedom is complicated.

Model	config	Degrees of Freedom
No three-factor	{1 1 2, {2 3 3}	$(I-1)(J-1)(K-1)$
One two-factor absent	{1 2, {3 3}	$(I-1)(J-1)K$
	{1 2, 2 3}	$(I-1)J(K-1)$
	{1 1, {2 3}	$I(J-1)(K-1)$
Two two-factor absent	{2, 3}	$(I-1)(JK-1)$
	{1, 3}	$(J-1)(IK-1)$
	{1, 2}	$(K-1)(IJ-1)$
No two-factor	{1 2 3}	$IJK-(I+J+K)+2$
Saturated	{1,2,3}	IJK

The Shape of the *table* Parameter Since variables are nested across columns and then across rows, any shape that conforms to the *dim* parameter is equivalent.

For example, the section "Generating a Table with Given Marginals" on page 710 presents data on a person's self-esteem for people classified according to their religion and their father's educational level. To save space, the educational levels are subsequently denoted by labels that indicate the typical number of years spent in school: "<12," "12," "<16," "16," and ">16."

The table would be encoded as follows:

```
dim={5 2 3};

table={
/* Father's Education:
                  <12    12   <16   16   >16
          Self-
   Relig  Esteem                            */
/* Cath-   Hi */ 575   388  100    77   51,
/* olic    Lo */ 267   153   40    37   19,

/* Jew-    Hi */ 117   102   67    87   62,
/*  ish    Lo */  48    35   18    12   13,

/* Prot-   Hi */ 359   233  109   197   90,
/* estant  Lo */ 159   173   47    82   32
        };
```

The same information for the same variables in the same order could also be encoded into an $n \times m$ table in two other ways. Recall that the product of entries in *dim* is nm and that m must equal the product of the first k entries of *dim* for some k. For this example, the product of the entries in *dim* is 30, and so the table must be 6×5, 3×10, or 1×30. The 3×10 table is encoded as concatenating rows 1–2, 3–4, and 5–6 to produce the following:

```
table={
/* Esteem: H I G H           L O W                    */
/*     <12  ...  >16    <12   ...  >16                 */

       575  ...   51    267   ...   19, /* Catholic  */
       117  ...   62     48   ...   13, /* Jewish    */
       359  ...   90    159   ...   32  /* Protestant*/
      };
```

The 1×30 table is encoded by concatenating all rows, as follows:

```
table={
/*     CATHOLIC            ...        PROTESTANT
   High         Low        ...      High       Low
<12 ... >16 <12 ... >16    ...  <12 ... >16 <12 ... >16
*/
575 ...  51 267 ...  19    ...   359 ... 90 159 ... 32
      };
```

ISEMPTY Function

ISEMPTY(*m*);

The ISEMPTY function is part of the IMLMLIB library. An empty matrix has no rows or columns. The ISEMPTY function returns 1 if its argument is an empty matrix; otherwise, the function returns 0 as shown in the following example:

```
free x;   /* an empty matrix */
isX = IsEmpty(x);
y = 1:5;
isY = IsEmpty(y);
print isX isY;
```

Figure 25.202 Results of the ISEMPTY Function

Output from Inverse Complex Cepstrum Function

isX	isY
1	0

ISSKIPPED Function

ISSKIPPED(*x*);

The ISSKIPPED function enables you to determine at run time whether any optional argument to a user-defined module was skipped. You can call the function only from within a module.

The ISSKIPPED function returns 0 if the symbol **x** was provided as an argument in the current call to the module. If the symbol was not provided (that is, it was skipped), the ISSKIPPED function returns 1.

The following module contains one required argument, **x**. The parameters **a** and **y** are optional. The first argument has a default value of 1, which means that **a** equals 1 if the first argument is not provided to the module. In contrast, the third argument does not have a default value. If the module is called without specifying the third parameter, the matrix **y** is the empty matrix. The following statements call the module with different combinations of supplied and skipped arguments.

```
start axpy(a=1, x, y=);
   if isskipped(y) then z = a#x;
   else                 z = a#x + y;
   return(z);
finish;

p = {1,2,3,4};
q = 1;
z1 = axpy( , p);    /* a and y skipped; a has default value */
z2 = axpy(2, p);    /* y skipped */
z3 = axpy(2, p, q); /* no parameter skipped */
print z1 z2 z3;
```

Figure 25.203 Skipping Module Arguments

Output from Inverse Complex Cepstrum Function

z1	z2	z3
1	2	3
2	4	5
3	6	7
4	8	9

ITSOLVER Call

CALL ITSOLVER(*x, error, iter, method, A, b* < , *precon*> < , *tol*> < , *maxiter*> < , *start*> < , *history*> **);**

The ITSOLVER subroutine solves a sparse linear system by using iterative methods.

The ITSOLVER call returns the following values:

x is the solution to *Ax=b*.

error is the final relative error of the solution.

iter is the number of iterations executed.

The input arguments to the ITSOLVER call are as follows:

method is the type of iterative method to use. The following values are valid:

- "CG" specifies a conjugate gradient algorithm. The matrix *A* must be symmetric and positive definite.
- "CGS" specifies a conjugate gradient squared algorithm, for general *A*.
- "MINRES" specifies a minimum residual algorithm, when *A* is symmetric indefinite.
- "BICG" specifies a biconjugate gradient algorithm, for general *A*.

A is the sparse coefficient matrix in the equation *Ax=b*. You can use SPARSE function to convert a matrix from dense to sparse storage.

b is a column vector, the right side of the equation *Ax=b*.

precon is the name of a preconditioning technique to use. The following values are valid:

- "NONE" specifies no preconditioning. This is the default behavior if the argument is not specified.
- "IC" specifies an incomplete Cholesky factorization. Specify this value when you specify "CG" or "MINRES" for the *method* argument.
- "DIAG" specifies a diagonal Jacobi preconditioner. Specify this value when you specify "CG" or "MINRES" for the *method* argument.
- "MILU" specifies a modified incomplete LU factorization. Specify this value when you specify "BICG" for the *method* argument.

tol	is the relative error tolerance.
maxiter	is the iteration limit.
start	is a starting point column vector.
history	is a matrix to store the relative error at each iteration.

The ITSOLVER call solves a sparse linear system by iterative methods, which involve updating a trial solution over successive iterations to minimize the error. The ITSOLVER call uses the technique specified in the *method* parameter to update the solution.

The input matrix A represents the coefficient matrix in sparse format; it is an $n \times 3$ matrix, where n is the number of nonzero elements. The first column contains the nonzero values, and the second and third columns contain the row and column locations for the nonzero elements, respectively. For the algorithms that assume symmetric A, only the lower triangular elements should be specified. The algorithm continues iterating to improve the solution until either the relative error tolerance specified in *tol* is satisfied or the maximum number of iterations specified in *maxiter* is reached. The relative error is defined as

$$\text{error} = \|Ax - b\|_2 / (\|b\|_2 + \epsilon)$$

where the $\|\cdot\|_2$ operator is the Euclidean norm and ϵ is a machine-dependent epsilon value to prevent any division by zero. If *tol* or *maxiter* is not specified in the call, then a default value of 10^{-7} is used for *tol* and 100,000 for *maxiter*.

The convergence of an iterative algorithm can often be enhanced by preconditioning the input coefficient matrix. The preconditioning option is specified with the *precon* parameter.

A starting trial solution can be specified with the *start* parameter; otherwise the ITSOLVER call generates a zero starting point. You can supply a matrix to store the relative error at each iteration with the *history* parameter. The *history* matrix should be allocated with enough elements to store the maximum number of iterations you expect.

You should always check the returned *error* and *iter* parameters to verify that the desired relative error tolerance is reached. If the tolerance is not reached, the program might continue the solution process with another ITSOLVER call, with *start* set to the latest result. You might also try a different *precon* option to enhance convergence.

For example, the following linear system has a coefficient matrix that contains several zeros:

$$\begin{bmatrix} 3 & 2 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & -10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

You can represent the matrix in sparse form and use the biconjugate gradient algorithm to solve the linear system, as shown in the following statements:

```
/* value     row column */
A = {  3       1      1,
       2       1      2,
       1.1     2      1,
       4       2      2,
       1       3      2,
```

```
       3.2      4       2,
     -10        3       3,
       3        4       4};

/* right hand side */
b = {1, 1, 1, 1};
maxiter = 10;
hist = j(maxiter,1,.);
start = {1,1,1,1};
tol = 1e-10;
call itsolver(x, error, iter, "bicg", A, b, "milu", tol,
maxiter, start, hist);
print x;
print iter error;
print hist;
```

Figure 25.204 Solution of a Linear System

Output from Inverse Complex Cepstrum Function

x
0.2040816
0.1938776
-0.080612
0.1265306

iter	error
3	5.011E-16

hist
0.0254375
0.0080432
5.011E-16
.
.
.
.
.
.
.

The following linear system also has a coefficient matrix with several zeros:

$$\begin{bmatrix} 3 & 1.1 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & 10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The following statements represent the matrix in sparse form and use the conjugate gradient algorithm solve the symmetric positive definite system:

```
/* value     row column */
A = { 3       1      1,
      1.1     2      1,
      4       2      2,
      1       3      2,
      3.2     4      2,
     10       3      3,
      3       4      4};

/* right hand side */
b = {1, 1, 1, 1};
call itsolver(x, error, iter, "CG", A, b);
print x, iter, error;
```

Figure 25.205 Solution to Sparse System

Output from Inverse Complex Cepstrum Function

x
2.68
-6.4
0.74
7.16

iter
4

error
2.847E-15

J Function

J(*nrow* < , *ncol*> < , *value*> **);**

The J function creates a matrix with *nrow* rows and *ncol* columns with all elements equal to *value*.

The arguments to the J function are as follows:

nrow	is a numeric matrix or literal that contains the number of rows.
ncol	is a numeric matrix or literal that contains the number of columns.
value	is a numeric or character matrix or literal for filling the rows and columns of the matrix.

If *ncol* is not specified, it defaults to *nrow*. If *value* is not specified, it defaults to 1. The REPEAT function and the SHAPE function also perform this operation, and they are more general.

Examples of the J function are as follows:

```
b = j(3, 4);
c = j(5, 2, "xyz");
print b, c;
```

Figure 25.206 Constant Matrices

Output from Inverse Complex Cepstrum Function

b			
1	1	1	1
1	1	1	1
1	1	1	1

c	
xyz	xyz
xyz	xyz
xyz	xyz
xyz	xyz
xyz	xyz

JROOT Function

JROOT(*order, n***);**

The JROOT function computes the first nonzero roots of a Bessel function of the first kind and the derivative of the Bessel function at each root. The function returns an $n \times 2$ matrix with the computed roots in the first column and the derivatives in the second column. You can evaluate the Bessel function itself by calling the JBESSEL function.

The arguments to the JROOT function are as follows:

order	is a scalar that denotes the order of the Bessel function, with *order*> -1. The order of a Bessel function is often indicated with the Greek subscript ν, so that J_ν indicates the Bessel function of order ν.
n	is a positive integer that denotes the number of roots.

The JROOT function returns a matrix in which the first column contains the first *n* roots of the Bessel function; these roots are the solutions to the equation

$$J_\nu(x_i) = 0, i = 1, \ldots, n$$

The second column of this matrix contains the derivatives $J'_\nu(x_i)$ of the Bessel function at each of the roots x_i. The expression $J_\nu(x)$ is a solution to the differential equation

$$x^2 \frac{d^2 J_\nu}{dx^2} + x \frac{dJ_\nu}{dx} + (x^2 - \nu^2) J_\nu = 0$$

One of the expressions for such a function is given by the series

$$J_\nu(x) = \left(\frac{1}{2}z\right)^\nu \sum_{k=0}^{\infty} \frac{\left(-\frac{1}{4}z^2\right)^k}{k!\Gamma(\nu + k + 1)}$$

where $\Gamma(\cdot)$ is the gamma function. See Abramowitz and Stegun (1972) for more details concerning the Bessel and gamma functions.

The root-finding algorithm is a Newton method coupled with a reasonable initial guess. For large values of *n* or ν, the algorithm could fail due to machine limitations. In this case, JROOT returns a matrix with zero rows and zero columns. The values that cause the algorithm to fail are machine-dependent.

The following statements compute the first few roots for the Bessel function of the first kind:

```
x = jroot(1, 4);
print x;
```

Figure 25.207 Roots of a Bessel Function

Output from Inverse Complex Cepstrum Function

x	
3.831706	-0.402759
7.0155867	0.3001158
10.173468	-0.249705
13.323692	0.2183594

To obtain only the roots, you can use the following statement, which extracts the first column of the returned matrix:

```
r = x[,1];
```

KALCVF Call

CALL KALCVF(*pred, vpred, filt, vfilt, data, lead, a, f, b, h, var* **<** *, z0* **>** **<** *, vz0* **>** **);**

The KALCVF subroutine computes the one-step prediction $z_{t+1|t}$ and the filtered estimate $z_{t|t}$, in addition to their covariance matrices. The call uses forward recursions, and you can also use it to obtain *k*-step estimates.

The input arguments to the KALCVF subroutine are as follows:

data is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.

lead is the number of steps to forecast after the end of the data.

a is an $N_z \times 1$ vector for a time-invariant input vector in the transition equation, or a $(T + \text{lead})N_z \times 1$ vector that contains input vectors in the transition equation.

f is an $N_z \times N_z$ matrix for a time-invariant transition matrix in the transition equation, or a $(T + \text{lead})N_z \times N_z$ matrix that contains transition matrices in the transition equation.

b is an $N_y \times 1$ vector for a time-invariant input vector in the measurement equation, or a $(T + \text{lead})N_y \times 1$ vector that contains input vectors in the measurement equation.

h is an $N_y \times N_z$ matrix for a time-invariant measurement matrix in the measurement equation, or a $(T + \text{lead})N_y \times N_z$ matrix that contains measurement matrices in the measurement equation.

var is an $(N_z + N_y) \times (N_z + N_y)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + \text{lead})(N_z + N_y) \times (N_z + N_y)$ matrix that contains variance matrices for the error in the transition equation and the error in the measurement equation—that is, $(\eta_t', \epsilon_t')'$.

z0 is an optional $1 \times N_z$ initial state vector $z_{1|0}'$.

vz0 is an optional $N_z \times N_z$ covariance matrix of an initial state vector $P_{1|0}$.

The KALCVF call returns the following values:

pred is a $(T + \text{lead}) \times N_z$ matrix that contains one-step predicted state vectors $(z_{1|0}, \ldots, z_{T+1|T}, z_{T+2|T}, \ldots, z_{T+\text{lead}|T})'$.

vpred is a $(T + \text{lead})N_z \times N_z$ matrix that contains mean square errors of predicted state vectors $(P_{1|0}, \ldots, P_{T+1|T}, P_{T+2|T}, \ldots, P_{T+\text{lead}|T})'$.

filt is a $T \times N_z$ matrix that contains filtered state vectors $(z_{1|1}, \ldots, z_{T|T})'$.

vfilt is a $TN_z \times N_z$ matrix that contains mean square errors of filtered state vectors $(P_{1|1}, \ldots, P_{T|T})'$.

The KALCVF call computes the conditional expectation of the state vector z_t given the observations, assuming that the mean and the variance of the initial state vector are known. The filtered value is the conditional expectation of the state vector z_t given the observations up to time t. For k-step forecasting where $k > 0$, the conditional expectation at time $t + k$ is computed given observations up to t. For notation, V_t and R_t are variances of η_t and ϵ_t, respectively, and G_t is a covariance of η_t and ϵ_t, and A^- stands for the generalized inverse of A. The filtered value and its covariance matrix are denoted $z_{t|t}$ and $P_{t|t}$, respectively. For $k > 0$, $z_{t+k|t}$ and $P_{t+k|t}$ stand for the k-step forecast of z_{t+k} and its mean square error. The Kalman filtering algorithm for one-step prediction and filtering is given as follows:

$$
\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + R_t \\
z_{t|t} &= z_{t|t-1} + P_{t|t-1} H_t' D_t^- \hat{\epsilon}_t \\
P_{t|t} &= P_{t|t-1} - P_{t|t-1} H_t' D_t^- H_t P_{t|t-1} \\
K_t &= (F_t P_{t|t-1} H_t' + G_t) D_t^- \\
z_{t+1|t} &= a_t + F_t z_{t|t-1} + K_t \hat{\epsilon}_t \\
P_{t+1|t} &= F_t P_{t|t-1} F_t' + V_t - K_t D_t K_t'
\end{aligned}
$$

And for k-step forecasting for $k > 1$,

$$
\begin{aligned}
z_{t+k|t} &= a_{t+k-1} + F_{t+k-1} z_{t+k-1|t} \\
P_{t+k|t} &= F_{t+k-1} P_{t+k-1|t} F_{t+k-1}' + V_{t+k-1}
\end{aligned}
$$

When you use the alternative transition equation

$$z_t = a_t + F_t z_{t-1} + \eta_t$$

the forward recursion algorithm is written

$$\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + H_t \mathbf{G}_t + G_t' H_t' + \mathbf{R}_t \\
z_{t|t} &= z_{t|t-1} + (P_{t|t-1} H_t' + G_t) D_t^- \hat{\epsilon}_t \\
P_{t|t} &= P_{t|t-1} - (P_{t|t-1} H_t' + G_t) D_t^- (H_t P_{t|t-1} + G_t') \\
K_t &= (F_{t+1} P_{t|t-1} H_t' + G_t) D_t^- \\
z_{t+1|t} &= a_{t+1} + F_{t+1} z_{t|t-1} + K_t \hat{\epsilon}_t \\
P_{t+1|t} &= F_{t+1} P_{t|t-1} F_{t+1}' + V_{t+1} - K_t D_t K_t'
\end{aligned}$$

And for k-step forecasting ($k > 1$),

$$\begin{aligned}
z_{t+k|t} &= a_{t+k} + F_{t+k} z_{t+k-1|t} \\
P_{t+k|t} &= F_{t+k} P_{t+k-1|t} F_{t+k}' + V_{t+k}
\end{aligned}$$

You can use the KALCVF call when you specify the alternative transition equation and $\mathbf{G}_t = \mathbf{0}$.

The initial state vector and its covariance matrix of the time-invariant Kalman filters are computed under the stationarity condition

$$\begin{aligned}
z_{1|0} &= (I - F)^- a \\
P_{1|0} &= (I - F \otimes F)^- \mathrm{vec}(V)
\end{aligned}$$

where F and V are the time-invariant transition matrix and the covariance matrix of transition equation noise, and $\mathrm{vec}(V)$ is an $N_z^2 \times 1$ column vector that is constructed by the stacking N_z columns of matrix V. Note that all eigenvalues of the matrix F are inside the unit circle when the SSM is stationary. When the preceding formula cannot be applied, the initial state vector estimate $z_{1|0}$ is set to a_1 and its covariance matrix $P_{1|0}$ is given by 10^6I. Optionally, you can specify initial values.

The KALCVF call accepts missing values in observations. If there is a missing observation, the filtered state vector for the missing observation is given by the one-step forecast.

The following program gives an example of the KALCVF call:

```
q = 2;
p = 2;
n = 10;
lead = 3;
total = n + lead;

call randseed(25735);
x = round( randfun(n||p,       "Normal"), 0.1);
f = round( randfun(q*total||q, "Normal"), 0.1);
a = round( randfun(total*q,    "Normal"), 0.1);
h = round( randfun(p*total||q, "Normal"), 0.1);
b = round( randfun(p*total,    "Normal"), 0.1);
```

```
temp = j(p+q, p+q);
do i = 1 to total;
   call randgen(temp, "Normal");
   temp = round(temp, 0.1);
   var = var//(temp*temp`);
end;

call kalcvf(pred, vpred, filt, vfilt, x, lead, a, f, b, h, var);

/* default initial state and covariance */
call kalcvs(sm, vsm, x, a, f, b, h, var, pred, vpred);
print sm[format=9.4] vsm[format=9.4];
```

Figure 25.208 Smoothed Estimate and Covariance

Output from Inverse Complex Cepstrum Function

sm		vsm	
0.1579	-0.2517	0.5652	0.5318
1.5140	-0.9182	0.5318	3.6498
0.7860	0.3102	0.8339	-0.2093
-1.6351	-0.5047	-0.2093	0.6248
-0.3077	-2.0140	0.3979	0.3933
-0.1868	0.2790	0.3933	0.6736
-0.1076	-0.0267	0.3506	-0.2237
-0.0667	-1.1527	-0.2237	0.2177
-0.1221	-0.7630	0.2084	0.3207
-0.6609	0.0271	0.3207	0.5045
		0.6630	0.3002
		0.3002	0.2219
		0.1042	0.1403
		0.1403	0.3469
		0.0552	-0.1018
		-0.1018	0.4853
		0.0903	-0.1627
		-0.1627	0.4561
		0.3894	-0.2838
		-0.2838	0.7716

KALCVS Call

CALL KALCVS(*sm, vsm, data, a, f, b, h, var, pred, vpred* < *, un* > < *, vun* > **);**

The KALCVS subroutine uses backward recursions to compute the smoothed estimate $z_{t|T}$ and its covariance matrix, $P_{t|T}$, where T is the number of observations in the complete data set.

The input arguments to the KALCVS subroutine are as follows.

data	is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.
a	is an $N_z \times 1$ vector for a time-invariant input vector in the transition equation, or a $TN_z \times 1$ vector that contains input vectors in the transition equation.
f	is an $N_z \times N_z$ matrix for a time-invariant transition matrix in the transition equation, or a $TN_z \times N_z$ matrix that contains T transition matrices.
b	is an $N_y \times 1$ vector for a time-invariant input vector in the measurement equation, or a $TN_y \times 1$ vector that contains input vectors in the measurement equation.
h	is an $N_y \times N_z$ matrix for a time-invariant measurement matrix in the measurement equation, or a $TN_y \times N_z$ matrix that contains T time-variant H_t matrices in the measurement equation.
var	is an $(N_z + N_y) \times (N_z + N_y)$ covariance matrix for the errors in the transition and the measurement equations, or a $T(N_z + N_y) \times (N_z + N_y)$ matrix that contains covariance matrices in the transition equation and measurement equation noises—that is, $(\eta_t', \epsilon_t')'$.
pred	is a $T \times N_z$ matrix that contains one-step forecasts $(z_{1\|0}, \ldots, z_{T\|T-1})'$.
vpred	is a $TN_z \times N_z$ matrix that contains mean square error matrices of predicted state vectors $(P_{1\|0}, \ldots, P_{T\|T-1})'$.
un	is an optional $1 \times N_z$ vector that contains u_T. The returned value is u_0.
vun	is an optional $N_z \times N_z$ matrix that contains U_T. The returned value is U_0.

The KALCVS call returns the following values:

sm	is a $T \times N_z$ matrix that contains smoothed state vectors $(z_{1\|T}, \ldots, z_{T\|T})'$.
vsm	is a $TN_z \times N_z$ matrix that contains covariance matrices of smoothed state vectors $(P_{1\|T}, \ldots, P_{T\|T})'$.

When the Kalman filtering is performed in the KALCVF call, the KALCVS call computes smoothed state vectors and their covariance matrices. The fixed-interval smoothing state vector at time t is obtained by the conditional expectation given all observations.

The smoothing algorithm uses one-step forecasts and their covariance matrices, which are given in the KALCVF call. For notation, $z_{t|T}$ is the smoothed value of the state vector z_t, and the mean square error matrix is denoted $P_{t|T}$. For smoothing,

$$
\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + R_t \\
K_t &= (F_t P_{t|t-1} H_t' + G_t) D_t^- \\
L_t &= F_t - K_t H_t \\
u_{t-1} &= H_t' D_t^- \hat{\epsilon}_t + L_t' u_t \\
U_{t-1} &= H_t' D_t^- H_t + L_t' U_t L_t \\
z_{t|T} &= z_{t|t-1} + P_{t|t-1} u_{t-1} \\
P_{t|T} &= P_{t|t-1} - P_{t|t-1} U_{t-1} P_{t|t-1}
\end{aligned}
$$

where $t = T, T-1, \ldots, 1$. The initial values are $u_T = \mathbf{0}$ and $\mathbf{U}_T = \mathbf{0}$.

When the SSM is specified by using the alternative transition equation

$$z_t = a_t + F_t z_{t-1} + \eta_t$$

the fixed-interval smoothing is performed by using the following backward recursions:

$$\begin{aligned}
\hat{\epsilon}_t &= y_t - b_t - H_t z_{t|t-1} \\
D_t &= H_t P_{t|t-1} H_t' + R_t \\
K_t &= F_{t+1} P_{t|t-1} H_t' D_t^- \\
L_t &= F_{t+1} - K_t H_t \\
u_{t-1} &= H_t' D_t^- \hat{\epsilon}_t + L_t' u_t \\
U_{t-1} &= H_t' D_t^- H_t + L_t' U_t L_t \\
z_{t|T} &= z_{t|t-1} + P_{t|t-1} u_{t-1} \\
P_{t|T} &= P_{t|t-1} - P_{t|t-1} U_{t-1} P_{t|t-1}
\end{aligned}$$

where it is assumed that $G_t = \mathbf{0}$.

You can use the KALCVS call regardless of the specification of the transition equation when $G_t = \mathbf{0}$. Harvey (1989) gives the following fixed-interval smoothing formula, which produces the same smoothed value:

$$\begin{aligned}
z_{t|T} &= z_{t|t} + P_t^*(z_{t+1|T} - z_{t+1|t}) \\
P_{t|T} &= P_{t|t} + P_t^*(P_{t+1|T} - P_{t+1|t}) P_t^{*\prime}
\end{aligned}$$

where

$$P_t^* = P_{t|t} F_t' P_{t+1|t}^-$$

under the shifted transition equation, but

$$P_t^* = P_{t|t} F_{t+1}' P_{t+1|t}^-$$

under the alternative transition equation.

The KALCVS call is accompanied by the KALCVF call, as shown in the following statements. Note that you do not need to specify UN and VUN.

```
call kalcvf(pred, vpred, filt, vfilt, y, 0, a, f, b, h, var);
call kalcvs(sm, vsm, y, a, f, b, h, var, pred, vpred);
```

You can also compute the smoothed estimate and its covariance matrix on an observation-by-observation basis. When the SSM is time invariant, the following example performs smoothing. In this situation, you should initialize UN and VUN as matrices of value 0, as shown in the following statements:

```
call kalcvf(pred, vpred, filt, vfilt, y, 0, a, f, b, h, var);
n = nrow(y);
nz = ncol(f);
un = j(1, nz, 0);
```

```
vun = j(nz, nz, 0);

do i = 1 to n;
   y_i = y[n-i+1,];
   pred_i  = pred[n-i+1,];
   vpred_i = vpred[(n-i)*nz+1:(n-i+1)*nz,];
   call kalcvs(sm_i, vsm_i, y_i, a, f, b, h, var,
               pred_i, vpred_i, un, vun);
   sm  = sm_i // sm;
   vsm = vsm_i // vsm;
end;
```

The KALCVF call has an example program that includes the KALCVS call.

KALDFF Call

CALL KALDFF(*pred, vpred, initial, s2, data, lead, int, coef, var, intd, coefd* < , *n0*> < , *at*> < , *mt*> < , *qt*> **);**

The KALDFF subroutine computes the one-step forecast of state vectors in an SSM by using the diffuse Kalman filter. The call estimates the conditional expectation of z_t, and also estimates the initial random vector, δ, and its covariance matrix.

The input arguments to the KALDFF subroutine are as follows:

data is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.

lead is the number of steps to forecast after the end of the data set.

int is an $(N_z + N_y) \times N_\beta$ matrix for a time-invariant fixed matrix, or a $(T + \text{lead})(N_z + N_y) \times N_\beta$ matrix that contains fixed matrices for the time-variant model in the transition equation and the measurement equation—that is, $(W_t', X_t')'$.

coef is an $(N_z + N_y) \times N_z$ matrix for a time-invariant coefficient, or a $(T + \text{lead})(N_z + N_y) \times N_z$ matrix that contains coefficients at each time in the transition equation and the measurement equation—that is, $(F_t', H_t')'$.

var is an $(N_z + N_y) \times (N_z + N_y)$ matrix for a time-invariant variance matrix for the error in the transition equation and the error in the measurement equation, or a $(T + \text{lead})(N_z + N_y) \times (N_z + N_y)$ matrix that contains covariance matrices for the error in the transition equation and the error in the measurement equation—that is, $(\eta_t', \epsilon_t')'$.

intd is an $(N_z + N_\beta) \times 1$ vector that contains the intercept term in the equation for the initial state vector z_0 and the mean effect β—that is, $(a', b')'$.

coefd is an $(N_z + N_\beta) \times N_\delta$ matrix that contains coefficients for the initial state δ in the equation for the initial state vector z_0 and the mean effect β—that is, $(A', B')'$.

n0 is an optional scalar including an initial denominator. If *n0*> 0, the denominator for $\hat{\sigma}_t^2$ is *n0* plus the number n_t of elements of $(y_1, \ldots, y_t)'$. If *n0*≤ 0 or *n0* is not specified, the denominator for $\hat{\sigma}_t^2$ is n_t. With *n0*≥ 0, the initial values, A_1, M_1, and Q_1, are assumed to be known and, hence, *at*, *mt*, and *qt* are used for input that contains the initial values. If the value of *n0* is negative or *n0* is not specified, the initial values for *at*, *mt*, and *qt* are computed. The value of *n0* is updated as $\max(n0, 0) + n_t$ after the KALDFF call.

at is an optional $kN_z \times (N_\delta + 1)$ matrix. If *n0*≥ 0, *at* contains $(A_1', \ldots, A_k')'$. However, only the first matrix $\mathbf{A}_1$ is used as input. When you specify the KALDFF call, *at* returns $(A'_{T-k+\text{lead}+1}, \ldots, A'_{T+\text{lead}})'$. If *n0* is negative or the matrix $\mathbf{A}_1$ contains missing values, $\mathbf{A}_1$ is automatically computed.

mt is an optional $kN_z \times N_z$ matrix. If *n0*≥ 0, *mt* contains $(M_1, \ldots, M_k)'$. However, only the first matrix M_1 is used as input. If *n0* is negative or the matrix M_1 contains missing values, *mt* is used for output, and it contains $(M_{T-k+\text{lead}+1}, \ldots, M_{T+\text{lead}})'$. Note that the matrix M_1 can be used as an input matrix if either of the off-diagonal elements is not missing. The missing element $M_1(i, j)$ is replaced by the nonmissing element $M_1(j, i)$.

qt is an optional $k(N_\delta+1)\times(N_\delta+1)$ matrix. If *n0*≥ 0, *qt* contains $(Q_1, \ldots, Q_k)'$. However, only the first matrix Q_1 is used as input. If *n0* is negative or the matrix Q_1 contains missing values, *qt* is used for output and contains $(Q_{T-k+\text{lead}+1}, \ldots, Q_{T+\text{lead}})'$. The matrix Q_1 can also be used as an input matrix if either of the off-diagonal elements is not missing since the missing element $Q_1(i, j)$ is replaced by the nonmissing element $Q_1(j, i)$.

The KALDFF call returns the following values:

pred is a $(T + \text{lead}) \times N_z$ matrix that contains estimated predicted state vectors $(\hat{z}_{1|0}, \ldots, \hat{z}_{T+1|T}, \hat{z}_{T+2|T}, \ldots, \hat{z}_{T+\text{lead}|T})'$.

vpred is a $(T + \text{lead})N_z \times N_z$ matrix that contains estimated mean square errors of predicted state vectors $(P_{1|0}, \ldots, P_{T+1|T}, P_{T+2|T}, \ldots, P_{T+\text{lead}|T})'$.

initial is an $N_d \times (N_d + 1)$ matrix that contains an estimate and its variance for initial state δ, that is, $(\hat{\delta}_T, \hat{\Sigma}_{\delta,T})$.

s2 is a scalar that contains the estimated variance $\hat{\sigma}_T^2$.

The KALDFF call computes the one-step forecast of state vectors in an SSM by using the diffuse Kalman filter. The SSM for the diffuse Kalman filter is written

$$\begin{aligned} y_t &= X_t\beta + H_t z_t + \epsilon_t \\ z_{t+1} &= W_t\beta + F_t z_t + \eta_t \\ z_0 &= a + A\delta \\ \beta &= b + B\delta \end{aligned}$$

where z_t is an $N_z \times 1$ state vector, y_t is an $N_y \times 1$ observed vector, and

$$\begin{aligned} \begin{bmatrix} \eta_t \\ \epsilon_t \end{bmatrix} &\sim N\left(\mathbf{0}, \sigma^2 \begin{bmatrix} V_t & G_t \\ G_t' & R_t \end{bmatrix}\right) \\ \delta &\sim N(\mu, \sigma^2\Sigma) \end{aligned}$$

It is assumed that the noise vector $(\eta_t', \epsilon_t')'$ is independent and δ is independent of the vector $(\eta_t', \epsilon_t')'$. The matrices, W_t, F_t, X_t, H_t, a, A, b, B, V_t, G_t, and R_t, are assumed to be known. The KALDFF call estimates the conditional expectation of the state vector z_t given the observations. The KALDFF subroutine also

produces the estimates of the initial random vector δ and its covariance matrix. For k-step forecasting where $k > 0$, the estimated conditional expectation at time $t + k$ is computed with observations given up to time t. The estimated k-step forecast and its estimated MSE are denoted $z_{t+k|t}$ and $P_{t+k|t}$ (for $k > 0$). $A_{t+k(\delta)}$ and $E_{t(\delta)}$ are last-column-deleted submatrices of A_{t+k} and E_t, respectively. The algorithm for one-step prediction is given as follows:

$$
\begin{aligned}
E_t &= (X_t B,\ y_t - X_t b) - H_t A_t \\
D_t &= H_t M_t H_t' + R_t \\
Q_{t+1} &= Q_t + E_t' D_t^- E_t \\
&= \begin{bmatrix} S_t & s_t \\ s_t' & q_t \end{bmatrix} \\
\hat{\sigma}_t^2 &= (q_t - s_t' S_t^- s_t)/n_t \\
\hat{\delta}_t &= S_t^- s_t \\
\hat{\Sigma}_{\delta,t} &= \hat{\sigma}_t^2 S_t^- \\
K_t &= (F_t M_t H_t' + G_t) D_t^- \\
A_{t+1} &= W_t(-B, b) + F_t A_t + K_t E_t \\
M_{t+1} &= (F_t - K_t H_t) M_t F_t' + V_t - K_t G_t' \\
z_{t+1|t} &= A_{t+1}(-\hat{\delta}_t', 1)' \\
P_{t+1|t} &= \hat{\sigma}_t^2 \mathbf{M}_{t+1} + A_{t+1(\delta)} \hat{\Sigma}_{\delta,t} A_{t+1(\delta)}'
\end{aligned}
$$

where n_t is the number of elements of $(y_1, \ldots, y_t)'$ plus $\max(n0, 0)$. Unless initial values are given and $n0 \geq 0$, initial values are set as follows:

$$
\begin{aligned}
A_1 &= W_1(-B, b) + F_1(-A, a) \\
M_1 &= V_1 \\
Q_1 &= \mathbf{0}
\end{aligned}
$$

For k-step forecasting where $k > 1$,

$$
\begin{aligned}
A_{t+k} &= W_{t+k-1}(-B, b) + F_{t+k-1} A_{t+k-1} \\
M_{t+k} &= F_{t+k-1} M_{t+k-1} F_{t+k-1}' + V_{t+k-1} \\
D_{t+k} &= H_{t+k} M_{t+k} H_{t+k}' + R_{t+k} \\
z_{t+k|t} &= A_{t+k}(-\hat{\delta}_t', 1)' \\
P_{t+k|t} &= \hat{\sigma}_t^2 M_{t+k} + A_{t+k(\delta)} \hat{\Sigma}_{\delta,t} A_{t+k(\delta)}'
\end{aligned}
$$

If there is a missing observation, the KALDFF call computes the one-step forecast for the observation that follows the missing observation as the two-step forecast from the previous observation.

An example that uses the KALDFF call is in the documentation for the KALDFS call.

KALDFS Call

CALL KALDFS(*sm, vsm, data, int, coef, var, bvec, bmat, initial, at, mt, s2* **<** *, un, vun* **>);**

The KALDFS subroutine computes the smoothed state vector and its mean square error matrix from the one-step forecast and mean square error matrix computed by KALDFF subroutine.

The input arguments to the KALDFS subroutine are as follows:

data	is a $T \times N_y$ matrix that contains data $(y_1, \ldots, y_T)'$.
int	is an $(N_z + N_y) \times N_\beta$ vector for a time-invariant intercept, or a $(T + \text{lead})(N_z + N_y) \times N_\beta$ vector that contains fixed matrices for the time-variant model in the transition equation and the measurement equation—that is, $(W_t', X_t')'$.
coef	is an $(N_z + N_y) \times N_z$ matrix for a time-invariant coefficient, or a $(T + \text{lead})(N_z + N_y) \times N_z$ matrix that contains coefficients at each time in the transition equation and the measurement equation—that is, $(F_t', H_t')'$.
var	is an $(N_z + N_y) \times (N_z + N_y)$ matrix for a time-invariant variance matrix for transition equation noise and the measurement equation noise, or a $(T + \text{lead})(N_z + N_y) \times (N_z + N_y)$ matrix that contains covariance matrices for the transition equation and measurement equation errors—that is, $(\eta_t', \epsilon_t')'$.
bvec	is an $N_\beta \times 1$ constant vector for the intercept for the mean effect β.
bmat	is an $N_\beta \times N_\delta$ matrix for the coefficient for the mean effect β.
initial	is an $N_\delta \times (N_\delta + 1)$ matrix that contains an initial random vector estimate and its covariance matrix—that is, $(\hat{\delta}_T, \hat{\Sigma}_{\delta,T})$.
at	is a $TN_z \times (N_\delta + 1)$ matrix that contains $(A_1', \ldots, A_T')'$.
mt	is a $(TN_z) \times N_z$ matrix that contains $(M_1, \ldots, M_T)'$.
s2	is the estimated variance in the end of the data set, $\hat{\sigma}_T^2$.
un	is an optional $N_z \times (N_\delta + 1)$ matrix that contains u_T. The returned value is u_0.
vun	is an optional $N_z \times N_z$ matrix that contains U_T. The returned value is U_0.

The KALDFS call returns the following values:

sm	is a $T \times N_z$ matrix that contains smoothed state vectors $(z_{1\|T}, \ldots, z_{T\|T})'$.
vsm	is a $TN_z \times N_z$ matrix that contains mean square error matrices of smoothed state vectors $(P_{1\|T}, \ldots, P_{T\|T})'$.

Given the one-step forecast and mean square error matrix in the KALDFF call, the KALDFS call computes a smoothed state vector and its mean square error matrix. Then the KALDFS subroutine produces an estimate of the smoothed state vector at time t—that is, the conditional expectation of the state vector z_t given all observations. Using the notations and results from the KALDFF subroutine section, the backward recursion algorithm for smoothing is denoted for $t = T, T-1, \ldots, 1$,

$$
\begin{aligned}
E_t &= (X_t B,\ y_t - X_t b) - H_t A_t \\
D_t &= H_t M_t H_t' + R_t \\
L_t &= F_t - (F_t M_t H_t' + G_t) D_t^- H_t \\
u_{t-1} &= H_t' D_t^- E_t + L_t' u_t \\
U_{t-1} &= H_t' D_t^- H_t + L_t' U_t L_t \\
z_{t|T} &= (A_t + M_t u_{t-1})(-\hat{\delta}_T', 1)' \\
C_t &= A_t + M_t u_{t-1} \\
P_{t|T} &= \hat{\sigma}_T^2 (M_t - M_t R_{t-1} M_t) + C_{t(\delta)} \hat{\Sigma}_{\delta,T} C_{t(\delta)}'
\end{aligned}
$$

where the initial values are $u_T = b0$ and $U_T = \mathbf{0}$, and $C_{t(\delta)}$ is the last-column-deleted submatrix of C_t. See De Jong (1991) for details about smoothing in the diffuse Kalman filter.

The KALDFS call is accompanied by the KALDFF call as shown in the following statements:

```
ny = ncol(y);
nz = ncol(coef);
nb = ncol(int);
nd = ncol(coefd);
at = j(nz, nd+1, .);
mt = j(nz, nz, .);
qt = j(nd+1, nd+1, .);
n0 = -1;
call kaldff(pred, vpred, initial, s2, y, 0, int, coef, var, intd,
            coefd, n0, at, mt, qt);
bvec = intd[nz+1:nz+nb,];
bmat = coefd[nz+1:nz+nb,];
call kaldfs(sm, vsm, x, int, coef, var, bvec, bmat,
            initial, at, mt, s2);
```

You can also compute the smoothed estimate and its covariance matrix observation by observation. When the SSM is time invariant, the following statements perform smoothing. You should initialize UN and VUN as matrices in which all elements are zero.

```
n  = nrow(y);
ny = ncol(y);
nz = ncol(coef);
nb = ncol(int);
nd = ncol(coefd);
at = j(nz, nd+1, .);
mt = j(nz, nz, .);
qt = j(nd+1, nd+1, .);
n0 = -1;
```

```
call kaldff(pred, vpred, initial, s2, y, 0, int, coef, var, intd,
            coefd, n0, at, mt, qt);
bvec = intd[nz+1:nz+nb,];
bmat = coefd[nz+1:nz+nb,];
un  = j(nz, nd+1, 0);
vun = j(nz, nz, 0);
do i = 1 to n;
   call kaldfs(sm_i, vsm_i, y[n-i+1], int, coef, var, bvec, bmat,
               initial, at, mt, s2, un, vun);
   sm  = sm_i // sm;
   vsm = vsm_i // vsm;
end;
```

The final example simulates random values for the data, intercept, coefficients, and variance matrix. It calls the KALDFF subroutine and uses the result as input to the KALDFS subroutine. The output is not shown.

```
proc iml;
q=4;
p=2;
qm=3;
qd=3;
n=10;
lead=0;
total=n+lead;
call randseed(25735);
/* simulate random values for input arguments */
x=j(n,p);              call randgen(x, "Normal");
int=j(total*(p+q),qm);call randgen(int, "Normal");
coef=j(total*(p+q),q);call randgen(coef, "Normal");
temp=j(p+q,p+q);
free var;
do ii=1 to total;
   call randgen(temp, "Normal");
   var=var//(temp*temp`);
end;
intd=j(q+qm,1);   call randgen(intd, "Normal");
coefd=j(q+qm,qd); call randgen(coefd, "Normal");

n0=-1;
at2=j(n*q,qd+1);
mt2=j(n*q,q);
qt2=j(n*(qd+1),qd+1);

call kaldff(pred,vpred,init,s2,
            x,lead,int,coef,var,intd,coefd,
            n0,at2,mt2,qt2);
print init,s2;

call kaldfs(sm,vsm,
            x,int,coef,var,intd[q+1:q+qm,],coefd[q+1:q+qm,],
            init,at2,mt2,s2);
print sm, vsm;
```

KPCATRAIN call

CALL KPCATRAIN(*eigenVal, eigenVec, centroids, rowMeans, scoreTrain, data, < , opt, optCluster>* **);**

Kernel principal component analysis (kernel PCA) is a nonlinear form of traditional PCA (Schölkopf, Smola, and Müller 1998). It uses the same basic idea as traditional PCA; that is, it seeks to project the set of data onto a low-dimensional subspace that captures the highest possible amount of variance in the data. Whereas traditional PCA performs a linear projection of the data onto a subset of the original space, kernel PCA instead uses a mapping function to embed the data onto a high-dimensional space referred to as $\mathcal{F}$ by a linear dimensionality reduction through the "kernel trick" in that space. Different kernels correspond to different mapping functions. In this way, you can find a nonlinear subspace that contains the data. The applications of kernel PCA include nonlinear dimensionality reduction, nonlinear data classification, kernel principal component regression, image denoising, and novelty detection.

The KPCATRAIN subroutine calculates the projections of training data onto the principal components in the space $\mathcal{F}$; it also calculates the eigenvector matrix and eigenvalue vector of the kernel matrix. By default, the KPCATRAIN subroutine applies k-means clustering to select representative points (cluster centroids) in the input data matrix and uses a low-rank approximation method for the kernel matrix construction. The low-rank approximation method greatly reduces the complexity of the eigenvector decomposition of the kernel matrix in terms of both computation and memory, and it achieves a performance that is comparable to the exact method.

The input arguments to the KPCATRAIN subroutine are as follows:

data is an $n \times m$ training data matrix that contains data $(x_1, \ldots, x_n)'$.

opt specifies an options vector. If an element of the options vector is a missing value, a default value is used. The components of *opt* are as follows:

opt[1] specifies the eigenvalue cutoff value to use to determine the rank. Note that the rank determined in this way depends on the scaling of the data. If the variables are measured in different scales, you might want to standardize the data. The cutoff value should be nonnegative and not greater than 1. By default, the value of *opt[1]* is 1E–8.

opt[2] specifies the kernel type:

0 linear kernel

1 polynomial kernel

2 Gaussian kernel

By default, the value of *opt[2]* is 2.

opt[3] specifies the kernel parameter, which is treated as follows:

- For the linear kernel type, this parameter is ignored.
- For the polynomial kernel type, this parameter is the polynomial degree, can take positive integer values, and has a default value of 3.
- For the Gaussian kernel type, the parameter is the bandwidth value, can take real positive values, and its default value is $\frac{0.25}{n(n-1)} \sum_{i,j=1}^{n} \left\| x_i - x_j \right\|^2$, where n is the number of rows in the training data matrix. This choice of default value comes from the idea in Kwok and Tsang (2004). The default parameter is not guaranteed

to produce good results. It is recommended that you tune this parameter by some systematic approach, such as cross validation.

opt[4] specifies the computation method:

0 low-rank approximation method

1 exact method

By default, the value of *opt[4]* is 0.

opt[5] specifies the number of principal components to use in scoring the training data. By default, the value of *opt[5]* is 4.

optCluster specifies another options vector, which contains the options that are related to k-means clustering for the low-rank approximation of the kernel matrix. If this vector contains missing values, default values are used for missing options. The components of *optCluster* are as follows:

optCluster[1] specifies the k-means clustering seed initialization type:

1 fast clustering, which spreads out the initial seeds in a heuristic way.

2 k-means++ (Arthur and Vassilvitskii 2007), which spreads out the initial seeds by introducing a weighted probability distribution in seed selection.

3 random, which selects initial seeds by pure randomization.

By default, the value of *optCluster[1]* is 1.

optCluster[2] specifies the number of clusters, c, in k-means clustering. This value is a positive integer larger than 1. If c is equal to or larger than n (the number of rows of input training data matrix), the exact method is used instead. By default, the value of *optCluster[2]* is 100.

optCluster[3] specifies the random seed to use in centroids initialization. This value is a positive integer no larger than 2,147,483,647. By default, the value of *optCluster[3]* is 0, which means the internal system time is used to initialize the random number stream.

optCluster[4] specifies the maximum number of iterations for recomputing cluster centroids. This value is a positive integer. By default, the value of *optCluster[4]* is 50.

optCluster[5] specifies the convergence criterion of the k-means clustering algorithm. This value is a positive real value. By default, the value of *optCluster[5]* is 0.02.

optCluster[6] specifies the number of restarts when *optCluster[1]* is 3 (random). The best k-means clustering result is selected among all the restarts. This value is a positive integer no larger than 2,048. By default, the value of *optCluster[6]* is 5.

The KPCATRAIN subroutine returns the following values:

eigenVal is an $r \times 1$ column vector that contains the scaled eigenvalues (divided by $n - 1$) of the centered kernel matrix, where r is the rank of centered kernel matrix or centered approximated kernel matrix, and n is the number of rows in the input training data matrix.

eigenVec is the $n \times r$ normalized (with respect to $\mathcal{F}$) eigenvector matrix obtained from eigendecomposition of the centered kernel matrix.

centroids is a $c \times m$ k-means clustering centroids matrix, where c is the number of clusters and m is the number of columns in the input training data matrix.

rowMeans is an $n \times 1$ column vector. Element i of this vector is the mean of the ith row in the kernel matrix.

scoreTrain is an $n \times q$ matrix that contains the projection of the training data onto the first q principal components in $\mathcal{F}$, where q is number of principal components.

The KPCATRAIN call implements an eigenvector decomposition on the centered kernel matrix. Currently, this call supports three types of kernel:

- linear kernel $k(x_i, x_j) = (x_i \cdot x_j)$, where $i, j \in (1, \ldots, n)$, which is equivalent to traditional PCA
- polynomial kernel $k(x_i, x_j) = (x_i \cdot x_j + 1)^d$, where d is the polynomial order
- Gaussian kernel $k(x_i, x_j) = e^{(-\|x_i - x_j\|^2/2s^2)}$, where s is the Gaussian kernel parameter (bandwidth)

In the exact method, the whole kernel matrix $\mathbf{K}_{n\times n} = \big(k(x_i, x_j)\big)_{ij}$ is constructed. The centered version can be calculated as $\tilde{\mathbf{K}}_{n\times n} = \mathbf{K}_{n\times n} - \mathbf{1}_{1/n}\mathbf{K}_{n\times n} - \mathbf{K}_{n\times n}\mathbf{1}_{1/n} + \mathbf{1}_{1/n}\mathbf{K}_{n\times n}\mathbf{1}_{1/n}$, where $\mathbf{1}_{1/n}$ is an $n \times n$ matrix with all elements equal to $1/n$. Then eigendecomposition is performed on $\tilde{\mathbf{K}} = U\Lambda U^T$. The normalized version of U with respect to space $\mathcal{F}$ is $U\Lambda^{-1/2}$, which is returned in *eigenVec*.

The low-rank approximation algorithm stems from the Nyström method (Baker 1977). The Nyström method can be treated as choosing a subset of c columns, $\mathbf{E} \in \mathbb{R}^{n\times c}$, from the kernel matrix and then reconstructing the complete kernel matrix by $\mathbf{K} \simeq \mathbf{E}\mathbf{W}^{-1}\mathbf{E}'$, where $\mathbf{W}$ is the intersection of the selected rows and columns of $\mathbf{K}$. The sampling scheme is important to the performance of Nyström method. In (Zhang, Tsang, and Kwok 2008), an error analysis is conducted and the resulting error bound suggests using k-means clustering as the sampling scheme. The greater the number of clusters chosen in k-means clustering, the more accurate the reconstruction. If the number of clusters equals the number of observations, the reconstruction will be exact. Low-rank approximation reduces the memory complexity of kernel PCA from $\mathcal{O}(n^2)$ to $\mathcal{O}(nc)$ and reduces computation complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(nc^2)$.

The computational details of a low-rank approximation are as follows:

1. The centroids are obtained from k-means clustering; they are denoted as z_i $(i = 1, \ldots, c)$.
2. The kernel matrix between the training data and the centroids is constructed as $E_{n\times c} = \big(k(x_i, z_j)\big)_{ij}$.
3. The kernel matrix between the centroids themselves is calculated as $\mathbf{W}_{c\times c} = \big(k(z_i, z_j)\big)_{ij}$.
4. $\mathbf{P}^{-1} = U\Lambda^{-1/2}U^T$ is formed, where $U\Lambda U^T$ is the eigendecomposition of W.
5. $G = \mathbf{E}\mathbf{P}^{-1}$ is formed, and the centered version $\tilde{G} = (G - \bar{G})$ is calculated, where $\bar{G}$ is column means of G.
6. Eigenvector decomposition on $\tilde{G}^T\tilde{G} = V\Lambda V^T$ is performed.
7. The centered kernel matrix is approximated as $\tilde{\mathbf{K}} = \tilde{G}\tilde{G}^T$, and the first c eigenvectors of $\tilde{\mathbf{K}}_{n\times n}$ can be approximated by $U \approx \tilde{G}V\Lambda^{-1}$.

After the eigenvector matrix U and diagonal eigenvalue matrix Λ are obtained, the projection of the training data onto the first q principal components can be calculated as $S = \tilde{\mathbf{K}}_{n\times n} U\Lambda^{-1/2} = U\Lambda U^T U\Lambda^{-1/2} = U\Lambda^{1/2}$.

Since kernel PCA is a nonlinear generalization of PCA, it can perform traditional PCA if you use a linear kernel and the exact method. For example, suppose you are interested in reproducing the principal component analysis from a call to the PRINCOMP procedure. The following call uses the Sashelp.Iris data set, which has four variables:

```
proc princomp data=sashelp.iris out=PCComponents cov n=3
              plots(only)=score(ncomp=2);
run;
```

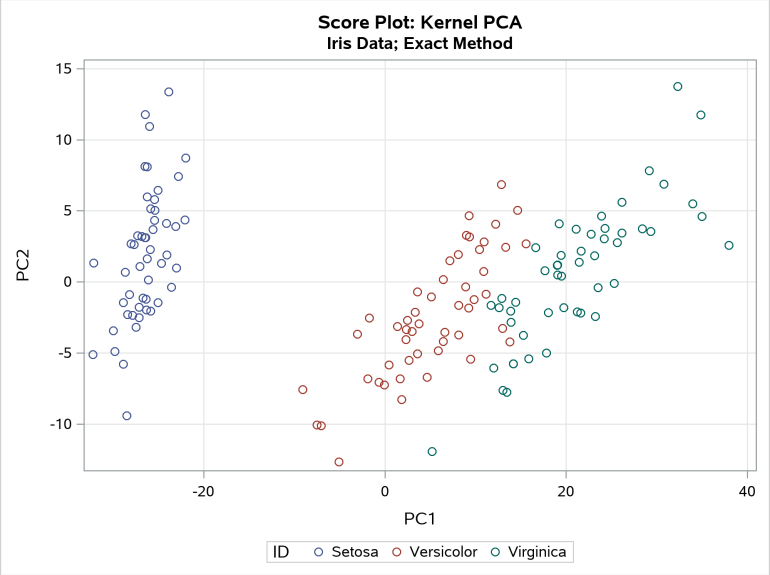
PROC PRINCOMP creates an output data set that contains the scored data. The scores are the projection of the data onto the first three eigenvectors of the covariance matrix. The following KPCATRAIN call performs an equivalent analysis. The *opt* vector requests that the analysis use a linear kernel and an exact method and project the data onto three principal components. The SCORE matrix returns the principal component scores. The first two scores are plotted against each other in Figure 25.210.

```
proc iml;
use sashelp.iris;
   read all var _NUM_ into X[c=varName];
   read all var "Species" into ID;
close;

/* traditional PCA */
opt = {.,          /* eigenvalue cutoff (default) */
       0,          /* linear kernel */
       .,          /* linear kernel */
       1,          /* exact method for PCA */
       3};         /* project on 3 PCs */
optClus = j(6,1,.);   /* default values */
call KPCATrain(eigenVal, eigenVec, centroids, RowMeans, score,
               X, opt, optClus);
print eigenVal;
title "Score Plot: Kernel PCA";
title2 "Iris Data; Exact Method";
call scatter(score[,1], score[,2]) grid={x y} group=ID label={"PC1" "PC2"};
```

Figure 25.209 Eigenvalues Obtained From Kernel PCA

eigenVal
422.82417
24.267075
7.82095
2.3835093

Figure 25.210 Projection of Iris Data onto the First Two Principal Components



Because eigenvectors are not unique, the columns of the SCORE matrix might differ in sign from the Prin1, Prin2, and Prin3 variables in the PCComponents data set (obtained from PROC PRINCOMP). For these data, the third column of the SCORE matrix is the negative of the Prin3 variable. For an example of kernel PCA with a nonlinear (Gaussian) kernel, see the example in the KPCASCORE function.

KPCASCORE function

KPCASCORE(*eigenVec, data, rowMeans, newdata, < , opt >***);**

The KPCASCORE function takes the results from the KPCATRAIN call as inputs and calculates the projections of the new data matrix onto the principal components in a high-dimensional space, $\mathcal{F}$.

The input arguments to the KPCASCORE function are as follows:

eigenVec is an $n \times r$ eigenvector matrix that is obtained from a KPCATRAIN call, where n is the number of rows in the input training data matrix and r is the rank of centered kernel matrix of the training data.

data is an $n \times m$ training data matrix that contains data $(x_1, \ldots, x_n)'$.

rowMeans is an $n \times 1$ column vector from the output of the KPCATRAIN call. Element i of this vector is the mean of the ith row in the kernel matrix.

newdata is a $p \times m$ new data matrix that contains the scoring data $(y_1, \ldots, y_p)'$.

opt is an optional vector that contains the scoring options. The second and third options should have the same values as those in the KPCATRAIN call.

opt[1] specifies the number of principal components to use in scoring the new data. This value should be a positive integer no larger than r (the rank of the centered kernel matrix). If the value of *opt[1]* is larger than r, r is used as *opt[1]*. By default, the value of *opt[1]* is 4.

opt[2] specifies the kernel type. It takes the same value as *opt[2]* in the KPCATRAIN call.

opt[3] specifies the kernel parameter. It takes the same value as *opt[3]* in the KPCATRAIN call.

The KPCASCORE function returns a $p \times q$ matrix that contains the projection of the new data onto the first q principal components in space $\mathcal{F}$, where q is the value of the *opt[1]* argument.

The KPCASCORE function calculates the kernel matrix between the training data and new data: $\mathbf{K}_{p\times n} = k(y_j, x_i)$ for $j = 1, \ldots, p$ and $i = 1, \ldots, n$. It also calculates the centered version, $\tilde{\mathbf{K}}_{p\times n} = \mathbf{K}_{p\times n} - \mathbf{1}_{1/n}\mathbf{K}_{n\times n} - \mathbf{K}_{p\times n}\mathbf{1'}_{1/n} + \mathbf{1}_{1/n}\mathbf{K}_{n\times n}\mathbf{1'}_{1/n}$, where $\mathbf{K}_{n\times n}$ is the kernel matrix of training data, $\mathbf{1}_{1/n}$ is a $p \times n$ matrix with all elements equal to $1/n$, and $\mathbf{1'}_{1/n}$ is an $n \times n$ matrix with all elements equal to $1/n$. Centering is done using the *rowMeans* argument. The projection of the new data onto the first q principal components can be computed as $S = \tilde{\mathbf{K}}_{m\times n} U\Lambda^{-1/2}$, where $U\Lambda^{-1/2}$ is the value of the *eigenVec* argument.

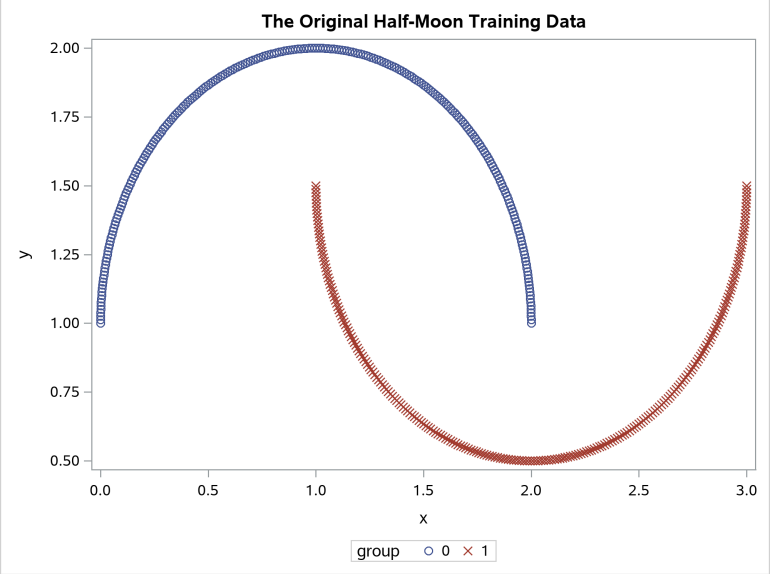
The following example first generates two nonlinear "half-moon" sets of data, one for training and the other for scoring. Then it calls the KPCATRAIN subroutine on the training half-moon data, computes the first two principal component scores, and plots them against each other in Figure 25.210. Then it calls the KPCASCORE function on the new "half-moon" data. The KPCASCORE function takes the outputs from the KPCATRAIN call as inputs, computes the first two principal component scores of new data, and plots them against each other in Figure 25.211.

```
/* Define functions for "half-moon" data generation */
start Linspace(a, b, n);
   if n<2 then return( b );
   incr = (b-a) / (n-1);
   return( do(a, b, incr) );
finish;

start make_moons(X, y, n=100, noise=., random_state);
    pi = constant("pi");
    n_out = floor(n/2);
    n_in = n - n_out;
    outer_circ_x = cos(Linspace(0, pi, n_out));
    outer_circ_y = sin(Linspace(0, pi, n_out));
    inner_circ_x = 1 - cos(Linspace(0, pi, n_in));
    inner_circ_y = 1 - sin(Linspace(0, pi, n_in)) - .5;
    X=(outer_circ_x||inner_circ_x)`||(outer_circ_y||inner_circ_y)`;
    y=(j(1,floor(n/2),0)||j(1,floor(n/2),1))`;
    Xy=X||y;
    call randseed(random_state, 1);
    obsIdx = sample(1:nrow(Xy), nrow(Xy), "NoReplace");
       X=Xy[obsIdx,1:ncol(Xy)-1];
       y=Xy[obsIdx,ncol(Xy)];
       Xn=j(nrow(X),2);
       if noise^=. then
        call randgen(Xn, "Normal", 0, noise);
       X=X+Xn;
finish;


/* Generate training "half-moon" data */
n1=500;
```

```
random_state_1=0;
run make_moons(X, y, n1, ., random_state_1);

moon_o=X||y;
create moon_o from moon_o[colname={"x" "y" "group"}];
append from moon_o;
close moon_o;

/* Set opt and optCluster vector, then call KPCATRAIN to score the training data */
opt={1e-15,2,0.18};
optCluster={2,100,.,.,.,.};
call KPCATRAIN(eigenVal, eigenVec, centroids, kernelMatRowMeans, scoreTrain, X, opt,
optCluster);

moon_t=scoreTrain[,1]||scoreTrain[,2]||y;
create moon_t from moon_t[colname={"x" "y" "group"}];
append from moon_t;
close moon_t;


/* Generate new "half-moon" data for scoring */
n2=200;
random_state_2=1;
run make_moons(X_new, y_new, n2, ., random_state_2);


/* Set score opt vector, then call KPCASCORE to score the new data */
opt={2,2,0.18};
score=KPCASCORE(eigenVec, X, kernelMatRowMeans, X_new, opt);

moonscore=score[,1]||score[,2]||y_new;
create moonscore from moonscore[colname={"x" "y" "group"}];
append from moonscore;
close moonscore;
quit;


/* Plot original training data, scores of training data, and scores of new data */
ods graphics on / attrpriority=none;
title "The Original Half-Moon Training Data";
proc sgplot data=moon_o;
    styleattrs datasymbols=(Circle X);
    scatter x=x y=y / group=group markerattrs=(size=7px);
run;

title "Score Plot: Kernel PCA";
title2 "Half-Moon Training Data; Approximation Method";
proc sgplot data=moon_t;
    xaxis label = "1st principal component";
    yaxis label = "2nd principal component";
    styleattrs datasymbols=(Circle X);
    scatter x=x y=y / group=group markerattrs=(size=7px);
run;
```

```
title "Score Plot: Kernel PCA";
title2 "Half-Moon New Data";
proc sgplot data=moonscore;
    xaxis label = "1st principal component";
    yaxis label = "2nd principal component";
    styleattrs datasymbols=(Circle X);
    scatter x=x y=y / group=group markerattrs=(size=7px);
run;
```

Figure 25.211 Original Half-Moon Training Data



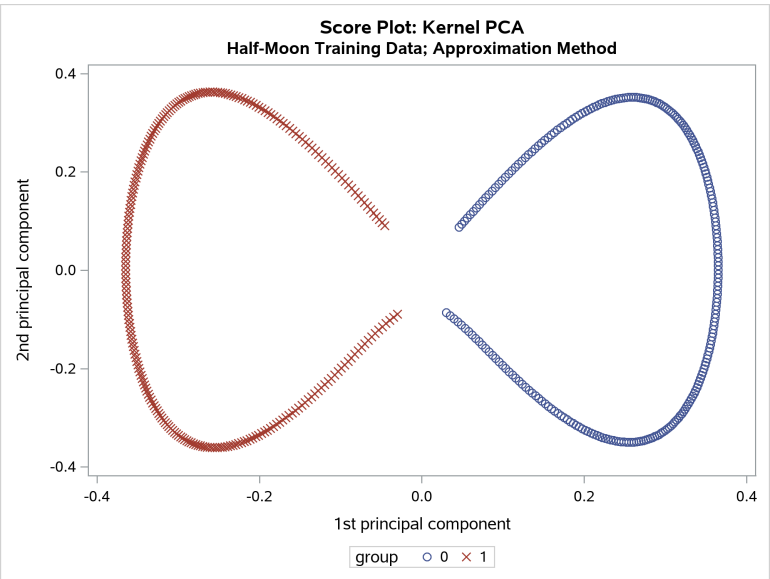
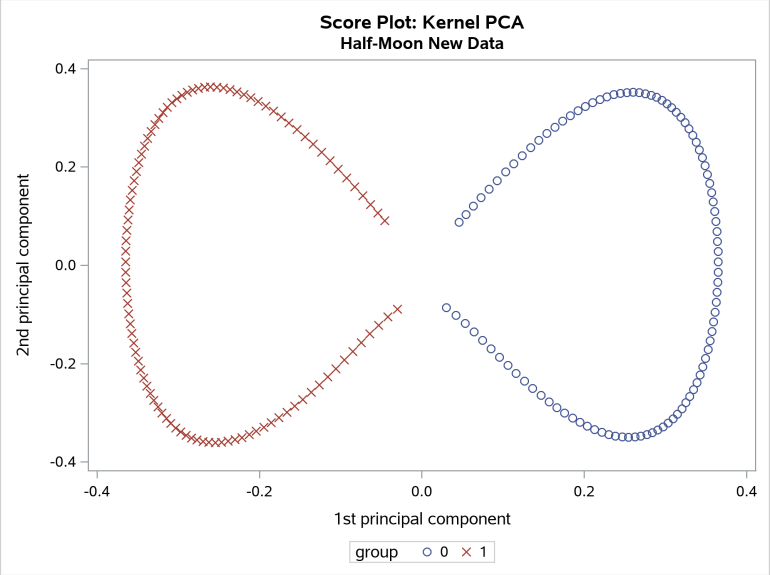


Figure 25.211 *continued*



As demonstrated in Figure 25.211, the first two principal component scores are plotted against each other for both training data and new data. The original nonlinear data are now linearly separable through the score plots.

KURTOSIS Function

KURTOSIS(*x*);

The KURTOSIS function is part of the IMLMLIB library. The KURTOSIS function returns the sample kurtosis for each column of a matrix. The sample kurtosis measures the heaviness of the tails of a data distribution. The KURTOSIS function returns an estimate for the *excess kurtosis*, which is 3 less than the standardized fourth central moment.

The KURTOSIS function returns the same sample kurtosis as the UNIVARIATE procedure. For a formula, see the section "Descriptive Statistics" in the chapter "The UNIVARIATE Procedure" in *Base SAS Procedures Guide: Statistical Procedures*.

The following example computes the kurtosis for each column of a matrix:

```
x = {1 0,
     2 1,
     4 2,
     8 3,
    16 . };
kurt = kurtosis(x);
print kurt;
```

Figure 25.212 Sample Kurtosis of Two Columns

kurt	
1.3037634	-1.2

LAG Function

LAG(*x* < , *lags*> < , *delRows*> **);**

The LAG function computes one or more lagged (shifted) values for time series data. The arguments are as follows:

x	specifies an $n \times 1$ numerical matrix of time series data.
lags	specifies integer lags. The *lags* argument can be an integer matrix with d elements. If so, the LAG function returns an $n \times d$ matrix where the ith column represents the ith lag applied to the time series. If the *lags* argument is not specified, a value of 1 is used.
delRows	is an binary flag variable that specifies whether to delete certain rows in the result matrix. By default, *delRows* is 0 and the output matrix contains the same number of rows as the input vector. If *delRows* is nonzero, then the first max(*lags*,0) rows and the last min(*lags*,0) rows of the result matrix are deleted. For example, if *x* has n elements and `y=LAG(x,3)`, then `y` is an n-element vector whose first three rows contain missing values. In contrast, `z=LAG(x,3,1)` returns a vector that has $n - 3$ elements and that equals `y[4:`*n*`]`.

The values of the *lags* argument are usually positive integers. A positive lag shifts the time series data backwards in time. A lag of 0 represents the original time series. A negative value for the *lags* argument shifts the time series data forward in time; this is sometimes called a *lead effect*. The LAG function is related to the DIF function.

For example, the following statements compute several lags:

```
x = {1,3,4,7,9};
lag = lag(x, {0 1 3});
print lag;
```

Figure 25.213 Lagged Data

lag		
1	.	.
3	1	.
4	3	.
7	4	1
9	7	3

LAMBERTW Function

LAMBERTW(*matrix* **<** *, branch* **>);**

The LAMBERTW function evaluates either branch of the real-valued Lambert W function (Corless et al. 1996). The Lambert W function is defined implicitly: for a real value x, $W(x)$ is the value that satisfies the equation $W \exp(W) = x$. Thus W is the inverse of the function $g(w) = w \exp(w)$.

The W function has two branches. The principal branch is denoted by W_0 and has the domain $x \geq -1/e$. The negative (lower) branch is denoted by W_{-1} and has the domain $-1/e \leq x < 0$. The Lambert W function is computed according to the method described by Barry, Culligan-Hensley, and Barry (1995).

The optional scalar argument *branch* determines the branch of the Lambert W function. If you omit the *branch* parameter or specify *branch* = 0, the principal branch is computed; otherwise the negative branch is computed. The following statements evaluate the principal branch of the Lambert W function at special values of x for which the exact value of $W(x)$ is known. The output is shown in Figure 25.214.

```
x     = -exp(-1) // 0 // -log(2)/2 // exp(1);
Exact = -1       // 0 // -log(2)   // 1      ;
w = LambertW(x);
print x w[L="w(x)"] Exact;
```

Figure 25.214 Values of the Lambert W Function

x	w(x)	Exact
-0.367879	-1	-1
0	0	0
-0.346574	-0.693147	-0.693147
2.7182818	1	1

To graph the Lambert W function, you can evaluate each branch of the function at a sequence of points. For example, you might evaluate the principal branch as follows:

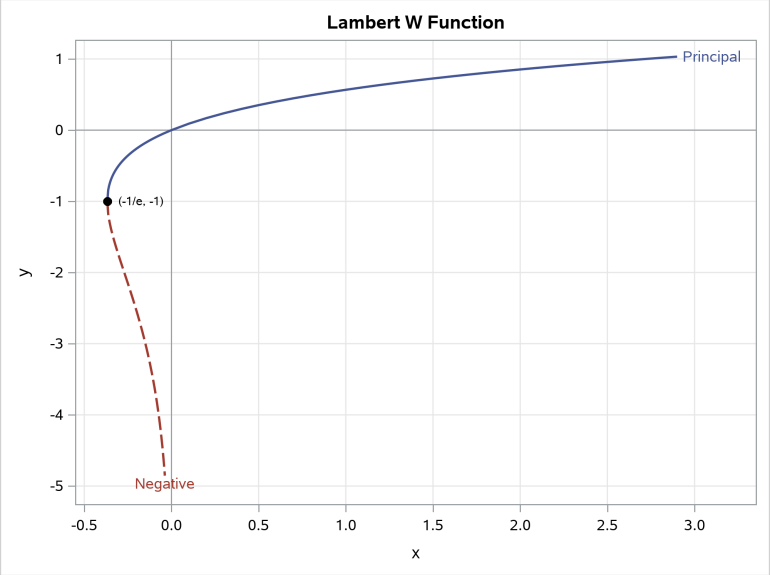
```
em = -exp(-1);
x = do( em, 0, 0.005 ) || do(0, 2.9, 0.1);
w0 = lambertW(x, 0);     /* principal branch */
```

In a similar way, you can compute values of the negative branch:

```
Upper = -5*exp(-5);      /* x value for which W(x) = -5 */
x = do(em, Upper, 0.005);
wm = lambertW(x, -1);     /* negative branch */
```

Figure 25.215 displays a graph of both branches of the Lambert W function.

Figure 25.215 The Two Branches of the Lambert W Function



The Lambert W function is used to simulate and compute quantiles for a generalized Gaussian distribution, which is a heavy-tailed distribution that has finite moments.

The derivative and integral of the Lambert W function are defined in terms of W. The derivative is $W'(x) = \exp(-W(x))/(1 + W(x))$. The indefinite integral is $\int W(x)\,dx = x\,[W(x) - 1 + 1/W(x)] + C$.

LAV Call

CALL LAV(*rc, xr, a, b* < , *x0*> < , *opt*> **);**

The LAV subroutine performs linear least absolute value regression by solving the L_1 norm minimization problem.

The LAV subroutine returns the following values:

rc is a scalar return code that indicates the reason for optimization termination.

rc	**Termination**
0	Successful
1	Successful, but approximate covariance matrix and standard errors cannot be computed
−1 or −3	Unsuccessful: error in the input arguments
−2	Unsuccessful: matrix A is rank-deficient $(\text{rank}(A) < n)$
−4	Unsuccessful: maximum iteration limit exceeded
−5	Unsuccessful: no solution found for ill-conditioned problem

xr specifies a vector or matrix with n columns. If the optimization process is not successfully completed, *xr* is a row vector with n missing values. If termination is successful and the *opt[3]* option is not set, *xr* is the vector with the optimal estimate, x^*. If termination is successful and

the *opt[3]* option is specified, *xr* is an $(n + 2) \times n$ matrix that contains the optimal estimate, x^*, in the first row, the asymptotic standard errors in the second row, and the $n \times n$ covariance matrix of parameter estimates in the remaining rows.

The input arguments to the LAV subroutine are as follows:

a specifies an $m \times n$ matrix A with $m \geq n$ and full column rank, $\mathrm{rank}(A) = n$. If you want to include an intercept in the model, you must include a column of ones in the matrix A.

b specifies the $m \times 1$ vector b.

x0 specifies an optional $n \times 1$ vector that specifies the starting point of the optimization.

opt is an optional vector used to specify options. If an element of the *opt* vector is missing, the default value is used.

- *opt[1]* specifies the maximum number *maxi* of outer iterations (this corresponds to the number of changes of the Huber parameter γ). The default is maxi $= \min(100, 10n)$. (The number of inner iterations is restricted by an internal threshold. If the number of inner iterations exceeds this threshold, a new outer iteration is started with an increased value of γ.)
- *opt[2]* specifies the amount of printed output. Higher values request additional output and include the output of lower values.

opt[2]	**Termination**
0	No output is printed.
1	Error and warning messages are printed.
2	The iteration history is printed (this is the default).
3	The n least squares (L_2 norm) estimates are printed if no starting point is specified, the L_1 norm estimates are always printed, and if *opt[3]* is set, the estimates are printed together with the asymptotic standard errors.
4	The $n \times n$ approximate covariance matrix of parameter estimates is printed if *opt[3]* is set.
5	The residual and predicted values for all m rows (equations) of A are printed.

- *opt[3]* specifies which estimate of the variance of the median of nonzero residuals be used as a factor for the approximate covariance matrix of parameter estimates and for the approximate standard errors (ASE). If *opt[3]*=0, the McKean-Schrader estimate (McKean and Schrader 1987) is used, and if *opt[3]*> 0, the Cox-Hinkley estimate (Cox and Hinkley 1974), with $v =$*opt[3]*, is used. The default behavior is that the covariance matrix is not computed.
- *opt[4]* specifies whether a computationally expensive test for necessary and sufficient optimality of the solution x is executed. The default behavior (*opt[4]*=0) is that the convergence test is not performed.

Missing values are not permitted in the *a* or *b* argument. The $x0$ argument is ignored if it contains any missing values. Missing values in the *opt* argument cause the default value to be used.

The LAV subroutine is designed for solving the unconstrained linear L_1 norm minimization problem,

$$\min_x L_1(x) \text{ where } L_1(x) = \|Ax - b\|_1 = \sum_{i=1}^{m} \left| \sum_{j=1}^{n} a_{ij} x_j - b_i \right|$$

for m equations with n (unknown) parameters $x = (x_1, \ldots, x_n)$. This is equivalent to estimating the unknown parameter vector, x, by least absolute value regression in the model

$$b = Ax + \epsilon$$

where b is the vector of n observations, A is the design matrix, and ϵ is a random error term.

An algorithm by Madsen and Nielsen (1993) is used, which can be faster for large values of m and n than the Barrodale and Roberts (1974) algorithm. The current version of the algorithm assumes that A has full column rank. Also, constraints cannot be imposed on the parameters in this version.

The L_1 norm minimization problem is more difficult to solve than the least squares (L_2 norm) minimization problem because the objective function of the L_1 norm problem is not continuously differentiable (the first derivative has jumps). A function that is continuous but not continuously differentiable is called *nonsmooth*. By using PROC NLP and the nonlinear optimization subroutines, you can obtain the estimates in linear and nonlinear L_1 norm estimation (even subject to linear or nonlinear constraints) as long as the number of parameters, n, is small. Using the nonlinear optimization subroutines, there are two ways to solve the nonlinear L_p norm, $p \geq 1$, problem:

- For small values of n, you can implement the Nelder-Mead simplex algorithm with the NLPNMS subroutine to solve the minimization problem in its original specification. The Nelder-Mead simplex algorithm does not assume a smooth objective function, does not take advantage of any derivatives, and therefore does not require continuous differentiability of the objective function. See the section "NLPNMS Call" on page 836 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 845 for details about the NLPQN subroutine.

Both approaches are successful only for a small number of parameters and good initial estimates. If you cannot supply good initial estimates, the optimal results of the corresponding nonlinear least squares (L_2 norm) estimation can provide fairly good initial estimates.

Gonin and Money (1989) show that the nonlinear L_1 norm estimation problem

$$\min_x \sum_{i=1}^{m} |f_i(x)|$$

can be reformulated as a linear optimization problem with nonlinear constraints in the following ways.

- as a linear optimization problem with $2m$ nonlinear inequality constraints in $m + n$ variables u_i and x_j,

$$\min_x \sum_{i=1}^{m} u_i \text{ subject to } \left.\begin{array}{rcl} f_i(x) - u_i & \leq & 0 \\ f_i(x) + u_i & \geq & 0 \\ u_i & \geq & 0 \end{array}\right\} \quad i = 1, \ldots, m$$

- as a linear optimization problem with $2m$ nonlinear equality constraints in $2m + n$ variables y_i, z_i, and x_j,

$$\min_x \sum_{i=1}^{m}(y_i + z_i) \text{ subject to } \left.\begin{array}{lcl} f_i(x) + y_i - z_i & = & 0 \\ y_i & \geq & 0 \\ z_i & \geq & 0 \end{array}\right\} \quad i = 1, \ldots, m$$

For linear functions $f_i(x) = \sum_{j=1}^{n}(a_{ij}x_j - b_i)$, $i = 1, \ldots, m$, you obtain linearly constrained linear optimization problems, for which the number of variables and constraints is on the order of the number of observations, m. The advantage that the algorithm by Madsen and Nielsen (1993) has over the Barrodale and Roberts (1974) algorithm is that its computational cost increases only linearly with m, and it can be faster for large values of m.

In addition to computing an optimal solution x^* that minimizes $L_1(x)$, you can also compute approximate standard errors and the approximate covariance matrix of x^*. The standard errors can be used to compute confidence limits.

The following example is the same one used for illustrating the LAV subroutine by Lee and Gentle (1986). A and b are as follows:

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & 2 \\ 1 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 2 \\ 1 \\ -1 \\ 2 \\ 4 \end{bmatrix}$$

The following statements specify the matrix A, the vector b, and the options vector *opt*. The options vector specifies that all output is printed (*opt[2]*=5), that the asymptotic standard errors and covariance matrix are computed based on the McKean-Schrader estimate λ of the variance of the median (*opt[3]*=0), and that the convergence test be performed (*opt[4]*=1).

```
a = { 0,  1, -1, -1,  2,  2 };
m = nrow(a);
a = j(m, 1, 1.) || a;
b = { 1,  2,  1, -1,  2,  4 };

opt= { . 5  0 1 };
call lav(rc, xr, a, b, , opt);
```

The first part of the output is shown in Figure 25.216. This output displays the least squares solution, which is used as the starting point. The estimates of the largest and smallest nonzero eigenvalues of $A'A$ give only an idea of the magnitude of these values, and they can be very crude approximations.

Figure 25.216 Least Squares Solution

LS Solution		
Est	1	1

The second part of the printed output shows the iteration history. It is shown in Figure 25.217.

Figure 25.217 Iteration History

LAV (L1) Estimation
Start with LS Solution
Start Iter: gamma=1 ActEqn=6

Iter	N Huber	Act Eqn	Rank	Gamma	L1(x)	F(Gamma)
1	1	2	2	0.9000	4.000000	2.200000
1	1	2	2	0.0000	4.000000	2.200000

The third part of the output is shown in Figure 25.218. This output displays the L_1 norm solution (first row) together with asymptotic standard errors (second row) and the asymptotic covariance matrix of parameter estimates. The ASEs are the square roots of the diagonal elements of this covariance matrix.

Figure 25.218 Parameter and Covariance Estimates

L1 Solution with ASE		
Est	1	1
ASE	0.4482711811	0.3310702082

Cov Matrix: McKean-Schrader	
0.2009470518	-0.054803741
-0.054803741	0.1096074828

The last part of the printed output shows the predicted values and residuals, as in Lee and Gentle (1986). It is shown in Figure 25.219.

Figure 25.219 Predicted and Residual Values

Predicted Values and Residuals

N	Observed	Predicted	Residual
1	1.0000	1.0000	0
2	2.0000	2.0000	0
3	1.0000	0.0000	1.000000
4	-1.0000	0.0000	-1.000000
5	2.0000	3.0000	-1.000000
6	4.0000	3.0000	1.000000

LCP Call

CALL LCP(*rc, w, z, m, q* < , *epsilon*> **);**

The LCP subroutine solves the linear complementarity problem:

$$\begin{aligned} \mathbf{w} &= \mathbf{M}\mathbf{z} + \mathbf{q} \\ \mathbf{w}'\mathbf{z} &= 0 \\ \mathbf{w}, \mathbf{z} &\geq 0 \end{aligned}$$

That is, given a matrix **M** and a vector **q**, the LCP subroutine computes orthogonal, nonnegative vectors **w** and **z** which satisfy the previous equations.

The input arguments to the LCP subroutine are as follows:

m is an $m \times m$ matrix.

q is an $m \times 1$ matrix.

epsilon is a scalar that defines virtual zero. The default value of *epsilon* is 1E–8.

The LCP subroutine returns the following matrices:

rc returns one of the following scalar return codes:

rc	**Termination**
0	A solution is found.
1	No solution is possible.
5	The solution is numerically unstable.
6	The subroutine could not obtain enough memory.

w returns an *m*-element column vector

z returns an *m*-element column vector

The following statements give a simple example:

```
q = {1, 1};
m = {1 0,
     0 1};
call lcp(rc, w, z, m, q);
print rc, w, z;
```

Figure 25.220 Solution to a Linear Complementarity Problem

rc
0

w
1
1

z
0
0

The next example shows the relationship between quadratic programming and the linear complementarity problem. Consider the linearly constrained quadratic program:

$$\begin{array}{rcll} \min \mathbf{c}'\mathbf{x} & + & \frac{1}{2}\mathbf{x}'\mathbf{H}\mathbf{x} & \\ \text{st. } \mathbf{G}\mathbf{x} & \geq & \mathbf{b} & \text{(QP)} \\ \mathbf{x} & \geq & 0 & \end{array}$$

If $\mathbf{H}$ is positive semidefinite, then a solution to the Kuhn-Tucker conditions solves QP. The Kuhn-Tucker conditions for QP are

$$\begin{array}{rcl} \mathbf{c} + \mathbf{H}\mathbf{x} & = & \mu + \mathbf{G}'\lambda \\ \lambda'(\mathbf{G}\mathbf{x} - \mathbf{b}) & = & 0 \\ \mu'\mathbf{x} & = & 0 \\ \mathbf{G}\mathbf{x} & \geq & \mathbf{b} \\ \\ x, \mu, \lambda & \geq & 0 \end{array}$$

In the linear complementarity problem, let

$$\begin{array}{rcl} \mathbf{M} & = & \begin{bmatrix} H & -G' \\ G & 0 \end{bmatrix} \\ \mathbf{w}' & = & (\mu' \mathbf{s}') \\ \mathbf{z}' & = & (\mathbf{x}'\lambda') \\ \mathbf{q}' & = & (\mathbf{c}' - \mathbf{b}) \end{array}$$

Then the Kuhn-Tucker conditions are expressed as finding $\mathbf{w}$ and $\mathbf{z}$ that satisfy

$$\begin{array}{rcl} \mathbf{w} & = & \mathbf{M}\mathbf{z} + \mathbf{q} \\ \mathbf{w}'\mathbf{z} & = & 0 \\ \mathbf{w}, \mathbf{z} & \geq & 0 \end{array}$$

From the solution $\mathbf{w}$ and $\mathbf{z}$ to this linear complementarity problem, the solution to QP is obtained; namely, $\mathbf{x}$ is the primal structural variable, $\mathbf{s} = \mathbf{G}\mathbf{x} - \mathbf{b}$ the surpluses, and $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$ are the dual variables. Consider a quadratic program with the following data:

$$\mathbf{C}' = (1245) \quad \mathbf{B}' = (11)$$

$$\mathbf{H} = \begin{bmatrix} 100 & 10 & 1 & 0 \\ 10 & 100 & 10 & 1 \\ 1 & 10 & 100 & 10 \\ 0 & 1 & 10 & 100 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 10 & 20 & 30 & 40 \end{bmatrix}$$

This problem is solved by using the LCP subroutine as follows:

```
/*---- Data for the Quadratic Program -----*/
c = {1, 2, 3, 4};
h = {100 10 1 0, 10 100 10 1, 1 10 100 10, 0 1 10 100};
g = {1 2 3 4, 10 20 30 40};
b = {1, 1};

/*----- Express the Kuhn-Tucker Conditions as an LCP ----*/
m = h || -g`;
m = m // (g || j(nrow(g),nrow(g),0));
q = c // -b;

/*----- Solve for a Kuhn-Tucker Point --------*/
call lcp(rc, w, z, m, q);

/*------ Extract the Solution to the Quadratic Program ----*/
x = z[1:nrow(h)];
print rc x;
```

Figure 25.221 Solution to a Quadratic Programming Problem

rc	x
0	0.0307522
	0.0619692
	0.0929721
	0.1415983

LEAVE Statement

LEAVE ;

The LEAVE statement stops the processing of the current DO loop and continues processing at the next statement following the DO loop. The LEAVE statement can be used only inside a DO loop.

You can use the LEAVE statement to exit a DO loop when a specified condition occurs. For example, the following program exits the DO loop if a missing value is encountered.

```
x = 1:10;
x[5] = .;

sum = 0;
do i = 1 to ncol(x);
   if x[i] = . then leave;  /* exit the DO loop */
   sum = sum + x[i];
end;
print i sum;
```

Figure 25.222 Sum of Values Until First Missing Value

i	sum
5	10

Notice that the value of the iteration counter (`i`) indicates the iteration at which the LEAVE statement was executed.

The LEAVE statement can be used to exit a DO loop in the middle of the loop. To exit a DO loop at the top of the loop, use a DO-WHILE statement. To exit a DO loop at the bottom of the loop, use a DO-UNTIL statement. To skip the current iteration but proceed to the next iteration, use a CONTINUE statement.

LENGTH Function

LENGTH(*matrix***);**

The LENGTH function takes a character matrix as an argument and produces a numeric matrix as a result. The result matrix has the same dimensions as the argument and contains the lengths of the corresponding string elements in *matrix*. The length of a string is equal to the position of the rightmost nonblank character in the string. If a string is entirely blank, its length value is set to 1. An example of the LENGTH function follows:

```
c = {"Hello" "My name is Jenny"};
b = length(c);
print b;
```

Figure 25.223 Length of Elements of a Character Matrix

b
5 16

See also the description of the NLENG function.

LINK Statement

LINK(*label***);**

statements **;**

label:*statements* **;**

RETURN ;

The LINK statement provides a way of calling a group of statements as if they were defined as a subroutine. When the LINK statement is executed, the program jumps immediately to the statement with the given *label* and begins executing statements from that point as it does for the GOTO statement. However, when the program executes a RETURN statement, the program returns to the statement that immediately follows the LINK statement, which is different behavior than the GOTO statement.

The LINK statement can be used only inside modules and DO groups. LINK statements can be nested within other LINK statements to any level. A RETURN statement without a LINK statement is executed the same as the STOP statement.

Instead of using a LINK statement, you can define a module and call the module by using a RUN statement.

An example that uses the LINK statement follows:

```
start a;
   x=1;
   y=2;
   link sum1; /* go to label; execute until return stmt */
   print z;
   stop;
   sum1:
      z=x+y;
   return;
finish a;

run a;
```

Figure 25.224 Result of Linking to a Group of Statements

z
3

LIST Statement

LIST <*range*> <**VAR** *operand*> <**WHERE***(expression)*> **;**

The LIST statement displays observations of a data set.

The arguments to the LIST statement are as follows:

range specifies a range of observations. You can specify a range of observations by using the ALL, CURRENT, NEXT, AFTER, and POINT keywords, as described in the section "Process a Range of Observations" on page 104.

operand specifies a set of variables. As described in the section "Select Variables with the VAR Clause" on page 105, you can specify variable names by using a matrix literal, a character matrix, an expression, or the _ALL_, _CHAR_, or _NUM_ keywords.

expression specifies observations to list. If you omit the WHERE clause, all observations are listed. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

The LIST statement displays 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*.

The following examples demonstrate the use of the LIST statement. The output is not shown.

```
use Sashelp.Class;

list all;                                /* lists whole data set */
list;                                  /* lists current observation */
list var{name age};           /* lists NAME and AGE in current obs */
list all where(age<=13); /* lists all obs where condition holds */
```

```
list next;                              /* lists next observation */
list point 18;                            /* lists observation 18 */
list point (10:15);          /* lists observations 10 through 15 */

close Sashelp.Class;
```

LISTADDITEM Call

CALL LISTADDITEM(*list, v,* < *, flag* > **);**

The ListAddItem subroutine adds a new item to end of a list. The length of the list increases by one. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

list specifies an existing list.

v specifies a SAS/IML variable or literal. It can be a matrix, table, list, or any other valid SAS/IML type, including an empty matrix. The value of *v* will become a new list element.

flag controls what happens to *v*. Valid values are:

- 'c' copies the data in *v* into the list. The *v* symbol is unchanged by the call. This is the default behavior.
- 'm' moves the data in *v* into the list. The *v* symbol is freed after the call.

The following statements create an empty list and then add two items:

```
L = ListCreate();               /* create empty list */
v1 = 1:3;
call ListAddItem(L, v1);        /* v1 is unchanged */
v2 = {"A", "Z"};
call ListAddItem(L, v2, 'm');   /* v2 is freed */
```

When you need to combine two lists, you can use the list concatenation operator (`||`). For example, if `S` and `T` are two lists, then `L = S || T` is the list that contains all items of `S` followed by all items of `T`.

LISTCREATE Function

LISTCREATE( );

LISTCREATE(*n* **);**

LISTCREATE(*names* **);**

The ListCreate function returns a new list. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

n returns a list that contains *n* empty items. You can then use the ListSetItem subroutine to set the value of each item.

names returns a named list whose length is the number of items in the *names* parameter. You can then use the ListSetItem subroutine to set the value of each item.

If you do not supply an argument, the function returns an empty list.

The following statements demonstrate several ways to create and fill a list:

```
L1 = ListCreate();                  /* L1 is an empty list */
call ListAddItem(L1, 1:3);          /* L1 now has one item */

L2 = ListCreate(2);                  /* allocate two items */
call ListSetItem(L2, 1, "A":"Z");   /* 1st item: character vector */
call ListSetItem(L2, 2, 5:1);       /* 2nd item: numeric vector */

L3 = ListCreate({"Hire Date", "Name", "Salary"});  /* named list */
call ListSetItem(L3, "Hire Date", {"01JUL1996"d "15JUN1997"d});
call ListSetItem(L3, "Name",      {"John" "Fred"});
call ListSetItem(L3, "Salary",    {72000  60000});
```

You can also use the list creation operator (`[]`) to create a list. During list creation, you can create named items by using name-value pairs. Names must be prefixed with the `#` symbol. You can specify a name by using a character literal, either with or without quotes. If the name contains a blank or a special character, you must surround the name with quotation marks. For example, the following statements replicate the list-creation statements in the previous example:

```
L1 = [];                            /* L1 is an empty list */
L1 = [ 1:3 ];                       /* L1 has one item */
L2 = [ "A":"Z", 5:1 ];              /* L2 has two items */
/* create a list with three named items */
L3 = [ #"Hire Date"={"01JUL1996"d "15JUN1997"d},
       #Name={"John" "Fred"},
       #Salary={72000  60000} ];
```

LISTDELETEITEM Call

CALL LISTDELETEITEM(*list, position* **);**

The ListDeleteItem subroutine deletes an item from a list. The length of the list is decreased by the number of items that are deleted. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

list specifies an existing list.

position specifies a numeric matrix of indices or a character matrix of item names. Names are not case sensitive.

The following statements show two ways to delete an item:

```
L2 = [ "A":"Z", 5:1 ];              /* L2 has two items */
call ListDeleteItem(L2, 1);         /* delete 1st item */

/* create a list with three named items */
L3 = [ #"Hire Date"={"01JUL1996"d "15JUN1997"d},
       #Name={"John" "Fred"},
       #Salary={72000  60000} ];
call ListDeleteItem(L3, "NAME");    /* delete Name item */
```

LISTDELETENAME Call

CALL LISTDELETENAME(*list, position* **);**

The ListDeleteName subroutine removes the names of one or more items, but the items themselves are not removed. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

list specifies an existing list.

position specifies a numeric matrix of indices or a character matrix of item names. Names are not case sensitive.

The first of the following statements creates a list that has four items. The call to the ListDeleteName subroutine deletes the names for the first and third items. After the deletion, the ListGetName function returns a four-element vector that has one nonmissing name and three missing values (blank strings), as shown in Figure 25.225.

```
L = [ #"Hire Date"={"01JUL1996"d "15JUN1997"d},
      #Name={"John" "Fred"},
      #Salary={72000  60000},
      #"Job Title"={"Manager" "Programmer"} ];
call ListDeleteName(L, {1 3});  /* delete 1st and 3rd names */
names = ListGetName(L);
print names;
```

Figure 25.225 Names of Items in a List

names	
Name	Job Title

LISTGETALLNAMES Function

LISTGETALLNAMES(*list)* **);**

The ListGetAllNames function returns a row vector that contains the names of all named items in a list. The number of columns in the row vector equals the number of named items. If there are no named items, the function returns an empty matrix. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The function takes one argument:

list specifies an existing list.

The first three of the following statements create a list that has two named items. The ListAddItem and ListInsertItem subroutines then add two unnamed items to the list. The ListGetAllNames function returns a row vector that has two elements. As shown in Figure 25.226, the vector contains the names of all named items.

```
L = [ #"Hire Date"={"01JUL1996"d "15JUN1997"d},
      #Salary={72000  60000} ];
call ListAddItem(L, 1:5);                    /* add unnamed item */
call ListInsertItem(L, 2, {"John" "Fred"}); /* insert unnamed item */

allNames = ListGetAllNames(L);
print allNames;
```

Figure 25.226 Names of Items in a List

allNames
Hire Date Salary

Notice that the ListGetAllNames function returns the names of all named items, in contrast to the ListGetName function which returns a blank string for unnamed items. To see the difference between the functions, use the ListGetName function in the preceding program.

LISTGETITEM Function

LISTGETITEM(*list, position* < *flag* > **);**

The ListGetItem function returns the value of a list item. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The function takes the following arguments:

list specifies an existing list.

position specifies a numeric matrix of indices or a character matrix of item names. Names are not case sensitive.

flag controls what happens to the list item after its value is copied. Valid values are:

- 'c' copies the list item, but does not change it. This is the default behavior.
- 'm' moves the list item. The item specified by *position* becomes empty, but the length of the list does not change.
- 'd' deletes the list item. The length of the list decreases by one, and the indices of subsequent items are adjusted. For example, if you delete the fourth item, the fifth item becomes the fourth item, the sixth item becomes the fifth item, and so on.

For example, the following statements create a list that contains four items and assign a matrix to each item. The first item is a 1×1 matrix, the second item is a 2×2 matrix, the third item is a 2×2 matrix, and the fourth item is a 4×4 matrix.

```
L = ListCreate(4);
do n = 1 to 4;
   call ListSetItem(L, n, j(n,n,n)); /* n x n matrix */
end;

x1 = ListGetItem(L, 1);      /* copy value */
x2 = ListGetItem(L, 2, 'd'); /* copy value; delete item */
x3 = ListGetItem(L, 3, 'm'); /* copy value; free item */
```

The first call to the ListGetItem function copies the first item from the list into the **x1** matrix. The list is unchanged.

The second call to the ListGetItem function copies the second item from the list into the **x2** matrix. The 'd' parameter requests that the second list item be deleted. The length of the list decreases by one. After the call returns, the second item is a 3×3 matrix and the third item is a 4×4 matrix.

The third call to the ListGetItem function copies the third item from the list into the **x3** matrix. The 'm' parameter requests that the third list item become an empty matrix, but leaves the length of the list unchanged.

After the program runs, the list contains three items: a 1×1 matrix, a 3×3 matrix, and an empty matrix.

Alternatively, if you want to copy an item from a list, you can use the list item operator (**$**) to extract the item. For example, the following statements set four items of a list and then retrieve the first item:

```
L = ListCreate(4);
do n = 1 to 4;
   L$n = j(n,n,n);    /* n x n matrix */
end;
x1 = L$1;             /* copy value */
```

LISTGETNAME Function

LISTGETNAME(*list* < , *position* > **);**

The ListGetName function returns the names of one or more items in a list. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The function takes the following arguments:

list specifies an existing list.

position specifies a numeric matrix of indices of items in the list.

The function returns a *k*-element character row vector, where *k* is the number of elements in the *position* vector. The character row vector contains a blank string for any specified item that does not have a name.

When called without the *position* argument, the function returns a character vector that contains the names of all items in the list, as shown in the following example:

```
L = [ #"Hire Date"={"01JUL1996"d "15JUN1997"d},   /* named list */
      #Name={"John" "Fred"},
      #Salary={72000  60000} ];
call ListAddItem(L, 1:5);                  /* add unnamed item */
names = ListGetName(L);
print names;
```

Figure 25.227 Names of Items in a List

names
Hire Date Name Salary

LISTGETSUBITEM Function

LISTGETSUBITEM(*list, position* < *, flag* > **);**

The ListGetSubItem function gets the value of an item in a nested sublist. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The function takes the following arguments:

list specifies an existing list.

position specifies a numeric matrix of indices or a character matrix of item names. Names are not case sensitive. You can also specify a list, which enables you to mix names and indices.

flag controls what happens to the list item after its value is copied. Valid values are:

- 'c' copies the list item but does not change it. This is the default behavior.
- 'm' moves the list item. The item specified by *position* becomes empty, but the length of the list does not change.
- 'd' deletes the list item. The length of the list decreases by one, and the indices of subsequent items are adjusted. For example, if you delete the fourth item, the fifth item becomes the fourth item, the sixth item becomes the fifth item, and so on.

Use this call to get the value of a nested sublist of the input list. The values of the position matrix are applied successively in row-major order to drill down into sublists.

For example, the following statements define a list that has three named items, one of which is a sublist. The first item in the list (which has the name "N") is copied into the matrix **M**. The second item in the sublist named "A" is copied into the matrix **M2**. The values of these matrices are shown in Figure 25.228.

```
numMat = 1:4;
charMat = {"x" "y" "z"};
list = [100, "Q"];                      /* simple list */
L = [#"N"=numMat, #"C"=charMat, #"A"=list];  /* multiple types in list  */
M = ListGetSubItem(L, "N");             /* item named "N" */
M2= ListGetSubItem(L, ["A", 2]);        /* 2nd item in sublist */
print M, M2;
```

Figure 25.228 Item of a Sublist

M
1 2 3 4

M2
Q

M
1 2 3 4

M2
Q

Equivalently, you can use the list item operator (**$**) to extract items from a list or from a sublist, as shown in the following:

```
M = L$"N";                          /* item named "N" */
M2= L$"A"$2;                        /* 2nd item in sublist */
M3= L$(["A", 2]);                   /* also 2nd item in sublist */
print M, M2;
```

LISTINDEX Function

LISTINDEX(*list, names* **);**

The ListIndex function returns the positions of the list items that have the specified *names*. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The function takes the following arguments:

list specifies an existing list.

position specifies a character matrix of item names. Names are not case sensitive.

The function returns a row vector that contains the indices that correspond to the specified names. If you specify a name that is not in the list, the ListIndex function returns a missing value, as shown in the following example:

```
L = ListCreate({"Hire Date", "Name", "Salary"});  /* named list */
k = ListIndex(L, {"Salary" "ID" "hire date"});    /* ID not valid name */
print k;
```

Figure 25.229 Indices for Names in a List

k
3 . 1

LISTINSERTITEM Call

CALL LISTINSERTITEM(*list, position, v* **<** *, flag* **>** **);**

The ListInsertItem subroutine adds a new item to a list. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The ListInsertItem subroutine inserts an item into a list. If the list has k items, you can insert a new item in positions $1, 2, \ldots, k+1$. If you insert an item at position i where $i \leq k$, then the existing items at positions $i, i+1, \ldots, k$ are shifted to the right to make room for the new item. In other words, the new item is inserted prior to the existing item in the specified position, and the length of the list increases by one.

The subroutine takes the following arguments:

list specifies an existing list.

position specifies the position where the item is to be inserted. Existing items at or subsequent to that position are shifted to the right. For example, the position 1 indicates that the item is inserted at the beginning of the list.

v specifies a SAS/IML variable or literal that will become a new list item. The value of *v* can be a matrix, table, list, or any other valid SAS/IML type, including an empty matrix.

flag controls what happens to *v*. Valid values are:

- 'c' copies the data in *v* into the list. The *v* symbol is unchanged by the call. This is the default behavior.
- 'm' moves the data in *v* into the list. The *v* symbol is freed after the call.

```
L = ListCreate();
call ListAddItem(L, 1:3);
call ListAddItem(L, {"Cat" "Dog"});
mat = {1 2, 3 4};
call ListInsertItem(L, 2, mat); /* insert 2x2 matrix in 2nd position */
```

The ListInsertItem function is more efficient than list concatenation. Although you can also insert the matrix by using the syntax `L = L[1] || [mat] || L[2]`, that syntax is inefficient and should be avoided.

LISTLEN Function

LISTLEN(*list* **);**

The ListLen function returns the number of items in a list. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following argument:

list specifies an existing list.

The items in a list might be empty, as shown in the following example:

```
L = ListCreate(3);              /* allocate array of 3 items */
numItems = ListLen(L);          /* numItems = 3 */
```

LISTSETITEM Call

CALL LISTSETITEM(*list, position, v* < , *flag* > **);**

The ListSetItem subroutine sets the value of one or more existing list items. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

list specifies an existing list.

position specifies a matrix of integer values or character values that specify the positions of the items to be set. You can also specify a list, which enables you to mix names and indices. Any existing items at the specified locations are overwritten.

v specifies a SAS/IML variable or literal that will become a new list item. The value of *v* can be a matrix, table, list, or any other valid SAS/IML type, including an empty matrix.

flag controls what happens to the *v* symbol. Valid values are:

'c' copies the data in *v* into the list. The *v* symbol is unchanged by the call. This is the default behavior.

'm' moves the data in *v* into the list. The *v* symbol is freed after the call.

The following statements use the ListCreate subroutine to allocate a list that has three items. The ListLen function is used in the DO loop to iterate over every item in the list. The ListSetItem subroutine is used to set the *n*th item to an $n \times n$ matrix.

```
/* create a list of matrices; use ListSetItem to fill */
L = ListCreate(3);             /* allocate array of 3 items */
do n = 1 to ListLen(L);        /* for each item in array */
   A = j(n, n, n-1);           /* define n x n matrix */
   call ListSetItem(L, n, A); /* assign n_th item of L */
end;
```

You can also specify a vector for the *position* argument, in which case the specified value is copied to multiple locations. For example, to set every item of a list to the value 0, use the following statement:

```
call ListSetItem(L, 1:ListLen(L), 0); /* assign all items of L */
```

You can use the list item operator (**$**) to set an item. For example, the following syntax is an alternative way to assign items in a loop:

```
do n = 1 to ListLen(L);        /* for each item in array */
   L$n = j(n, n, n-1);         /* assign n x n matrix */
end;
```

LISTSETNAME Call

CALL LISTSETNAME(*list, position, name* **);**

The ListSetName subroutine sets the name of an item. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

list specifies an existing list.

position specifies a matrix of integer values or character values that specify the items to be assigned names. If the *position* argument specifies an item that already has a name, the item is renamed.

name specifies a character matrix that has the same number of elements as *position*. The *i*th name is assigned to the item in the *i*th position.

The following example sets the names of three items in a list:

```
L = ListCreate();
call ListAddItem(L, {"Red" "Green" "Blue"});  /* 1st item */
call ListAddItem(L, {"Bird" "Cat" "Dog"});    /* 2nd item */
call ListAddItem(L, {"Coffee" "Tea" "Milk"}); /* 3rd item */
call ListSetName(L, 1:3, {"Colors" "Pets" "Beverages"});
```

Equivalently, you can use the list creation operator (`[]`) to create a literal list that contains named items, as follows:

```
L = [#Colors    = {"Red" "Green" "Blue"},
     #Pets      = {"Bird" "Cat" "Dog"},
     #Beverages = {"Coffee" "Tea" "Milk"} ];
```

LISTSETSUBITEM Call

CALL LISTSETSUBITEM(*list, position, v* < *, flag* > **);**

The ListSetSubitem subroutine sets the value of an item in a nested sublist. The values of the *position* matrix are applied successively in row-major order to drill down into sublists. For examples and a general discussion of using lists, see Chapter 10, "Lists and Data Structures."

The subroutine takes the following arguments:

list specifies an existing list.

position specifies a numeric matrix of indices or a character matrix of item names. Names are not case-sensitive.

v specifies a SAS/IML variable or literal that will become a new list item. The value of *v* can be a matrix, table, list, or any other valid SAS/IML type, including an empty matrix.

flag controls what happens to the sublist item after its value is copied. Valid values are:

- 'c' copies the *v* value but does not change it. This is the default behavior.
- 'm' copies the *v* value to the list item and frees the *v* symbol.

Use this call to set the value of a nested sublist.

```
call ListSetSubItem(L, {"12" "M"}, {"Joe", "Rick", "Simon"});
```

For example, the following statements define a list that has three items, one of which is a sublist. The call to the ListSetSubItem subroutine changes the value of the second item in the sublist.

```
numMat = 1:4;
charMat = {"x" "y" "z"};
list = [100, "Q"];                      /* simple list */
L = [#"N"=numMat, #"C"=charMat, #"A"=list];  /* multiple types in list  */
call ListSetSubItem(L, {3 2}, "R");     /* change 2nd item in sublist */
call ListSetSubItem(L, ["A", 2], "S"); /* also change 2nd item in sublist */
```

Equivalently, you can use the list item operator (**$**) to set items in a sublist. You can use multiple operators to access sublists, as in **L$"A"$2**. You can also use a matrix or list to specify the indices or names that determine the item, as shown in the following:

```
L$"A"$2 = "R";                          /* change 2nd item in sublist */
L$(["A", 2]) = "S";                     /* also change 2nd item in sublist */
L${3 2} = "T";                          /* third way */
```

LMS Call

CALL LMS(*sc, coef, wgt, opt, y* < , *x* > < , *sorb* > **);**

The LMS subroutine performs least median of squares (LMS) robust regression (sometimes called *resistant* regression) by minimizing the *h*th-ordered squared residual. The subroutine is able to detect outliers and perform a least squares regression on the remaining observations.

The algorithm used in the LMS subroutine is based on the PROGRESS program of Rousseeuw and Hubert (1996), which is an updated version of Rousseeuw and Leroy (1987). In the special case of regression through the origin with a single regressor, Barreto and Maharry (2006) show that the PROGRESS algorithm does not, in general, find the slope that yields the least median of squares. Starting with SAS/IML 9.2, the LMS subroutine uses the algorithm of Barreto and Maharry (2006) to obtain the correct LMS slope in the case of regression through the origin with a single regressor. In this case, input arguments that are specific to the PROGRESS algorithm are ignored and output specific to the PROGRESS algorithm is suppressed.

The value of *h* can be specified, but in most applications the default value works well and the results seem to be quite stable toward different choices of *h*.

In the following discussion, *N* is the number of observations and *n* is the number of regressors. The input arguments to the LMS subroutine are as follows:

opt specifies an options vector. The options vector can be a vector of missing values, which results in default values for all options. The components of *opt* are as follows:

- *opt[1]* specifies whether an intercept is used in the model (*opt[1]*=0) or not (*opt[1]*≠ 0). If *opt[1]*=0, then a column of ones is added as the last column to the input matrix **X**; that is, you do not need to add this column of ones yourself. The default is *opt[1]*=0.
- *opt[2]* specifies the amount of printed output. Higher values request additional output and include the output of lower values.

0 prints no output except error messages.

1 prints all output except (1) arrays of $O(N)$, such as weights, residuals, and diagnostics; (2) the history of the optimization process; and (3) subsets that result in singular linear systems.

2 additionally prints arrays of $O(N)$, such as weights, residuals, and diagnostics; also prints the case numbers of the observations in the best subset and some basic history of the optimization process.

3 additionally prints subsets that result in singular linear systems.

The default is *opt[2]*=0.

opt[3] specifies whether only LMS is computed or whether, additionally, least squares (LS) and weighted least squares (WLS) regression are computed.

0 computes only LMS.

1 computes, in addition to LMS, weighted least squares regression on the observations with *small* LMS residuals (where *small* is defined by opt[8]).

2 computes, in addition to LMS, unweighted least squares regression.

3 adds both unweighted and weighted least squares regression to LMS regression.

The default is *opt[3]*=0.

opt[4] specifies the quantile h to be minimized. This is used in the objective function. The default is *opt[4]*$= h = \left[\frac{N+n+1}{2}\right]$, which corresponds to the highest possible breakdown value. This is also the default of the PROGRESS program. The value of h should be in the range $\frac{N}{2} + 1 \leq h \leq \frac{3N}{4} + \frac{n+1}{4}$

opt[5] specifies the number N_{Rep} of generated subsets. Each subset consists of n observations $(k_1, \ldots, k_n)$, where $1 \leq k_i \leq N$. The total number of subsets that contain n observations out of N observations is

$$N_{\text{tot}} = \binom{N}{n} = \frac{\prod_{j=1}^{n}(N - j + 1)}{\prod_{j=1}^{n} j}$$

where n is the number of parameters including the intercept.

Due to computer time restrictions, not all subset combinations of n observations out of N can be inspected for larger values of N and n. Specifying a value of $N_{\text{Rep}} < N_{\text{tot}}$ enables you to save computer time at the expense of computing a suboptimal solution.

If *opt[5]* is zero or missing, the default number of subsets is taken from the following table.

n	1	2	3	4	5	6	7	8	9	10
N_{lower}	500	50	22	17	15	14	0	0	0	0
N_{upper}	10^6	1414	182	71	43	32	27	24	23	22
N_{Rep}	500	1000	1500	2000	2500	3000	3000	3000	3000	3000

n	11	12	13	14	15
N_{lower}	0	0	0	0	0
N_{upper}	22	22	22	23	23
N_{Rep}	3000	3000	3000	3000	3000

If the number of cases (observations) N is smaller than N_{lower}, then all possible subsets are used; otherwise, N_{Rep} subsets are chosen randomly. This means that an exhaustive search is performed for *opt[5]*=−1. If N is larger than N_{upper}, a note is printed in the log file that indicates how many subsets exist.

opt[6] is not used.

opt[7] specifies whether the last argument *sorb* contains a given parameter vector $\mathbf{b}$ or a given subset for which the objective function should be evaluated.

0 *sorb* contains a given subset index.

1 *sorb* contains a given parameter vector $\mathbf{b}$.

The default is *opt[7]*=0.

opt[8] is relevant only for LS and WLS regression (*opt[3]* > 0). It specifies whether the covariance matrix of parameter estimates and approximate standard errors (ASEs) are computed and printed.

0 does not compute covariance matrix and ASEs.

1 computes covariance matrix and ASEs but prints neither of them.

2 computes the covariance matrix and ASEs but prints only the ASEs.

3 computes and prints both the covariance matrix and the ASEs.

The default is *opt[8]*=0.

y refers to an N response vector.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors. If opt[1] is zero or missing, an intercept $\mathbf{x}_{n+1} \equiv 1$ is added by default as the last column of $\mathbf{X}$. If the matrix $\mathbf{X}$ is not specified, $\mathbf{y}$ is analyzed as a univariate data set.

sorb refers to an n vector that contains either of the following:

- n observation numbers of a subset for which the objective function should be evaluated; this subset can be the start for a pairwise exchange algorithm if *opt[7]* is specified.
- n given parameters $\mathbf{b} = (b_1, \ldots, b_n)$ (including the intercept, if necessary) for which the objective function should be evaluated.

Missing values are not permitted in *x* or *y*. Missing values in *opt* cause the default value to be used.

The LMS subroutine returns the following values:

sc is a column vector that contains the following scalar information, where rows 1–9 correspond to LMS regression and rows 11–14 correspond to either LS or WLS:

sc[1] the quantile h used in the objective function

sc[2] number of subsets generated

sc[3] number of subsets with singular linear systems

sc[4] number of nonzero weights w_i

sc[5] lowest value of the objective function F_{LMS} attained

sc[6] preliminary LMS scale estimate S_P

sc[7] final LMS scale estimate S_F

sc[8] robust R square (*coefficient of determination*)

sc[9] asymptotic consistency factor

If *opt[3]* > 0, then the following are also set:

sc[11] LS or WLS objective function (sum of squared residuals)

sc[12] LS or WLS scale estimate

sc[13] R square value for LS or WLS

sc[14] F value for LS or WLS

For *opt[3]*=1 or *opt[3]*=3, these rows correspond to WLS estimates; for *opt[3]*=2, these rows correspond to LS estimates.

coef is a matrix with n columns that contains the following results in its rows:

coef[1,] LMS parameter estimates

coef[2,] indices of observations in the best subset

If *opt[3]* > 0, then the following are also set:

coef[3,] LS or WLS parameter estimates

coef[4,] approximate standard errors of LS or WLS estimates

coef[5,] t values

coef[6,] p-values

coef[7,] lower boundary of Wald confidence intervals

coef[8,] upper boundary of Wald confidence intervals

For *opt[3]*=1 or *opt[3]*=3, these rows correspond to WLS estimates; for *opt[3]*=2, these rows correspond to LS estimates.

wgt is a matrix with N columns that contains the following results in its rows:

wgt[1,] weights (1 for small residuals; 0 for large residuals)

wgt[2,] residuals $r_i = y_i - \mathbf{x}_i\mathbf{b}$

wgt[3,] resistant diagnostic u_i (the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider results for Brownlee (1965) stackloss data. The three explanatory variables correspond to measurements for a plant that oxidizes ammonia to nitric acid on 21 consecutive days.

- x_1 air flow to the plant
- x_2 cooling water inlet temperature
- x_3 acid concentration

The response variable y_i contains the permillage of ammonia lost (stackloss). The data are also given by Rousseeuw and Leroy (1987) and Osborne (1985). Rousseeuw and Leroy (1987) cite a large number of papers where this data set was analyzed and state that most researchers "concluded that observations 1, 3, 4, and 21 were outliers," and that some people also reported observation 2 as an outlier.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5,985 different subsets of 4 observations out of 21. If you decide not to specify `opt[5]`, the LMS subroutine chooses N_{rep} =2,000 random sample subsets. Since there is a large number of subsets with singular linear systems, which you do not want to print, choose `opt[2]=2` for reduced printed output.

```
   /* X1  X2  X3   Y  Stackloss data */
aa = { 1  80  27  89  42,
       1  80  27  88  37,
       1  75  25  90  37,
       1  62  24  87  28,
       1  62  22  87  18,
       1  62  23  87  18,
       1  62  24  93  19,
       1  62  24  93  20,
       1  58  23  87  15,
       1  58  18  80  14,
       1  58  18  89  14,
       1  58  17  88  13,
       1  58  18  82  11,
       1  58  19  93  12,
       1  50  18  89   8,
       1  50  18  86   7,
       1  50  19  72   8,
       1  50  19  79   8,
       1  50  20  80   9,
       1  56  20  82  15,
       1  70  20  91  15 };

a = aa[, 2:4]; b = aa[, 5];
opt = j(8, 1, .);
opt[2]= 2;    /* ipri */
opt[3]= 3;    /* ilsq */
opt[8]= 3;    /* icov */

call lms(sc, coef, wgt, opt, b, a);
```

The first portion of the output displays descriptive statistics, as shown in Figure 25.230:

Figure 25.230 Descriptive Statistics

LMS: The 13th ordered squared residual will be minimized.

Median and Mean		
	Median	Mean
VAR1	58	60.428571429
VAR2	20	21.095238095
VAR3	87	86.285714286
Intercep	1	1
Response	15	17.523809524

Dispersion and Standard Deviation		
	Dispersion	StdDev
VAR1	5.930408874	9.1682682584
VAR2	2.965204437	3.160771455
VAR3	4.4478066555	5.3585712381
Intercep	0	0
Response	5.930408874	10.171622524

The next portion of the output shows the least squares estimates and the covariance of the estimates. Information about the residuals are also displayed, but are not shown in Figure 25.231.

Figure 25.231 Least Squares Estimates

Unweighted Least-Squares Estimation

LS Parameter Estimates						
Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.7156402	0.13485819	5.31	<.0001	0.45132301	0.97995739
VAR2	1.29528612	0.36802427	3.52	0.0026	0.57397182	2.01660043
VAR3	-0.1521225	0.15629404	-0.97	0.3440	-0.4584532	0.15420818
Intercep	-39.919674	11.8959969	-3.36	0.0038	-63.2354	-16.603949

Sum of Squares = 178.8299616

Degrees of Freedom = 17

LS Scale Estimate = 3.2433639182

Cov Matrix of Parameter Estimates				
	VAR1	VAR2	VAR3	Intercep
VAR1	0.0181867302	-0.036510675	-0.007143521	0.2875871057
VAR2	-0.036510675	0.1354418598	0.0000104768	-0.651794369
VAR3	-0.007143521	0.0000104768	0.024427828	-1.676320797
Intercep	0.2875871057	-0.651794369	-1.676320797	141.51474107

R-squared = 0.9135769045

F(3,17) Statistic = 59.9022259

Probability = 3.0163272E-9

The LMS subroutine prints results for the 2,000 random subsets. Figure 25.232 shows the iteration history, the best subset of observations that are used to form estimates, and the estimated parameters. The subroutine also displays residual information (not shown).

Figure 25.232 Least Median Squares Estimates

There are 5985 subsets of 4 cases out of 21 cases.

The algorithm will draw 2000 random subsets of 4 cases.

Random Subsampling for LMS

Subset	Singular	Best Criterion	Percent
500	23	0.163262	25
1000	55	0.140519	50
1500	79	0.140519	75
2000	103	0.126467	100

Minimum Criterion= 0.1264668282

Least Median of Squares (LMS) Method

Minimizing 13th Ordered Squared Residual.

Highest Possible Breakdown Value = 42.86 %

Random Selection of 2103 Subsets

Among 2103 subsets 103 is/are singular.

Observations of Best Subset			
15	11	19	10

Estimated Coefficients			
VAR1	VAR2	VAR3	Intercep
0.75	0.5	0	-39.25

Observations 1, 3, 4, and 21 have scaled residuals larger than 2.0 (table not shown) and are considered outliers. The corresponding WLS estimates are shown in Figure 25.233:

Figure 25.233 Weighted Least Squares Estimates

LMS Objective Function = 0.75

Preliminary LMS Scale = 1.0478510755

Robust R Squared = 0.96484375

Final LMS Scale = 1.2076147288

LMS Residuals

N	Observed	Estimated	Residual	Res / S
1	42.000000	34.250000	7.750000	6.417610
2	37.000000	34.250000	2.750000	2.277216
3	37.000000	29.500000	7.500000	6.210590
4	28.000000	19.250000	8.750000	7.245688
5	18.000000	18.250000	-0.250000	-0.207020
6	18.000000	18.750000	-0.750000	-0.621059
7	19.000000	19.250000	-0.250000	-0.207020
8	20.000000	19.250000	0.750000	0.621059
9	15.000000	15.750000	-0.750000	-0.621059
10	14.000000	13.250000	0.750000	0.621059
11	14.000000	13.250000	0.750000	0.621059
12	13.000000	12.750000	0.250000	0.207020
13	11.000000	13.250000	-2.250000	-1.863177
14	12.000000	13.750000	-1.750000	-1.449138
15	8.000000	7.250000	0.750000	0.621059
16	7.000000	7.250000	-0.250000	-0.207020
17	8.000000	7.750000	0.250000	0.207020
18	8.000000	7.750000	0.250000	0.207020
19	9.000000	8.250000	0.750000	0.621059
20	15.000000	12.750000	2.250000	1.863177
21	15.000000	23.250000	-8.250000	-6.831649

Distribution of Residuals

MinRes	1st Qu.	Median	Mean	3rd Qu.	MaxRes
-8.25	-0.5	0.25	0.9047619048	0.75	8.75

Figure 25.233 *continued*

Resistant Diagnostic		
N	U	Resistant Diagnostic
1	10.448052	2.278040
2	7.931751	1.729399
3	10.000000	2.180349
4	11.666667	2.543741
5	2.729730	0.595176
6	3.486486	0.760176
7	4.729730	1.031246
8	4.243243	0.925175
9	3.648649	0.795533
10	3.759835	0.819775
11	4.605767	1.004218
12	4.925169	1.073859
13	3.888889	0.847914
14	4.586421	1.000000
15	5.297030	1.154938
16	4.009901	0.874299
17	6.679576	1.456381
18	4.305340	0.938715
19	4.019976	0.876495
20	3.000000	0.654105
21	11.000000	2.398384

Median(U)= 4.5864208797

Weighted Least-Squares Estimation

RLS Parameter Estimates Based on LMS						
Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.79768556	0.06743906	11.83	<.0001	0.66550742	0.9298637
VAR2	0.57734046	0.16596894	3.48	0.0041	0.25204731	0.9026336
VAR3	-0.0670602	0.06160314	-1.09	0.2961	-0.1878001	0.05367975
Intercep	-37.652459	4.73205086	-7.96	<.0001	-46.927108	-28.37781

Weighted Sum of Squares = 20.400800254

Degrees of Freedom = 13

RLS Scale Estimate = 1.2527139846

Cov Matrix of Parameter Estimates				
	VAR1	VAR2	VAR3	Intercep
VAR1	0.0045480273	-0.007921409	-0.001198689	0.0015681747
VAR2	-0.007921409	0.0275456893	-0.00046339	-0.065017508
VAR3	-0.001198689	-0.00046339	0.0037949466	-0.246102248
Intercep	0.0015681747	-0.065017508	-0.246102248	22.392305355

Figure 25.233 *continued*

Weighted R-squared = 0.9750062263

F(3,13) Statistic = 169.04317954

Probability = 1.158521E-10

There are 17 points with nonzero weight.

Average Weight = 0.8095238095

LOAD Statement

LOAD < **MODULE**=*(module-list)* > < *matrix-list* > **;**

The LOAD statement loads modules and matrix values from the current library storage into the current workspace.

The arguments to the LOAD statement are as follows:

module-list	is a list of modules.
matrix-list	is a list of matrices.

For example, to load three modules A, B, and C and one matrix X, use the following statement:

```
load module=(A B C) X;
```

The special operand _ALL_ can be used to load all matrices or all modules. For example, if you want to load all matrices, use the following statement:

```
load _all_;
```

If you want to load all modules, use the following statement:

```
load module=_all_;
```

To load all matrices and modules stored in the library storage, you can enter the LOAD command without any arguments, as follows:

```
load;
```

The storage library can be specified by using a RESET STORAGE command. The default library is Work.Imlstor. For more information, see Chapter 19 and the descriptions of the STORE, REMOVE, RESET, and SHOW statements.

LOC Function

LOC(*matrix***);**

The LOC function finds nonzero elements of a matrix. It creates a $1 \times n$ row vector, where n is the number of nonzero elements in the argument matrix. Missing values are treated as zeros. The values in the resulting row vector are the locations of the nonzero elements in the argument (in row-major order).

For example, consider the following statements:

```
a = {1 0 2 3 0};
b = loc(a);
print b;
```

Because the first, third, and fourth elements of **a** are nonzero, these statements result in the row vector shown in Figure 25.234:

Figure 25.234 Location of Nonzero Elements

b
1 3 4

If every element of the argument vector is 0, the result is empty; that is, **b** has zero rows and zero columns.

The LOC function is useful for subscripting parts of a matrix that satisfy some condition. For example, the following statements create a matrix **y** that contains the rows of **x** that have a positive element in the diagonal of **x**:

```
x = {1  1  0,
     0 -2  2,
     0  0  3};
y = x[loc(vecdiag(x)>0), ];
print y;
```

Figure 25.235 Rows with Positive Diagonal Elements

y
1 1 0
0 0 3

LOG Function

LOG(*matrix***);**

The LOG function is the scalar function that takes the natural logarithm of each element of the argument matrix. An example of a valid statement follows:

```
c = {1 2 3};
b = log(c);
print b;
```

Figure 25.236 Natural Logarithms

b		
0	0.6931472	1.0986123

LOGABSDET Function

LOGABSDET(*matrix***);**

The LOGABSDET function computes the natural logarithm of the absolute value of the determinant of a matrix, along with the sign of the determinant. The logarithm value is returned as the first element of the returned matrix, and the sign of the determinant is returned as the second element. The value −1 signifies a negative determinant, +1 signifies a positive determinant, and 0 signifies a zero determinant. If the determinant is 0, a missing value is returned in the first element for the logarithm value. This function works even if the value of the determinant is greater than the maximum value possible on the computer.

The following example computes the value of the log of the absolute value of the determinant and then checks it against the determinant value from the DET function:

```
z = {1 2 3,
     4 9 6,
     7 8 9};
det1 = det(z);
x = logabsdet(z);
print z;
print "Log of the absolute value of det(z) " (x[1]);
print "sign of det(z) " (x[2]);

/* use the choose() function to help convert x to det(z) */

det2 = choose(x[2],exp(x[1]) * x[2], 0);

print "det(z) = " det1 "determinant from LogAbsDet(z) = " det2;
```

Figure 25.237 Example LogAbsDet Function Call

z		
1	2	3
4	9	6
7	8	9

Log of the absolute value of det(z)	3.871201

sign of det(z)	-1

	det1		det2
det(z) =	-48	determinant from LogAbsDet(z) =	-48

LP Call

CALL LP(*rc, x, dual, a, b* **<** *, cntl* **>** **<** *, u* **>** **<** *, l* **>** **<** *, basis* **>** **);**

The LP subroutine is a legacy subroutine that solves a linear programming problem. Although the LP subroutine continues to be supported, the LPSOLVE subroutine, which was introduced in SAS/IML 13.1,

is more efficient and provides an input format that is easier to use. SAS/IML programmers should use the LPSOLVE subroutine.

You can find the documentation for the LP subroutine in earlier releases of the *SAS/IML User's Guide*.

LPSOLVE Call

CALL LPSOLVE(*rc, objvalue, x, dual, reducost, c, a, b* < *, cntl*> < *, rowsense*> < *, range*> < *, l*> < *, u*> **);**

The LPSOLVE subroutine solves a linear programming problem. It uses a different input format and solver options from the LP call and is the preferred method for solving linear programming problems.

The input arguments to the LPSOLVE subroutine are as follows:

c is a vector of dimension n of objective function coefficients. A missing value is treated as 0.

a is an $m \times n$ matrix of the technological coefficients. A missing value is treated as 0.

b is a vector of dimension m of constraints' right-hand sides (RHS). For a range constraint, *b* is its constraint upper bound. A missing value is treated as 0.

cntl is an optional vector that contains one to eight elements that represent the LPSOLVE subroutine's control options. The default value is used if an option is not specified or its value is a missing value. If *cntl*=(*objsense, printlevel, maxtime, maxiter, presolve, algorithm,scaling, tol*), then

- *objsense* specifies whether it is a minimization or a maximization problem, where 1 specifies minimization and −1 specifies maximization. The default value is 1.
- *printlevel* specifies the type of messages printed to the log. A value of 0 prints warning and error messages only, whereas 1 prints solution information in addition to warning and error messages. The default value is 0.
- *maxtime* specifies an upper bound of running time in seconds. The default value is effectively unbounded.
- *maxiter* specifies the maximum number of iterations to be processed. The default value is effectively unbounded.
- *presolve* specifies the presolve option, where 0 indicates no presolve and 1 indicates an automatic presolve option.The default value is 1.
- *algorithm* specifies the type of solver, where 1 specifies primal simplex, 2 specifies dual simplex, and 3 specifies interior point algorithm. The default value is 2.
- *scaling* specifies whether to scale the problem matrix, where 0 turns off scaling and 1 turns on scaling. The default value is 1.
- *tol* specifies a feasibility and optimality tolerance. The default value is 10^{-6}.

rowsense is an optional row vector of dimension m that specifies the sense of each constraint. The values can be E, L, G, or R for equal, less than or equal to, greater than or equal to, or range constraint. If this vector is missing, the solver treats the constraints as E type constraints.

range is an optional row vector of dimension *m* that specifies the range of the constraints. The row sense for a range constraint is R. For the non-range constraints, the corresponding values are ignored. For a range constraint, the range value is the difference between its constraint lower bound and its constraint upper bound *b*, so it must be nonnegative.

l is an optional column vector of dimension *n* that specifies lower bounds on the decision variables. If you do not specify *l* or *l[j]* has a missing value, then the lower bound of variable *j* is assumed to be 0.

u is an optional column vector of dimension *n* that specifies upper bounds on the decision variables. If you do not specify *u* or *u[j]* has a missing value, the upper bound of variable *j* is assumed to be infinity.

The LPSOLVE subroutine returns the following values:

rc returns one of the following scalar return codes:

rc	**Termination Reason**
0	The solution is optimal.
1	The time limit was exceeded.
2	The maximum number of iterations was exceeded.
3	The solution is infeasible.
4	The solution is unbounded or infeasible.
5	The subroutine could not obtain enough memory.
6	The subroutine failed to solve the problem.

objvalue returns the optimal or final objective value at termination.

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*.

reducost returns reduced cost in a column vector of length *n*.

The LPSOLVE subroutine solves linear programs. A standard linear program has the following formulation:

$$\min c^T \mathbf{x}$$
$$\text{subject to } \mathbf{A}\mathbf{x}\{\geq, =, \leq\}\mathbf{b}$$
$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$$

If only c, $\mathbf{A}$, and $\mathbf{b}$ are present, then LPSOLVE solves the following linear programming problem by default:

$$\min c^T \mathbf{x}$$
$$\text{subject to } \mathbf{A}\mathbf{x} = \mathbf{b}$$
$$0 \leq \mathbf{x}$$

The primal and dual simplex solvers implement the two-phase simplex method. In phase I, the solver tries to find a feasible solution. If it does not find a feasible solution the LP is infeasible; otherwise, the solver

enters phase II to solve the original LP. The interior point solver implements a primal-dual predictor-corrector interior point algorithm.

Consider the following example:

$$\max(X_1 + X_2)$$
$$\text{subject to } 2X_1 + 0.5X_2 - X_3 \leq 1$$
$$0.2X_1 + 5X_2 - X_4 \leq 1$$
$$0 \leq X_i \leq 9 \text{ for i} = 1, 2, 3, 4$$

The problem is solved by using the following statements:

```
object = {  1   1   0   0 };
coef   = {  2  .5  -1   0,
           .2   5   0  -1};
b =      { 1,   1 };
l = { 0 0 0 0 };
u = { 9 9 9 9 };
rowsense = {'L','L'};
cntl= -1;
call lpsolve (rc,objv,x,dual,rd,object,coef,b,cntl,rowsense,,l,u);
print objv, x, dual, rd;
```

Figure 25.238 Example LPSOLVE Call

objv
6.3636364

x
4.5454545
1.8181818
9
9

dual	
0.4848485	0.1515152

rd
0
0
0.4848485
0.1515152

LTS Call

CALL LTS(*sc, coef, wgt, opt, y* < *, x* > < *, sorb* > **);**

The LTS subroutine performs least trimmed squares (LTS) robust regression by minimizing the sum of the *h* smallest squared residuals. The subroutine also detects outliers and perform a least squares regression on the

remaining observations. The LTS subroutine implements the FAST-LTS algorithm described by Rousseeuw and Van Driessen (1998).

The value of h can be specified, but for many applications the default value works well and the results seem to be quite stable toward different choices of h.

In the following discussion, N is the number of observations and n is the number of regressors. The input arguments to the LTS subroutine are as follows:

opt specifies an options vector. The options vector can be a vector of missing values, which results in default values for all options. The components of *opt* are as follows:

opt[1] specifies whether an intercept is used in the model (*opt[1]*=0) or not (*opt[1]*$\neq$ 0). If *opt[1]*=0, then a column of ones is added as the last column to the input matrix $\mathbf{X}$; that is, you do not need to add this column of ones yourself. The default is *opt[1]*=0.

opt[2] specifies the amount of printed output. Higher values request additional output and include the output of lower values.

- 0 prints no output except error messages.
- 1 prints all output except (1) arrays of $O(N)$, such as weights, residuals, and diagnostics; (2) the history of the optimization process; and (3) subsets that result in singular linear systems.
- 2 additionally prints arrays of $O(N)$, such as weights, residuals, and diagnostics; it also prints the case numbers of the observations in the best subset and some basic history of the optimization process.
- 3 additionally prints subsets that result in singular linear systems.

The default is *opt[2]*=0.

opt[3] specifies whether only LTS is computed or whether, additionally, least squares (LS) and weighted least squares (WLS) regression are computed:

- 0 computes only LTS.
- 1 computes, in addition to LTS, weighted least squares regression on the observations with *small* LTS residuals (where *small* is defined by opt[8]).
- 2 computes, in addition to LTS, unweighted least squares regression.
- 3 adds both unweighted and weighted least squares regression to LTS regression.

The default is *opt[3]*=0.

opt[4] specifies the quantile h to be minimized. This is used in the objective function. The default is *opt[4]*$= h = \left[\frac{N+n+1}{2}\right]$, which corresponds to the highest possible breakdown value. This is also the default of the PROGRESS program. The value of h should be in the range $\frac{N}{2} + 1 \leq h \leq \frac{3N}{4} + \frac{n+1}{4}$.

opt[5] specifies the number N_{Rep} of generated subsets. Each subset consists of n observations $(k_1, \ldots, k_n)$, where $1 \leq k_i \leq N$. The total number of subsets that contain n observations out of N observations is

$$N_{\text{tot}} = \binom{N}{n} = \frac{\prod_{j=1}^{n}(N - j + 1)}{\prod_{j=1}^{n} j}$$

where n is the number of parameters including the intercept.

Due to computer time restrictions, not all subset combinations of n observations out of N can be inspected for larger values of N and n. Specifying a value of $N_{\text{Rep}} < N_{\text{tot}}$ enables you to save computer time at the expense of computing a suboptimal solution.

When *opt[5]* is zero or missing:

- If $N > 600$, the default FAST-LTS algorithm constructs up to five disjoint random subsets with sizes as equal as possible, but not to exceed 300. Inside each subset, the algorithm chooses $500/5 = 100$ subset combinations of n observations.
  The number of subsets is taken from the following table:

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
N_{lower}	500	50	22	17	15	14	0	0	0	0
N_{upper}	10^6	1414	182	71	43	32	27	24	23	22
N_{Rep}	500	1000	1500	2000	2500	3000	3000	3000	3000	3000

n	**11**	**12**	**13**	**14**	**15**
N_{lower}	0	0	0	0	0
N_{upper}	22	22	22	23	23
N_{Rep}	3000	3000	3000	3000	3000

- If the number of cases (observations) N is smaller than N_{lower}, then all possible subsets are used; otherwise, fixed 500 subsets for FAST-LTS or N_{Rep} subsets for algorithm before SAS/IML 8.1 are chosen randomly. This means that an exhaustive search is performed for *opt[5]*=−1. If N is larger than N_{upper}, a note is printed in the log file that indicates how many subsets exist.

opt[6] is not used.

opt[7] specifies whether the last argument *sorb* contains a given parameter vector $\mathbf{b}$ or a given subset for which the objective function should be evaluated.

0 *sorb* contains a given subset index.

1 *sorb* contains a given parameter vector $\mathbf{b}$.

The default is *opt[7]*=0.

opt[8] is relevant only for LS and WLS regression (*opt[3]* > 0). It specifies whether the covariance matrix of parameter estimates and approximate standard errors (ASEs) are computed and printed.

0 does not compute covariance matrix and ASEs.

1 computes covariance matrix and ASEs but prints neither of them.

2 computes the covariance matrix and ASEs but prints only the ASEs.

3 computes and prints both the covariance matrix and the ASEs.

The default is *opt[8]*=0.

opt[9] is relevant only for LTS. If *opt[9]*=0, the algorithm FAST-LTS of Rousseeuw and Van Driessen (1998) is used. If *opt[9]* = 1, the algorithm of Rousseeuw and Leroy (1987) is used. The default is *opt[9]*=0.

y a response vector with N observations.

x an $N \times n$ matrix $\mathbf{X}$ of regressors. If opt[1] is zero or missing, an intercept $\mathbf{x}_{n+1} \equiv 1$ is added by default as the last column of $\mathbf{X}$. If the matrix $\mathbf{X}$ is not specified, $\mathbf{y}$ is analyzed as a univariate data set.

sorb refers to an n vector that contains either of the following:

- n observation numbers of a subset for which the objective function should be evaluated; this subset can be the start for a pairwise exchange algorithm if *opt[7]* is specified.
- n given parameters $\mathbf{b} = (b_1, \ldots, b_n)$ (including the intercept, if necessary) for which the objective function should be evaluated.

Missing values are not permitted in x or y. Missing values in *opt* cause the default value to be used.

The LTS subroutine returns the following values:

sc is a column vector that contains the following scalar information, where rows 1–9 correspond to LTS regression and rows 11–14 correspond to either LS or WLS:

sc[1] the quantile h used in the objective function

sc[2] number of subsets generated

sc[3] number of subsets with singular linear systems

sc[4] number of nonzero weights w_i

sc[5] lowest value of the objective function F_{LTS} attained

sc[6] preliminary LTS scale estimate S_P

sc[7] final LTS scale estimate S_F

sc[8] robust R square (*coefficient of determination*)

sc[9] asymptotic consistency factor

If *opt[3]* > 0, then the following are also set:

sc[11] LS or WLS objective function (sum of squared residuals)

sc[12] LS or WLS scale estimate

sc[13] R square value for LS or WLS

sc[14] F value for LS or WLS

For *opt[3]*=1 or *opt[3]*=3, these rows correspond to WLS estimates; for *opt[3]*=2, these rows correspond to LS estimates.

coef is a matrix with n columns that contains the following results in its rows:

coef[1,] LTS parameter estimates

coef[2,] indices of observations in the best subset

If *opt[3]* > 0, then the following are also set:

coef[3,] LS or WLS parameter estimates

coef[4,] approximate standard errors of LS or WLS estimates

coef[5,] *t*values

coef[6,] *p*-values

coef[7,] lower boundary of Wald confidence intervals

coef[8,] upper boundary of Wald confidence intervals

For *opt[3]*=1 or *opt[3]*=3, these rows correspond to WLS estimates; for *opt[3]*=2, these rows correspond to LS estimates.

wgt is a matrix with N columns that contains the following results in its rows:

wgt[1,] weights (1 for small residuals; 0 for large residuals)

wgt[2,] residuals $r_i = y_i - \mathbf{x}_i\mathbf{b}$

wgt[3,] resistant diagnostic u_i (the resistant diagnostic cannot be computed for a perfect fit when the objective function is zero or nearly zero)

Example

Consider Brownlee (1965) stackloss data used in the example for the LMS subroutine.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5,985 different subsets of 4 observations out of 21. If you decide not to specify `opt[5]`, the FAST-LTS algorithm chooses 500 random sample subsets, as in the following statements:

```
    /* X1  X2  X3   Y  Stackloss data */
aa = { 1  80  27  89  42,
       1  80  27  88  37,
       1  75  25  90  37,
       1  62  24  87  28,
       1  62  22  87  18,
       1  62  23  87  18,
       1  62  24  93  19,
       1  62  24  93  20,
       1  58  23  87  15,
       1  58  18  80  14,
       1  58  18  89  14,
       1  58  17  88  13,
       1  58  18  82  11,
       1  58  19  93  12,
       1  50  18  89   8,
       1  50  18  86   7,
       1  50  19  72   8,
       1  50  19  79   8,
       1  50  20  80   9,
       1  56  20  82  15,
       1  70  20  91  15 };

a = aa[, 2:4]; b = aa[, 5];
opt = j(8, 1, .);
```

```
opt[2]= 1;     /* ipri */
opt[3]= 3;     /* ilsq */
opt[8]= 3;     /* icov */

call lts(sc, coef, wgt, opt, b, a);
```

Figure 25.239 Least Trimmed Squares

LTS: The sum of the 13 smallest squared residuals will be minimized.

Median and Mean

	Median	Mean
VAR1	58	60.428571429
VAR2	20	21.095238095
VAR3	87	86.285714286
Intercep	1	1
Response	15	17.523809524

Dispersion and Standard Deviation

	Dispersion	StdDev
VAR1	5.930408874	9.1682682584
VAR2	2.965204437	3.160771455
VAR3	4.4478066555	5.3585712381
Intercep	0	0
Response	5.930408874	10.171622524

Unweighted Least-Squares Estimation

LS Parameter Estimates

Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.7156402	0.13485819	5.31	<.0001	0.45132301	0.97995739
VAR2	1.29528612	0.36802427	3.52	0.0026	0.57397182	2.01660043
VAR3	-0.1521225	0.15629404	-0.97	0.3440	-0.4584532	0.15420818
Intercep	-39.919674	11.8959969	-3.36	0.0038	-63.2354	-16.603949

Sum of Squares = 178.8299616

Degrees of Freedom = 17

LS Scale Estimate = 3.2433639182

Cov Matrix of Parameter Estimates

	VAR1	VAR2	VAR3	Intercep
VAR1	0.0181867302	-0.036510675	-0.007143521	0.2875871057
VAR2	-0.036510675	0.1354418598	0.0000104768	-0.651794369
VAR3	-0.007143521	0.0000104768	0.024427828	-1.676320797
Intercep	0.2875871057	-0.651794369	-1.676320797	141.51474107

Figure 25.239 *continued*

R-squared = 0.9135769045

F(3,17) Statistic = 59.9022259

Probability = 3.0163272E-9

Least Trimmed Squares (LTS) Method

Least Trimmed Squares (LTS) Method

Minimizing Sum of 13 Smallest Squared Residuals.

Highest Possible Breakdown Value = 42.86 %

Random Selection of 517 Subsets

Among 517 subsets 17 is/are singular.

The best half of the entire data set obtained after full iteration consists of the cases:

5 6 7 8 9 10 11 12 15 16 17 18 19

Estimated Coefficients

VAR1	VAR2	VAR3	Intercep
0.7409210642	0.3915267228	0.0111345398	-37.32332647

LTS Objective Function = 0.474940583

Preliminary LTS Scale = 0.9888435617

Robust R Squared = 0.9745520119

Final LTS Scale = 1.0360272594

Weighted Least-Squares Estimation

RLS Parameter Estimates Based on LTS

Variable	Estimate	Approx Std Err	t Value	Pr > |t|	Lower WCI	Upper WCI
VAR1	0.75694055	0.07860766	9.63	<.0001	0.60287236	0.91100874
VAR2	0.45353029	0.13605033	3.33	0.0067	0.18687654	0.72018405
VAR3	-0.05211	0.05463722	-0.95	0.3607	-0.159197	0.054977
Intercep	-34.05751	3.82881873	-8.90	<.0001	-41.561857	-26.553163

Figure 25.239 *continued*

Weighted Sum of Squares = 10.273044977

Degrees of Freedom = 11

RLS Scale Estimate = 0.9663918355

Cov Matrix of Parameter Estimates				
	VAR1	VAR2	VAR3	Intercep
VAR1	0.0061791648	-0.005776855	-0.002300587	-0.034290068
VAR2	-0.005776855	0.0185096933	0.0002582502	-0.069740883
VAR3	-0.002300587	0.0002582502	0.0029852254	-0.131487406
Intercep	-0.034290068	-0.069740883	-0.131487406	14.659852903

Weighted R-squared = 0.9622869127

F(3,11) Statistic = 93.558645037

Probability = 4.1136826E-8

There are 15 points with nonzero weight.

Average Weight = 0.7142857143

The run has been executed successfully.

The preceding program produces the following output associated with the LTS analysis. In this analysis, observations, 1, 2, 3, 4, 13, and 21 have scaled residuals larger than 2.5 (table not shown) and are considered outliers.

See the documentation for the LMS subroutine for additional details.

LUPDT Call

CALL LUPDT(*lup, bup, sup, L, z* < *, b* > < *, y* > < *, ssq* > **);**

The LUPDT subroutine provides updating and downdating for rank deficient linear least squares solutions, complete orthogonal factorization, and Moore-Penrose inverses.

The LUPDT subroutine returns the following values:

lup is an $n \times n$ lower triangular matrix L that is updated or downdated by using the q rows in Z.

bup is an $n \times p$ matrix B of right-hand sides that is updated or downdated by using the q rows in Y. If *b* is not specified, *bup* is not accessible.

sup is a p vector of square roots of residual sum of squares that is updated or downdated by using the q rows in Y. If *ssq* is not specified, *sup* is not accessible.

The input arguments to the LUPDT subroutine are as follows:

L specifies an $n \times n$ lower triangular matrix $\mathbf{L}$ to be updated or downdated by q row vectors z stored in the $q \times n$ matrix Z. Only the lower triangle of *L* is used; the upper triangle can contain any information.

z is a $q \times n$ matrix Z used rowwise to update or downdate the matrix L.

b specifies an optional $n \times p$ matrix $\mathbf{B}$ of right-hand sides that have to be updated or downdated simultaneously with *L*. If *b* is specified, the argument *y* must be specified.

y specifies an optional $q \times p$ matrix Y used rowwise to update or downdate the right-hand-side matrix *b*.

ssq specifies an optional $p \times 1$ vector that, if *b* is specified, specifies the square root of the error sum of squares that should be updated or downdated simultaneously with *L* and *b*.

The relevant formula for the LUPDT call is $\tilde{\mathbf{L}}\tilde{\mathbf{L}}' = \mathbf{L}\mathbf{L}' + \mathbf{Z}\mathbf{Z}'$. See the section "Complete QR Decomposition with LUPDT" on page 970 in the documentation for the RZLIND call.

MAD Function

MAD(*x* **<** *, method* **>);**

The MAD function computes the univariate (scaled) median absolute deviation of each column of the input matrix.

The arguments to the MAD function are as follows:

x is an $n \times p$ input data matrix.

method is an optional string argument with the following values:

"MAD" for computing the median absolute deviation (MAD); this is the default.

"NMAD" for computing the normalized version of MAD

"SN" for computing S_n

"QN" for computing Q_n

For simplicity, the following descriptions assume that the input argument *x* is a column vector. The notation x_i means the *i*th element of the column vector *x*.

The MAD function can be used for computing one of the following three robust scale estimates:

- median absolute deviation (MAD) or normalized form of MAD,

  $$\text{MAD}_n = b * \text{med}_i^n \; |x_i - \text{med}_j^n \; x_j|$$

  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,

  $$S_n = c_n * \text{med}_i \; \text{med}_j \; |x_i - x_j|$$

  where the outer median is a low median (order statistic of rank $\left[\frac{n+1}{2}\right]$) and the inner median is a high median (order statistic of rank $\left[\frac{n}{2} + 1\right]$), and where c_n is a scalar that depends on sample size n.

- Q_n is another efficient alternative to MAD. It is based on the kth-order statistic of the $\binom{n}{2}$ inter-point distances,

$$Q_n = d_n * \{|x_i - x_j|; \quad i < j\}_{(k)} \quad \text{with} \quad k \approx \binom{n}{2}/4$$

  where d_n is a scalar similar to but different from c_n. See Rousseeuw and Croux (1993) for more details.

The scalars c_n and d_n are defined as follows:

$$c_n = 1.1926* \begin{cases} 0.743 & \text{for n=2} \\ 1.851 & \text{for n=3} \\ 0.954 & \text{for n=4} \\ 1.351 & \text{for n=5} \\ 0.993 & \text{for n=6} \\ 1.198 & \text{for n=7} \\ 1.005 & \text{for n=8} \\ 1.131 & \text{for n=9} \\ n/(n-0.9) & \text{for other odd n} \\ 1.0 & \text{otherwise} \end{cases} \qquad d_n = 2.2219* \begin{cases} 0.399 & \text{for n=2} \\ 0.994 & \text{for n=3} \\ 0.512 & \text{for n=4} \\ 0.844 & \text{for n=5} \\ 0.611 & \text{for n=6} \\ 0.857 & \text{for n=7} \\ 0.669 & \text{for n=8} \\ 0.872 & \text{for n=9} \\ n/(n+1.4) & \text{for other odd n} \\ n/(n+3.8) & \text{otherwise} \end{cases}$$

Example

The following example uses the univariate data set of Barnett and Lewis (1978). The data set is used in Chapter 15 to illustrate the univariate LMS and LTS estimates.

```
b = {3, 4, 7, 8, 10, 949, 951};

rmad1 = mad(b);
rmad2 = mad(b,"mad");
rmad3 = mad(b,"nmad");
rmad4 = mad(b,"sn");
rmad5 = mad(b,"qn");
print "Default MAD=" rmad1,
      "Common MAD =" rmad2,
      "MAD*1.4826 =" rmad3,
      "Robust S_n =" rmad4,
      "Robust Q_n =" rmad5;
```

Figure 25.240 Median Absolute Deviations

	rmad1
Default MAD=	4

	rmad2
Common MAD =	4

	rmad3
MAD*1.4826 =	5.9304089

Figure 25.240 *continued*

rmad4
Robust S_n = 7.143674

rmad5
Robust Q_n = 5.7125049

MAGIC Function

MAGIC(*n*);

The MAGIC function is part of the IMLMLIB library. The MAGIC function returns an $n \times n$ magic square for $n > 2$. The matrix M is a magic square if it contains the integers $1, 2, \ldots, n^2$. If s is the trace of M, then M satisfies the following conditions:

- The sum of every row is s.
- The sum of every column is s.
- The sum of the antidiagonal is s.

There are many algorithms for creating magic squares. The algorithm implemented in the MAGIC function is based on Moler (2011).

The MAGIC function is mainly used to generate examples for documentation, discussion forums, books, and so forth. The following example displays two magic squares:

```
m3 = Magic(3);
m4 = Magic(4);
print m3, m4;
```

Figure 25.241 Magic Squares of Size 3 and 4

m3		
8	1	6
3	5	7
4	9	2

m4			
16	2	3	13
5	11	10	8
9	7	6	12
4	14	15	1

MAHALANOBIS Function

MAHALANOBIS(*x*, < , *center*> < , *cov*> **);**

The MAHALANOBIS function is part of the IMLMLIB library. The MAHALANOBIS function returns the Mahalanobis distance between *center* and the rows of *x*, measured according to the Mahalanobis metric. The arguments are as follows:

x	specifies an $n \times p$ numerical matrix that contains n points in p-dimensional space.
center	is a $1 \times p$ numerical vector that contains a point in p-dimensional space. The function returns the distances from the rows of *x* to *center*. If *center* is not specified, the sample mean, $\bar{x}$, is used.
cov	is an $n \times n$ covariance matrix that specifies the metric that is used to compute distances. If *cov* is the identity matrix, then the function returns the usual Euclidean distance. If *cov* is not specified, the sample covariance matrix of *x* is used. In this case, the number of rows of *x* must be strictly greater than the number of columns, so that the covariance matrix is nonsingular.

If u and c are p-dimensional row vectors and S is a covariance matrix, then the Mahalanobis distance between u and c is

$$d(u,c) = \left[(u-c)S^{-1}(u-c)'\right]^{1/2}$$

The following statements compute the Mahalanobis distance between the rows of *x* and the point (1, 1):

```
x = {1 0,
     0 1,
    -1 0,
     0 -1};
center = {1 1};
cov = {4 1,
       1 9};
maha = mahalanobis(x, center, cov);
print maha;
```

Figure 25.242 Mahalanobis Distance between Pairs of Points

maha
0.3380617
0.5070926
1.0141851
0.7745967

When the *cov* argument is an identity matrix, the Mahalanobis distance simplifies to the usual Euclidean distance. See the DISTANCE function for more information.

MARG Call

CALL MARG(*locmar, marginal, dim, table, config***);**

The MARG subroutine evaluates marginal totals in a multiway contingency table.

The input arguments to the MARG subroutine are as follows:

locmar	is a returned matrix that contains a vector of indices to each new set of marginal totals under the model specified by *config*. A marginal total is exhibited for each level of the specified marginal. These indices help locate particular totals.
marginal	is a return vector of marginal totals.
dim	is an input matrix. If the problem contains v variables, then *dim* is $1 \times v$ row vector. The value *dim*[i] is the number of possible levels for variable i in a contingency table.
table	is an input matrix. The *table* argument specifies an array of the number of observations at each level of each variable. Variables are nested across columns and then across rows.
config	is an input matrix. The *config* argument specifies which marginal totals to evaluate. Each column of *config* specifies a distinct marginal in the model under consideration.

The matrix *table* must conform in size to the contingency table specified in *dim*. In particular, if *table* is $n \times m$, the product of the entries in the *dim* vector must equal nm. In addition, there must be some integer k such that the product of the first k entries in *dim* equals m. See the description of the IPF function for more information about specifying *table*.

For example, consider the three-dimensional table discussed in the IPF call, based on data that appear in Christensen (1997). The table presents data on a person's self-esteem for people classified according to their religion and their father's educational level.

		Father's Educational Level				
Religion	Self-Esteem	Not HS Grad	HS Grad	Some Coll	Coll Grad	Post Coll
Catholic	High	575	388	100	77	51
	Low	267	153	40	37	19
Jewish	High	117	102	67	87	62
	Low	48	35	18	12	13
Protestant	High	359	233	109	197	90
	Low	159	173	47	82	32

As explained in the IPF call documentation, the father's education level is Variable 1, self-esteem is Variable 2, and religion is Variable 3.

The following program encodes this table, uses the MARG call to compute a two-way marginal table by summing over the third variable and a one-way marginal by summing over the first two variables.

```
dim={5 2 3};

table={
/* Father's Education:
           NotHSGrad HSGrad Col ColGrad PostCol
          Self-
   Relig  Esteem                          */
/* Cath-   Hi */ 575   388  100    77   51,
/* olic    Lo */ 267   153   40    37   19,

/* Jew-    Hi */ 117   102   67    87   62,
/*  ish    Lo */  48    35   18    12   13,

/* Prot-   Hi */ 359   233  109   197   90,
/* estant  Lo */ 159   173   47    82   32
        };

config = { 1 3,
           2 0 };
call marg(locmar, marginal, dim, table, config);
print locmar, marginal;
```

Figure 25.243 Marginal Totals in a Three-Way Table

locmar	
1	11

	marginal												
	COL1	COL2	COL3	COL4	COL5	COL6	COL7	COL8	COL9	COL10	COL11	COL12	COL13
ROW1	1051	723	276	361	203	474	361	105	131	64	1707	561	1481

The first marginal total is contained in locations 1 through 10 of the **marginal** vector, which is shown in Figure 25.243. It represents the results of summing **table** over the religion variable. The first entry of **marginal** is the number of subjects with high self-esteem whose fathers did not graduate from high school ($1051 = 575 + 117 + 359$). The second entry is the number of subjects with high self-esteem whose fathers were high school graduates ($723 = 388 + 102 + 233$). The tenth entry is the number of subjects with low self-esteem whose fathers had some post-collegiate education ($64 = 19 + 13 + 32$).

The second marginal is contained in locations 11 through 13 of the **marginal** vector. It represents the results of summing **table** over the education and self-esteem variables. The eleventh entry of the **marginal** vector is the number of Catholics in the study. The thirteenth entry is the number of Protestants.

You can also extract the marginal totals into separate vectors, as shown in the following statements:

```
/* Examine marginals: The name indicates the
   variable(s) that are NOT summed over.
   The locmar variable tells where to index
   into the marginal variable. */
Var12_Marg = marginal[1:(locmar[2]-1)];
Var12_Marg = shape(Var12_Marg, dim[2], dim[1]);
Var3_Marg  = marginal[locMar[2]:ncol(marginal)];
print Var12_Marg, Var3_Marg;
```

Figure 25.244 Marginal Totals

Var12_Marg				
1051	723	276	361	203
474	361	105	131	64

Var3_Marg
1707
561
1481

MATTRIB Statement

MATTRIB *name* <**ROWNAME**=*row-name*> <**COLNAME**=*column-name*> <**LABEL**=*label*> <**FORMAT**=*format*> **;**

The MATTRIB subroutine associates printing attributes with matrices.

The input arguments to the MATTRIB subroutine are as follows:

name	is a character matrix or quoted literal that contains the name of a matrix.
row-name	is a matrix or quoted literal that specifies row names.
column-name	is a matrix or quoted literal that specifies column names.
label	is a character matrix or quoted literal that associates a label with the matrix. The *label* argument has a maximum length of 256 characters.
format	is a valid SAS format.

The MATTRIB statement associates printing attributes with matrices. Each matrix can be associated with a ROWNAME= matrix and a COLNAME= matrix, which are used whenever the matrix is printed to label the rows and columns, respectively. The statement is written as the keyword MATTRIB followed by a list of one or more names and attribute associations. It is not necessary to specify all attributes. The attribute associations are applied to the previous *name*. Thus, the following statement associates a row name RA and a column name CA to **a**, and a column name CB to **b**:

```
a = {1 2 3, 4 5 6};
ra = {"Row 1", "Row 2"};
ca = 'C1':'C3';
b = 1:4;
cb = {"A" "B" "C" "D"};
mattrib a rowname=ra colname=ca  b colname=cb;
print a, b;
```

Figure 25.245 Matrix Attributes

a

	C1	C2	C3
Row 1	1	2	3
Row 2	4	5	6

b

A	B	C	D
1	2	3	4

You cannot group names. The following statement does not associate anything with **a**. In fact, it clears any attributes that were previously associated with **a**.

```
mattrib a b colname=cb;
print a, b;
```

Figure 25.246 Modified Matrix Attributes

a

	C1	C2	C3
Row 1	1	2	3
Row 2	4	5	6

b

A	B	C	D
1	2	3	4

The values of the associated matrices are not looked up until they are needed. Thus, they need not have values at the time the MATTRIB statement is specified; the values can be assigned at any time before the object matrix is printed. Because the associated matrices must available when a PRINT statement is executed, do not use temporary matrices to assign attributes.

The attributes continue to bind with the matrix until reassigned with another MATTRIB statement. To eliminate an attribute, specify EMPTY as the name (for example, ROWNAME=EMPTY). Use the SHOW NAMES statement to view current matrix attributes.

The following example demonstrates all options in the MATTRIB statement:

```
rows = "xr1":"xr3";
cols = "cl1":"cl4";
x = {1 1 1 1,
     2 2 2 2,
     3 3 3 3};
mattrib x rowname=rows
          colname=cols
          label={"My Matrix, x"}
          format=5.2;
print x;
```

Figure 25.247 Matrix Attributes

My Matrix, x				
	cl1	cl2	cl3	cl4
xr1	1.00	1.00	1.00	1.00
xr2	2.00	2.00	2.00	2.00
xr3	3.00	3.00	3.00	3.00

MAX Function

MAX(*matrix1 < , matrix2, . . . , matrix15* **>);**

The MAX function returns the maximum value of a matrix or set of matrices. The matrices can be numeric or character.

For numeric arguments, the MAX function returns a single numeric value that is the largest element among all arguments. For character arguments, the MAX function returns the character string that is largest in the ASCII order. For character arguments, the size of the result is the maximum number of characters among the arguments.

There can be as many as 15 argument matrices. The function checks for missing numeric values and does not include them in the result. If all arguments are missing, then the machine's most negative representable number is the result.

If you want to find the elementwise maximums of the corresponding elements of two matrices, use the maximum operator (<>).

An example that uses the MAX function follows:

```
c = {1 -123 13 56 128 -81 12};
b = max(c);
print b;
```

Figure 25.248 Maximum Value of a Matrix

b
128

MAXQFORM Call

CALL MAXQFORM(*rc, maxq, V, b < , best >* **);**

The MAXQFORM subroutine computes the subsets of a matrix system that maximize the quadratic form.

If V and b are an $n \times n$ matrix and an $n \times 1$ vector, respectively, then the MAXQFORM function computes the subsets of components s such that $b'[s]V^{-1}[s,s]b[s]$ is maximized.

The MAXQFORM subroutine returns the following values:

rc is one of the following scalar return codes:

0	normal return
1	error: the number of elements of b is too large to process
2	error: V is not positive semidefinite

maxq is an $m \times (n+2)$ matrix, where m is the total number of subsets computed and n is the number of elements of *b*. The value of m depends on the value of *best* and is equal to $2^n - 1$ if *best* is not specified. Each row of *maxq* contains information for a selected subset of *V* and *b*. The first element of the row is the number of components in the subset. The second element is the value of the quadratic form. The following elements of the row are either 0 or 1, to indicate whether the corresponding components of *V* and *b* are included in the subset.

The input arguments to the MAXQFORM subroutine are as follows:

V specifies an $n \times n$ positive semidefinite matrix. Often this is generated as a crossproduct matrix, $X'X$, where X is a $k \times n$ matrix.

b specifies an $n \times 1$ vector. Often this arises as $X'y$, where X is a $k \times n$ matrix, and y is a $k \times 1$ vector.

best specifies an optional scalar. If *best* is specified with the value p, then the p subsets with the largest value for the quadratic form are returned for each subset size.

The leaps and bounds algorithm by Furnival and Wilson (1974) computes the maximum value of quadratic forms for subsets of components. Many statistics computed as a quadratic form can then be used as the criterion for the method of subset selection. These include the regression sum of squares, Wald statistics, and score statistics.

Consider the following fitness data, which consists of observations with values for age measured in years, weight measured in kilograms, time to run 1.5 miles measured in minutes, heart rate while resting, heart rate while running, maximum heart rate recorded while running, and oxygen intake rate while running measured in milliliters per kilogram of body weight per minute.

```
fit = {
  44  89.47  11.37  62  178  182  44.609,
  40  75.07  10.07  62  185  185  45.313,
  44  85.84   8.65  45  156  168  54.297,
  42  68.15   8.17  40  166  172  59.571,
  38  89.02   9.22  55  178  180  49.874,
  47  77.45  11.63  58  176  176  44.811,
  40  75.98  11.95  70  176  180  45.681,
  43  81.19  10.85  64  162  170  49.091,
  44  81.42  13.08  63  174  176  39.442,
  38  81.87   8.63  48  170  186  60.055,
  44  73.03  10.13  45  168  168  50.541,
  45  87.66  14.03  56  186  192  37.388,
  45  66.45  11.12  51  176  176  44.754,
  47  79.15  10.60  47  162  164  47.273,
  54  83.12  10.33  50  166  170  51.855,
  49  81.42   8.95  44  180  185  49.156,
  51  69.63  10.95  57  168  172  40.836,
  51  77.91  10.00  48  162  168  46.672,
  48  91.63  10.25  48  162  164  46.774,
  49  73.37  10.08  67  168  168  50.388,
  57  73.37  12.63  58  174  176  39.407,
  54  79.38  11.17  62  156  165  46.080,
  52  76.32   9.63  48  164  166  45.441,
  50  70.87   8.92  48  146  155  54.625,
  51  67.25  11.08  48  172  172  45.118,
  54  91.63  12.88  44  168  172  39.203,
  51  73.71  10.47  59  186  188  45.790,
  57  59.08   9.93  49  148  155  50.545,
  49  76.32   9.40  56  186  188  48.673,
  48  61.24  11.50  52  170  176  47.920,
  52  82.78  10.50  53  170  172  47.467 };
```

Use the following statement to center the data:

```
fitc = fit - fit[:,];
```

Now compute the crossproduct matrices, as follows:

```
x = fitc[, 1:6];
y = fitc[, 7];
xpx = x`*x;
xpy = x`*y;
```

The following statements compute the best three regression sums of squares for each size of regressor set:

```
call maxqform(rc, maxq, xpx, xpy, 3);
print maxq;
```

Figure 25.249 Best Three Regression Sums of Squares

	maxq						
1	632.9001	0	0	1	0	0	0
1	135.78285	0	0	0	1	0	0
1	134.84474	0	0	0	0	1	0
2	650.66573	1	0	1	0	0	0
2	648.26218	0	0	1	0	1	0
2	634.46746	0	0	1	0	0	1
3	690.55086	1	0	1	0	1	0
3	689.60921	0	0	1	0	1	1
3	665.55064	1	0	1	0	0	1
4	712.45153	1	0	1	0	1	1
4	695.14669	1	1	1	0	1	0
4	694.5988	0	1	1	0	1	1
5	721.97309	1	1	1	0	1	1
5	712.63302	1	0	1	1	1	1
5	696.05218	1	1	1	1	1	0
6	722.54361	1	1	1	1	1	1

MCD Call

CALL MCD(*sc, coef, dist, opt, x***);**

The MCD subroutine computes the minimum covariance determinant estimator. The MCD call is the robust estimation of multivariate location and scatter, defined by minimizing the determinant of the covariance matrix computed from *h* points. The algorithm for the MCD subroutine is based on the FAST-MCD algorithm given by Rousseeuw and Van Driessen (1999).

These robust locations and covariance matrices can be used to detect multivariate outliers and leverage points. For this purpose, the MCD subroutine provides a table of robust distances.

In the following discussion, *N* is the number of observations and *n* is the number of regressors. The input arguments to the MCD subroutine are as follows:

opt refers to an options vector with the following components (missing values are treated as default values):

opt[1] specifies the amount of printed output. Higher option values request additional output and include the output of lower values.

- 0 prints no output except error messages.
- 1 prints most of the output.
- 2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.
- 3 additionally prints how many subsets result in singular linear systems.

The default is *opt[1]*=0.

opt[2] specifies whether the classical, initial, and final robust covariance matrices are printed. The default is *opt[2]*=0. The final robust covariance matrix is always returned in *coef*.

opt[3] specifies whether the classical, initial, and final robust correlation matrices are printed or returned. The default is *opt[3]*=0.

0 does not return or print.

1 prints the robust correlation matrix.

2 returns the final robust correlation matrix in *coef*.

3 prints and returns the final robust correlation matrix.

opt[4] specifies the quantile h used in the objective function. The default is *opt[4]*= $h = [\frac{N+n+1}{2}]$. If the value of h is specified outside the range $\frac{N}{2} + 1 \leq h \leq \frac{3N}{4} + \frac{n+1}{4}$, it is reset to the closest boundary of this region.

opt[5] specifies the number N_{Rep} of subset generations. This option is the same as described for the LMS subroutine and the LTS subroutine. Due to computer time restrictions, not all subset combinations can be inspected for larger values of N and n.

When *opt[5]* is zero or missing:

- If $N > 600$, up to five disjoint random subsets are constructed with sizes as equal as possible, but not to exceed 300. Inside each subset, $N_{\text{Rep}} = 500/5 = 100$ subset combinations of n observations are chosen.
- If $N \leq 600$, the number of subsets is taken from the following table.

n	**1**	**2**	**3**	**4**	**5**	**6**	**7 or more**
N_{lower}	500	50	22	17	15	14	0

- If the number of observations N is smaller than N_{lower}, as given in the table, then all possible subsets are used; otherwise, $N_{\text{Rep}} = 500$ subsets are chosen randomly. This means that an exhaustive search is performed for *opt[5]*=−1.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors.

Missing values are not permitted in x. Missing values in *opt* cause default values to be used for each option.

The MCD subroutine returns the following values:

sc is a column vector that contains the following scalar information:

sc[1] the quantile h used in the objective function

sc[2] number of subsets generated

sc[3] number of subsets with singular linear systems

sc[4] number of nonzero weights w_i

sc[5] lowest value of the objective function F_{MCD} attained (smallest determinant)

sc[6] Mahalanobis-like distance used in the computation of the lowest value of the objective function F_{MCD}

sc[7] the cutoff value used for the outlier decision

coef is a matrix with n columns that contains the following results in its rows:

coef[1,]	location of ellipsoid center
coef[2,]	eigenvalues of final robust scatter matrix
coef[3:2+n,]	the final robust scatter matrix for *opt[2]*=1 or *opt[2]*=3
coef[2+n+1:2+2n,]	the final robust correlation matrix for *opt[3]*=1 or *opt[3]*=3

dist is a matrix with N columns that contains the following results in its rows:

dist[1,]	Mahalanobis distances
dist[2,]	robust distances based on the final estimates
dist[3,]	weights (1 for small robust distances; 0 for large robust distances)

Example

Consider the Brownlee (1965) stackloss data used in the example for the MVE subroutine.

For $N = 21$ and $n = 4$ (three explanatory variables including intercept), you obtain a total of 5,985 different subsets of 4 observations out of 21. If you decide not to specify `opt[5]`, the MCD algorithm chooses 500 random sample subsets, as in the following statements:

```
    /* Int 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];             /* X1-X3 */
opt = j(8, 1, .);
opt[1] = 2;               /* ipri */
opt[2] = 1;               /* pcov: print COV */
opt[3] = 1;               /* pcor: print CORR */

call mcd(sc, xmcd, dist, opt, a);
```

A portion of the output is shown in the following figures. Figure 25.250 shows a summary of the MCD algorithm and the final h points selected.

Figure 25.250 Summary of MCD

Fast MCD by Rousseeuw and Van Driessen	
Number of Variables	3
Number of Observations	21
Default Value for h	12
Specified Value for h	12
Breakdown Value	42.86
- Highest Possible Breakdown Value -	

Figure 25.251 shows the observations that were chosen that are used to form the robust estimates.

Figure 25.251 Selected Observations

MCD Estimates (Obtained by Subsampling and Iteration)

The best half of the entire data set obtained after full iteration consists of the cases:

4	5	6	7	8	9	10	11	12	13	14	20

Figure 25.252 shows the MCD estimators of the location, scatter matrix, and correlation matrix. The MCD scatter matrix is multiplied by a factor to make it consistent with the data that come from a single Gaussian distribution.

Figure 25.252 MCD Estimators

MCD Location Estimate		
VAR1	VAR2	VAR3
59.5	20.833333333	87.333333333

MCD Scatter Matrix Estimate			
	VAR1	VAR2	VAR3
VAR1	5.1818181818	4.8181818182	4.7272727273
VAR2	4.8181818182	7.6060606061	5.0606060606
VAR3	4.7272727273	5.0606060606	19.151515152

Consistent Scatter Matrix			
	VAR1	VAR2	VAR3
VAR1	8.6578437815	8.0502757968	7.8983838007
VAR2	8.0502757968	12.708297013	8.4553211199
VAR3	7.8983838007	8.4553211199	31.998580526

Figure 25.253 shows the classical Mahalanobis distances, the robust distances, and the weights that identify the outlying observations (that is, leverage points when explaining y with these three regressor variables).

Figure 25.253 Robust Distances

Classical Distances and Robust (Rousseeuw) Distances

Unsquared Mahalanobis Distance and

Unsquared Rousseeuw Distance of Each Observation

N	Mahalanobis Distances	Robust Distances	Weight
1	2.253603	12.173282	0
2	2.324745	12.255677	0
3	1.593712	9.263990	0
4	1.271898	1.401368	1.000000
5	0.303357	1.420020	1.000000
6	0.772895	1.291188	1.000000
7	1.852661	1.460370	1.000000
8	1.852661	1.460370	1.000000
9	1.360622	2.120590	1.000000
10	1.745997	1.809708	1.000000
11	1.465702	1.362278	1.000000
12	1.841504	1.667437	1.000000
13	1.482649	1.416724	1.000000
14	1.778785	1.988240	1.000000
15	1.690241	5.874858	0
16	1.291934	5.606157	0
17	2.700016	6.133319	0
18	1.503155	5.760432	0
19	1.593221	6.156248	0
20	0.807054	2.172300	1.000000
21	2.176761	7.622769	0

Robust distances are based on reweighted estimates.

The cutoff value is the square root of the 0.975 quantile of the chi square distribution with 3 degrees of freedom.

Points whose robust distance exceeds 3.0575159206 have received a zero weight in the last column above.

There were 9 such points in the data.

These may include boundary cases.

Only points whose robust distance is substantially larger than the cutoff should be considered outliers.

MEAN Function

MEAN(*x* < , *method*> < , *param*> **);**

The MEAN function computes a sample mean of data. The arguments are as follows:

x specifies an $n \times p$ numerical matrix. The MEAN function computes means of the p columns of this matrix.

method specifies the method used to compute the mean. This argument is optional. The following are valid values:

- "arithmetic" specifies that arithmetic means be computed. This is the default value.
- "trimmed" specifies that trimmed means be computed. The number of observations that are trimmed is determined by the *param* option.
- "winsorized" specifies that Winsorized means be computed. The number of observations that are Winsorized is determined by the *param* option.

param specifies the number of observations trimmed or Winsorized. (This argument is ignored when "arithmetic" is specified for the *method* argument.) The default value for *param* is 0.1, which corresponds to trimming or Winsorizing 10% of the observations with the lowest values and 10% of the observations with the largest values.

The *method* argument is not case-sensitive. The first four characters are used to determine the value. For example, "WINS", "Winsor", and "winsorized" specify the same option.

The MEAN function uses the same algorithms as the UNIVARIATE procedure for computing the means, trimmed means, and Winsorized means. For additional details and formulas, see the UNIVARIATE procedure documentation (especially the TRIMMED= and WINSORIZED= options) in the *Base SAS Procedures Guide: Statistical Procedures*.

The *param* argument determines how many observations are trimmed (or Winsorized). The value for this argument can be an integer or a proportion. If the value is an integer k, then k observations are trimmed, provided that k is between 0 and half the number of nonmissing observations. If value is a proportion p in the interval $[0, 0.5)$, then the number of observations trimmed is equal to the smallest integer that is greater than or equal to np, where n is the number of nonmissing observations.

The following example demonstrates basic usage:

```
x = {5,6,6,6,7,7,7,8,8,15};
mean = mean(x);
trim = mean(x, "trimmed", 0.2); /* 20% of obs */
winsor = mean(x, "winsorized", 1); /* one obs */
print mean trim winsor;
```

Figure 25.254 Arithmetic, Trimmed, and Winsorized Means

mean	trim	winsor
7.5	6.8333333	6.9

The MEAN function operates on columns of matrices. If *x* is an $n \times p$ matrix, the function returns a $1 \times p$ row vector. The value of the *j*th element is the mean for the *j*th column of the matrix, as the following example demonstrates:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
```

```
     7 2 13};
mean = mean(x);
print mean;
```

Figure 25.255 Arithmetic Mean of Columns

mean		
6	4	8

Missing values in a column are excluded from the computation. The default behavior of the MEAN function is identical to the subscript reduction operator that computes the mean. That is, **mean(x)** and **x[:,]** both compute the means of the columns of **x**. See the section "Subscript Reduction Operators" on page 55 for more information about subscript reduction operators.

MEDIAN Function

MEDIAN(*matrix***);**

The MEDIAN function is part of the IMLMLIB library. The MEDIAN function returns the median value for each column in the $n \times m$ matrix argument. 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 excluded from the computation. If all values in a column are missing, the return value for that column is missing. An example of the MEDIAN function follows:

```
x = {1 3,
     2 3,
     4 9,
    10 0};
med = median(x);
print med;
```

Figure 25.256 Median of Columns

med	
3	3

MILPSOLVE Call

CALL MILPSOLVE(*rc, objvalue, x, relgap, c, a, b* **<** *, cntl* **>** **<** *, coltype* **>** **<** *, rowsense* **>** **<** *, range* **>** **<** *, l* **>** **<** *, u* **>** **);**

The MILPSOLVE subroutine solves a mixed integer linear programming problem. For complete functionality the SAS/OR product must also be installed, otherwise the maximum number of variables and maximum number of constraints is restricted to 500.

The input arguments to the MILPSOLVE subroutine are as follows:

c is a vector of dimension n of objective function coefficients. A missing value is treated as 0.

a is an $m \times n$ matrix of the technological coefficients. A missing value is treated as 0.

b is a vector of dimension m of constraints' right-hand sides (RHS). For a range constraint, *b* is the constraint upper bound. A missing value is treated as 0.

cntl is an optional vector that contains the control parameters for the MILPSOLVE subroutine. The vector can contain from one to 14 options. These options are a subset of the options that are supported by the OPTMILP procedure in SAS/OR software. For a detailed description of the options, see *SAS/OR User's Guide: Mathematical Programming*. A default value is used when an option is not specified or its value is a missing value. If *cntl*=(*objsense, printlevel, maxtime, maxnodes, relobjgap, presolver, cuts, heuristics, probe, nodesel, varsel, conflictsearch, inttol, tol*), then

- *objsense* specifies whether the problem is a minimization or a maximization problem, where 1 specifies a minimization problem and -1 specifies a maximization problem. The default value is 1.
- *printlevel* specifies the type of messages printed to the log. A value of 0 prints warning and error messages only, whereas 1 prints solution information in addition to warning and error messages. The default value is 0.
- *maxtime* specifies an upper bound of running time in seconds. The default value is effectively unbounded.
- *maxnodes* specifies the maximum number of branch-and-bound nodes to be processed. The default value is no limit.
- *relobjgap* specifies a stopping criterion that is based on the best integer objective and the objective of the best remaining node. The stopping criterion is

  $$|\text{BestInteger} - \text{BestBound}|/(10^{-10} + |\text{BestBound}|)$$

  The default value is 10^{-4}.
- *presolver* specifies a presolve option, where 0 specifies that no presolve is performed, and 1 specifies that an automatic presolve is performed. The default value is 1.
- *cuts* specifies a cuts option, where 0 specifies that no cuts are made, and 1 specifies that automatic cuts are made. The default value is 0.
- *heuristics* specifies a heuristics option, where 0 specifies that no heuristics are used, and 1 specifies that heuristics are automatically used. The default value is 1.
- *probe* specifies a probe option, where 0 specifies that no probing is performed, and 1 specifies that probing is automatically performed. The default value is 1.
- *nodesel* specifies the node selection strategy, where -1 specifies automatic selection, 0 chooses the node that has the best relaxed objective, 1 chooses the node that has the best estimate of the integer objective value, and 2 chooses the most recently created node. The default value is -1.
- *varsel* specifies the rule for selecting the branching variable, where -1 uses automatic branching variable selection, 0 chooses the variable that has maximum infeasibility, 1 chooses the variable that has minimum infeasibility, 2 chooses a branching variable based on pseudocost, and 3 uses the strong branching variable selection strategy. The default value is -1.

conflictsearch specifies a conflict search option, where 0 specifies no conflict search and 1 specifies automatic conflict search. The default value is 1.

inttol specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. . The value can be any number between 0.0 and 0.5. The default value is 10^{-5}.

tol specifies a feasibility and optimality tolerance. The value can be any number between 10^{-9} and 10^{-4}. The default value is 10^{-6}.

coltype is an optional column vector of dimension *n* that specifies the type of each variable. The values can be C, B, or I for continuous, binary, or integer variable. If this vector is missing or *coltype[j]* has a missing value, the solver treats variable *j* as a binary variable if both *l* and *u* bounds are not specified or as a continuous variable otherwise.

rowsense is an optional row vector of dimension *m* that specifies the sense of each constraint. The values can be E, L, G, or R for equal, less than or equal to, greater than or equal to, or range constraint. If this vector is missing, the solver treats the constraints as E type constraints.

range is an optional row vector of dimension *m* that specifies the range of the constraints. The row sense of a range constraint is R. For the non-range constraints, the corresponding values are ignored. For a range constraint, the range value is the difference between its constraint lower bound and its constraint upper bound *b*, so it must be nonnegative.

l is an optional column vector of dimension *n* that specifies lower bounds on the decision variables. If you do not specify *l* or *l[j]* has a missing value, then the lower bound of variable *j* is assumed to be 0.

u is an optional column vector of dimension *n* that specifies upper bounds on the decision variables. If you do not specify *u* or *u[j]* has a missing value, the upper bound of variable *j* is assumed to be 1 for a binary or integer variable, or infinity for the continuous variable.

The MILPSOLVE subroutine returns the following values:

rc returns one of the following scalar return codes:

rc	**Termination Reason**
0	The solution is integer optimal.
1	The time limit was exceeded.
2	The number of node limit was exceeded.
3	The solution is infeasible.
4	The solution is unbounded or infeasible.
5	The subroutine could not obtain enough memory.
6	The subroutine failed to solve the problem.

objvalue returns the optimal or final objective value at termination.

x returns the current primal solution in a column vector of length *n*.

relgap returns the relative gap between the current best integer objective and the objective of the best remaining node.

The MILPSOLVE subroutine is a solver for general mixed integer linear programs (MILPs).

A standard mixed integer linear program has the formulation:

$$\min c^T \mathbf{x}$$
$$\text{subject to } \mathbf{A}\mathbf{x}\{\geq, =, \leq\}\mathbf{b}$$
$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$$
$$\text{where } x_i \text{ is integer for some subset of indices}$$

If only c, $\mathbf{A}$, and $\mathbf{b}$ are present, then the MILPSOLVE subroutine solves the following integer programming problem by default:

$$\min c^T \mathbf{x}$$
$$\text{subject to } \mathbf{A}\mathbf{x} = \mathbf{b}$$
$$\text{all } x_i \text{ are binary}$$

The MILPSOLVE subroutine implements a linear-programming-based branch-and-bound algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The MILPSOLVE subroutine also implements advanced techniques such as presolving, probing, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm. These techniques are explained in more detail in the chapter "The OPTMILP Procedure" in *SAS/OR User's Guide: Mathematical Programming*.

Consider the following example:

$$\min(X_1 + X_2)$$
$$\text{subject to } 2X_1 + 0.5X_2 - X_3 \leq 1$$
$$0.2X_1 + 5X_2 - X_4 \leq 1$$
$$X_i \text{ is binary for } i = 1, 2, 3, 4$$

The problem is solved by using the following statements:

```
object = {  1   1   0   0 };
coef   = {  2  .5  -1   0,
           .2   5   0  -1};
b =      { 1,   1 };
rowsense = {L,L};
cntl = -1;
call milpsolve(rc,objv,x,relgap,object,coef,b,cntl,,rowsense);
print objv, x, relgap;
```

Figure 25.257 Example MILPSOLVE Call

objv
1

Figure 25.257 *continued*

x
1
0
1
0

relgap
0

MIN Function

MIN(*matrix1 < , matrix2, . . . , matrix15>* **);**

The MIN function returns the minimum value of a matrix or set of matrices. The matrices can be numeric or character.

The MIN function produces a single numeric value (or a character string value) that is the smallest element (lowest character string value) in all arguments. There can be as many as 15 argument matrices. The function checks for missing numeric values and excludes them from the result. If all arguments are missing, then the machine's largest representable number is the result.

If you want to find the elementwise minimums of the corresponding elements of two matrices, use the element minimum operator (><).

For character arguments, the size of the result is the size of the largest of all arguments.

The following statements use the MIN function to compute the minimum value of a vector:

```
c = {1 -123 13 56 128 -81 12};
b = min(c);
print b;
```

Figure 25.258 Minimum Value

b
-123

MOD Function

MOD(*value, divisor***);**

The MOD function returns the remainder of the division of elements of the first argument by elements of the second argument.

The arguments to the MOD function are as follows:

value is a numeric matrix or literal that contains the dividend.

divisor is a numeric matrix or literal that contains the divisor.

If either operand is a scalar, the MOD function performs the operation for each element of the matrix with the scalar value. If either operand is a row or column vector, then the operation is performed by using that vector on each of the rows or columns of the matrix.

Unlike the MOD function in Base SAS software, the MOD function in SAS/IML software does not perform any numerical "fuzzing" to return an exact zero when the result would otherwise be very small. Thus the results of the SAS/IML MOD function is more similar to the MODZ function in Base SAS software.

An example of a valid statement follows:

```
c = {-7 14 20 -81 23};
b = mod(c, 4);
print b;
```

Figure 25.259 Remainders after Division

b
-3 2 0 -1 3

MODULEI Call

CALL MODULEI(*control, modname,* < *matrix1, . . . , matrix13*> **);**

The MODULEI subroutine calls an external routine that does not return a value.

The input arguments to the MODULEI subroutine are as follows:

control is a character matrix that contains a control string.

modname is a character matrix that contains the name of the external routine to be called.

matrix specifies matrix parameters to be passed to the external routine.

The CALL MODULEI routine executes a routine *modname* that resides in an external shared library with the specified arguments.

The MODULEI call routine is similar to the MODULE call routine that is available in the SAS DATA step. It is also closely related to the MODULEIN function, which returns a scalar numeric value, and the MODULEIC function, which returns a character value. CALL MODULEI builds a parameter list by using the information in the arguments and a routine description and argument attribute table that you define in a separate file. The attribute table is a sequential text file that contains descriptions of the routines that you can invoke with the CALL MODULEI routine and MODULEIN and MODULEIC functions. The purpose of the table is to define how CALL MODULEI should interpret its supplied arguments when it builds a parameter list to pass to the external routine. The attribute table should contain a description for each external routine that you intend to call, and descriptions of each argument associated with that routine. This enables you to call external routines that have been compiled in different programming languages that use different calling and matrix representation conventions.

Before you invoke CALL MODULEI, you must define the fileref of SASCBTBL to point to the external file that contains the attribute table. You can name the file whatever you want when you create it. You can then use matrices as arguments to CALL MODULEI and ensure that these arguments are properly converted before being passed to the external routine. The exact syntax for the attribute table is system-dependent, and can be found in the *SAS Companion* for your operating system. Attempting to use CALL MODULEI for a module without a correct attribute table entry can cause the SAS System to fail.

MODULEIC Function

MODULEIC(*control, modname, < matrix1, . . . , matrix13>* **);**

The MODULEIC subroutine calls an external routine that returns a character value.

The arguments to the MODULEIC function are as follows:

control	is a character matrix that contains a control string.
modname	is a character matrix that contains the name of the external routine to be called.
matrix	specifies matrix parameters to be passed to the external routine.

The MODULEIC routine executes a routine *modname* that resides in an external shared library with the specified arguments and that returns a character value.

The description of this function is identical to the description of the MODULEI call, except that the MODULEIC function returns a character value from the external routine. See the MODULEI call for a full description of the function and its arguments.

MODULEIN Function

MODULEIN(*control, modname, < matrix1, . . . , matrix13>* **);**

The MODULEIN subroutine calls an external routine that returns a numerical value.

The arguments to the MODULEIN function are as follows:

control	is a character matrix that contains a control string.
modname	is a character matrix that contains the name of the external routine to be called.
matrix	specifies matrix parameters to be passed to the external routine.

The MODULEIN routine executes a routine *modname* that resides in an external shared library with the specified arguments and that returns a numeric value.

The description of this function is identical to the description of the MODULEI call, except that the MODULEIN function returns a scalar numeric value from the external routine. See the MODULEI call for a full description of the function and its arguments.

This example invokes the CHANGI routine from the TRYMOD.DLL module on a Windows platform. Use the following attribute table.

```
routine changi module=trymod returns=long;
arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;
```

The following statements call the CHANGI function:

```
proc iml;
   ones = J(4,5,1);
   i = do(10, 40, 10);
   j = 4:8;
   x1 = i` # ones + j;

   y1=x1;
   x2=x1;
   y2=y1;
   rc=modulein("*i","changi",6,x2);
```

MODULESTACK Function

MODULESTACK();

The MODULESTACK function returns a vector of the names of all modules in the module call stack. The first element in the vector is the name of the current module, the second element is the name of the module that called the current module, and so forth. The last element in the vector is always MAIN, which represents the main SAS/IML program. In other words, if **s = ModuleStack()**, then **s[1]** is the name of the current module, **s[2]** is the name of the parent module, and so forth.

The MODULESTACK function is useful as a diagnostic and debugging tool. You can use it to determine the sequence of modules that call a specific module. The following program provides an example. The program defines two modules. The MOD1 module calls the MODULESTACK function and prints the result. The MOD2 module calls the MOD1 module. The main program calls both modules.

```
start Mod1(n);
   stack = ModuleStack();
   print stack[label="Stack In Mod1"];
   return (2*n);
finish;

start Mod2(n);
   return ( 3*Mod1(n) );  /* call Mod1 from Mod2 */
finish;

x = Mod1(1);              /* call Mod1 from main scope */
y = Mod2(2);
```

The results are shown in Figure 25.260. The first table in the output shows that the MOD1 module was called from the program's main scope. The second table indicates that the MOD1 module was called from the MOD2 function, which in turn was called from the program's main scope.

Figure 25.260 Stack of Module Calls



MVE Call

CALL MVE(*sc, coef, dist, opt, x* < , *s* > **);**

The MVE subroutine computes the robust estimation of multivariate location and scatter, defined by minimizing the volume of an ellipsoid that contains *h* points.

The MVE subroutine computes the minimum volume ellipsoid estimator. These robust locations and covariance matrices can be used to detect multivariate outliers and leverage points. For this purpose, the MVE subroutine provides a table of robust distances.

In the following discussion, *N* is the number of observations and *n* is the number of regressors. The input arguments to the MVE subroutine are as follows:

opt refers to an options vector with the following components (missing values are treated as default values):

opt[1] specifies the amount of printed output. Higher option values request additional output and include the output of lower values.

- 0 prints no output except error messages.
- 1 prints most of the output.
- 2 additionally prints case numbers of the observations in the best subset and some basic history of the optimization process.
- 3 additionally prints how many subsets result in singular linear systems.

The default is *opt[1]*=0.

opt[2] specifies whether the classical, initial, and final robust covariance matrices are printed. The default is *opt[2]*=0. The final robust covariance matrix is always returned in *coef*.

opt[3] specifies whether the classical, initial, and final robust correlation matrices are printed or returned. The default is *opt[3]*=0.

0 does not return or print.

1 prints the robust correlation matrix.

2 returns the final robust correlation matrix in *coef*.

3 prints and returns the final robust correlation matrix.

opt[4] specifies the quantile h used in the objective function. The default is *opt[5]*= $h = \left[\frac{N+n+1}{2}\right]$. If the value of h is specified outside the range $\frac{N}{2}+1 \leq h \leq \frac{3N}{4}+\frac{n+1}{4}$, 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}, as given in the table, then all possible subsets are used; otherwise, N_{Rep} subsets are chosen randomly. This means that an exhaustive search is performed for *opt[5]*= -1. If N is larger than N_{upper}, a note is printed in the log file that indicates how many subsets exist.

x refers to an $N \times n$ matrix $\mathbf{X}$ of regressors. Missing values are not permitted in x.

s refers to an $n+1$ vector that contains $n+1$ observation numbers of a subset for which the objective function should be evaluated, where n is the number of parameters. In other words, the MVE algorithm computes the minimum volume of the ellipsoid that contains the observation numbers contained in s.

The MVE subroutine returns the following values:

sc is a column vector that contains the following scalar information:

sc[1] the quantile h used in the objective function

sc[2] number of subsets generated

sc[3] number of subsets with singular linear systems

sc[4] number of nonzero weights w_i

sc[5] lowest value of the objective function F_{MVE} attained (volume of smallest ellipsoid found)

sc[6] Mahalanobis-like distance used in the computation of the lowest value of the objective function F_{MVE}

sc[7] the cutoff value used for the outlier decision

coef is a matrix with *n* columns that contains the following results in its rows:

coef[1,]	location of ellipsoid center
coef[2,]	eigenvalues of final robust scatter matrix
coef[3:2+n,]	the final robust scatter matrix for *opt[2]*=1 or *opt[2]*=3
coef[2+n+1:2+2n,]	the final robust correlation matrix for *opt[3]*=1 or *opt[3]*=3

dist is a matrix with *N* columns that contains the following results in its rows:

dist[1,]	Mahalanobis distances
dist[2,]	robust distances based on the final estimates
dist[3,]	weights (1 for small robust distances; 0 for large robust distances)

Example

Consider results for Brownlee (1965) stackloss data. The three explanatory variables correspond to measurements for a plant that oxidizes ammonia to nitric acid on 21 consecutive days:

- x_1 air flow to the plant
- x_2 cooling water inlet temperature
- x_3 acid concentration

The response variable y_i contains the permillage of ammonia lost (stackloss). These data are also given by Rousseeuw and Leroy (1987).

```
          /* X1  X2  X3   Y    Stackloss data */
aa = { 1  80  27  89  42,
       1  80  27  88  37,
       1  75  25  90  37,
       1  62  24  87  28,
       1  62  22  87  18,
       1  62  23  87  18,
       1  62  24  93  19,
       1  62  24  93  20,
       1  58  23  87  15,
       1  58  18  80  14,
       1  58  18  89  14,
       1  58  17  88  13,
       1  58  18  82  11,
       1  58  19  93  12,
       1  50  18  89   8,
       1  50  18  86   7,
       1  50  19  72   8,
       1  50  19  79   8,
       1  50  20  80   9,
       1  56  20  82  15,
       1  70  20  91  15 };
```

Rousseeuw and Leroy (1987) cite a large number of papers where this data set was analyzed and state that most researchers "concluded that observations 1, 3, 4, and 21 were outliers"; some people also reported observation 2 as an outlier.

By default, subroutine MVE chooses only 2,000 randomly selected subsets in its search. There are in total 5,985 subsets of 4 cases out of 21 cases, as shown in Figure 25.261, which is produced by the following statements:

```
a = aa[, 2:4];
opt = j(8, 1, .);
opt[1] = 2;               /* ipri */
opt[2] = 1;               /* pcov: print COV */
opt[3] = 1;               /* pcor: print CORR */
opt[5] = -1;              /* nrep: use all subsets */

call mve(sc, xmve, dist, opt, a);
```

The first part of the output (Figure 25.261) shows the classical scatter and correlation matrix, along with the means of each variable.

Figure 25.261 Classical Estimates of Scatter and Location

Classical Covariance Matrix

	VAR1	VAR2	VAR3
VAR1	84.057142857	22.657142857	24.571428571
VAR2	22.657142857	9.9904761905	6.6214285714
VAR3	24.571428571	6.6214285714	28.714285714

Classical Correlation Matrix

	VAR1	VAR2	VAR3
VAR1	1	0.781852333	0.5001428749
VAR2	0.781852333	1	0.3909395378
VAR3	0.5001428749	0.3909395378	1

Classical Mean	
VAR1	60.428571429
VAR2	21.095238095
VAR3	86.285714286

The second part of the output (Figure 25.262) shows the results of the optimization (complete subset sampling):

Figure 25.262 Subset Sampling and Optimal Subset

Subset	Singular	Best Criterion	Percent
1497	22	253.312431	25
2993	46	224.084073	50
4489	77	165.830053	75
5985	156	165.634363	100

Figure 25.262 *continued*

Observations of Best Subset			
7	10	14	20

Initial MVE Location Estimates	
VAR1	58.5
VAR2	20.25
VAR3	87

Initial MVE Scatter Matrix

	VAR1	VAR2	VAR3
VAR1	34.829014749	28.413143611	62.32560534
VAR2	28.413143611	38.036950318	58.659393261
VAR3	62.32560534	58.659393261	267.63348175

The third part of the output (Figure 25.263) shows the optimization results after local improvement:

Figure 25.263 Robust Estimates of Scatter and Location

Robust MVE Location Estimates	
VAR1	56.705882353
VAR2	20.235294118
VAR3	85.529411765

Robust MVE Scatter Matrix

	VAR1	VAR2	VAR3
VAR1	23.470588235	7.5735294118	16.102941176
VAR2	7.5735294118	6.3161764706	5.3676470588
VAR3	16.102941176	5.3676470588	32.389705882

Eigenvalues of Robust Scatter Matrix	
VAR1	46.597431018
VAR2	12.155938483
VAR3	3.423101087

Robust Correlation Matrix

	VAR1	VAR2	VAR3
VAR1	1	0.6220269501	0.5840361335
VAR2	0.6220269501	1	0.375278187
VAR3	0.5840361335	0.375278187	1

The final output (Figure 25.264) presents a table that contains the classical Mahalanobis distances, the robust distances, and the weights that identify the outlying observations (that is leverage points when explaining y with these three regressor variables):

Figure 25.264 Distances and Weights

Classical Distances and Robust (Rousseeuw) Distances

Unsquared Mahalanobis Distance and

Unsquared Rousseeuw Distance of Each Observation

N	Mahalanobis Distances	Robust Distances	Weight
1	2.253603	5.528395	0
2	2.324745	5.637357	0
3	1.593712	4.197235	0
4	1.271898	1.588734	1.000000
5	0.303357	1.189335	1.000000
6	0.772895	1.308038	1.000000
7	1.852661	1.715924	1.000000
8	1.852661	1.715924	1.000000
9	1.360622	1.226680	1.000000
10	1.745997	1.936256	1.000000
11	1.465702	1.493509	1.000000
12	1.841504	1.913079	1.000000
13	1.482649	1.659943	1.000000
14	1.778785	1.689210	1.000000
15	1.690241	2.230109	1.000000
16	1.291934	1.767582	1.000000
17	2.700016	2.431021	1.000000
18	1.503155	1.523316	1.000000
19	1.593221	1.710165	1.000000
20	0.807054	0.675124	1.000000
21	2.176761	3.657281	0

MinRes	1st Qu.	Median	Mean	3rd Qu.	MaxRes
0.6751244996	1.5084120761	1.7159242054	2.2282960174	2.0831826658	5.6373573538

NAME Function

NAME(*arguments***);**

The NAME function returns the names of the arguments in a column vector. The *arguments* parameter specifies the names of existing matrices.

In the following example, **N** is a 2×1 character matrix that contains the character values 'Seq' and 'Const':

```
Seq = 1:3;
Const = -1;
N = name(Seq, Const);
do i = 1 to nrow(N);
   msg = "Values of Matrix " + N[i];
   x = value(N[i]);
   print x[label=msg];
end;
```

Figure 25.265 Matrix Names

Values of Matrix Seq
1 2 3

Values of Matrix Const
-1

A primary use of the NAME function is in writing macros in which you want to use an argument for both its name and its value.

NCOL Function

NCOL(*matrix***);**

The NCOl function returns the number of columns in its matrix argument. If the matrix has not been given a value, the NCOL function returns a value of 0.

For example, following statements display the number of columns of the matrix **m**:

```
m = {1 2 3, 4 5 6, 3 2 1, 4 3 2, 5 4 3};
p = ncol(m);
print p;
```

Figure 25.266 Number of Columns in a Matrix

p
3

NDX2SUB Function

NDX2SUB(*dim, indices* **);**

The NDX2SUB function is part of the IMLMLIB library. The NDX2SUB function converts indices of a matrix into subscripts for the matrix. The arguments are as follows:

dim	specifies the dimensions of the matrix. For example, the value of this argument might be the 1×2 vector that is returned from the DIMENSION function.
indices	specifies the elements of a matrix, enumerated in row-major order.

The indices of an $n \times p$ matrix are the elements $1, 2, \ldots, np$. The indices enumerate the elements in row-major order: the first p indices enumerate the first row, the next p indices enumerate the second row, and so forth. The NDX2SUB function converts indices to subscripts, which are pairs (i, j) such that $1 \leq i \leq n$ and $1 \leq j \leq p$.

You can use the module to display the rows and columns of elements that satisfy a certain condition. For example, the following statements locate all the even numbers in a matrix and then call the NDX2SUB function to find the subscripts of the even elements:

```
x = {1  2  3,
     4  5  6,
     7  8  9,
    10 11 12};
idx = loc( mod(x, 2)=0 );
dim = nrow(x) || ncol(x);
s = ndx2sub(dim, idx);
print s;
```

Figure 25.267 Subscripts That Correspond to Indices

s
1 2
2 1
2 3
3 2
4 1
4 3

You can also use the NDX2SUB function to keep track of indices and subscripts of multidimensional arrays. Although the SAS/IML language does not support multidimensional arrays, a common technique is to store the elements of a $d_1 \times d_2 \times \ldots \times d_k$ array in a two-dimensional matrix with $d_1 \times d_2 \times \ldots \times d_{k-1}$ rows and d_k columns. For example, you can store the contents of four 3×3 arrays in a single 12×3 matrix, as shown in the following program:

```
/* Store four 3x3 matrices in a 12x3 matrix
  (each group of three rows is a matrix) */
dim = {4 3 3};
m = j(12, 3);
p = 9; /* = prod(dim[2:ncol(dim)]) */
do i = 1 to 4;
   startNdx = 1 + (i-1)*p;
   endNdx   = i*p;
   ndx = startNdx:endNdx; /* get indices for i_th matrix */
   m[ndx] = i; /* assign or extract matrix */
   subscripts = ndx2sub(dim, ndx); /* or get subscripts */
end;
print m;
```

Figure 25.268 Storing Smaller Matrices inside a Larger Matrix

m
1 1 1
1 1 1
1 1 1
2 2 2
2 2 2
2 2 2
3 3 3
3 3 3
3 3 3
4 4 4
4 4 4
4 4 4

To convert from subscripts to indices, see the SUB2NDX function.

NLENG Function

NLENG(*matrix***);**

The NLENG function returns a single numeric value that is the size in bytes of each element in *matrix*. All matrix elements have the same size. For English text, this size is also the number of characters that can be stored in a matrix element.

If the matrix does not have a value, then the NLENG function returns a value of 0. This function is different from the LENGTH function, which returns the size of each element of a character matrix, omitting the trailing blanks.

The following statements demonstrate the NLENG function:

```
m = {"ab " "ijklm  ",
     "x"   " "       };
len = nleng(m);
print len;
```

Figure 25.269 Number of Bytes in Each Matrix Element

len
7

Nonlinear Optimization and Related Subroutines

The following list shows the syntax for nonlinear optimization subroutines. Subsequent sections describe each subroutine in detail.

- conjugate gradient optimization method:

CALL NLPCG(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "grd" >* **);**

- double-dogleg optimization method:

  CALL NLPDD(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "grd" >* **);**

- Nelder-Mead simplex optimization method:

  CALL NLPNMS(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "nlc" >* **);**

- Newton-Raphson optimization method:

  CALL NLPNRA(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" >, < , "grd" > < , "hes" >* **);**

- Newton-Raphson ridge optimization method:

  CALL NLPNRR(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "grd" > < , "hes" >* **);**

- (dual) quasi-Newton optimization method:

  CALL NLPQN(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "grd" > < , "nlc" > < , "jacnlc" >* **);**

- quadratic optimization method:

  CALL NLPQUA(*rc, xr, quad, x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , lin >* **);**

- trust-region optimization method:

  CALL NLPTR(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "grd" > < , "hes" >* **);**

The following list shows the syntax for optimization subroutines that use least squares methods. Subsequent sections describe each subroutine in detail.

- hybrid quasi-Newton least squares methods:

  CALL NLPHQN(*rc, xr, "fun", x0, opt < , blc > < , tc > < , par > < , "ptit" > < , "jac" >* **);**

- Levenberg-Marquardt least squares method:

  CALL NLPLM(*rc, xr, "fun", x0, opt < , blc > < , tc > < , par > < , "ptit" > < , "jac" >* **);**

The following list shows the syntax for supplementary subroutines that are often used in conjunction with optimization subroutines. Subsequent sections describe each subroutine in detail.

- approximate derivatives by finite differences:

  CALL NLPFDD(*f, g, h, "fun", x0 < , par> < , "grd">* **);**

- feasible point subject to constraints:

  CALL NLPFEA(*xr, x0, blc < , par>* **);**

NOTE: The names of the optional arguments can be used as keywords. For example, the following statements are equivalent:

```
call nlpnrr(rc,xr,"fun",x0,,,ter,,,"grad");
call nlpnrr(rc,xr,"fun",x0) tc=ter grd="grad";
```

All the optimization subroutines require at least two input arguments:

- The NLPQUA subroutine requires the *quad* matrix argument, which specifies the symmetric matrix $\mathbf{G}$ of the quadratic problem. The input can be dense or sparse.
- Other optimization subroutines require the *"fun"* argument, which specifies a module that defines the objective function or functions. For least squares subroutines, the FUN module must return a column vector of length m that corresponds to the values of the m functions $f_1(x), \ldots, f_m(x)$, each evaluated at the point $x = (x_1, \ldots, x_n)$. For other subroutines, the FUN module must return the value of the objective function $f = f(x)$ evaluated at the point x.
- The argument *x0* specifies a row vector that defines the number of parameters n. If *x0* is a feasible point, it represents a starting point for the iterative optimization process. Otherwise, a linear programming algorithm is called at the start of each optimization subroutine to replace the input *x0* by a feasible starting point.

The other arguments that can be used as input are described in the following list. As indicated in the previous lists, not all input arguments apply to each subroutine.

Note that you can specify optional arguments with the *keyword=argument* syntax.

The following list describes each argument:

opt indicates an options vector that specifies details of the optimization process, such as particular updating techniques and whether the objective function is to be maximized instead of minimized. See the section "Options Vector" on page 335 for details.

blc specifies a constraint matrix that defines lower and upper bounds for the n parameters in addition to general linear equality and inequality constraints. For details, see the section "Parameter Constraints" on page 333.

tc specifies a vector of thresholds that correspond to the termination criteria tested in each iteration. See the section "Termination Criteria" on page 339 for details.

par specifies a vector of control parameters that can be used to modify the algorithms if the default settings do not complete the optimization process successfully. For details, see the section "Control Parameters Vector" on page 346.

"ptit" specifies a module that replaces the subroutine used to print the iteration history and test the termination criteria. If the *"ptit"* module is specified, the matrix specified by the *tc* argument has no effect. See the section "Termination Criteria" on page 339 for details.

"grd" specifies a module that computes the gradient vector, $g = \nabla f$, at a given input point x. See the section "Objective Function and Derivatives" on page 326 for details.

"hes" specifies a module that computes the $n \times n$ Hessian matrix, $\mathbf{G} = \nabla^2 f$, at a given input point x. See the section "Objective Function and Derivatives" on page 326 for details.

"jac" specifies a module that computes the $m \times n$ Jacobian matrix, $\mathbf{J} = (\nabla f_i)$, of the m least squares functions at a given input point x. See the section "Objective Function and Derivatives" on page 326 for details.

"nlc" specifies a module that computes general equality and inequality constraints. This is the method by which nonlinear constraints must be specified. For details, see the section "Parameter Constraints" on page 333.

"jacnlc" specifies a module that computes the Jacobian matrix of first-order derivatives of the equality and inequality constraints specified by the NLC module. For details, see the section "Parameter Constraints" on page 333.

lin specifies the linear part of the quadratic optimization problem. See the section "NLPQUA Call" on page 850 for details.

The modules that can be used as input arguments for the subroutines (*"fun," "grd," "hes," "jac," "ptit," "nlc,"* and *"jacnlc"*) accept only a single input parameter $x = (x_1, \ldots, x_n)$. You can provide more input parameters for these modules by using the GLOBAL clause. See the section "Using the GLOBAL Clause" on page 71 for an example.

All the optimization subroutines return the following results:

- The scalar return code *rc* indicates the reason for the termination of the optimization process. A return code $rc > 0$ indicates a successful termination that corresponds to one of the specified termination criteria. A return code $rc < 0$ indicates unsuccessful termination—that is, that the result *xr* is unreliable. See the section "Definition of Return Codes" on page 326 for more details.
- The row vector *xr*, which has length n, contains the optimal point when $rc > 0$.

NLPCG Call

CALL NLPCG(*rc, xr, "fun", x0 < , opt > < , blc > < , tc > < , par > < , "ptit" > < , "grd" >* **);**

The NLPCG subroutine uses the conjugate gradient method to solve a nonlinear optimization problem.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPCG subroutine requires function and gradient calls; it does not need second-order derivatives. The gradient vector contains the first derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$g(x) = \nabla f(x) = \left(\frac{\partial f}{\partial x_j} \right)$$

If you do not specify a module with the *"grd"* argument, the first-order derivatives are approximated by finite difference formulas by using only function calls. The NLPCG algorithm can require many function and gradient calls, but it requires less memory than other subroutines for unconstrained optimization. In general, many iterations are needed to obtain a precise solution, but each iteration is computationally inexpensive. You can specify one of four update formulas for generating the conjugate directions with the fourth element of the *opt* input argument.

Value of *opt[4]*	Update Method
1	Automatic restart method of Powell (1977) and Beale (1972). This is the default.
2	Fletcher-Reeves update (Fletcher 1987)
3	Polak-Ribiere update (Fletcher 1987)
4	Conjugate-descent update of Fletcher (1987)

The NLPCG subroutine is useful for optimization problems with large n. For the unconstrained or boundary-constrained case, the NLPCG method requires less memory than other optimization methods. (The NLPCG method allocates memory proportional to n, whereas other methods allocate memory proportional to n^2.) During n successive iterations, uninterrupted by restarts or changes in the working set, the conjugate gradient algorithm computes a cycle of n conjugate search directions. In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step size α that satisfies the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit for the step size. You can specify other line-search algorithms with the fifth element of the *opt* argument.

For an example of the NLPCG subroutine, see the section "Constrained Betts Function" on page 319.

NLPDD Call

CALL NLPDD(*rc, xr, "fun", x0 < , opt> < , blc> < , tc> < , par> < , "ptit"> < , "grd"> ***);**

The NLPDD subroutine uses the double-dogleg method to solve a nonlinear optimization problem.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The double-dogleg optimization method combines the ideas of the quasi-Newton and trust-region methods. In each iteration, the algorithm computes the step, $s^{(k)}$, as a linear combination of the steepest descent or ascent search direction, $s_1^{(k)}$, and a quasi-Newton search direction, $s_2^{(k)}$, as follows:

$$s^{(k)} = \alpha_1 s_1^{(k)} + \alpha_2 s_2^{(k)}$$

The step $s^{(k)}$ must remain within a specified trust-region radius (Fletcher 1987). Hence, the NLPDD subroutine uses the dual quasi-Newton update but does not perform a line search. You can specify one of two update formulas with the fourth element of the *opt* input argument.

Value of *opt[4]*	Update Method
1	Dual BFGS update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual DFP update of the Cholesky factor of the Hessian matrix.

The double-dogleg optimization technique works well for medium to moderately large optimization problems, in which the objective function and the gradient are much faster to compute than the Hessian. The

implementation is based on Dennis and Mei (1979) and Gay (1983), but it is extended for boundary and linear constraints. The NLPDD subroutine generally needs more iterations than the techniques that require second-order derivatives (NLPTR, NLPNRA, and NLPNRR), but each of the NLPDD iterations is computationally inexpensive. Furthermore, the NLPDD subroutine needs only gradient calls to update the Cholesky factor of an approximate Hessian.

In addition to the standard iteration history, the NLPDD routine prints the following information:

- The heading *lambda* refers to the parameter λ of the double-dogleg step. A value of 0 corresponds to the full (quasi-) Newton step.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero.

The following statements invoke the NLPDD subroutine to solve the constrained Betts optimization problem (see the section "Constrained Betts Function" on page 319):

```
start F_BETTS(x);
   f = .01 * x[1] * x[1] + x[2] * x[2] - 100;
   return(f);
finish F_BETTS;

con = {  2 -50  .   .,
        50  50  .   .,
        10  -1  1  10};
x = {-1 -1};
opt = {0 1};
call nlpdd(rc, xres, "F_BETTS", x, opt, con);
```

Figure 25.270 shows the iteration history. The optimization converged after six iterations.

Figure 25.270 Constrained Optimization

Note: Initial point was changed to be feasible for boundary and linear constraints.

Double Dogleg Optimization

Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

Without Parameter Scaling

Gradient Computed by Finite Differences

Parameter Estimates	2
Lower Bounds	2
Upper Bounds	2
Linear Constraints	1

Optimization Start			
Active Constraints	0	**Objective Function**	-98.5376
Max Abs Gradient Element	2	**Radius**	1

Figure 25.270 *continued*

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Lambda	Slope of Search Direction
1	0	2	0	-99.54678	1.0092	0.1346	6.012	-1.805
2	0	3	0	-99.59120	0.0444	0.1279	0	-0.0228
3	0	5	0	-99.90252	0.3113	0.0624	0	-0.209
4	0	6	1	-99.96000	0.0575	0.00432	0	-0.0975
5	0	7	1	-99.96000	4.66E-6	0.000079	0	-458E-8
6	0	8	1	-99.96000	1.559E-9	0	0	-16E-10

Optimization Results			
Iterations	6	**Function Calls**	9
Gradient Calls	8	**Active Constraints**	1
Objective Function	-99.96	**Max Abs Gradient Element**	0
Slope of Search Direction	-1.56621E-9	**Radius**	1

GCONV convergence criterion satisfied.

The optimal value for the function is returned in the **`xres`** vector, which is displayed in Figure 25.271.

Figure 25.271 The Optimal Value

	xres
2	-9.612E-8

NLPFDD Call

CALL NLPFDD(*f, g, h, "fun", x0, < , par> < , "grd"*> **);**

The NLPFDD subroutine uses the finite-differences method to approximate derivatives.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPFDD subroutine can be used for the following tasks:

- If the module *"fun"* returns a scalar, the NLPFDD subroutine computes the function value *f*, the gradient vector *g*, and the Hessian matrix *h*, all evaluated at the point *x0*.
- If the module *"fun"* returns a column vector of *m* function values, the subroutine assumes that a least squares function is specified, and it computes the function vector *f*, the Jacobian matrix **J**, and the crossproduct of the Jacobian matrix $\mathbf{J}'\mathbf{J}$ at the point *x0*. In this case, you must set the first element of the *par* argument to *m*.

If any of the results cannot be computed, the subroutine returns a missing value for that result.

You can specify the following input arguments with the NLPFDD subroutine:

- The *"fun"* argument refers to a module that returns either a scalar value or a column vector of length m. This module returns the value of the objective function or, for least squares problems, the values of the m functions that the objective function comprises.
- The *x0* argument is a vector of length n that defines the point at which the functions and derivatives should be computed.
- The *par* argument is a vector that defines options and control parameters. The *par* argument in the NLPFDD call is different from the one used in the optimization subroutines.
- The *"grd"* argument is optional and refers to a module that returns a vector that defines the gradient of the function at *x0*. If the *"fun"* argument returns a vector of values instead of a scalar, the *"grd"* argument is ignored.

If the *"fun"* module returns a scalar, the subroutine returns the following values:

- *f* is the value of the function at the point *x0*.
- *g* is a vector that contains the value of the gradient at the point *x0*. If you specify the *"grd"* argument, the gradient is computed from that module. Otherwise, the approximate gradient is computed by a finite difference approximation by using calls of the function module in a neighborhood of *x0*.
- *h* is a matrix that contains a finite difference approximation of the value of the Hessian at the point *x0*. If you specify the *"grd"* argument, the Hessian is computed by calls of that module in a neighborhood of *x0*. Otherwise, it is computed by calls of the function module in a neighborhood of *x0*.

If the *"fun"* module returns a vector, the subroutine returns the following values:

- *f* is a vector that contains the values of the m functions that comprise objective function at the point *x0*.
- *g* is the $m \times n$ Jacobian matrix $\mathbf{J}$, which contains the first-order derivatives of the functions with respect to the parameters, evaluated at *x0*. It is computed by finite difference approximations in a neighborhood of *x0*.
- *h* is the $n \times n$ crossproduct of the Jacobian matrix, $\mathbf{J}^T\mathbf{J}$. It is computed by finite difference approximations in a neighborhood of *x0*.

The *par* argument is a vector of length 3.

- *par[1]* corresponds to the *opt[1]* argument in the optimization subroutines. This argument is relevant only to least squares optimization methods, in which case it specifies the number of functions returned by the module *"fun"*. If *par[1]* is missing or is smaller than 1, it is set to 1.
- *par[2]* corresponds to the *opt[8]* argument in the optimization subroutines. It determines what type of approximation is to be used and how the finite difference interval, h, is to be computed. See the section "Finite-Difference Approximations of Derivatives" on page 331 for details.
- *par[3]* corresponds to the *par[8]* argument in the optimization subroutines. It specifies the number of accurate digits in evaluating the objective function. The default is $-\log_{10}(\epsilon)$, where ϵ is the machine precision.

If you specify a missing value in the *par* argument, the default value is used.

The NLPFDD subroutine is particularly useful for checking your analytical derivative specifications of the *"grd"*, *"hes"*, and *"jac"* modules. You can compare the results of the modules with the finite difference approximations of the derivatives of *f* at the point *x0* to verify your specifications.

In the unconstrained Rosenbrock problem (see the section "Unconstrained Rosenbrock Function" on page 315), the objective function is

$$f(x) = 50(x_2 - x_1^2)^2 + \frac{1}{2}(1 - x_1)^2$$

The gradient and the Hessian are

$$g'(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 200x_1^3 - 200x_1x_2 + x_1 - 1 \\ -100x_1^2 + 100x_2 \end{bmatrix}$$

$$\mathbf{H}(x,y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 600x_1^2 - 200x_2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix}$$

At the point $x = (2, 7)$, these matrices evaluate to

$$g'(2,7) = \begin{bmatrix} -1199 \\ 300 \end{bmatrix}$$

$$\mathbf{H}(2,7) = \begin{bmatrix} 1001 & -400 \\ -400 & 100 \end{bmatrix}$$

The following statements define the Rosenbrock function and use the NLPFDD call to compute the gradient and the Hessian:

```
start F_ROSEN(x);
   y1 = 10 * (x[2] - x[1] * x[1]);
   y2 =  1 - x[1];
   f  = 0.5 * (y1 * y1 + y2 * y2);
   return(f);
finish F_ROSEN;
x = {2 7};
call nlpfdd(crit, grad, hess, "F_ROSEN", x);
print grad;
print hess;
```

Figure 25.272 Gradient and Hessian at a Point

grad	
-1199	300.00001

hess	
1000.9998	-400.0018
-400.0018	99.999993

If the Rosenbrock problem is considered from a least squares perspective, the two functions are

$$\begin{aligned} f_1(x) &= 10(x_2 - x_1^2) \\ f_2(x) &= 1 - x_1 \end{aligned}$$

The Jacobian and the crossproduct of the Jacobian are

$$\begin{aligned} \mathbf{J} &= \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \end{bmatrix} \\ \mathbf{J}^T\mathbf{J} &= \begin{bmatrix} 400x_1^2 + 1 & -200x_1 \\ -200x_1 & 100 \end{bmatrix} \end{aligned}$$

At the point $x = (2, 7)$, these matrices evaluate to

$$\begin{aligned} \mathbf{J}(2,7) &= \begin{bmatrix} -40 & 10 \\ -1 & 0 \end{bmatrix} \\ \mathbf{J}^T\mathbf{J}|_{(2,7)} &= \begin{bmatrix} 1601 & -400 \\ -400 & 100 \end{bmatrix} \end{aligned}$$

The following statements define the Rosenbrock problem in a least squares framework and use the NLPFDD call to compute the Jacobian and the crossproduct matrix. Since the value of the PARMS variable, which is used for the *par* argument, is 2, the NLPFDD subroutine allocates memory for a least squares problem with two functions, $f_1(x)$ and $f_2(x)$.

```
start F_ROSEN(x);
   y = j(2, 1, 0);
   y[1] = 10 * (x[2] - x[1] * x[1]);
   y[2] =  1 - x[1];
   return(y);
finish F_ROSEN;
x     = {2 7};
parms = 2;
call nlpfdd(fun, jac, crpj, "F_ROSEN", x, parms);
print jac;
print crpj;
```

Figure 25.273 Jacobian and Crossproduct Matrix at a Point

jac	
-40	10
-1	0

crpj	
1601	-400
-400	100

NLPFEA Call

CALL NLPFEA(*xr, x0, blc* < *, par*> **);**

The NLPFEA subroutine computes feasible points subject to constraints.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPFEA subroutine tries to compute a point that is feasible subject to a set of boundary and linear constraints. You can specify boundary and linear constraints that define an empty feasible region, in which case the subroutine returns missing values.

You can specify the following input arguments with the NLPFEA subroutine:

- *x0* is a row vector that defines the coordinates of a point that is not necessarily feasible for a set of linear and boundary constraints.
- *blc* is an $m \times n$ matrix that defines a set of m boundary and linear constraints. See the section "Parameter Constraints" on page 333 for details.
- *par* is a vector of length two. The argument is different from the one used in the optimization subroutines. The first element sets the LCEPS parameter, which controls how precisely the returned point must satisfy the constraints. The second element sets the LCSING parameter, which specifies the criterion for deciding when constraints are considered linearly dependent. For details, see the section "Control Parameters Vector" on page 346.

The NLPFEA subroutine returns the *xr* argument. The result is a vector that contains either the n coordinates of a feasible point, which indicates that the subroutine was successful, or missing values, which indicates that the subroutine could not find a feasible point.

The following statements call the NLPFEA subroutine with the constraints from the Betts problem (see the section "Constrained Betts Function" on page 319) and an initial infeasible point $x_0 = (-17, -61)$. The subroutine returns the feasible point $(2, -50)$ as the vector XFEAS.

```
con = {  2 -50  .   .,
        50  50  .   .,
        10  -1  1  10};
x = {-17. -61};
call nlpfea(xfeas, x, con);
print xfeas;
```

Figure 25.274 Feasible Point

xfeas
6.8 -40

NLPHQN Call

CALL NLPHQN(*rc, xr, "fun", x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"jac"*> **);**

The NLPHQN subroutine uses a hybrid quasi-Newton least squares method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPHQN subroutine uses one of the Fletcher and Xu (1987) hybrid quasi-Newton methods. Refer also to Al-Baali and Fletcher (1985) and Al-Baali and Fletcher (1986). In each iteration, the subroutine uses a criterion to decide whether a Gauss-Newton or a dual quasi-Newton search direction is appropriate. You can choose one of three criteria (HY1, HY2, or HY3) proposed by Fletcher and Xu (1987) with the sixth element of the *opt* vector. The default is HY2. The subroutine computes the crossproduct Jacobian (for the Gauss-Newton step), updates the Cholesky factor of an approximate Hessian (for the quasi-Newton step), and performs a line search to compute an approximate minimum along the search direction. The default line-search technique used by the NLPHQN method is designed for least squares problems ((Lindström and Wedin 1984) and (Al-Baali and Fletcher 1986)), but you can specify a different line-search algorithm with the fifth element of the *opt* argument. See the section "Options Vector" on page 335 for details.

You can specify two update formulas with the fourth element of the *opt* argument as indicated in the following table.

Value of *opt[4]*	Update Method
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.

The NLPHQN subroutine needs approximately the same amount of working memory as the NLPLM subroutine, and in most applications, the latter seems to be superior. Hence, the NLPHQN method is recommended only when the NLPLM method encounters problems.

NOTE: In least squares subroutines, you must set the first element of the *opt* vector to m, the number of functions.

In addition to the standard iteration history, the NLPHQN subroutine prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that, on the basis of the Fletcher and Xu (1987) criterion, the subroutine used a Gauss-Newton search direction instead of a quasi-Newton search direction.
- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The following statements use the NLPHQN call to solve the unconstrained Rosenbrock problem (see the section "Unconstrained Rosenbrock Function" on page 315).

```
title "Test of NLPHQN subroutine: No Derivatives";
start F_ROSEN(x);
   y = j(1, 2, 0);
   y[1] = 10 * (x[2] - x[1] * x[1]);
   y[2] =  1 - x[1];
   return(y);
finish F_ROSEN;

x = {-1.2 1};
opt = {2 2};
call nlphqn(rc, xr, "F_ROSEN", x, opt);
```

Figure 25.275 Optimization Results

Test of NLPHQN subroutine: No Derivatives

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	-1.200000	-107.799999
2	X2	1.000000	-44.000000

Value of Objective Function = 12.1

Test of NLPHQN subroutine: No Derivatives

Hybrid Quasi-Newton LS Minimization

Dual Broyden - Fletcher - Goldfarb - Shanno Update (DBFGS)

Version HY2 of Fletcher & Xu (1987)

Gradient Computed by Finite Differences

CRP Jacobian Computed by Finite Differences

Parameter Estimates	2
Functions (Observations)	2

Optimization Start

Active Constraints	0	Objective Function	12.1
Max Abs Gradient Element	107.7999987		

Figure 25.275 *continued*

Iteration		Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
1		0	3	0	7.22423	4.8758	56.9322	0.0616	-628.8
2	*	0	5	0	0.97090	6.2533	2.3017	0.266	-14.448
3	*	0	7	0	0.81911	0.1518	3.7839	0.119	-1.942
4		0	9	0	0.69103	0.1281	5.5103	2.000	-0.144
5		0	19	0	0.47345	0.2176	8.8638	11.854	-0.194
6	*	0	21	0	0.35906	0.1144	9.8735	0.253	-0.947
7	*	0	22	0	0.23342	0.1256	10.1490	0.398	-0.718
8	*	0	24	0	0.14799	0.0854	11.6248	1.346	-0.467
9	*	0	26	0	0.00948	0.1385	2.6276	1.443	-0.296
10	*	0	28	0	1.98904E-6	0.00948	0.00609	0.938	-0.0190
11	*	0	30	0	7.0819E-10	1.988E-6	0.000749	1.003	-398E-8
12	*	0	32	0	2.0242E-21	7.08E-10	1.77E-10	1.000	-14E-10

Optimization Results			
Iterations	12	Function Calls	33
Jacobian Calls	13	Gradient Calls	19
Active Constraints	0	Objective Function	2.024176E-21
Max Abs Gradient Element	1.771198E-10	Slope of Search Direction	-1.416374E-9

ABSGCONV convergence criterion satisfied.

Test of NLPHQN subroutine: No Derivatives

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function
1	X1	1.000000	1.771198E-10
2	X2	1.000000	-1.19804E-10

Value of Objective Function = 2.024176E-21

NLPLM Call

CALL NLPLM(*rc, xr, "fun", x0, opt* < *, blc* > < *, tc* > < *, par* > < *, "ptit"* > < *, "jac"* > **);**

The NLPLM subroutine uses the Levenberg-Marquardt least squares method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPLM subroutine uses the Levenberg-Marquardt method, which is an efficient modification of the trust-region method for nonlinear least squares problems and is implemented as in Moré (1978). This is the recommended algorithm for small to medium least squares problems. Large least squares problems

can often be processed more efficiently with other subroutines, such as the NLPCG subroutine and the NLPQN subroutine. In each iteration, the NLPLM subroutine solves a quadratically constrained quadratic minimization problem that restricts the step to the boundary or interior of an n-dimensional elliptical trust region.

The m functions $f_1(x), \ldots, f_m(x)$ are computed by the module specified with the *"fun"* module argument. The $m \times n$ Jacobian matrix, $\mathbf{J}$, contains the first-order derivatives of the m functions with respect to the n parameters, as follows:

$$\mathbf{J}(x) = (\nabla f_1, \ldots, \nabla f_m) = \left(\frac{\partial f_i}{\partial x_j} \right)$$

You can specify $\mathbf{J}$ with the *"jac"* module argument; otherwise, the subroutine computes it with finite difference approximations. In each iteration, the subroutine computes the crossproduct of the Jacobian matrix, $\mathbf{J}^T\mathbf{J}$, to be used as an approximate Hessian.

NOTE: In least squares subroutines, you must set the first element of the *opt* vector to m, the number of functions.

In addition to the standard iteration history, the NLPLM subroutine also prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that the computed Hessian approximation was singular and had to be ridged with a positive value.
- The heading *lambda* represents the Lagrange multiplier, λ. This has a value of zero when the optimum of the quadratic function approximation is inside the trust region, in which case a trust-region-scaled Newton step is performed. It is greater than zero when the optimum is at the boundary of the trust region, in which case the scaled Newton step is too long to fit in the trust region and a quadratically constrained optimization is done. Large values indicate optimization difficulties, and as in Gay (1983), a negative value indicates the special case of an indefinite Hessian matrix.
- The heading *rho* refers to ρ, the ratio between the achieved and predicted difference in function values. Values that are much smaller than 1 indicate optimization difficulties. Values close to or larger than 1 indicate that the trust region radius can be increased.

See the section "Unconstrained Rosenbrock Function" on page 315 for an example that uses the NLPLM subroutine to solve the unconstrained Rosenbrock problem.

NLPNMS Call

CALL NLPNMS(*rc, xr, "fun", x0* **<** *, opt* **>** **<** *, blc* **>** **<** *, tc* **>** **<** *, par* **>** **<** *, "ptit"* **>** **<** *, "nlc"* **>);**

The NLPNMS subroutine uses the Nelder-Mead simplex method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The Nelder-Mead simplex method is one of the subroutines that can solve optimization problems with nonlinear constraints. It does not use any derivatives, and it does not assume that the objective function has continuous derivatives. However, the objective function must be continuous. The NLPNMS technique uses a large number of function calls, and it can be unable to generate precise results when $n > 40$.

The NLPNMS subroutine uses the following simplex algorithms:

- For unconstrained or only boundary-constrained problems, the original Nelder-Mead simplex algorithm is implemented and extended to boundary constraints. This algorithm does not compute the objective for infeasible points, and it is invoked if the *"nlc"* module argument is not specified and the *blc* argument contains at most two rows (corresponding to lower and upper bounds).
- For linearly or nonlinearly constrained problems, a slightly modified version of Powell's constrained optimization by linear approximations (COBYLA) implementation is used (Powell 1992). 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 (Powell 1992) is a sequential trust-region algorithm that tries to maintain a regularly shaped simplex throughout the iterations. The algorithm uses a monotone-decreasing radius, ρ, of a spheric trust region. The modification implemented in the NLPNMS call permits an increase of the trust-region radius ρ in special situations. A sequence of iterations is performed with a constant trust-region radius ρ until the computed function reduction is much less than the predicted reduction. Then, the trust-region radius ρ is reduced. The trust-region radius is increased only if the computed function reduction is relatively close to the predicted reduction and if the simplex is well-shaped. The start radius, ρ_{beg}, can be specified with the second element of the *par* argument, and the final radius, ρ_{end}, can be specified with the ninth element of the *tc* argument. Convergence to small values of ρ_{end}, or high-precision convergence, can require many calls of the function and constraint modules and can result in numerical problems. The main reasons for the slow convergence of the COBYLA algorithm are as follows:

- Linear approximations of the objective and constraint functions are used locally.
- Maintaining the regularly shaped simplex and not adapting its shape to nonlinearities yields very small simplexes for highly nonlinear functions, such as fourth-order polynomials.

To allocate memory for the vector returned by the *"nlc"* module argument, you must specify the total number of nonlinear constraints with the tenth element of the *opt* argument. If any of the constraints are equality constraints, the number of equality constraints must be specified by the eleventh element of the *opt* argument. See the section "Parameter Constraints" on page 333 for details.

For more information about the special sets of termination criteria used by the NLPNMS algorithms, see the section "Termination Criteria" on page 339.

In addition to the standard iteration history, the NLPNMS subroutine prints the following information. For unconstrained or boundary-constrained problems, the subroutine also prints the following:

- *difcrit*, which, in this subroutine, refers to the difference between the largest and smallest function values of the $n + 1$ simplex vertices
- *std*, which is the standard deviation of the function values of the simplex vertices

- *deltax*, which is the vertex length of a restarted simplex. If there are convergence problems, the algorithm restarts the iteration process with a simplex of smaller vertex length.
- *size*, which is the average L_1 distance of the simplex vertex with the smallest function value to the other simplex vertices

For linearly and nonlinearly constrained problems, the subroutine prints the following:

- *conmax* is the maximum constraint violation.
- *meritf* is the value of the merit function, Φ.
- *difmerit* is the difference between adjacent values of the merit function.
- ρ is the trust-region radius.

The following statements uses the NLPNMS call to solve the Rosen-Suzuki problem (see the section "Rosen-Suzuki Problem" on page 321), which has three nonlinear constraints. Figure 25.276 is a partial listing of the output:

```
start F_HS43(x);
   f = x*x` + x[3]*x[3] - 5*(x[1] + x[2]) - 21*x[3] + 7*x[4];
   return(f);
finish F_HS43;
start C_HS43(x);
   c = j(3,1,0.);
   c[1] = 8 - x*x` - x[1] + x[2] - x[3] + x[4];
   c[2] = 10 - x*x` - x[2]*x[2] - x[4]*x[4] + x[1] + x[4];
   c[3] = 5 - 2.*x[1]*x[1] - x[2]*x[2] - x[3]*x[3]
            - 2.*x[1] + x[2] + x[4];
   return(c);
finish C_HS43;

x = j(1, 4, 1);
opt = j(1, 11, .);
opt[2] = 3; opt[10] = 3; opt[11] = 0;
call nlpnms(rc, xres, "F_HS43", x, opt, , , , , "C_HS43");
```

Figure 25.276 Nelder-Mead Simplex Optimization

Optimization Start Parameter Estimates		
N	Parameter	Estimate
1	X1	1.000000
2	X2	1.000000
3	X3	1.000000
4	X4	1.000000

Figure 25.276 *continued*

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

Iteration	Restarts	Function Calls	Objective Function	Maximum Constraint Violation	Merit Function	Merit Function Change	Ratio Between Actual and Predicted Change
1	0	12	-52.80342	4.3411	-42.3031	12.803	1.000
2	0	17	-39.51475	0.0227	-39.3797	-2.923	0.250
3	0	53	-44.02098	0.00949	-43.9727	4.593	0.0625
4	0	62	-44.00214	0.000833	-43.9977	0.0249	0.0156
5	0	72	-44.00009	0.000033	-43.9999	0.00226	0.0039
6	0	79	-44.00000	1.783E-6	-44.0000	0.00007	0.0010
7	0	90	-44.00000	1.363E-7	-44.0000	1.74E-6	0.0002
8	0	94	-44.00000	1.543E-8	-44.0000	5.33E-7	0.0001

Figure 25.276 *continued*

Optimization Results			
Iterations	8	**Function Calls**	95
Restarts	0	**Objective Function**	-44.00000003
Maximum Constraint Violation	1.5430589E-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 *?*

The 2 nonlinear constraints which are marked with *?* are not satisfied at the accuracy specified by the LCEPSILON= option. However, the default value of this option seems to be too strong to be applied to nonlinear constraints.

NLPNRA Call

CALL NLPNRA(*rc, xr, "fun", x0 < , opt> < , blc> < , tc> < , par> < , "ptit"> < , "grd"> < , "hes"> ***);**

The NLPNRA subroutine uses the Newton-Raphson method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPNRA algorithm uses a pure Newton step at each iteration when both the Hessian is positive definite and the Newton step successfully reduces the value of the objective function. Otherwise, it performs a combination of ridging and line-search to compute successful steps. If the Hessian is not positive definite, a multiple of the identity matrix is added to the Hessian matrix to make it positive definite (Eskow and Schnabel 1991).

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$. It requires continuous first- and second-order derivatives of the objective function inside the feasible region. If second-order derivatives are computed efficiently and precisely, the NLPNRA method does not need many function, gradient, and Hessian calls, and it can perform well for medium to large problems.

Using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and can contain significant rounding errors. If you use the *"grd"* input argument to specify a module that computes first-order derivatives analytically, you can reduce drastically the computation time for numerical second-order derivatives. The computation of the finite difference approximation for the Hessian matrix generally uses only n calls of the module that specifies the gradient.

In each iteration, a line search is done along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation. You can specify other line-search algorithms with the fifth element of the *opt* argument. See the section "Options Vector" on page 335 for details.

In unconstrained and boundary constrained cases, the NLPNRA algorithm can take advantage of diagonal or sparse Hessian matrices that are specified by the input argument *"hes"*. To use sparse Hessian storage, the value of the ninth element of the *opt* argument must specify the number of nonzero Hessian elements returned by the Hessian module. See the section "Objective Function and Derivatives" on page 326 for more details.

In addition to the standard iteration history, the NLPNRA subroutine prints the following information:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The following statements invoke the NLPNRA subroutine to solve the constrained Betts optimization problem (see the section "Constrained Betts Function" on page 319). The iteration history follows.

```
start F_BETTS(x);
   f = .01 * x[1] * x[1] + x[2] * x[2] - 100;
   return(f);
finish F_BETTS;

con = {  2 -50  .   .,
        50  50  .   .,
        10  -1  1  10};
x = {-1 -1};
opt = {0 2};
call nlpnra(rc, xres, "F_BETTS", x, opt, con);
```

Figure 25.277 Newton-Raphson Optimization

Note: Initial point was changed to be feasible for boundary and linear constraints.

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound Constraint
1	X1	6.800000	0.136000	2.000000	50.000000
2	X2	-1.000000	-2.000000	-50.000000	50.000000

Figure 25.277 *continued*

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		

Iteration		Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
1		0	2	0	-98.81551	0.2779	1.8000	0.100	-2.925
2	*	0	3	0	-99.40840	0.5929	1.2713	0.294	-2.365
3	*	0	4	1	-99.87460	0.4662	0.5845	0.540	-1.182
4		0	5	1	-99.96000	0.0854	0.000025	1.000	-0.171
5		0	6	1	-99.96000	1.54E-10	0	1.000	-31E-11

Optimization Results			
Iterations	5	**Function Calls**	7
Hessian Calls	6	**Active Constraints**	1
Objective Function	-99.96	**Max Abs Gradient Element**	0
Slope of Search Direction	-3.07388E-10	**Ridge**	0

GCONV convergence criterion satisfied.

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	**X1**	2.000000	0.040000	Lower BC
2	**X2**	-4.860653E-9	0	

Value of Objective Function = -99.96

Linear Constraints Evaluated at Solution

1 10.00000 = -10.0000 + 10.0000 * X1 - 1.0000 * X2

NLPNRR Call

CALL NLPNRR(*rc, xr, "fun", x0* < , *opt*> < , *blc*> < , *tc*> < , *par*> < , *"ptit"*> < , *"grd"*> < , *"hes"*> **);**

The NLPNRR subroutine uses a Newton-Raphson ridge method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPNRR algorithm uses a pure Newton step when both the Hessian is positive definite and the Newton step successfully reduces the value of the objective function. Otherwise, a multiple of the identity matrix is added to the Hessian matrix.

The subroutine uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and the Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$. It requires continuous first- and second-order derivatives of the objective function inside the feasible region.

Note that using only function calls to compute finite difference approximations for second-order derivatives can be computationally very expensive and can contain significant rounding errors. If you use the *"grd"* input argument to specify a module that computes first-order derivatives analytically, you can reduce drastically the computation time for numerical second-order derivatives. The computation of the finite difference approximation for the Hessian matrix generally uses only n calls of the module that specifies the gradient.

The NLPNRR method performs well for small- to medium-sized problems, and it does not need many function, gradient, and Hessian calls. However, if the gradient is not specified analytically by using the *"grd"* module argument, or if the computation of the Hessian module specified with the *"hes"* argument is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms might be more efficient.

In addition to the standard iteration history, the NLPNRR subroutine prints the following information:

- The heading *ridge* refers to the value of the nonnegative ridge parameter. A value of zero indicates that a Newton step is performed. A value greater than zero indicates either that the Hessian approximation is zero or that the Newton step fails to reduce the optimization criterion. A large value can indicate optimization difficulties.
- The heading *rho* refers to ρ, the ratio of the achieved difference in function values and the predicted difference, based on the quadratic function approximation. A value that is much smaller than 1 indicates possible optimization difficulties.

The following statements invoke the NLPNRR subroutine to solve the constrained Betts optimization problem (see the section "Constrained Betts Function" on page 319). The iteration history follows.

```
start F_BETTS(x);
   f = .01 * x[1] * x[1] + x[2] * x[2] - 100;
   return(f);
finish F_BETTS;

con = {  2 -50  .   .,
        50  50  .   .,
        10  -1  1  10};
x = {-1 -1};
opt = {0 2};
call nlpnrr(rc, xres, "F_BETTS", x, opt, con);
```

Figure 25.278 Newton-Raphson Optimization

Note: Initial point was changed to be feasible for boundary and linear constraints.

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound Constraint
1	X1	6.800000	0.136000	2.000000	50.000000
2	X2	-1.000000	-2.000000	-50.000000	50.000000

Value of Objective Function = -98.5376

Linear Constraints

1 59.00000 : 10.0000 <= + 10.0000 * X1 - 1.0000 * X2

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		

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Ridge	Ratio Between Actual and Predicted Change
1	0	2	1	-99.87337	1.3358	0.5887	0	0.706
2	0	3	1	-99.96000	0.0866	0.000040	0	1.000
3	0	4	1	-99.96000	4.07E-10	0	0	1.014

Optimization Results

Iterations	3	**Function Calls**	5
Hessian Calls	4	**Active Constraints**	1
Objective Function	-99.96	**Max Abs Gradient Element**	0
Ridge	0	**Actual Over Pred Change**	1.0135158294

GCONV convergence criterion satisfied.

Figure 25.278 *continued*

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	X1	2.000000	0.040000	Lower BC
2	X2	0.000000134	0	

Value of Objective Function = -99.96

Linear Constraints Evaluated at Solution

1 10.00000 = -10.0000 + 10.0000 * X1 - 1.0000 * X2

NLPQN Call

CALL NLPQN(*rc, xr, "fun", x0* **<** *, opt* **>** **<** *, blc* **>** **<** *, tc* **>** **<** *, par* **>** **<** *, "ptit"* **>** **<** *, "grd"* **>** **<** *, "nlc"* **>** **<** *, "jacnlc"* **>** **);**

The NLPQN subroutine uses a quasi-Newton method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPQN subroutine uses (dual) quasi-Newton optimization techniques, and it is one of the two available subroutines that can solve problems with nonlinear constraints. These techniques work well for medium to moderately large optimization problems where the objective function and the gradient are much faster to compute than the Hessian matrix. The NLPQN subroutine does not need to compute second-order derivatives, but it generally requires more iterations than the techniques that compute second-order derivatives.

The two categories of problems solved by the NLPQN subroutine are unconstrained or linearly constrained problems and nonlinearly constrained problems. Unconstrained or linearly constrained problems do not use the *"nlc"* or *"jacnlc"* module arguments, whereas nonlinearly constrained problems use the arguments to specify the nonlinear constraints and the Jacobian matrix of their first-order derivatives, respectively.

The type of optimization problem specified determines the algorithm that the subroutine invokes. The algorithms are very different, and they use different sets of termination criteria. For more details, see the section "Termination Criteria" on page 339.

Unconstrained or Linearly Constrained Quasi-Newton Optimization

The NLPQN subroutine invokes this algorithm if you do not specify the *"nlc"* argument. Using the fourth element of the *opt* argument, you can specify two update formulas for either the original quasi-Newton algorithm or the dual quasi-Newton algorithm, as indicated in the following table:

Value of *opt[4]*	Update Method
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.
3	Original Broyden, Fletcher, Goldfarb, and Shanno (BFGS) update of the inverse Hessian matrix.
4	Original Davidon, Fletcher, and Powell (DFP) update of the inverse Hessian matrix.

In each iteration, a line search is performed along the search direction to find an approximate optimum of the objective function. The default line-search method uses quadratic interpolation and cubic extrapolation to obtain a step size that satisfies the Goldstein conditions. One of the Goldstein conditions can be violated if the feasible region defines an upper limit of the step size. Violating the left-side Goldstein condition can affect the positive definiteness of the quasi-Newton update. In these cases, either the update is skipped or the iterations are restarted with an identity matrix that results in the steepest descent or ascent search direction. You can specify line-search algorithms different from the default method with the fifth element of the *opt* argument.

The following statements invoke the NLPQN subroutine to solve the Rosenbrock problem (see the section "Unconstrained Rosenbrock Function" on page 315):

```
start F_ROSEN(x);
   y1 = 10 * (x[2] - x[1] * x[1]);
   y2 =  1 - x[1];
   f  = 0.5 * (y1 * y1 + y2 * y2);
   return(f);
finish F_ROSEN;
x = {-1.2 1};
opt = {0 2 . 2};
call nlpqn(rc, xr, "F_ROSEN", x, opt);
```

Since *opt[4]*=2, the DDFP update is performed. The gradient is approximated by finite differences since no module is specified that computes the first-order derivatives. Part of the iteration history follows. In addition to the standard iteration history, the NLPQN subroutine prints the following information for unconstrained or linearly constrained problems:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

Figure 25.279 Quasi-Newton Optimization

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective 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		

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
1	0	4	0	2.06405	10.0359	0.7917	0.0340	-628.8
2	0	7	0	1.92035	0.1437	8.6301	6.557	-0.0363
3	0	10	0	1.78089	0.1395	11.0943	8.193	-0.0288
4	0	13	0	1.33331	0.4476	7.6069	33.376	-0.0269
5	0	17	0	1.13400	0.1993	0.9386	15.438	-0.0260
6	0	22	0	0.93915	0.1948	3.5290	11.537	-0.0233
7	0	24	0	0.84821	0.0909	4.8308	8.124	-0.0193
8	0	30	0	0.54334	0.3049	4.1770	35.143	-0.0186
9	0	32	0	0.46593	0.0774	0.9479	8.708	-0.0178
10	0	37	0	0.35322	0.1127	2.5981	10.964	-0.0147
11	0	40	0	0.26381	0.0894	3.3028	13.590	-0.0121
12	0	41	0	0.20282	0.0610	0.6451	10.000	-0.0116
13	0	46	0	0.11714	0.0857	1.6603	11.395	-0.0102
14	0	51	0	0.07149	0.0456	2.4050	11.559	-0.0074
15	0	53	0	0.04746	0.0240	0.5628	6.868	-0.0071
16	0	58	0	0.02759	0.0199	1.3282	5.365	-0.0055
17	0	60	0	0.01625	0.0113	1.9246	5.882	-0.0035
18	0	62	0	0.00475	0.0115	0.6357	8.068	-0.0032
19	0	66	0	0.00167	0.00307	0.4810	2.336	-0.0022
20	0	70	0	0.0005952	0.00108	0.6043	3.287	-0.0006
21	0	72	0	0.0000771	0.000518	0.0289	2.329	-0.0004
22	0	76	0	1.92115E-6	0.000075	0.0365	1.772	-0.0001
23	0	78	0	2.39905E-8	1.897E-6	0.00158	1.159	-331E-8
24	0	80	0	5.0935E-11	2.394E-8	0.000016	0.967	-46E-9
25	0	119	0	3.9538E-11	1.14E-11	7.963E-7	1.061	-19E-13

Figure 25.279 *continued*

Optimization Results			
Iterations	25	**Function Calls**	120
Gradient Calls	107	**Active Constraints**	0
Objective Function	3.953809E-11	**Max Abs Gradient Element**	7.9633202E-7
Slope of Search Direction	-1.88008E-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 = 3.953809E-11

Nonlinearly Constrained Quasi-Newton Optimization

The algorithm used for nonlinearly constrained quasi-Newton optimization is an efficient modification of Powell's variable metric constrained watchdog (VMCWD) algorithm (Powell 1978a, 1982a). A similar but older algorithm (VF02AD) is part of the Harwell library. Both the VMCWD and VF02AD algorithms use Fletcher's VE02AD algorithm, which is also part of the Harwell library, for positive definite quadratic programming. This NLPQN implementation uses a quadratic programming subroutine that updates and downdates the Cholesky factor when the active set changes (Gill et al. 1984). The nonlinear NLPQN algorithm is not a feasible point algorithm, and the value of the objective function is not required to decrease monotonically. Instead, the algorithm tries to reduce a linear combination of objective function and constraint violations.

The following are similarities and differences between this algorithm and Powell's VMCWD algorithm:

- You can use the sixth element of the *opt* argument to modify the algorithm used by the NLPQN subroutine. If you specify *opt[6]*=2, which is the default, the evaluation of the Lagrange vector μ is performed the same way as described in Powell (1982b). However, 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 (1978b) is performed.

- Instead of updating an approximate Hessian matrix, this algorithm uses the dual BFGS or dual DFP update that updates the Cholesky factor of an approximate Hessian. If the condition of the updated matrix gets too bad, the algorithm restarts with a positive diagonal matrix. At the end of the first iteration after each restart, the Cholesky factor is scaled.

- The Cholesky factor is loaded into the quadratic programming subroutine, which ensures positive definiteness of the problem. During the quadratic programming step, the Cholesky factor of the projected Hessian matrix $Z_k^T G Z_k$ is updated simultaneously with QT decomposition when the active set changes. See Gill et al. (1984) for more information.

- The line-search strategy is very similar to that of Powell's algorithm, but this algorithm does not call for derivatives during the line search. Therefore, this algorithm generally needs fewer derivative calls than function calls, whereas the VMCWD algorithm always requires the same number of derivative calls as function calls. Also, Powell's line-search method sometimes uses steps that are too long during the early iterations. In those cases, you can use the second element of the *par* argument to restrict the step length α in the first five iterations. See the section "Control Parameters Vector" on page 346 for more details.
- The watchdog strategy is also similar to that of Powell's algorithm. However, this algorithm does not return automatically after a fixed number of iterations to a previous, more optimal point. A return to such a point is further delayed if the observed function reduction is close to the expected function reduction of the quadratic model.
- Although Powell's termination criterion, the FTOL2 criterion, can still be used, the NLPQN implementation uses, by default, two other termination criteria (GTOL and ABSGTOL).

This algorithm is automatically invoked if the *"nlc"* argument is specified. The module specified with the *"nlc"* argument must return a vector of length n_c, where n_c is the total number of constraints. Letting n_e be the number of equality constraints, the constraints must be of the following form:

$$\begin{aligned} c_i(x) &= 0, \quad i = 1, \ldots, n_c \\ c_i(x) &\geq 0, \quad i = n_e + 1, \ldots, n_c \end{aligned}$$

The first n_c elements of the returned vector contain the c_i for the equality constraints, and the remaining elements contain the c_i for the inequality constraints.

NOTE: You must specify the total number of constraints with the tenth element of the *opt* argument, and if there are any equality constraints, you must specify their number, n_e, with the eleventh element of the *opt* argument.

The nonlinear NLPQN algorithm requires the Jacobian matrix of the first-order derivatives of the n_c constraints returned by the module specified by the *"nlc"* argument. You can provide these derivatives by specifying a module with the *"jacnlc"* argument. This module must return the Jacobian matrix **J** of first-order partial derivatives. That is, **J** is an $n_c \times n$ matrix such that the entry in the *i*th row and *j*th column is given by

$$\mathbf{J}(i, j) = \frac{\partial c_i}{\partial x_j}$$

If you specify an *"nlc"* module without specifying a *"jacnlc"* argument, finite difference approximations of the first-order derivatives of the constraints are used. You can use the ninth element of the *par* argument to specify the number of accurate digits used in evaluating the constraints.

You can specify two update formulas with the fourth element of the *opt* argument as indicated in the following table:

Value of *opt[4]*	Update Method
1	Dual Broyden, Fletcher, Goldfarb, and Shanno (DBFGS) update of the Cholesky factor of the Hessian matrix. This is the default.
2	Dual Davidon, Fletcher, and Powell (DDFP) update of the Cholesky factor of the Hessian matrix.

This algorithm uses its own line-search technique. None of the options and parameters that control the line search in the other algorithms apply in the nonlinear NLPQN algorithm, with the exception of the second element of the *par* vector, which can be used to restrict the length of the step size in the first five iterations.

See Example 17.8 for an example where you need to specify a value for the second element of the *par* argument. The values of the fourth, fifth, and sixth elements of the *par* vector, which control the processing of linear and boundary constraints, are valid only for the quadratic programming subroutine used in each iteration of the NLPQN call. For a simple example of the NLPQN subroutine, see the section "Rosen-Suzuki Problem" on page 321.

NLPQUA Call

CALL NLPQUA(*rc, xr, quad, x0* <, *opt*> <, *blc*> <, *tc*> <, *par*> <, *"ptit"*> <, *lin*> **);**

The NLPQUA subroutine computes an optimum value of a quadratic objective function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPQUA subroutine uses a fast algorithm for maximizing or minimizing the quadratic objective function

$$\frac{1}{2}x^T\mathbf{G}x + g^T x + c$$

subject to boundary constraints and general linear equality and inequality constraints. The algorithm is memory-consuming for problems with general linear constraints.

The matrix $\mathbf{G}$ must be symmetric but not necessarily positive definite (or negative definite for maximization problems). The constant term c affects only the value of the objective function, not its derivatives or the optimal point x^*.

The algorithm is an active-set method in which the update of active boundary and linear constraints is done separately. The QT decomposition of the matrix A_k of active linear constraints is updated iteratively (Gill et al. 1984). If n_f is the number of free parameters (that is, n minus the number of active boundary constraints) and n_a is the number of active linear constraints, then $\mathbf{Q}$ is an $n_f \times n_f$ orthogonal matrix that contains null space Z in its first $n_f - n_a$ columns and range space Y in its last n_a columns. The matrix $\mathbf{T}$ is an $n_a \times n_a$ triangular matrix of the form $t_{ij} = 0$ for $i < n - j$. The Cholesky factor of the projected Hessian matrix $Z_k^T\mathbf{G}Z_k$ is updated simultaneously with the QT decomposition when the active set changes.

The objective function is specified by the input arguments *quad* and *lin*, as follows:

- The *quad* argument specifies the symmetric $n \times n$ Hessian matrix, $\mathbf{G}$, 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 $\mathbf{G}$ has many zero elements. You can specify an $nn \times 3$ matrix in which each row represents one of the nn nonzero elements of $\mathbf{G}$. The first column specifies the row location in $\mathbf{G}$, 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 c 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 c of the objective function.

As in the other optimization subroutines, you can use the *blc* argument to specify boundary and general linear constraints, and you must provide a starting point *x0* to determine the number of parameters. If *x0* is not feasible, a feasible initial point is computed by linear programming, and the elements of *x0* can be missing values.

Assuming nonnegativity constraints $x \geq 0$, the quadratic optimization problem is solved with the LCP call, which solves the linear complementarity problem.

Choosing a sparse (or dense) input form of the *quad* argument does not mean that the algorithm used in the NLPQUA subroutine is necessarily sparse (or dense). If the following conditions are satisfied, the NLPQUA algorithm stores and processes the matrix $\mathbf{G}$ as sparse:

- No general linear constraints are specified.
- The memory needed for the sparse storage of $\mathbf{G}$ is less than 80% of the memory needed for dense storage.
- $\mathbf{G}$ is not a diagonal matrix. If $\mathbf{G}$ is diagonal, it is stored and processed as a diagonal matrix.

The sparse NLPQUA algorithm uses a modified form of minimum degree Cholesky factorization (George and Liu 1981).

In addition to the standard iteration history, the NLPNRA subroutine prints the following information:

- The heading *alpha* is the step size, α, computed with the line-search algorithm.
- The heading *slope* refers to $g^T s$, the slope of the search direction at the current parameter iterate $x^{(k)}$. For minimization, this value should be significantly smaller than zero. Otherwise, the line-search algorithm has difficulty reducing the function value sufficiently.

The Betts problem (see the section "Constrained Betts Function" on page 319) can be expressed as a quadratic problem in the following way:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad G = \begin{bmatrix} 0.02 & 0 \\ 0 & 2 \end{bmatrix}, \quad g = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad c = -100$$

Then

$$\frac{1}{2} x^T G x - g^T x + c = 0.5[0.02x_1^2 + 2x_2^2] - 100 = 0.01x_1^2 + x_2^2 - 100$$

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 };
con  = {  2. -50.  .   .,
         50.  50.  .   .,
         10.  -1. 1. 10.};
x = { -1. -1.};
opt = {0 2};
call nlpqua(rc, xres, quad, x, opt, con, , , , lin);
```

The *quad* argument specifies the **G** matrix, and the *lin* argument specifies the g vector with the value of c appended as the last element. The matrix `con` specifies the boundary constraints and the general linear constraint.

The iteration history follows.

Figure 25.280 Quadratic Optimization

Note: Initial point was changed to be feasible for boundary and linear constraints.

Optimization Start

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Lower Bound Constraint	Upper Bound Constraint
1	X1	6.800000	0.136000	2.000000	50.000000
2	X2	-1.000000	-2.000000	-50.000000	50.000000

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		

Iteration	Restarts	Function Calls	Active Constraints	Objective Function	Objective Function Change	Max Abs Gradient Element	Step Size	Slope of Search Direction
1	0	2	1	-99.87349	1.3359	0.5882	0.706	-2.925
2	0	3	1	-99.96000	0.0865	0	1.000	-0.173

Optimization Results

Iterations	2	**Function Calls**	4
Gradient Calls	3	**Active Constraints**	1
Objective Function	-99.96	**Max Abs Gradient Element**	0
Slope of Search Direction	-0.173010381		

ABSGCONV convergence criterion satisfied.

Figure 25.280 *continued*

Optimization Results

Parameter Estimates

N	Parameter	Estimate	Gradient Objective Function	Active Bound Constraint
1	X1	2.000000	0.040000	Lower BC
2	X2	0	0	

Value of Objective Function = -99.96

Linear Constraints Evaluated at Solution

1 10.00000 = -10.0000 + 10.0000 * X1 - 1.0000 * X2

NLPTR Call

CALL NLPTR(*rc, xr, "fun", x0 < , opt> < , blc> < , tc> < , par> < , "ptit"> < , "grd"> < , "hes"> ***);**

The NLPTR subroutine uses a trust-region method to compute an optimum value of a function.

See the section "Nonlinear Optimization and Related Subroutines" on page 822 for a listing of all NLP subroutines. See Chapter 17 for a description of the arguments of NLP subroutines.

The NLPTR subroutine is a trust-region method. The algorithm uses the gradient $g^{(k)} = \nabla f(x^{(k)})$ and Hessian matrix $\mathbf{G}^{(k)} = \nabla^2 f(x^{(k)})$ and requires that the objective function $f = f(x)$ has continuous first- and second-order derivatives inside the feasible region.

The $n \times n$ Hessian matrix $\mathbf{G}$ contains the second derivatives of the objective function f with respect to the parameters $x_1, \ldots, x_n$, as follows:

$$G(x) = \nabla^2 f(x) = \left(\frac{\partial^2 f}{\partial x_j \partial x_k} \right)$$

The trust-region method works by optimizing a quadratic approximation to the nonlinear objective function within a hyperelliptic trust region. This trust region has a radius, Δ, that constrains the step size that corresponds to the quality of the quadratic approximation. The method is implemented by using Dennis, Gay, and Welsch (1981), Gay (1983), and Moré and Sorensen (1983).

Finite difference approximations for second-order derivatives that use only function calls are computationally very expensive. If you specify first-order derivatives analytically with the *"grd"* module argument, you can drastically reduce the computation time for numerical second-order derivatives. Computing the finite difference approximation for the Hessian matrix $\mathbf{G}$ generally uses only n calls of the module that computes the gradient analytically.

The NLPTR method performs well for small- to medium-sized problems and does not need many function, gradient, and Hessian calls. However, if the gradient is not specified by using the *"grd"* argument or if the computation of the Hessian module, as specified by the *"hes"* module argument, is computationally expensive, one of the (dual) quasi-Newton or conjugate gradient algorithms might be more efficient.

In addition to the standard iteration history, the NLPTR subroutine prints the following information:

- Under the heading *Iter*, an asterisk (*) printed after the iteration number indicates that the computed Hessian approximation was singular and had to be ridged with a positive value.
- The heading *lambda* represents the Lagrange multiplier, λ. This has a value of zero when the optimum of the quadratic function approximation is inside the trust region, in which case a trust-region-scaled Newton step is performed. It is greater than zero when the optimum is at the boundary of the trust region, in which case the scaled Newton step is too long to fit in the trust region and a quadratically constrained optimization is done. Large values indicate optimization difficulties, and as in Gay (1983), a negative value indicates the special case of an indefinite Hessian matrix.
- The heading *radius* refers to Δ, the radius of the trust region. Small values of the radius combined with large values of λ in subsequent iterations indicate optimization problems.

For an example of the use of the NLPTR subroutine, see the section "Unconstrained Rosenbrock Function" on page 315.

NORM Function

NORM(*x*, < , *method*> **);**

The NORM function computes the vector or matrix norm of *x*. The norm depends on the metric specified by the *method* argument. The arguments are as follows:

x	specifies a numeric vector with n elements or an $n \times p$ numeric matrix.
method	is an optional argument that specifies the method used to specify the norm. The *method* argument is either a numeric value, *method*≥ 1, or a case-insensitive character value. The valid options are given in the following sections.

Methods for Vector Norms

If *x* is a vector, then a vector norm is computed. The following are valid values of the *method* argument:

"L1"	specifies that the function compute the 1-norm: $\|x\|_1 = \Sigma_k \|x_k\|$. An equivalent alias is "CityBlock" or "Manhattan".
"L2"	specifies that the function compute the Euclidean 2-norm: $\|x\|_2 = \sqrt{(x'x)} = (\Sigma_k \|x_k\|^2)^{1/2}$. This is the default value. An equivalent alias is "Euclidean" or "Frobenius".
"LInf"	specifies that the function compute the ∞-norm: $\|x\|_\infty = \max_k \|x_k\|$. An equivalent alias is "Chebyshev".
p	is a numeric value, $p \geq 1$, that specifies the p-norm: $\|x\|_p = (\Sigma_k \|x_k\|^p)^{1/p}$, $p \geq 1$.

Methods for Matrix Norms

For an $n \times p$ matrix A such that $n > 1$ and $p > 1$, the *method* argument has the following valid values:

"Frobenius"	specifies the Frobenius norm: $\|A\|_F = \left(\Sigma_{i=1}^{n}\Sigma_{j=1}^{p}\|a_{ij}\|^2\right)^{1/2}$. This is the default value.
"L1"	specifies the matrix 1-norm: $\|A\|_1 = \max_{1\le j\le p}\Sigma_{i=1}^{n}\|a_{ij}\|$. This norm computes the maximum of the absolute column sums.
"L2"	specifies the matrix 2-norm, which is equivalent to the square root of the largest eigenvalue of the $A'A$ matrix. This quantity can be expensive to compute because the function internally computes eigenvalues.
"LInf"	specifies the ∞-norm: $\|A\|_\infty = \max_{1\le i\le n}\Sigma_{j=1}^{p}\|a_{ij}\|$. This norm computes the maximum of the absolute row sums.

The matrix p-norm is not available unless $p \in \{1, 2, \infty\}$.

The following statements compute vector norms:

```
/* compute vector norms */
v = 1:5;
vn1 = norm(v, "L1");
vn2 = norm(v, "L2");
vnInf = norm(v, "LInf");
print vn1 vn2 vnInf;
```

Figure 25.281 Vector Norms

vn1	vn2	vnInf
15	7.4161985	5

You can also compute matrix norms, as follows:

```
x = {1 2, 3 4};
mn1 = norm(x, "L1");
mnF = norm(x, "Frobenius");
mnInf = norm(x, "LInf");
print mn1 mnF mnInf;
```

Figure 25.282 Matrix Norms

mn1	mnF	mnInf
6	5.4772256	7

The NORM function returns a missing value if any element of the argument contains a missing value.

NROW Function

NROW(*matrix***);**

The NROW function returns the number of rows in its matrix argument. If the matrix has not been given a value, the NROW function returns a value of 0.

For example, following statements display the number of rows of the matrix **m**:

```
m = {1 2 3, 4 5 6, 3 2 1, 4 3 2, 5 4 3};
n = nrow(m);
print n;
```

Figure 25.283 Number of Rows in a Matrix

n
5

NUM Function

NUM(*matrix***);**

The NUM function produces a numeric representation of elements in a character matrix. If you have a character matrix for which each element is a string representation of a number, the NUM function produces a numeric matrix with dimensions that are the same as the dimensions of the argument and with elements that are the numeric representations (double-precision floating-point) of the corresponding elements of the argument.

For example, following statements display the result of converting a character matrix to a numeric matrix:

```
c = {"1" "2" "3"};
reset print;       /* display values and type of matrices */
m = num(c);
```

Figure 25.284 Numeric Matrix

m	1 row	3 cols	(numeric)

1	2	3

You can also use the PUTN function in Base SAS software to apply a SAS format to each element of a numeric matrix. The resulting matrix is character-valued.

See also the description of the CHAR function, which converts numeric matrices into character matrices.

ODE Call

CALL ODE(*r, "dername", c, t, h* **<** *, "jacobian"* **>** **<** *, eps* **>** **<** *, "SAS-data-set"* **>** **);**

The ODE subroutine performs numerical integration of first-order vector differential equations of the form

$$\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y}(t)) \quad \text{with } \mathbf{y}(0) = \mathbf{c}$$

The ODE subroutine returns the following values:

r is a numeric matrix that contains the results of the integration over connected subintervals. The number of columns in *r* is equal to the number of subintervals of integration as defined by the argument *t*. In case of any error in the integration on any subinterval, partial results are not reported in *r*.

The input arguments to the ODE subroutine are as follows:

"dername" specifies the name of a module used to evaluate the integrand.

c specifies an initial value vector for the variable y.

t specifies a sorted vector that describes the limits of integration over connected subintervals. The simplest form of the vector *t* contains only the limits of the integration on one interval. The first component of *t* should contain the initial value, and the second component should be the final value of the independent variable. For more advanced usage of the ODE subroutine, the vector *t* can contain more than two components. The components of the vector must be sorted in ascending order. Two consecutive components of the vector *t* are interpreted as a subinterval. The ODE subroutine reports the final result of integration at the right endpoint of each subinterval. This information is vital if $f(\cdot)$ has internal points of discontinuity. To produce accurate solutions, it is essential that you provide the location of these points in the variable *t*. The continuity of the forcing function is vital to the internal control of error.

h specifies a numeric vector that contains three components: the minimum allowable step size, *hmin*; the maximum allowable step size, *hmax*; and the initial step size to start the integration process, *hinit*.

"jacobian" optionally specifies the name of a module that is used to evaluate the Jacobian analytically. The Jacobian is the matrix *J*, with

$$J_{ij} = \frac{\partial f_i}{\partial y_j}$$

If the *"jacobian"* module is not specified, the ODE subroutine uses a finite-difference method to approximate the Jacobian. The keyword for this option is J.

eps specifies a scalar that indicates the required accuracy. It has a default value of 1E–4. The keyword for this option is EPS.

SAS-data-set is an optional argument that specifies the name of a valid predefined SAS data set name. The data set is used to save the successful independent and dependent variables of the integration at each step. The keyword for this option is DATA.

The ODE subroutine is an adaptive, variable-order, variable-step-size, stiff integrator based on implicit backward-difference methods. See Aiken (1985), Bickart and Picel (1973), Donelson and Hansen (1971), Gaffney (1984), and Shampine (1978). The integrator is an implicit predictor-corrector method that locally attempts to maintain the prescribed precision *eps* relative to

$$d = \max_{0 \le t \le T} (\|y(t)\|_\infty, 1)$$

As you can see from the expression, this quantity is dynamically updated during the integration process and can help you to understand the validity of the results reported by the subroutine.

A Linear Differential Equation

Consider the differential equation

$$\frac{dy}{dt} = -ty \text{ with } y = 0.5 \text{ at } t = 0$$

The following statements attempt to find the solution at $t = 1$:

```
/* Define the integrand */
start fun(t,y);
   v = -t*y;
   return(v);
finish;

/* Call ODE */
c   = 0.5;
t   = {0 1};
h   = {1E-12 1 1E-5};
call ode(r1, "FUN", c, t, h);
print r1[format=E21.14];
```

Figure 25.285 Solution to a Differential Equation at $t = 1$

r1
3.03432290135600E-01

In this case, the integration is carried out over $(0, 1)$ to give the value of y at $t = 1$. The optional parameter *eps* has not been specified, so it is internally set to 1E–4. Also, the optional parameter *"jacobian"* has not been specified, so finite-difference methods are used to estimate the Jacobian. The accuracy of the answer can be increased by specifying *eps*. For example, set EPS=1E–7, as follows:

```
call ode(r2, "FUN", c, t, h) eps=1E-7;
print r2[format =E21.14];
```

Figure 25.286 A Solution with Increased Accuracy

r2
3.03265687354960E-01

Compare this value to $0.5e^{-0.5} = 3.03265329856310E-01$ and observe that the result is correct through the sixth decimal digit and has an error relative to 1 that is $O(1E-7)$.

If the solution was desired at 1 and 2 with an accuracy of 1E–7, you would use the following statements:

```
t   = {0 1 2};
h   = {1E-12 1 1E-5};
call ode(r3, "FUN", c, t, h) eps=1E-7;
print r3[format=E21.14];
```

Figure 25.287 A Solution at Two Times

r3	
3.03265687354960E-01	6.76677185425390E-02

Note that **r3** contains the solution at $t = 1$ in the first column and at $t = 2$ in the second column.

A Discontinuous Forcing Function

Now consider the smoothness of the forcing function $f(\cdot)$. For the purpose of estimating errors, adaptive methods require some degree of smoothness in the function $f(\cdot)$. If this smoothness is not present in $f(\cdot)$ over the interior and including the left endpoint of the subinterval, the reported result does not have the desired accuracy. The function $f(\cdot)$ must be at least continuous. If the function does not meet this requirement, you should specify the discontinuity as an intermediate point. For example, consider the differential equation

$$\frac{dy}{dt} = \begin{cases} t & \text{if } t < 1 \\ 0.5t^2 & \text{if } t \geq 1 \end{cases}$$

To find the solution at $t = 2$, use the following statements:

```
/* Define the integrand */
start fun(t,y);
    if t < 1 then v = t;
    else v = .5*t*t;
    return(v);
finish;

c   = 0;
t   = {0 2};
h   = {1E-12 1. 1E-5};
call ode(r1, "FUN", c, t, h) eps=1E-12;
print r1[format=E21.14];
```

Figure 25.288 Numerical Solution Across a Discontinuity

r1
1.66666626639430E+00

In the preceding case, the integration is carried out over a single interval, $(0, 2)$. The optional parameter *eps* is specified to be 1E–12. The optional parameter *"jacobian"* is not specified, so finite-difference methods are used to estimate the Jacobian.

Note that the value of **r1** does not have the required accuracy (it should contain a 12 decimal-place representation of 5/3), although no error message is produced. The reason is that the function is not continuous at the point $t = 1$. Even the lowest-order method cannot produce a local reliable error estimate near the point of discontinuity. To avoid this problem, you can create subintervals so that the integration is carried out first over $(0, 1)$ and then over $(1, 2)$. The following statements implement this method:

```
c   = 0;
t   = {0 1 2};
h   = {1E-12 1 1E-5};
call ode(r2, "FUN", c, t, h) eps=1E-12;
print r2[format=E21.14];
```

Figure 25.289 Numerical Solution on Subintervals

r2	
5.00000000003280E-01	1.66666666667280E+00

The variable **r2** contains the solutions at both $t = 1$ and $t = 2$, and the errors are of the specified order. Although there is no interest in the solution at the point $t = 1$, the advantage of specifying subintervals with no discontinuities is that the function $f(\cdot)$ is infinitely differentiable in each subinterval.

A Piecewise Continuous Forcing Function

When $f(\cdot)$ is continuous, the ODE subroutine can compute the integration to the specified precision, even if the function is defined piecewise. Consider the differential equation

$$\frac{dy}{dt} = \begin{cases} t & \text{if } t < 1 \\ t^2 & \text{if } t \geq 1 \end{cases}$$

The following statements finds the solution at $t = 2$. Since the function $f(\cdot)$ is continuous, the requirements for error control are satisfied.

```
/* Define the integrand */
start fun(t,y);
    if t < 1 then v = t;
    else v = t*t;
    return(v);
finish;

c   = 0.5;
t   = {0 2};
h   = {1E-12 1 1E-5};
call ode(r, "FUN", c, t, h) eps=1E-12;
print r[format=E21.14];
```

Figure 25.290 Numerical Solution Across a Discontinuity

r
3.33333333334260E+00

Comparing Numerical Integration with an Eigenvalue Decomposition

This example compares the ODE subroutine to an eigenvalue decomposition for stiff-linear systems. In the problem

$$\frac{dy}{dt} = Ay \quad \text{with} \quad y(0) = c$$

where A is a symmetric constant matrix, the solution can be written in terms of the eigenvalue decomposition as

$$y(t) = Ue^{Dt}U'c$$

where U is the matrix of eigenvectors and D is a diagonal matrix with the eigenvalues on its diagonal.

The following statements produce two solutions, one by using the ODE subroutine and the other by using the eigenvalue decomposition:

```
/* Define the integrand */
start fun(t,x) global(a,count);
   count = count+1;
   v = a*x;
   return(v);
finish;

/* Define the Jacobian */
start jac(t,x) global(a);
   return(a);
finish;

a = {-1000 -1 -2 -3,
        -1 -2  3 -1,
        -2  3 -4 -3,
        -3 -1 -3 -5 };

count = 0;
t = {0 1 2};
h = {1E-12  1 1E-5};
eps = 1E-9;
c = {1, 0, 0, 0 };
call ode(z, "FUN", c, t, h)  eps=eps j="JAC";
print z[format=E21.14];
print count;
```

Figure 25.291 Numerical Integration of a Linear System

z	
1.85787365323370E-06	6.58581419290380E-06
-1.76251618648820E-03	-6.35540479968280E-03
-1.56685608330750E-03	-5.72429355251570E-03
1.01207978487750E-03	3.73655984584120E-03

count
437

```
/* Do the eigenvalue decomposition */
start eval(t) global(d,u,c);
   v = u*diag(exp(d*t))*u`*c;
   return(v);
finish;

call eigen(d,u,a);
free z1;
do i = 1 to nrow(t)*ncol(t)-1;
   z1 = z1 || (eval(t[i+1]));
end;
print z1[format=E21.14];
```

Figure 25.292 Analytic Solution of a Linear System

z1	
1.85787839378720E-06	6.58580950202810E-06
-1.76251639451200E-03	-6.35540294085790E-03
-1.56685625917120E-03	-5.72429205508220E-03
1.01207878768800E-03	3.73655890904620E-03

Is this an $O(1E-9)$ result? Note that for the problem

$$d = \max_{0 \le t \le T} (\|y(t)\|_\infty, 1) = 1$$

with the 1E–6 result, the ODE subroutine should attempt to maintain an accuracy of 1E–9 relative to 1. Therefore, the 1E–6 result should have almost three correct decimal places. At $t = 2$, the first component of **z** is 6.58597048842310E–06, while its more accurate value is 6.58580950203220E–06, showing an $O(1E-10)$ error.

Troubleshooting

The ODE subroutine can fail for problems with unusual qualitative properties, such as finite escape time in the interval of integration (that is, the solution goes towards infinity at some finite time). In such cases, try testing with different subintervals and different levels of accuracy to gain some qualitative information about the behavior of the solution of the differential equation.

ODSGRAPH Call

CALL ODSGRAPH(*name, template, matrix1* <, *matrix2, . . . , matrix13*> **);**

The ODSGRAPH subroutine renders an ODS statistical graph that is defined by a template.

The input arguments to the ODSGRAPH subroutine are as follows:

name is a character matrix or quoted literal that assigns a name to the graph. The name is used to identify the output graph in the SAS Results window.

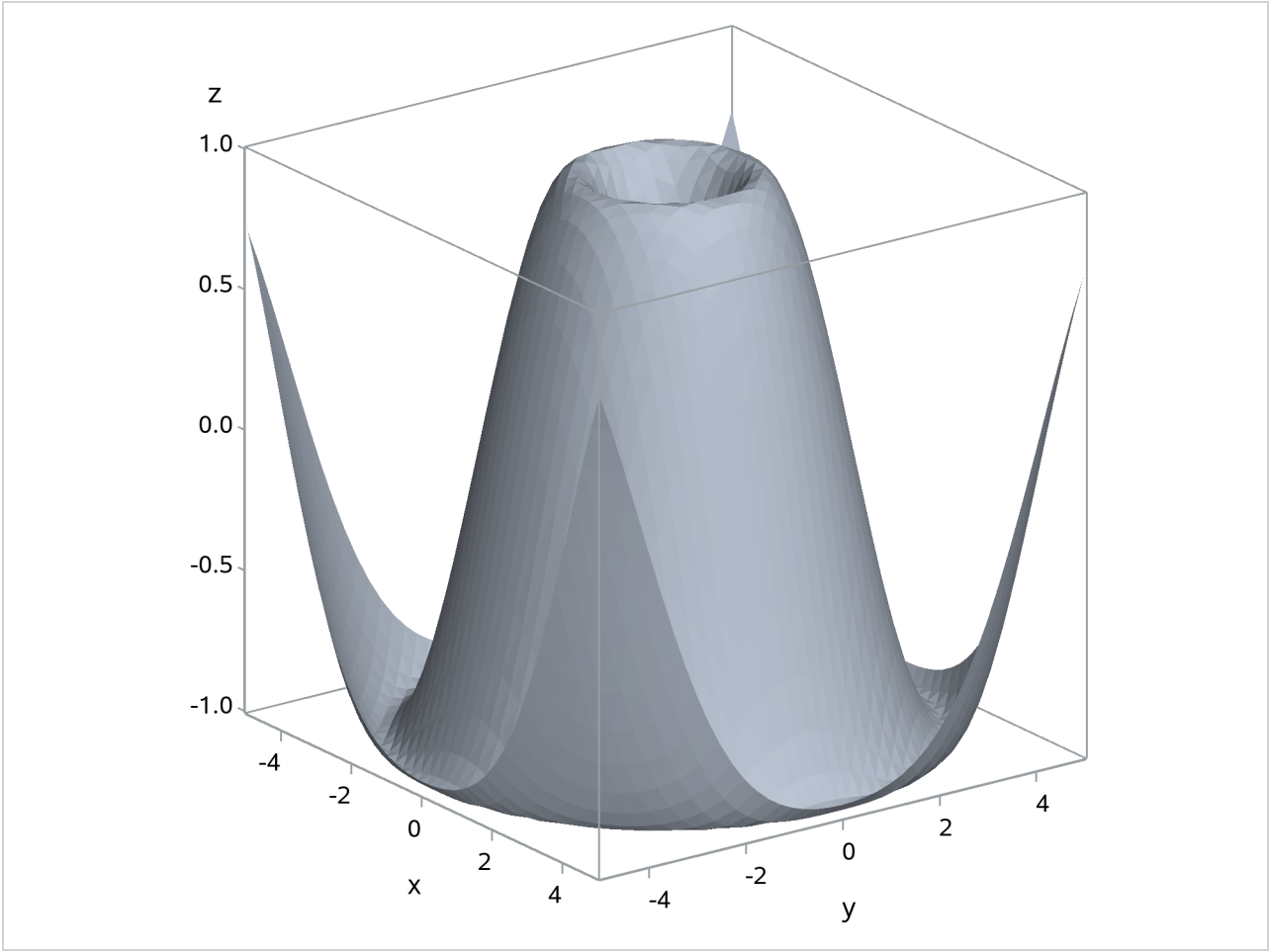
template is a character matrix or quoted literal that names the template used to render the graph.

matrix is a matrix whose columns are supplied to the template. You can specify up to 13 arguments. The name of each column must be specified by using the MATTRIB statement or the COLNAME= option in a READ statement.

The ODSGRAPH subroutine (which requires a SAS/GRAPH license) renders a graph defined by the input template. Data for the graph are in the columns of the matrix arguments. Column names are assigned to the matrices by using the MATTRIB statement or by using the COLNAME= option in a READ statement. This is illustrated in the following example, which produces a three-dimensional surface plot:

```
proc template;
  define statgraph SurfacePlot;
    BeginGraph;
    layout overlay3d;
      surfaceplotparm x=x y=y z=z / surfacetype=fill;
    endlayout;
    EndGraph;
  end;
run;

ods graphics on;
title "Surface Plot";

proc iml;
XDiv = do( -5, 5, 0.25 );
YDiv = do( -5, 5, 0.25 );
nX = ncol(XDiv);
nY = ncol(YDiv);
x = shape(repeat(XDiv, nY, 1), 0, 1);
y = shape(repeat(YDiv`, 1, nX), 0, 1);
z = sin( sqrt( x##2 + y##2 ) );
matrix = x || y || z;
mattrib matrix colname={"x" "y" "z"};
call odsgraph("surface", "SurfacePlot", matrix);
quit;

ods graphics off;
```

Figure 25.293 ODS Graphic



In the example, the TEMPLATE procedure defines a template for a surface plot. The ODSGRAPH subroutine calls ODS to render the graph by using the layout in the template. (You can render the graph in any ODS destination.) The data for the graph are contained in a matrix. The MATTRIB statement associates the columns of the matrix with the variable names required by the template.

You can also create graphs from data that are read from a data set. If x, y, and z are variables in a data set, then the following statements plot these variables:

```
use myData;
read all into matrix[colname = c];
call odsgraph("surface", "SurfacePlot", matrix);
```

Since column names created via a READ statement are permanently associated with the INTO matrix, you do not need to use a MATTRIB statement for this example.

The sample programs distributed with SAS/IML software include other examples of plots that are available by using ODS Statistical Graphics.

OPSCAL Function

OPSCAL(*mlevel, quanti* < *, qualit*> **);**

The OPSCAL function rescales qualitative data to be a least squares fit to quantitative data.

The arguments to the OPSCAL function are as follows:

mlevel	specifies a scalar that has one of two values. When *mlevel* is 1, the *qualit* matrix is at the nominal measurement level; when *mlevel* is 2, it is at the ordinal measurement level.
quanti	specifies an $m \times n$ matrix of quantitative information assumed to be at the interval level of measurement.
qualit	specifies an $m \times n$ matrix of qualitative information whose level of measurement is specified by *mlevel*. When *qualit* is omitted, *mlevel* must be 2 and a temporary *qualit* is constructed that contains the integers from 1 to n in the first row, from $n + 1$ to $2n$ in the second row, from $2n + 1$ to $3n$ in the third row, and so forth, up to the integers $(m - 1)n$ to mn in the last(mth) row. You cannot specify *qualit* as a character matrix.

The result of the OPSCAL function is the optimal scaling transformation of the qualitative (nominal or ordinal) data in *qualit*. The optimal scaling transformation that results has the following properties:

- It is a least squares fit to the quantitative data in *quanti*.
- It preserves the qualitative measurement level of *qualit*.

When *qualit* is at the nominal level of measurement, the optimal scaling transformation result is a least squares fit to *quanti*, given the restriction that the category structure of *qualit* must be preserved. If element i of *qualit* is in category c, then element i of the optimum scaling transformation result is the mean of all those elements of *quanti* that correspond to elements of *qualit* that are in category c.

For example, the following statements create the vector shown in Figure 25.294:

```
quanti = {5  4  6  7  4  6  2  4  8  6};
qualit = {6  6  2 12  4 10  4 10  8  6};
os = opscal(1, quanti, qualit);
print os;
```

Figure 25.294 Optimal Scaling Transformation of Nominal Data

os
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 discrete-nominal transformation (Young 1981).

When *qualit* is at the ordinal level of measurement, the optimal scaling transformation result is a least squares fit to *quanti*, given the restriction that the ordinal structure of *qualit* must be preserved. This is done by determining blocks of elements of *qualit* so that if element i of *qualit* is in block b, then element i of the result

is the mean of all those *quanti* elements that correspond to block *b* elements of *qualit* so that the means are (weakly) in the same order as the elements of *qualit*.

For example, consider these statements, which produce the transformation shown in Figure 25.295:

```
os2 = opscal(2, quanti, qualit);
print os2;
```

Figure 25.295 Optimal Scaling Transformation of Ordinal Data

os2
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 least squares monotonic transformation (Kruskal 1964b).

You can omit the *qualit* argument, as shown in the following statements:

```
quanti = {5  3  6  7  5  7  8  6  7  8};
os3 = opscal(2, quanti);
print os3;
```

These statements are equivalent to specifying

```
qualit = 1:10;
```

The result is shown in Figure 25.296.

Figure 25.296 Optimal Scaling Transformation

os3
4 4 6 6 6 7 7 7 7 8

ORPOL Function

ORPOL(*x* < , *maxdegree*> < , *weights*> **);**

The ORPOL function generates orthogonal polynomials on a discrete set of points.

The arguments to the ORPOL function are as follows:

x	is an $n \times 1$ vector of values on which the polynomials are to be defined.
maxdegree	specifies the maximum degree polynomial to be computed. If *maxdegree* is omitted, the default value is $\min(n, 19)$. If *weights* is specified, you must also specify *maxdegree*.
weights	specifies an $n \times 1$ vector of nonnegative weights associated with the points in *x*. If you specify *weights*, you *must* also specify *maxdegree*. If *maxdegree* is not specified or is specified incorrectly, the default weights (all weights are 1) are used.

The ORPOL matrix function generates orthogonal polynomials evaluated at the n points contained in *x* by using the algorithm of Emerson (1968). The result is a column-orthonormal matrix $\mathbf{P}$ with n rows and *maxdegree*+1 columns such that $\mathbf{P}'\text{diag}(\text{weights})\mathbf{P} = \mathbf{I}$. The result of evaluating the polynomial of degree $j-1$ at the ith element of *x* is stored in $\mathbf{P}[i, j]$.

The maximum number of nonzero orthogonal polynomials (r) that can be computed from the vector and the weights is the number of distinct values in the vector, ignoring any value associated with a zero weight.

The polynomial of maximum degree has degree of $r-1$. If the value of *maxdegree* exceeds $r-1$, then columns $r+1$, $r+2$,..., *maxdegree*+1 of the result are set to 0. In this case,

$$\mathbf{P}'\text{diag}(\text{weights})\mathbf{P} = \begin{bmatrix} I(r) & 0 \\ 0 & 0 \end{bmatrix}$$

The following statements create a matrix with three orthogonal columns, as shown in Figure 25.297:

```
x = T(1:5);
P = orpol(x,2);
print P;
```

Figure 25.297 Orthogonal Polynomials

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 shown by the following statements:

```
reset fuzz;          /* print tiny numbers as zero */
w = j(ncol(x),1,1); /* default weight is all ones */
/* Verify orthonormal */
L = P`*diag(w)*P;
print L;
```

Some reference books on orthogonal polynomials do not normalize the columns of the matrix that represents the orthogonal polynomials. For example, a textbook might give the following as a fourth-degree polynomial evaluated on evenly spaced data:

```
textbookPoly = { 1 -2  2 -1  1,
                 1 -1 -1  2 -4,
                 1  0 -2  0  6,
                 1  1 -1 -2 -4,
                 1  2  2  1  1 };
```

To compare this representation to the normalized representation that the ORPOL function produces, use the following program:

```
/* Normalize the columns of textbook representation */
normalPoly = textbookPoly;
do i = 1 to ncol( normalPoly );
   v = normalPoly[,i];
   norm = sqrt(v[##]);
   normalPoly[,i] = v / norm;
end;

/* Compare the normalized matrix with ORPOL */
x = T(1:5); /* Any evenly spaced data gives the same answer */
imlPoly = orpol( x, 4 );

diff = imlPoly - normalPoly;
maxDiff = abs(diff)[<>];
reset fuzz; /* print tiny numbers as zero */
print maxDiff;
```

Figure 25.298 Normalizing a Matrix

maxDiff
0

Polynomial Regression

A typical use for orthogonal polynomials is to fit a polynomial to a set of data. Given a set of points (x_i, y_i), $i = 1, \ldots, m$, the classical theory of orthogonal polynomials says that the best approximating polynomial of degree d is given by

$$f_d = \sum_{i=1}^{d+1} c_i P_i$$

where $c_i = \langle y, P_i \rangle / \langle P_i, P_i \rangle$ and where P_i is the ith column of the matrix $\mathbf{P}$ 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 following program, the weight vector is used to overweight or underweight particular data points. The researcher has reasons to doubt the accuracy of the first measurement. The last data point is also underweighted because it is a leverage point and is believed to be an outlier. The second data point was measured twice and is overweighted. (Rerunning the program with a weight vector of all ones and examining the new values of the `fit` variable is a good way to understand the effect of the weight vector.)

```
x = {0.1, 2,   3,  5,    8,   10, 20};
y = {0.5, 1, 0.1, -1, -0.5, -0.8, 0.1};

/* The second measurement was taken twice.
   The first and last data points are underweighted
   because of uncertainty in the measurements.  */
w = {0.5, 2,   1,  1,    1,    1, 0.2};
maxDegree = 4;
P = orpol(x,maxDegree,w);

/* The best fit by a polynomial of degree k is
   Sum c_i P_i  where c_i = <f,P_i> */
start InnerProduct(f,g,w);
   h = f#g#w;
   return (h[+]);
finish;

c = j(1,maxDegree+1);
do i = 1 to maxDegree+1;
   c[i] = InnerProduct(y, P[,i], w);
end;

FitResults = j(maxDegree+1,2);
do k = 1 to maxDegree+1;
   fit = P[,1:k] * c[1:k];
   resid = y - fit;
   FitResults[k,1] = k-1;       /* degree of polynomial */
   FitResults[k,2] = resid[##]; /* sum of square errors */
end;
print FitResults[colname={"Degree" "SSE"}];
```

Figure 25.299 Statistics for an Orthogonal Polynomial Regression

FitResults	
Degree	SSE
0	3.1733014
1	4.6716722
2	1.3345326
3	1.3758639
4	0.8644558

Testing Linear Hypotheses

The ORPOL function can also be used to test linear hypotheses. Suppose you have an experimental design with k factor levels. (The factor levels can be equally or unequally spaced.) At the ith level, you record n_k observations, $i = 1 \ldots k$. If $n_1 = n_2 = \ldots = n_k$, then the design is said to be *balanced*; otherwise it is *unbalanced*. You want to fit a polynomial model to the data and then ask how much variation in the data is explained by the linear component, how much variation is explained by the quadratic component after the linear component is taken into account, and so on for the cubic, quartic, and higher-level components.

To be completely concrete, suppose you have four factor levels (1, 4, 6, and 10) and that you record seven measurements at first level, two measurements at the second level, three measurements at the third level, and

four measurements at the fourth level. This is an example of an unbalanced and unequally spaced factor-level design. The following program uses orthogonal polynomials to compute the Type I sum of squares for the linear hypothesis. (The program works equally well for balanced designs and for equally spaced factor levels.)

The following program calls the ORPOL function to generate the orthogonal polynomial matrix **P**, and uses it to form the Type I hypothesis matrix **L**. The program then uses the DESIGN function to generate **X**, the design matrix associated with the experiment. The program then computes **b**, the estimated parameters of the linear model. Since **L** was expressed in terms of the orthogonal polynomial matrix **P**, the computations involved in forming the Type I sum of squares are considerably simplified.

```
/* unequally spaced and unbalanced factor levels */
levels = { 1,1,1,1,1,1,1,
           4,4,
           6,6,6,
           10,10,10,10};

/* data for y. Make sure the data are sorted
   according to the factor levels */
y = {2.804823, 0.920085, 1.396577, -0.083318,
     3.238294, 0.375768, 1.513658,            /* level 1 */
     3.913391, 3.405821,                      /* level 4 */
     6.031891, 5.262201, 5.749861,            /* level 6 */
    10.685005, 9.195842, 9.255719,  9.204497 /* level 10 */
};

a      = {1,4,6,10}; /* spacing */
trials = {7,2,3,4};  /* sample sizes */
maxDegree = 3; /* model with Intercept,a,a##2,a##3 */

P = orpol(a,maxDegree,trials);

/* Test linear hypotheses:
   How much variation is explained by the
   i_th polynomial component after components
   0..(i-1) have been taken into account? */

/* the columns of L are the coefficients of the
   orthogonal polynomial contrasts */
L = diag(trials)*P;

/* form design matrix */
x = design(levels);

/* compute b, the estimated parameters of the
   linear model. b is the mean of the y values
   at each level.
   b = ginv(x'*x) * x` * y
   but since x is the output from DESIGN, then
   x`*x = diag(trials) and so
   ginv(x`*x) = diag(1/trials) */
b = diag(1/trials)*x`*y;
```

```
/* (L`*b)[i] is the best linear unbiased estimated
   (BLUE) of the corresponding orthogonal polynomial
   contrast */
blue = L`*b;

/* The variance of (L`*b) is
   var(L`*b) = L`*ginv(x`*x)*L
     =[P`*diag(trials)]*diag(1/trials)*[diag(trials)*P]
     = P`*diag(trials)*P
     = Identity          (by definition of P)

   Therefore the standardized square of
   (L`*b) is computed as
   SS1[i] = (blue[i]*blue[i])/var(L`*b)[i,i])
          = (blue[i])##2                          */

SS1 = blue # blue;
rowNames = {"Intercept" "Linear" "Quadratic" "Cubic"};
print SS1[rowname=rowNames format=11.7 label="Type I SS"];
```

Figure 25.300 indicates that most of the variation in the data can be explained by a first-degree polynomial.

Figure 25.300 Statistics for an Orthogonal Polynomial Regression

	Type I SS
Intercept	331.8783538
Linear	173.4756050
Quadratic	0.4612604
Cubic	0.0752106

Generating Families of Orthogonal Polynomials

There are classical families of orthogonal polynomials (for example, Legendre, Laguerre, Hermite, and Chebyshev) that arise in the study of differential equations and mathematical physics. These "named" families are orthogonal on particular intervals (a, b) with respect to the inner product $\int_b^a f(x)g(x)w(x)\,dx$. The functions returned by the ORPOL function are *different* from these named families because the ORPOL function uses a different inner product. There are no built-in functions that can automatically generate these families; however, you can write a program to generate them.

Each named polynomial family $\{p_j\}$, $j \geq 0$ satisfies a three-term recurrence relation of the form

$$p_j = (A_j + xB_j)p_{j-1} - C_j p_{j-2}$$

where the constants A_j, B_j, and C_j are relatively simple functions of j. To generate these "named" families, use the three-term recurrence relation for the family. The recurrence relations are found in references such as Abramowitz and Stegun (1972) or Thisted (1988).

For example, the so-called Legendre polynomials (represented P_j for the polynomial of degree j) are defined on $(-1, 1)$ with the weight function $w(x) = 1$. They are standardized by requiring that $P_j(1) = 1$ for all $j \geq 0$. Thus $P_0(x) = 1$. The linear polynomial $P_1(x) = a + bx$ is orthogonal to P_0 so that

$$\int_{-1}^{1} P_1(x)P_0(x)\,dx = \int_{-1}^{1} a + bx\,dx = 0$$

which implies $a = 0$. The standardization $P_1(1) = 1$ implies that $P_1(x) = x$. The remaining Legendre polynomials can be computed by looking up the three-term recurrence relation: $A_j = 0$, $B_j = (2j - 1)/j$, and $C_j = (j - 1)j$. The following program computes Legendre polynomials evaluated at a set of points:

```
maxDegree = 6;
/* evaluate polynomials at these points */
x = T( do(-1,1,0.05) );

/* define the standard Legendre Polynomials
   Using the 3-term recurrence with
   A[j]=0, B[j]=(2j-1)/j, and C[j]=(j-1)/j
   and the standardization P_j(1)=1
   which implies P_0(x)=1, P_1(x)=x. */
legendre = j(nrow(x), maxDegree+1);
legendre[,1] = 1; /* P_0 */
legendre[,2] = x; /* P_1 */

do j = 2 to maxDegree;
   legendre[,j+1] = (2*j-1)/j # x # legendre[,j] -
                    (j-1)/j # legendre[,j-1];
end;
```

ORTVEC Call

CALL ORTVEC(*w, r, rho, lindep, v* **<** *, q* **>** **);**

The ORVEC subroutine provides columnwise orthogonalization and stepwise QR decomposition by using the Gram-Schmidt process.

The ORTVEC subroutine returns the following values:

w is an $m \times 1$ vector. If the Gram-Schmidt process converges (*lindep*=0), *w* is orthonormal to the columns of Q, which is assumed to have $n \leq m$ (nearly) orthonormal columns. If the Gram-Schmidt process does not converge (*lindep*=1), *w* is a vector of missing values. For stepwise QR decomposition, *w* is the $(n + 1)$th orthogonal column of the matrix Q. If the *q* argument is not specified, *w* is the normalized value of the vector *v*,

$$w = \frac{v}{\sqrt{v'v}}$$

r is a $n \times 1$ vector. If the Gram-Schmidt process converges (*lindep*=0), *r* contains Fourier coefficients. If the Gram-Schmidt process does not converge (*lindep*=1), *r* is a vector of missing values. If the *q* argument is not specified, *r* is a vector with zero dimension. For stepwise QR decomposition, *r* contains the n upper triangular elements of the $(n + 1)$th column of R.

rho is a scalar value. If the Gram-Schmidt process converges (*lindep*=0), *rho* specifies the distance from *w* to the range of Q. Even if the Gram-Schmidt process converges, if *rho* is sufficiently small, the vector *v* can be linearly dependent on the columns of Q. If the Gram-Schmidt process does not converge (*lindep*=1), *rho* is set to 0. For stepwise QR decomposition, *rho* contains the diagonal element of the $(n + 1)$th column of R. In formulas, the value *rho* is denoted by ρ.

lindep returns a value of 1 if the Gram-Schmidt process does not converge in 10 iterations. A value of 1 often indicates that the input vector *v* is linearly dependent on the *n* columns of the input matrix *Q*. In that case, *rho* is set to 0, and the results *w* and *r* contain missing values. If *lindep*=0, the Gram-Schmidt process did converge, and the results *w*, *r*, and *rho* are computed.

The input arguments to the ORTVEC subroutine are as follows:

v specifies an $m \times 1$ vector *v* that is to be orthogonalized to the *n* columns of *Q*. For stepwise QR decomposition of a matrix, *v* is the $(n+1)$th matrix column before its orthogonalization.

q specifies an optional $m \times n$ matrix *Q* that is assumed to have $n \leq m$ (nearly) orthonormal columns. Thus, the $n \times n$ matrix $Q'Q$ should approximate the identity matrix. The column orthonormality assumption is not tested in the ORTVEC call. If it is violated, the results are not predictable. The argument *q* can be omitted or can have zero rows and columns. For stepwise QR decomposition of a matrix, *q* contains the first *n* matrix columns that are already orthogonal.

The relevant formula for the ORTVEC subroutine is

$$v = \mathbf{Qr} + \rho w$$

In the formula, if the $m \times n$ matrix *Q* has *n* (nearly) orthonormal columns, the vector *v* is orthogonal to the columns of *Q* and ρ is the distance from *w* to the range of *Q*.

There are two special cases:

- If $m > n$, ORTVEC normalizes the result *w*, so that *w*′ *w*= 1.
- If $m = n$, the output vector *w* is the null vector.

The case $m < n$ is not possible since *Q* is assumed to have *n* (nearly) orthonormal columns.

To initialize a stepwise QR decomposition, the ORTVEC subroutine can be called to normalize *v* only (that is, to compute $w = v/\sqrt{v'v}$ and $\rho = \sqrt{v'v}$). There are two ways to accomplish this:

- 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 `q={}`).

In both cases, *r* is a column vector with zero rows.

The ORTVEC subroutine is useful for the following applications:

- performing stepwise QR decomposition. Compute *Q* and *R*, so that $A = QR$, where *Q* is column orthonormal, $Q'Q = I$, and *R* is upper triangular. The *j*th step is applied to the *j*th column, *v*, of *A*, and it computes the *j*th column *w* of *Q* and the *j*th column, $(r\ \rho\ 0)'$, of *R*.
- computing the $m \times (m-n)$ null space matrix, Q_2, that corresponds to an $m \times n$ range space matrix, Q_1 $(m > n)$, by the following stepwise process:
  1. Set $v = e_i$ (where e_i is the *i*th unit vector) and try to make it orthogonal to all column vectors of Q_1 and the already generated Q_2.

2. If the subroutine is successful, append *w* to Q_2; otherwise, try $v = e_{i+1}$.

The 4×3 matrix Q contains the unit vectors e_1, e_3, and e_4. The column vector *v* is pairwise linearly independent with the three columns of Q. As expected, the ORTVEC subroutine computes the vector *w* as the unit vector e_2 with $u = (1, 1, 1)$ and $\rho = 1$.

```
q = { 1  0  0,
      0  0  0,
      0  1  0,
      0  0  1 };
v = { 1, 1, 1, 1 };
call ortvec(w,u,rho,lindep,v,q);
print rho u w;
```

Figure 25.301 Matrix Orthogonalization

rho	u	w
1	1	0
	1	1
	1	0
		0

Stepwise QR Decomposition Example

You can perform the QR decomposition of the linearly independent columns of an $m \times n$ matrix **A** with the following statements:

```
a = {1  2  1,
     2  4  2,
     1  4 -1,
     1  0  3}; /* use any matrix A */
nind = 0;  ndep = 0;  dmax = 0.;
n = ncol(a);  m = nrow(a); ind = j(1,n,0);
free q;
do j = 1 to n;
   v = a[ ,j];
   call ortvec(w,u,rho,lindep,v,q);
   aro = abs(rho);
   if aro > dmax then dmax = aro;
   if aro <= 1.e-10 * dmax then lindep = 1;
   if lindep = 0 then do;
      nind = nind + 1;
      q = q || w;
      if nind = n then r = r || (u // rho);
      else r = r || (u // rho // j(n-nind,1,0.));
   end;
   else do;
      print "Column " j " is linearly dependent.";
      ndep = ndep + 1; ind[ndep] = j;
   end;
end;
print q r;
```

Figure 25.302 QR Decomposition of Independent Columns

q		r	
0.3779645	0	2.6457513	5.2915026
0.7559289	0	0	2.8284271
0.3779645	0.7071068	0	0
0.3779645	-0.707107		

Next, process the remaining (dependent) columns of **A**:

```
do j = 1 to ndep;
   k = ind[ndep-j+1];
   v = a[ ,k];
   call ortvec(w,u,rho,lindep,v,q);
   if lindep = 0 then do;
      nind = nind + 1;
      q = q || w;
      if nind = n then r = r || (u // rho);
      else r = r || (u // rho // j(n-nind,1,0.));
   end;
end;
print q r;
```

Figure 25.303 QR Decomposition of Dependent Columns

q			r		
0.3779645	0	-0.239046	2.6457513	5.2915026	2.6457513
0.7559289	0	-0.478091	0	2.8284271	-2.828427
0.3779645	0.7071068	0.5976143	0	0	1.327E-16
0.3779645	-0.707107	0.5976143			

You can also use the ORTVEC subroutine to compute the null space in the last columns of **Q**:

```
do i = 1 to m;
   if nind < m then do;
     v = j(m,1,0.); v[i] = 1.;
     call ortvec(w,u,rho,lindep,v,q);
     aro = abs(rho);
     if aro > dmax then dmax = aro;
     if aro <= 1.e-10 * dmax then lindep = 1;
     if lindep = 0 then do;
       nind = nind + 1;
       q = q || w;
     end;
     else print "Unit vector" i "linearly dependent.";
   end;
end;
if nind < m then do;
   print "This is theoretically not possible.";
end;
print q;
```

Figure 25.304 Final Orthogonal Matrix

q			
0.3779645	0	-0.239046	0.8944272
0.7559289	0	-0.478091	-0.447214
0.3779645	0.7071068	0.5976143	0
0.3779645	-0.707107	0.5976143	-3.1E-17

In the example, if you define $\mathbf{Q}_2$ to be the last two columns of $\mathbf{Q}$, then $\mathbf{Q}_2' A = \mathbf{0}$.

PACKAGE Statement

PACKAGE *keyword* < *options* > **;**

The PACKAGE statement supports installing, uninstalling, and using packages. A package consists of SAS/IML source code, documentation, data sets, and sample programs. Packages are a convenient way for programmers to download and install functions that extend the functionality of SAS/IML software. For information about how to use packages, see Chapter 11, "Packages."

Packages are supported only on Linux and Windows operating systems.

The following statements are documented separately:

- PACKAGE HELP statement
- PACKAGE INFO statement
- PACKAGE INSTALL statement
- PACKAGE LIBNAME statement
- PACKAGE LIST statement
- PACKAGE LOAD statement
- PACKAGE UNINSTALL statement

As described in the section "Collections of Packages" on page 176, packages are stored in one of three *collections*:

- The PRIVATE collection contains packages that were installed by a user for personal use.
- The PUBLIC collection contains packages that were installed by a system administrator in a public location and are intended to be used by multiple users at a site.
- The SYSTEM collection is installed as part of SAS/IML software and is available to any user. These packages were written by SAS/IML developers and are supported by SAS Technical Support. The SYSTEM collection is similar to the IMLMLIB library of modules, except that a package in the SYSTEM collection must be loaded before the functions are available for use.

When any PACKAGE statement encounters the name of a package, it searches for the package first in the PRIVATE collection, then in the PUBLIC collection, and finally in the SYSTEM collection. When it finds a package with the specified name, it stops searching. If a version of a package exists in the PUBLIC collection, you can install a newer version of the same package in the PRIVATE collection. By default, the PACKAGE statement will find the newer version. You can explicitly specify the PUBLIC collection if you want to use the older version.

PACKAGE HELP Statement

PACKAGE HELP *packagename*<**(***collection* **)**> **;**

If an installed package provides documentation, the PACKAGE HELP statement displays the documentation. The documentation often includes the syntax and purpose of the SAS/IML modules that the package defines. For more information about packages, see Chapter 11, "Packages."

You must specify the following argument:

packagename
specifies the name of the package.

You can also specify the following option in parentheses:

collection
requests that only the specified collection be searched. Valid values for *collection* are PRIVATE, PUBLIC, and SYSTEM. For more information about collections, see the PACKAGE statement.

The following statement displays the help file for the AboveBelow package, which is installed in the SYSTEM collection:

```
package help AboveBelow;
```

The following statement searches only the SYSTEM collection:

```
package help AboveBelow(system);
```

If a package is named Pkg1, then valid names for the help files are *Pkg1.pdf*, *Pkg1.txt*, *index.htm*, and *index.html*. The help files are stored in the **help** subdirectory for the package. The file that is displayed depends on the files that exist and the interface that you use to run SAS/IML software as follows:

- The IML procedure echoes the file *Pkg1.txt* to the SAS Log window.
- The SAS/IML Studio application displays the first of the following files that it finds:
  1. *Pkg1.pdf*
  2. *index.htm* or *index.html*
  3. *Pkg1.txt* (opened in Notepad)

PACKAGE INFO Statement

PACKAGE INFO *packagename*< **(***collection* **)** > . . . **;**

The PACKAGE INFO statement displays information about one or more installed packages.

You must specify the following argument:

packagename
specifies the name of the package. You can specify multiple names in a single statement.

You can also specify the following option in parentheses:

collection
requests that only the specified collection be searched. Valid values for *collection* are PRIVATE, PUBLIC, and SYSTEM. For more information about collections, see the PACKAGE statement.

The displayed information comes from the *info.txt* file for the package. For example, the following statement displays information about the AboveBelow package, which is installed in the SYSTEM collection. The output is shown in Figure 25.305.

```
package info AboveBelow;
```

Figure 25.305 Information about an Installed Package

Package Information	
Name	AboveBelow
Description	Counts values above and below a specified value
Author	SAS/IML Development Team
Collection	System
Version	1.0
Requires IML	14.1
Directory	C:\Program Files\SASHome\SASFoundation\9.4\iml\sasmisc\packages\AboveBelow

PACKAGE INSTALL Statement

PACKAGE INSTALL *"fullname"*< **(***collection* < **PARSETIME** > **)** > . . . **;**

PACKAGE INSTALL *fileref*< **(***collection* < **PARSETIME** > **)** > . . . **;**

Packages are distributed as ZIP files. The PACKAGE INSTALL statement unzips the ZIP file and installs the package in a predetermined location.

You can specify the package to be installed in two ways:

"fullname"
specifies the ZIP file's fully qualified name (which includes both the path and the file name) in quotation marks. You can specify multiple names in a single statement.

fileref
specifies a file reference that was previously created by the FILENAME statement.

You can also specify the following options in parentheses:

collection
specifies the collection into which the package is to be installed. You can specify PUBLIC or PRIVATE; you cannot install a package in the SYSTEM collection. By default, the package is installed in the PRIVATE collection. At some sites, you might need administrative privileges in order to install a package in the PUBLIC collection.

PARSETIME
requests that the package be installed at parse time, rather than at run time. This option applies only to IMLPLUS programs in the SAS/IML Studio environment.

For example, the following statement installs the Pkg1 package from the ZIP file named **`C:\Packages\Pkg1.zip`**:

```
package install "C:\Packages\Pkg1.zip";
```

The following statements are an equivalent way to install the Pkg1 package:

```
filename ThePkg "C:\Packages\Pkg1.zip";
package install ThePkg;
```

The PACKAGE INSTALL statement creates a directory named *Pkg1* in a directory whose location is system-dependent and then unzips the contents of the ZIP file into that directory. The previous examples do not specify a collection, so the package is installed in the PRIVATE collection.

You should install a package only once. SAS/IML software does not allow you overwrite a package that is already installed. To install a newer version of a package in the same collection, you must uninstall the older version and then install the newer version.

Some operating systems (notably Linux) are case-sensitive. When you specify a ZIP file on a case-sensitive operating system, be sure to match the case of the file name in the operating system.

PACKAGE LIBNAME Statement

PACKAGE LIBNAME *libref packagename* **;**

The PACKAGE LIBNAME statement creates a SAS libref that points to the **`data`** directory for a package. You must load a package before you can use the PACKAGE LIBNAME statement.

You must specify the following arguments:

libref
specifies the name of a SAS libref to be created.

packagename
specifies the name of the package.

The PACKAGE LIBNAME statement is used to access data in a package. For examples, the following statements read data from the AboveBelow package, which is installed in the SYSTEM collection:

```
package load AboveBelow;
package libname ABdata AboveBelow;
/* use the libref to read data that the package provides */
use ABdata.example;
   read all var _NUM_ into X[colname=varNames];
close ABdata.example;
print varNames;
```

Figure 25.306 Reading Data from a Package

varNames		
x1	x2	x3

The READ statement reads the numerical variables into the matrix **X**. Figure 25.306 shows that the example data set contains numerical variables named x1, x2, and x3.

You must specify exactly one package in the PACKAGE LIBNAME statement.

PACKAGE LIST Statement

PACKAGE LIST < *collection* > **;**

The PACKAGE LIST statement lists the packages that are installed in the specified collection (or in all collections if a collection is not specified). For more information about collections, see the PACKAGE statement.

You can specify the following option:

collection
: requests that only packages in the specified collection be listed. Valid values for *collection* are PRIVATE, PUBLIC, and SYSTEM. By default, the PACKAGE LIST statement lists packages in all collections.

For example, the following statement displays packages that are installed in the SYSTEM collection:

```
proc iml;
package list system;
```

Figure 25.307 System Packages

System Packages	
Name	**Version**
AboveBelow	1.0

Notice that the PACKAGE LIST statement acts on *collections* of packages, whereas the other PACKAGE statements apply to individual packages. The *collection* argument is not specified inside parentheses.

PACKAGE LOAD Statement

PACKAGE LOAD *packagename*<(*collection* <**VERSION=***version*>)> . . . **;**

The PACKAGE LOAD statement executes the package source files. Usually the source files define SAS/IML modules. However, a package might also define SAS/IML matrices, create data sets, define macro variables, or carry out a computation.

You must specify the following argument:

packagename
specifies the name of the package. You can specify multiple names in a single statement.

You can also specify the following options in parentheses:

collection
requests that only the specified collection be searched for the file to be executed. Valid values for *collection* are PRIVATE, PUBLIC, and SYSTEM. For more information about collections, see the PACKAGE statement.

VERSION=*version*
specifies the minimum required version number for loading a package. A version number is a text string that contains up to four numbers that are separated by decimal points. For example, valid values are "1.0", "2.7.1", and "3.1.4.1". When the PACKAGE LOAD statement finds a specified package, it checks the package version. If the package version is greater than or equal to *version*, the package is loaded. Otherwise, the statement displays an error message.

The following statement loads the AboveBelow package, which is installed in the SYSTEM collection:

```
package load AboveBelow;
```

The PACKAGE LOAD statement reads the *info.txt* file and uses the SourceFiles keyword to determine which files contain the source code for the package. (For more information about the *info.txt* file, see the section "Step 2: Create the Package Information File" on page 178.) The statement executes the specified files, which are located in the **source** directory of the package's root directory. For the AboveBelow package, the PACKAGE LOAD statement defines several SAS/IML modules. The SAS Log window indicates that new modules were defined.

As a second example, the following PACKAGE LOAD statement loads a package called *Pkg1* from the PUBLIC collection. The load succeeds if the installed version of the package is greater than or equal to version 1.2, but it fails if the installed version of the package is less than 1.2. For example, the load succeeds if the installed version is 1.2, 1.3, or 2.1. The load fails if the installed version is 1.1.

```
package load Pkg1(public version="1.2");
```

The PACKAGE LOAD statement is similar to the EXECUTEFILE subroutine but differs in the following ways:

- The PACKAGE LOAD statement can execute multiple files. The EXECUTEFILE subroutine executes a single file.

- The PACKAGE LOAD statement keeps track of which packages have been loaded. A subsequent PACKAGE LOAD statement does not reread and rerun the package source files. In contrast, the EXECUTEFILE subroutine rereads and reruns the file every time it is called.

The IMLPlus language in the SAS/IML Studio application executes the PACKAGE LOAD statement at parse time. That is, SAS/IML Studio reads the source files and defines the modules before it executes any statements.

PACKAGE UNINSTALL Statement

PACKAGE UNINSTALL *packagename*< **(***collection* < **NOWARN** > < **PARSETIME** > **)** > . . . **;**

The PACKAGE UNINSTALL statement uninstalls a package that is installed in the specified collection.

You must specify the following argument:

packagename
specifies the name of the package. You can specify multiple names in a single statement.

You can also specify the following options in parentheses:

collection
requests that only the specified collection be searched for the package to uninstall. Valid values for *collection* are PRIVATE and PUBLIC. By default, the PACKAGE UNINSTALL statement looks for the package in the PRIVATE collection. If you have administrative privileges to uninstall a package in the PUBLIC collection, you can specify the PUBLIC collection in parentheses after the package name. You cannot uninstall a package in the SYSTEM collection.

NOWARN
requests that no warning message be reported if the specified package is not installed.

PARSETIME
requests that the package be installed at parse time, rather than at run time. This option applies only to IMLPLUS programs in the SAS/IML Studio environment.

For example, if the Pkg1 package is installed in the PRIVATE collection, the following statement uninstalls the package from that collection:

```
package uninstall Pkg1;
```

PARENTNAME Function

PARENTNAME(*"argument"***);**

The PARENTNAME function enables you to determine the name of a matrix that is passed into a SAS/IML module. If an argument is skipped or the argument is called with an expression, the PARENTNAME function returns a blank character.

```
start GetName(a=0);
   pa = ParentName("a");
   return( pa );
finish;

x = 1:5;
n1 = GetName(x);                    /* name is "x" */
n2 = GetName(1:5);                  /* temporary matrix, no name */
n3 = GetName();                     /* skipped argument, no name */
print n1 n2 n3;
```

Figure 25.308 Names of Matrices That Are Passed into a Module

n1	n2	n3
x		

PALETTE Function

PALETTE(*name, numColors* **);**

The PALETTE function is part of the IMLMLIB library. The PALETTE function returns a palette of colors that are suitable for using in a discrete heat map or a choropleth map. The colors are appropriate to use for the COLORRAMP= option of the HEATMAPDISC subroutine.

The following example gets several color palettes:

```
BuGn4 = Palette("BuGn", 4);
BrBG5 = Palette("BrBG", 5);
Past6 = Palette("Pastel1", 6);
print BuGn4, BrBG5, Past6;
```

Figure 25.309 Color Palettes

BuGn4			
CXEDF8FB	CXB2E2E2	CX66C2A4	CX238B45

BrBG5				
CXA6611A	CXDFC27D	CXF5F5F5	CX80CDC1	CX018571

Past6					
CXFBB4AE	CXB3CDE3	CXCCEBC5	CXDECBE4	CXFED9A6	CXFFFFCC

The color specification and palette names were designed by Cynthia Brewer (Brewer 2013) and are described at http://ColorBrewer.org. The color schemes are copyright 2002 by Cynthia Brewer, Mark Harrower, and The Pennsylvania State University. The ColorBrewer color schemes are made available under the Apache License, Version 2.0.

The sequential schemes support between three and nine colors. The diverging schemes support between three and 11 colors. The qualitative schemes support between three and eight colors, with some palettes supporting as many as 12 colors. Table 25.3–Table 25.5 show the names of the color palettes and the number of colors that each supports.

Table 25.3 Sequential Color Schemes

Name	Max Colors
BLUES	9
GREENS	9
GREYS	9
ORANGES	9
PURPLES	9
REDS	9
BUGN	9
BUPU	9
GNBU	9
ORRD	9
PUBU	9
PUBUGN	9
PURD	9
RDPU	9
YLGN	9
YLGNBU	9
YLORBR	9
YLORRD	9

Table 25.4 Diverging Color Schemes

Name	Max Colors
BRBG	11
PIYG	11
PRGN	11
PUOR	11
RDBU	11
RDGY	11
RDYLBU	11
RDYLGN	11
SPECTRAL	11

Table 25.5 Qualitative Color Schemes

Name	Max Colors
ACCENT	8
DARK2	8
PAIRED	12
PASTEL1	9
PASTEL2	8
SET1	9
SET2	8
SET3	12

The following SAS/IML statements create heat maps that demonstrate the color palettes. Palettes with seven colors are shown in Figure 25.310–Figure 25.312.

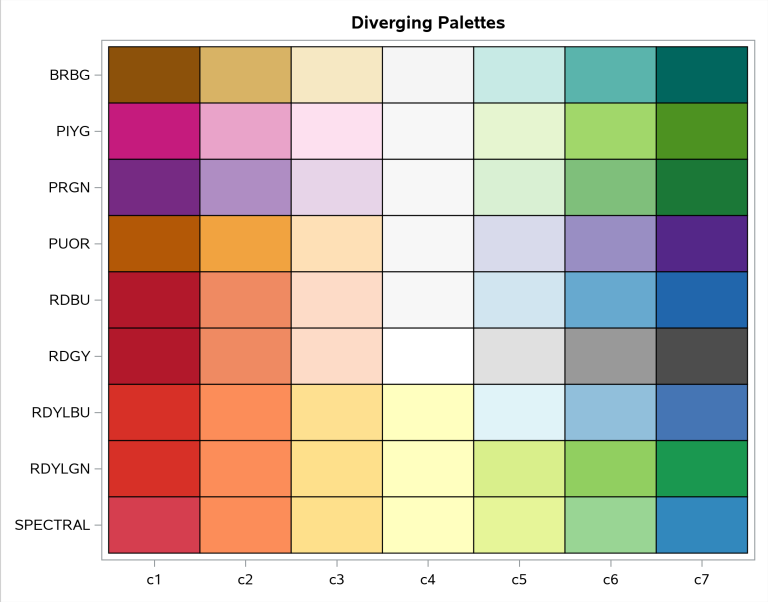
```
/* show N-color heat map for specified palette names */
start ShowSchemes(Names, N, _title);
   Name = colvec(Names);
   NumPalettes = nrow(Name);
   R = j(NumPalettes, N, "        "); /* R = ramp */
   do i = 1 to NumPalettes;
      R[i, ] = Palette(Name[i], N);
   end;

   xnames = "c1":("c"+strip(char(N)));
   x = j(nrow(R), ncol(R), .);
   idx = loc(R^=" ");
   x[idx] = 1:ncol(idx);
   colors = R[idx];
   run HeatmapDisc(x, colors) xvalues=xnames yvalues=Name
                              title=_title ShowLegend=0;
finish;

/* sequential palettes: Use for ordered value */
seq = {BLUES GREENS GREYS ORANGES PURPLES REDS /* monochrome */
       BUGN BUPU GNBU ORRD PUBU PUBUGN PURD RDPU YLGN YLGNBU YLORBR YLORRD};
ods graphics / width = 600 height=1000;
run ShowSchemes(seq, 7, "Sequential Palettes");
```

Figure 25.310 Sequential Palettes



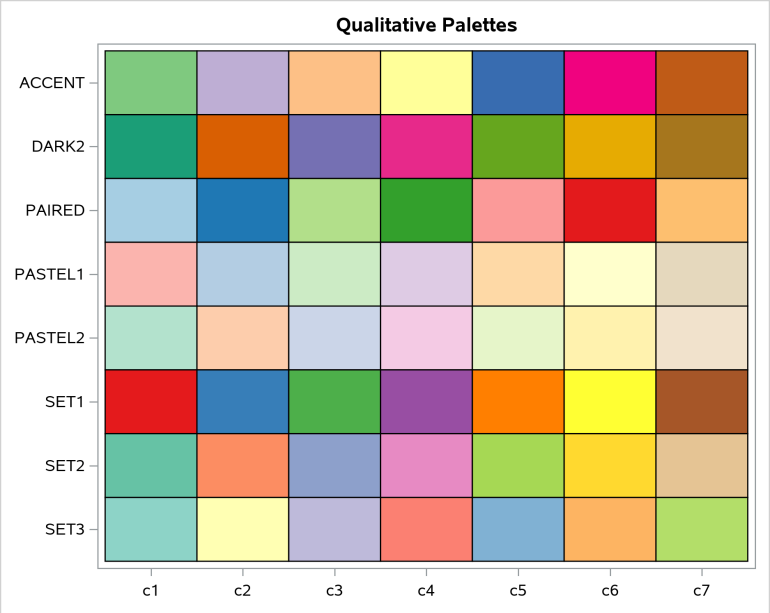

```
/* diverging palettes: Use to show high/low relative to a central value */
div = {BRBG   PIYG  PRGN   PUOR    RDBU    RDGY  RDYLBU  RDYLGN SPECTRAL};
ods graphics / width = 700 height=550;
run ShowSchemes(div, 7, "Diverging Palettes");
```

Figure 25.311 Diverging Palettes




```
/* qualitative palettes: Use to show nominal value (for example, race) */
qual = {ACCENT DARK2 PAIRED PASTEL1 PASTEL2 SET1  SET2    SET3};
ods graphics / width = 600 height=480;
run ShowSchemes(qual, 7, "Qualitative Palettes");
```

Figure 25.312 Qualitative Palettes



PAUSE Statement

PAUSE < *expression* | * > **;**

The PAUSE statement interrupts the execution of a module.

The arguments to the PAUSE statement are as follows:

expression	is a character matrix or quoted literal that contains a message to print.
*	suppresses any messages.

The PAUSE statement stops execution of a module, saves the calling chain so that execution can resume later (by a RESUME statement), prints a pause message that you can specify, and puts you in immediate mode so you can enter more statements.

You can specify an operand in the PAUSE statement to supply a message to be printed for the pause prompt. If no operand is specified, the following default message is printed:

```
Paused in module MyModule.
```

In this case, *MyModule* is the name of the module that contains the pause. If you want to suppress all messages in a PAUSE statement, use an asterisk as the operand, as follows:

```
pause *;
```

The PAUSE statement should be specified only in modules. It generates a warning if executed in immediate mode.

When an error occurs while executing inside a module, PROC IML automatically behaves as though a PAUSE statement was issued. PROC IML also enters "immediate mode" within the module environment. You can correct the error and then resume execution by submitting a RESUME statement .

PROC IML supports pause processing of both subroutine and function modules. See also the description of the SHOW statement which uses the PAUSE option.

POLYROOT Function

POLYROOT(*vector***);**

The POLYROOT function computes the zeros of a real polynomial. The *vector* argument is an $n \times 1$ (or $1 \times n$) vector that contains the coefficients of an $(n-1)$ degree polynomial with the coefficients arranged in order of decreasing powers.

The POLYROOT function returns the array r, which is an $(n-1) \times 2$ matrix that contains the roots of the polynomial. The first column of r contains the real part of the complex roots, and the second column contains the imaginary part. If a root is real, the imaginary part is 0.

The POLYROOT function finds the real and complex roots of a polynomial with real coefficients.

The POLYROOT function uses an algorithm proposed by Jenkins and Traub (1970) to find the roots of the polynomial. The algorithm is not guaranteed to find all roots of the polynomial. An appropriate warning message is issued when one or more roots cannot be found. If r is given as a root of the polynomial $P(x)$, then $1 + P(r) = 1$, based on the rounding error of the computer that is employed.

For example, you can use the following statements to find the roots of the polynomial

$$P(x) = 0.2567x^4 + 0.1570x^3 + 0.0821x^2 - 0.3357x + 1$$

```
p = {0.2567 0.1570 0.0821 -0.3357 1};
r = polyroot(p);
print r;
```

Figure 25.313 Roots of a Quartic Polynomial

r	
0.8383029	0.8514519
0.8383029	-0.851452
-1.144107	1.1914525
-1.144107	-1.191452

The polynomial has two conjugate pairs of roots that, within machine precision, are given by $r = 0.8383029 \pm 0.8514519i$ and $r = -1.144107 \pm 1.1914525i$.

PRINT Statement

PRINT < *matrices* > < *(expression)* > < *"message"* > < *pointer-controls* > < *[options]* > **;**

The PRINT statement displays the values of matrices or literals.

The arguments to the PRINT statement are as follows:

matrices	are the names of matrices.
(*expression*)	is an expression in parentheses that is evaluated. The result of the evaluation is printed. The evaluation of a subscripted matrix used as an expression results in printing the submatrix.
"message"	is a message in quotes.
pointer-controls	control the pointer for printing. For example, a comma (,) skips a single line and a slash (/) skips to a new page.
options	are described in the following list.

The following *options* can appear in the PRINT statement. They are specified in brackets after the matrix name to which they apply.

COLNAME=*matrix*

specifies the name of a 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. The *matrix* argument is usually a character matrix, but numeric matrices are also supported. You can 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 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 elements in *matrix* are traversed in row-major order. The *matrix* argument is usually character, but numeric matrices are also supported. You can use the following RESET AUTONAME statement to automatically label rows as ROW1, ROW2, and so on:

```
reset autoname;
```

For example, the following statements print a matrix in the 12.2 format with column and row labels:

```
x = {45.125 50.500,
     75.375 90.825};
r = {"Div A"  "Div B"};
c = {"Amount" "Net Pay"};
print x[rowname=r colname=c format=12.2];
```

Figure 25.314 Matrix with Row and Column Labels

	x	
	Amount	Net Pay
Div A	45.13	50.50
Div B	75.38	90.83

To permanently associate the preceding options with a matrix name, see the description of the MATTRIB statement.

If there is not enough room to print all the matrices across the page, then one or more matrices are printed out in the next group. If there is not enough room to print all the columns of a matrix across the page, then the columns are continued on a subsequent line.

The spacing between adjacent matrices can be controlled by the SPACES= option of the RESET statement. The FW= option of the RESET statement can be used to control the number of print positions used to print each numeric element. The RESET WIDE option can be used to suppress the automatic row and column headers during printing. For more print-related options, see the description of the RESET statement.

To print part of a matrix or a temporary expression, enclose the expression in parentheses:

```
y=1:10;
print (y[1:3])[format=5.1]; /* prints first few elements */
print (sum(y))[label="sum"];
```

Figure 25.315 Printing Temporary Matrices

1.0
2.0
3.0

sum
55

PROD Function

PROD(*matrix1 < , matrix2, . . . , matrix15>* **);**

The PROD function returns as a single numeric value the product of all nonmissing elements in all arguments. You can pass in as many as 15 numeric matrices as arguments. The PROD function checks for missing values and does not include them in the product. It returns missing if all values are missing.

For example, consider the following statements:

```
a = {2 1, . 3};
b = prod(a);
print b;
```

Figure 25.316 Output from the PROD Function

b
6

For a single argument with at least one nonmissing value, the PROD function is identical to the subscript reduction operator that computes the product. That is, **`prod(x)`** and **`x[#]`** both compute the product of the elements of **`x`**. See the section "Subscript Reduction Operators" on page 55 for more information about subscript reduction operators.

PRODUCT Function

PRODUCT(*a, b < , dim>* **);**

The PRODUCT function multiplies matrices of polynomials.

The arguments to the PRODUCT function are as follows:

a is an $m \times (ns)$ numeric matrix. The first $m \times n$ submatrix contains the constant terms of the polynomials, the second $m \times n$ submatrix contains the first-order terms, and so on.

b is an $n \times (pt)$ matrix. The first $n \times p$ submatrix contains the constant terms of the polynomials, the second $n \times p$ submatrix contains the first-order terms, and so on.

dim is a 1×1 matrix, with value $p > 0$. The value of this matrix is used to set the dimension p of the matrix b. If omitted, the value of p is set to 1.

The PRODUCT function multiplies matrices of polynomials. The value returned is the $m \times (p(s + t - 1))$ matrix of the polynomial products. The first $m \times p$ submatrix contains the constant terms, the second $m \times p$ submatrix contains the first-order terms, and so on.

The PRODUCT function can be used to multiply the matrix operators employed in a multivariate time series model of the form

$$\Phi_1(B)\Phi_2(B)\mathbf{Y}_t = \Theta_1(B)\Theta_2(B)\epsilon_t$$

where $\Phi_1(B)$, $\Phi_2(B)$, $\Theta_1(B)$, and $\Theta_2(B)$ are matrix polynomial operators whose first matrix coefficients are identity matrices. Often $\Phi_2(B)$ and $\Theta_2(B)$ represent seasonal components that are isolated in the modeling process but multiplied with the other operators when forming predictors or estimating parameters. The RATIO function is often employed in a time series context as well.

For example, the following statements demonstrate the PRODUCT function:

```
m1 = {1 2 3 4,
      5 6 7 8};
m2 = {1 2 3,
      4 5 6};
r = product(m1, m2, 1);
print r;
```

Figure 25.317 A Product of Matrices of Polynomials

r			
9	31	41	33
29	79	105	69

PURGE Statement

PURGE ;

The PURGE data processing statement is used to remove observations marked for deletion and to renumber the remaining observations. This closes the gaps created by deleted records. Execution of this statement can be time-consuming because it involves rewriting the entire data set.

CAUTION: Any indexes associated with the data set are lost after a purge.

When you quit PROC IML, observations marked for deletion are *not* automatically purged.

The following example creates a data set named A. The EDIT statement opens the data set for editing. The DELETE statement marks several observations for deletion. As shown in Figure 25.318, the observations are not removed and renumbered until the PURGE statement executes.

```
data a;
do i=1 to 10;
   output;
end;
run;

proc iml;
edit a;
pts = 3:8;
delete point pts;
list all;

purge;
list all;
```

Figure 25.318 Deleting and Purging Observations

```
 OBS         i

------ ---------

     1    1.0000

     2    2.0000

     9    9.0000

    10   10.0000



 OBS         i

------ ---------

     1    1.0000

     2    2.0000

     3    9.0000

     4   10.0000
```

PUSH Call

CALL PUSH(*argument1* **<** *, argument2, . . . , argument15* **>);**

The PUSH subroutine pushes character arguments that contain valid SAS statements (usually SAS/IML statements or global statements) to the input command stream. You can specify up to 15 arguments. Any statements in the input command queue are executed when the module is paused (see the PAUSE statement), which happens when one of the following occurs:

- An execution error occurs within a module.
- An interrupt is issued.
- A PAUSE statement executes.

The pushed string is read before any other lines of input. If you call the PUSH subroutine several times, the strings pushed each time are ahead of the less recently pushed strings. If you would rather place the lines after others in the input stream, use the QUEUE call.

The strings you push do not appear on the log.

CAUTION: Do not push too many statements at one time. Pushing too many statements causes problems that can result in exiting the SAS System.

For more information about the input command stream, see Chapter 20.

An example that uses the PUSH subroutine follows:

```
start;
   code='reset pagesize=25;';
   call push(code,'resume;');
   pause;
   /* show that pagesize was set to 25 during   */
   /* a PAUSE state of a module                 */
   show options;
finish;
run main;
```

Figure 25.319 Result of a PUSH Statement

```
Options: noautoname center noclip

         deflib=WORK (system-specific-pathname)

         nodetails noflow nofuzz fw=9

         imlmlib=SASHELP.IMLMLIB  linesize=98 nolog

         name pagesize=25 noprint noprintall spaces=1

         userlib=WORK.IMLSTOR(not open)

         nowide
```

PUT Statement

PUT < *operand* > < *record-directives* > < *positionals* > < *format* > **;**

The PUT statement writes data to an external file. The arguments to the PUT statement are as follows:

operand specifies the value you want to output to the current position in the record. The *operand* can be either a variable name, a literal value, or an expression in parentheses. The *operand* can be followed immediately by an output format specification.

record-directives start new records. There are three types:

- *holding* @ is used at the end of a PUT statement to hold the current record so that you can continue to write more data to the record with later PUT statements. Otherwise, the next record is used for the next PUT statement.
- / writes out the current record and begins forming a new record.
- > *operand* specifies that the next record written start at the indicated byte position in the file (for RECFM=N files only). The *operand* is a literal number, a variable name, or an expression in parentheses. For example:

```
put >3 x 3.2;
```

positionals specify the column on the record to which the PUT statement should go. There are two types of positionals:

- @ *operand* specifies to go to the indicated column, where *operand* is a literal number, a variable name, or an expression in parentheses. For example, @30 means to go to column 30.

+ operand	specifies that the indicated number of columns are to be skipped, where *operand* is a literal number, a variable name, or an expression in parentheses.
format	specifies a valid SAS or user-defined output format. These are of the form *w.d* or $*w*. for standard numeric and character formats, respectively, where *w* is the width of the field and *d* is the decimal parameter, if any. They can also be a named format of the form *NAMEw.d*, where *NAME* is the name of the format. If the width is unspecified, then a default width is used; this is 9 for numeric variables.

The PUT statement writes to the file specified in the previously executed FILE statement, putting the values from matrices. The statement is described in detail in Chapter 8.

The PUT statement is a sequence of positionals and record directives, variables, and formats. An example that uses the PUT statement follows:

```
/* output variable A in column 1 using a 6.4 format */
/* Skip 3 columns and output X using an 8.4 format  */
put @1 a 6.4 +3 x 8.4;
```

PV Function

PV(*times,flows,freq,rates***);**

The PV function returns a scalar that contains the present value of the cash flows based on the specified frequency and rates.

The arguments to the function are as follows:

times	is an $n \times 1$ column vector of times. Elements should be nonnegative.
flows	is an $n \times 1$ column vector of cash flows.
freq	is a scalar that represents the base of the rates to be used for discounting the cash flows. If positive, it represents discrete compounding as the reciprocal of the number of compoundings per period. If zero, it represents continuous compounding. If –1, the rates represent per-period discount factors. No other negative values are accepted.
rates	is an $n \times 1$ column vector of rates to be used for discounting the cash flows. Elements should be positive.

A general present value relationship can be written as

$$P = \sum_{k=1}^{K} c(k)D(t_k)$$

where P is the present value of the asset, $\{c(k)\}, k = 1, \ldots, K$, is the sequence of cash flows from the asset, t_k is the time to the kth cash flow in periods from the present, and $D(t)$ is the discount function for time t. The discount factors are as follows:

- with per-unit-time-period discount factors d_t:

  $$D(t) = d_t^t$$

- with continuous compounding:

  $$D(t) = e^{-r_t t}$$

- with discrete compounding:

$$D(t) = (1 + fr)^{-t/f}$$

  where $f > 0$ is the frequency, the reciprocal of the number of compoundings per unit time period.

The following statements present an example of using the PV function in the DATA step:

```
data a;
  pv = mort(., 438.79, 0.10/12, 30*12);
run;
proc print data=a; run;
```

Figure 25.320 Present Value Computation (DATA Step)

OBS	pv
1	50000.48

You can do the same computation by using the PV function in SAS/IML software. The first example uses a monthly rate; the second example uses an annual rate.

```
proc iml;
/*  If rate is specified as annual rate divided by 12 and FREQ=1,
 *  then results are equal to those computed by the MORT function. */
timesn = t(1:360);
flows = repeat(438.79, 360);
rate = repeat(0.10/12, 360);
freq = 1;
pv = pv(timesn, flows, freq, rate);
print pv;

/*  If rate is specified as annual rate, then the cash flow TIMES
 *  need to be specified in 1/12 increments and the FREQ=1/12.
 *  This produces the same result as the previous PV call. */
timesn = t(do(1/12, 30, 1/12));
flows = repeat(438.79, 360);
rate = repeat(0.10, 360); /* specify annual rate  */
freq = 1/12;              /* 12 compoundings annually */
pv = pv(timesn, flows, freq, rate);
print pv;
```

Figure 25.321 Present Value Computation (PROC IML)

pv
50000.48

pv
50000.48

QNTL Call

CALL QNTL(*q, x,* < *, probs*> < *, method*> **);**

The QNTL subroutine computes sample quantiles for data. The arguments are as follows:

q specifies a matrix to contain the quantiles of the *x* matrix.

x specifies an $n \times p$ numerical matrix of data. The QNTL subroutine computes quantiles for each column of the matrix.

probs specifies a numeric vector of probabilities used to compute the quantiles. If this option is not specified, the vector $\{0.25, 0.5, 0.75\}$ is used, resulting in the quartiles of the data. For convenience, a probability of 0 returns the minimum value of *x*, and a probability of 1 returns the maximum value.

method specifies the method used to compute the quantiles. These methods correspond to those defined by using the PCTLDEF= option in the UNIVARIATE procedure. For details, see the section "Calculating Percentiles" of the documentation for the CORR procedure in the *Base SAS Procedures Guide: Statistical Procedures*.

The following values are valid:

- 1 specifies that quantiles are computed according to a weighted average.
- 2 specifies that quantiles are computed by choosing an observation closest to some quantity.
- 3 specifies that quantiles are computed by using the empirical distribution function.
- 4 specifies that quantiles are computed according to a different weighted average.
- 5 specifies that quantiles are computed by using average values of the empirical distribution function. This is the default value.

If *x* is an $n \times p$ matrix, the QNTL subroutine computes a $k \times p$ matrix where k is the dimension of the *probs* matrix. The quantiles are returned in the *q* matrix, as shown in the following example:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
     7 2 13};
call qntl(q, x);
print q[rowname={"P25", "P50", "P75"}];
```

Figure 25.322 Quantiles

	q		
P25	6	2	5
P50	6	2	9
P75	6	7	10

You can use the MATTRIB statement to permanently assign row names to the matrix that contains the quantiles, as shown in the following statements:

```
p = {0.25 0.50 0.75};
labels = "P" + strip(putn(100*p, "best5."));
mattrib q rowname=labels;
print q;
```

Figure 25.323 Rownames for Quantiles

	q		
P25	6	2	5
P50	6	2	9
P75	6	7	10

You can specify the optional arguments in either of two ways: by specifying an argument positionally or by specifying a keyword/value pair, as shown in the following statements.

```
x = T(1:100);
p = do(0.1, 0.9, 0.1);
call qntl(q1, x, p);
call qntl(q2, x) probs=p; /* equivalent */
```

QR Call

CALL QR(*q, r, piv, lindep, a* < , *ord* > < , *b* > **);**

The QR subroutine produces the QR decomposition of a matrix by using Householder transformations.

The QR subroutine returns the following values:

q specifies an orthogonal matrix $\mathbf{Q}$ that is the product of the Householder transformations applied to the $m \times n$ matrix $\mathbf{A}$, if the *b* argument is not specified. In this case, the $\min(m, n)$ Householder transformations are applied, and *q* is an $m \times m$ matrix. If the *b* argument is specified, *q* is the $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$ that has the transposed Householder transformations $\mathbf{Q}'$ applied on the p columns of the argument matrix $\mathbf{B}$.

r specifies a $\min(m, n) \times n$ upper triangular matrix $\mathbf{R}$ that is the upper part of the $m \times n$ upper triangular matrix $\widetilde{\mathbf{R}}$ of the QR decomposition of the matrix $\mathbf{A}$. The matrix $\widetilde{\mathbf{R}}$ of the QR decomposition can be obtained by vertical concatenation (by using the operator //) of the $(m - \min(m, n)) \times n$ zero matrix to the result matrix $\mathbf{R}$.

piv specifies an $n \times 1$ vector of permutations of the columns of $\mathbf{A}$; that is, on return, the QR decomposition is computed, not of $\mathbf{A}$, but of the permuted matrix whose columns are $[\mathbf{A}_{\text{piv}[1]} \cdots \mathbf{A}_{\text{piv}[n]}]$. The vector *piv* corresponds to an $n \times n$ permutation matrix $\mathbf{\Pi}$.

lindep is the number of linearly dependent columns in matrix $\mathbf{A}$ detected by applying the $\min(m, n)$ Householder transformations in the order specified by the argument vector *piv*.

The input arguments to the QR subroutine are as follows:

a specifies an $m \times n$ matrix **A** that is to be decomposed into the product of the orthogonal matrix **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 **A**. 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 n_l 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 n_l initial columns and before the n_u 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 n_u columns of **A** that are to be processed. If $n > m$, some of these columns are not processed.

The default is *ord[j]*=j, in which case the Householder transformations are processed in the same order in which the columns are stored in matrix **A** (without pivoting).

b specifies an optional $m \times p$ matrix **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 **Q**.

The QR subroutine decomposes an $m \times n$ matrix **A** into the product of an $m \times m$ orthogonal matrix **Q** and an $m \times n$ upper triangular matrix $\widetilde{\mathbf{R}}$, so that

$$\mathbf{A\Pi} = \mathbf{Q}\widetilde{\mathbf{R}}, \mathbf{Q}'\mathbf{Q} = \mathbf{QQ}' = \mathbf{I}_m$$

by means of $\min(m, n)$ Householder transformations.

The $m \times m$ orthogonal matrix **Q** is computed only if the last argument *b* is not specified, as in the following example:

```
call qr(q, r, piv, lindep, a, ord);
```

In many applications, the number of rows, *m*, is very large. In these cases, the explicit computation of the $m \times m$ matrix **Q** might require too much memory or time.

In the usual case where $m > n$,

$$\mathbf{A} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

$$\widetilde{\mathbf{R}} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}$$

$$\mathbf{Q} = [\mathbf{Q}_1 \; \mathbf{Q}_2], \qquad \widetilde{\mathbf{R}} = \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix}$$

where $\mathbf{R}$ is the result returned by the QR subroutine.

The n columns of matrix $\mathbf{Q}_1$ provide an orthonormal basis for the n columns of $\mathbf{A}$ and are called the *range space* of $\mathbf{A}$. Since the $m - n$ columns of $\mathbf{Q}_2$ are orthogonal to the n columns of $\mathbf{A}$, $\mathbf{Q}_2'\mathbf{A} = \mathbf{0}$, they provide an orthonormal basis for the orthogonal complement of the columns of $\mathbf{A}$ and are called the *null space* of $\mathbf{A}$.

In the case where $m < n$,

$$\mathbf{A} = \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\widetilde{\mathbf{R}} = \mathbf{R} = \begin{bmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \end{bmatrix}$$

Specifying the argument *ord* as an n vector lets you specify a special order of the columns in matrix $\mathbf{A}$ on which the Householder transformations are applied. There are two special cases:

- If you do not specify the *ord* argument, the default values $ord[j] = j$ are used. In this case, Householder transformations are done in the same order in which the columns are stored in $\mathbf{A}$ (without pivoting).
- If you set all components of *ord* to zero, the Householder transformations are done in order of decreasing Euclidean norms of the columns of $\mathbf{A}$.

To check the QR decomposition, use the following statements to compute the three residual sum of squares (represented by the variables SS0, SS1, and SS2), which should be close to zero:

```
a = shape(1:20, 5);
m = nrow(a); n = ncol(a);
ord = j(1, n, 0);
call qr(q, r, piv, lindep, a);
ss0 = ssq(a[ ,piv] - q[,1:n] * r);
ss1 = ssq(q * q` - i(m));
ss2 = ssq(q` * q - i(m));
print ss0 ss1 ss2;
```

Figure 25.324 Result of a QR Decomposition

ss0	ss1	ss2
6.231E-28	3.026E-31	2.535E-31

If the QR subroutine detects linearly dependent columns while processing matrix **A**, the column order given in the result vector *piv* can differ from an explicitly specified order in the argument vector *ord*. If a column of **A** is found to be linearly dependent on columns already processed, this column is swapped to the end of matrix **A**. The order of columns in the result matrix **R** corresponds to the order of columns processed in **A**. The swapping of a linearly dependent column of **A** to the end of the matrix corresponds to the swapping of the same column in **R** and leads to a zero row at the end of the upper triangular matrix **R**.

The scalar result *lindep* counts the number of linearly dependent columns that are detected in constructing the first $\min(m, n)$ Householder transformations in the order specified by the argument vector *ord*. The test of linear dependence depends on the singularity criterion, which is 1E–8 by default.

Solving the linear system $Rx = Q'b$ with an upper triangular matrix **R** whose columns are permuted corresponding to the result vector *piv* leads to a solution x with permuted components. You can reorder the components of x by using the index vector *piv* at the left-hand side of an expression, as follows:

```
a = {3  0  0 -1,
     0  1  2  0,
     4 -4 -1  1,
    -1  2  3  4};
b = {-1, 8, -3, 28};

n = ncol(a); p = ncol(b);
ord = j(1, n, 0);
call qr(qtb, r, piv, lindep, a, ord, b);
print piv;

x = j(n,1);
x[piv] = inv(r) * qtb[1:n, 1:p];
print x;
```

Figure 25.325 Solution to a Linear System

piv
1 4 2 3

x
1
2
3
4

The Full-Rank Linear Least Squares Problem

This example solves the full-rank linear least squares problem. Specify the argument *b* as an $m \times p$ matrix **B**, as follows:

```
call qr(q, r, piv, lindep, a, ord, b);
```

When you specify the *b* argument, the QR subroutine computes the matrix $Q'B$ (instead of Q) as the result *q*. Now you can compute the p least squares solutions x_k of an overdetermined linear system with an $m \times n, m > n$ coefficient matrix A, rank(A) = n, and p right-hand sides b_k stored as the columns of the $m \times p$ matrix B:

$$\min_{x_k} \|Ax_k - b_k\|^2, k = 1, \ldots, p$$

where $\|\cdot\|$ is the Euclidean vector norm. This is accomplished by solving the p upper triangular systems with back substitution:

$$x_k = Pi'R^{-1}Q_1'b_k, k = 1, \ldots, p$$

For most applications, the number of rows of **A**, m, is much larger than n, the number of columns of **A**, or p, the number of right-hand sides. In these cases, you are advised not to compute the large $m \times m$ matrix **Q** (which can consume too much memory and time) if you can solve your problem by computing only the smaller $m \times p$ matrix $\mathbf{Q}'\mathbf{B}$ implicitly.

For example, use the first five columns of the 6×6 Hilbert matrix **A**, as follows:

```
a= {  36      -630      3360     -7560      7560     -2772,
    -630     14700    -88200    211680   -220500     83160,
    3360    -88200    564480  -1411200   1512000   -582120,
   -7560    211680  -1411200   3628800  -3969000   1552320,
    7560   -220500   1512000  -3969000   4410000  -1746360,
   -2772     83160   -582120   1552320  -1746360    698544 };
aa = a[, 1:5];
b= { 463, -13860, 97020, -258720, 291060, -116424};

m = nrow(aa); n = ncol(aa); p = ncol(b);
call qr(qtb, r, piv, lindep, aa, , b);

if lindep=0 then do;
   x=inv(r)*qtb[1:n];
   print x;  /* x solves aa*x=b */
end;
else /* handle linear dependence */;
```

Figure 25.326 Solution to Least Squares Problem

x
1
0.5
0.3333333
0.25
0.2

Note that you are using only the first n rows, $\mathbf{Q}_1'\mathbf{B}$, of the **qtb** matrix. The IF-THEN statement of the preceding example can be replaced by the more efficient TRISOLV function:

```
if lindep=0 then
   x = trisolv(1, r, qtb[1:n], piv);
```

For information about solving rank-deficient linear least squares problems, see the RZLIND call.

QUAD Call

CALL QUAD(*r, "fun", points* **<** *, eps* **>** **<** *, peak* **>** **<** *, scale* **>** **<** *, msg* **>** **<** *, cycles* **>** **);**

The QUAD subroutine performs numerical integration of scalar functions in one dimension over infinite, connected semi-infinite, and connected finite intervals.

The QUAD subroutine returns the following value:

r is a numeric vector that contains the results of the integration. The size of *r* is equal to the number of subintervals defined by the argument *points*. If the numerical integration fails on a particular subinterval, the corresponding element of *r* is set to missing.

The input arguments to the QUAD subroutine are as follows:

"fun" specifies the name of a module used to evaluate the integrand.

points specifies a sorted vector that provides the limits of integration over connected subintervals. The simplest form of the vector provides the limits of the integration on one interval. The first element of *points* should contain the left limit. The second element should be the right limit. A missing value of `.M` in the left limit is interpreted as $-\infty$, and a missing value of `.P` is interpreted as $+\infty$. For more advanced usage of the QUAD call, *points* can contain more than two elements. The elements of the vector must be sorted in an ascending order. Each two consecutive elements in *points* defines a subinterval, and the subroutine reports the integration over each specified subinterval. The use of subintervals is important because the presence of internal points of discontinuity in the integrand hinders the algorithm.

eps is an optional scalar that specifies the desired relative accuracy. It has a default value of 1E–7. You can specify *eps* by using the EPS= keyword.

peak is an optional scalar that is the approximate location of a maximum of the integrand. By default, it has a location of 0 for infinite intervals, a location that is one unit away from the finite boundary for semi-infinite intervals, and a centered location for bounded intervals. You can specify *peak* by using the PEAK= keyword.

scale is an optional scalar that is the approximate estimate of any scale in the integrand along the independent variable (see the examples). It has a default value of 1. You can specify *scale* by using the SCALE= keyword.

msg is an optional character scalar that restricts the number of messages produced by the QUAD subroutine. If *msg* = "NO" then it does not produce any warning messages. You can specify *msg* by using the MSG= keyword.

cycles is an optional integer that indicates the maximum number of refinements the QUAD subroutine can make in order to achieve the required accuracy. It has a default value of 8. You can specify *cycles* by using the CYCLES= keyword.

If the dimensions of any optional argument are 0×0, the QUAD subroutine uses its default value.

The QUAD subroutine is a numerical integrator based on adaptive Romberg-type integration techniques. See Rice (1973), Sikorsky (1982), Sikorsky and Stenger (1984), Stenger (1973a), Stenger (1973b), and Stenger (1978). Many adaptive numerical integration methods (Ralston and Rabinowitz 1978) start at one end of the interval and proceed towards the other end, working on subintervals while locally maintaining a certain prescribed precision. This is not the case with the QUAD call. The QUAD subroutine is an adaptive global-type integrator that produces a quick, rough estimate of the integration result and then refines the estimate until it achieves the prescribed accuracy. This gives the subroutine an advantage over Gauss-Hermite and Gauss-Laguerre quadratures (Ralston and Rabinowitz 1978; Squire 1987), particularly for infinite and semi-infinite intervals, because those methods perform only a single evaluation.

A Simple Example

Consider the integral

$$\int_0^{\infty} e^{-t}\, dt$$

The following statements evaluate this integral:

```
/* Define the integrand */
start fun(t);
   v = exp(-t);
   return(v);
finish;

a = {0  .P};
call quad(z, "fun", a);
print z[format=E21.14];
```

Figure 25.327 Result of Numerical Integration on a Semi-Infinite Domain

z
9.99999999595190E-01

The integration is carried out over the interval $(0, \infty)$, as specified by the **a** variable. The missing value in the second element of **a** is interpreted as ∞. The values of EPS=1E–7, PEAK=1, SCALE=1, and CYCLES=8 are used by default.

The following statements integrate the same exponential function over two subintervals:

```
a   = {0  3 .P };
call quad(z2, "fun", a);
print z2[format=E21.14];
```

Figure 25.328 Result of Numerical Integration on Two Intervals

z2
9.50212930994570E-01
4.97870683477090E-02

Notice that the elements of **a** are in ascending order. The integration is carried out over $(0, 3)$ and $(3, \infty)$, and the corresponding results are shown in the output. The values of EPS=1E–7, PEAK=1, SCALE=1, and CYCLES=8 are used by default. To obtain the results of integration over $(0, \infty)$, use the SUM function on the elements of the **z2** vector, as follows:

```
b = sum(z2);
print b[format=E21.14];
```

Figure 25.329 Result of Numerical Integration on Two Intervals

b
9.99999999342280E-01

Using the PEAK= Option

The *peak* and *scale* options enable you to avoid analytically changing the variable of the integration in order to produce a well-conditioned integrand that permits the numerical evaluation of the integration.

Consider the integration

$$\int_0^{\infty} e^{-10000t}\, dt$$

The following statements evaluate this integral:

```
start fun2(t);
   v = exp(-10000*t);
   return(v);
finish;

a = {0  .P};
/* Either syntax can be used                    */
/*   call quad(z, "fun2", a, 1E-10, 0.0001); or */
call quad(z3, "fun2", a) eps=1E-10 peak=0.0001;
print z3[format=E21.14];
```

Figure 25.330 Result of Specifying PEAK= Option

z3
9.99999999998990E-05

The integration is performed over the semi-infinite interval $(0, \infty)$. The default values of SCALE=1 and CYCLES=8 are used. However, the default value of *peak* is 1 for this semi-infinite interval, which is not a good estimate of the location of the function's maximum. If you do not specify a *peak* value, the integration cannot be evaluated to the desired accuracy, a message is printed to the SAS log, and a missing value is returned. Note that *peak* can still be set to 1E–7 and the integration will be successful.

The evaluation of the integrand at *peak* must be nonzero for the computation to continue. You should adjust the value of *peak* to get a nonzero evaluation at *peak* before trying to adjust *scale*. Reducing *scale* decreases the initial step size and can lead to an increase in the number of function evaluations per step at a linear rate.

Using the SCALE= Option

Consider the integration

$$\int_{-\infty}^{\infty} e^{-100000(t-3)^2}\, dt$$

The integrand is essentially zero except on a small interval close to $t = 3$. The following statements evaluate this integral:

```
/* Define the integrand */
start fun3(t);
   v = exp(-100000*(t-3)*(t-3));
   return(v);
finish;

a = { .M  .P };
call quad(z4, "fun3", a) eps=1E-10 peak=3 scale=0.001;
print z4[format=E21.14];
```

Figure 25.331 Result of Specifying the SCALE= Option

z4
5.60499121639830E-03

The integration is carried out over the infinite interval $(-\infty, \infty)$. The default value of CYCLES=8 has been used. The integrand has its maximum value at $t = 3$, so the PEAK=3 option is specified.

If you use the default value of *scale*, the integral cannot be evaluated to the desired accuracy, and a missing value is returned. The variables *scale* and *cycles* can be used to increase the number of possible function evaluations; the number of possible function evaluations increases linearly with the reciprocal of *scale*, but it potentially increases in an exponential manner when *cycles* is increased. Increasing the number of function evaluations increases execution time.

Two-Dimensional Integration

When you perform double integration, you must separate the variables between the iterated integrals. There should be a clear distinction between the variable of the one-dimensional integration and the parameters that are passed to the integrand. Another important consideration is specifying the correct limits of integration.

For example, suppose you want to compute probabilities for the standard bivariate normal distribution with correlation ρ. In particular, if an observation (x, y) is drawn from the distribution, what is probability that

$x \leq a$ and $y \leq b$ for given values of a and b?

The bivariate normal probability is given by the following double integral:

$$\text{probbnrm}(a, b, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{a} \int_{-\infty}^{b} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}\right) dy\, dx$$

The inner integral is

$$g(x, b, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \int_{-\infty}^{b} \exp\left(-\frac{x^2 - 2\rho xy + y^2}{2(1-\rho^2)}\right) dy$$

with parameters x and ρ, and the limits of integration are from $-\infty$ to b. The outer integral is then

$$\text{probbnrm}(a, b, \rho) = \int_{-\infty}^{a} g(x, b, \rho)\, dx$$

with the limits from $-\infty$ to a.

You can write a function module with parameters a, b, ρ that computes the bivariate normal probability. In the following statements, the function module is called NORCDF2 because it compute the CDF of the bivariate normal distribution. The NORCDF2 module calls the QUAD subroutine on the MARGINAL module, which computes the outer integral. The MARGINAL module, in turn, uses the QUAD function to evaluate inner integral. The integrand of the inner integral is defined in the NORPDF2 module.

```
/*-----------------------------------------------------*/
/* This function is the density function and requires  */
/* the variable T (passed in the argument)             */
/* and a list of global parameters, YV, RHO, COUNT     */
/*-----------------------------------------------------*/
start norpdf2(t) global(yv,rho,count);
   count = count+1;
   q=(t#t-2#rho#t#yv+yv#yv)/(1-rho#rho);
   p=exp(-q/2);
   return(p);
finish;

/*-----------------------------------------------------*/
/* The outer integral                                  */
/* The limits of integration are .M to YY              */
/* YV is passed as a parameter to the inner integral*/
/*-----------------------------------------------------*/
start marginal(v) global(yy,yv,eps);
   interval = .M || yy;
   if ( v < -12 ) then return(0);
   yv = v;
   call quad(pm, "NORPDF2", interval) eps=eps;
   return(pm);
finish;

/*-----------------------------------------------------*/
/* Global parameters: YY, RHO, EPS                     */
```

```
/* EPS is set from IML                                */
/*---------------------------------------------------*/
start norcdf2(a, b, rrho) global(yy,rho,eps);
   rho = rrho; /* copy arguments (local variables) to global list */
   yy = b;

   interval= .M || a; /* upper/lower limits for outer integral */
   call quad(p,"MARGINAL",interval) eps=eps;

   pi = constant("Pi");
   per = p /(2#pi#sqrt(1-rho#rho)); /* scale the value from QUAD */
   return(per);
finish;

/*-----------------------------------------------------*/
/* Main Program: set up global constants and call QUAD */
/*-----------------------------------------------------*/
count = 0;
eps = 1E-11;

p =  norcdf2(2, 1, 0.1);
print p[format=E21.14], count;
```

Figure 25.332 Result of Numerical Integration of a Double Iterated Integral

p
8.23640898880300E-01

count
250453

The variable COUNT contains the number of times the NORPDF2 module is called. Note that the value computed by the NORCDF2 module is very close to that returned by the PROBBNRM function, which computes probabilities for the bivariate normal model, as shown by the following statements:

```
/* Compute the value with the PROBBNRM function */
pp = probbnrm(2,1,0.1);
print pp[format=E21.14];
```

Figure 25.333 Result of Numerical Integration of a Double Iterated Integral

pp
8.23640898880500E-01

Note the following:

- The iterated inner integral cannot have a left endpoint of $-\infty$. For large values of v, the inner integral does not contribute to the answer but still needs to be computed to the required relative accuracy. Therefore, either cut off the function (when $v \leq -12$), as in the MARGINAL module in the preceding example, or have the intervals start from a reasonable cutoff value. In addition, the QUAD subroutine stops if the integrands appear to be identically 0 (probably caused by underflow) over the interval of integration.

- This method of integration (iterated, one-dimensional integrals) is extremely conservative and requires unnecessary function evaluations. In this example, the QUAD subroutine for the inner integration lacks information about the final value that the QUAD subroutine for the outer integration is trying to refine. The lack of communication between the two QUAD routines can cause useless computations to be performed in the inner integration.

  To illustrate this idea, let the relative error be 1E–11 and let the answer delivered by the outer integral be close to 0.8, as in this example. Any computation of the inner execution of the QUAD call that yields 0.8E–11 or less does not contribute to the final answer of the QUAD subroutine for the outer integral. However, the inner integral lacks this information, and for a given value of the parameter yv, it attempts to compute an answer with much more precision than is necessary. The lack of communication between the two QUAD subroutines prevents the introduction of better cutoffs. Although this method can be inefficient, the final calculations are accurate.

QUARTILE Function

QUARTILE(*matrix***);**

The QUARTILE function is part of the IMLMLIB library. Given an $n \times m$ data matrix, the QUARTILE function returns a $5 \times m$ matrix. The rows of the return matrix contain the minimum, lower quartile, median, upper quartile, and maximum values (respectively) for the data in *matrix*. Missing values are excluded from the computation. If all values in a column are missing, the return values for that column are missing.

```
use Sashelp.Class;
read all var _NUM_ into X[colname=varNames];
close Sashelp.Class;
q = quartile(X);
rn = {"Minimum" "Q1" "Median" "Q2" "Maximum"};
print q[rowname=rn colname=varNames];
```

Figure 25.334 Quartiles

	q		
	Age	Height	Weight
Minimum	11	51.3	50.5
Q1	12	57.5	84
Median	13	62.8	99.5
Q2	15	66.5	112.5
Maximum	16	72	150

For the computation of arbitrary quantiles, see the documentation for the QNTL call.

QUEUE Call

CALL QUEUE(*argument1 < , argument2, . . . , argument15>* **);**

The QUEUE subroutine places character arguments that contain valid SAS statements (usually SAS/IML statements or global statements) at the end of the input command stream. You can specify up to 15 arguments.

Each argument to the QUEUE subroutine is a character matrix or quoted literal that contains valid SAS statements.

The queued string is read after other lines of input already in the queue. If you want to push the lines in front of other lines already in the queue, use the PUSH subroutine instead. Any statements queued to the input command queue get executed when the module is paused (see the PAUSE statement), which happens when one of the following occurs:

- An execution error occurs within a module.
- An interrupt is issued.
- A PAUSE statement executes.

The strings you queue do not appear on the log.

CAUTION: Do not queue too many statements at one time. Queuing too many statements can cause problems that can result in exiting the SAS System.

For more examples, see Chapter 20.

An example that uses the QUEUE subroutine follows:

```
start mod(x);
   code="x=0;";
   call queue (code,"resume;");
   pause;
finish;

x=1;
run mod(x);
print x;
```

Figure 25.335 Result of Evaluating Queued Statements

x
0

QUIT Statement

QUIT ;

Use the QUIT statement to exit PROC IML. If a DATA or PROC statement is encountered, QUIT is implied. The QUIT statement is executed immediately; therefore, you cannot use QUIT as an executable statement (that is, as part of a module or conditional clause). However, you can use the ABORT statement as an executable statement.

PROC IML closes all open data sets and files when a QUIT statement is encountered. Workspace and symbol spaces are freed up. If you need to use any matrix values or any module definitions in a later session, you must store them in a storage library before you quit.

RANCOMB Function

RANCOMB(*n, k* < *, numcomb*> **);**

RANCOMB(*set, k* < *, numcomb*> **);**

The RANCOMB function generates random combinations of *k* elements taken from a set of *n* elements. The random number seed is set by the RANDSEED subroutine.

The first argument, *set*, can be a scalar or a vector. If *set* is a scalar, the function returns indices in the range 1–*n*. If *set* is a vector, the number of elements of the vector determines *n* and the RANCOMB function returns elements of *set*.

By default, the RANCOMB function returns a single random combination with one row and *k* columns. If the *numcomb* argument is specified, the function returns a matrix with *numcomb* rows and *k* columns. Each row of the returned matrix represents a single combination.

The following statements generate five random combinations of two elements from the set $\{1, 2, 3, 4\}$:

```
n = 4;
k = 2;
call randseed(1234);
c = rancomb(n, k, 5);
print c;
```

Figure 25.336 Random Pairwise Combinations of Four Items

c
1 4
1 2
2 4
2 3
1 3

The function can return combinations for arbitrary numerical or character matrices. For example, the following statements generate five random pairwise combinations of four elements:

```
d = rancomb({A B C D}, 2, 5);
print d;
```

Figure 25.337 Random Pairwise Combinations of Four Characters

d
A D
A B
A D
B D
A B

RANDDIRICHLET Function

RANDDIRICHLET(*N, Shape*);

The RANDDIRICHLET function is part of the IMLMLIB library. The RANDDIRICHLET function generates a random sample from a Dirichlet distribution, which is a multivariate generalization of the beta distribution.

The input parameters are as follows:

N	is the number of observations to sample.
Shape	is a $1 \times (p+1)$ vector of shape parameters for the distribution, Shape$[i] > 0$.

The RANDDIRICHLET function returns an $N \times p$ matrix that contains N random draws from the Dirichlet distribution.

If $X = \{X_1\ X_2\ \ldots\ X_p\}$ with $\sum_{i=1}^{p} X_i < 1$ and $X_i > 0$ follows a Dirichlet distribution with shape parameter $\alpha = \{\alpha_1\ \alpha_2\ \ldots\ \alpha_{p+1}\}$, then

- the probability density function for x is

$$f(x;\alpha) = \frac{\Gamma(\sum_{i=1}^{p+1}\alpha_i)}{\prod_{i=1}^{p+1}\Gamma(\alpha_i)} \prod_{i=1}^{p} {x_i}^{\alpha_i - 1}(1 - x_1 - x_2 - \ldots - x_p)^{\alpha_{p+1}-1}$$

- if $p = 1$, the probability distribution is a beta distribution.
- if $\alpha_0 = \Sigma_{i=1}^{p+1}\alpha_i$, then
  - the expected value of X_i is α_i/α_0.
  - the variance of X_i is $\alpha_i(\alpha_0 - \alpha_i)/(\alpha_0^2(\alpha_0 + 1))$.
  - the covariance of X_i and X_j is $-\alpha_i\alpha_j/(\alpha_0^2(\alpha_0 + 1))$.

The following example generates 1,000 samples from a two-dimensional Dirichlet distribution. Each row of the returned matrix x is a row vector sampled from the Dirichlet distribution. The following example computes the sample mean and covariance and compares them with the expected values:

```
call randseed(1);
n = 1000;
Shape = {2, 1, 1};
x = RandDirichlet(n,Shape);
d = nrow(Shape)-1;
s = Shape[1:d];
Shape0 = sum(Shape);
Mean = s`/Shape0;
Cov = -s*s` / (Shape0##2*(Shape0+1));
/* replace diagonal elements with variance */
Variance = s#(Shape0-s) / (Shape0##2*(Shape0+1));
do i = 1 to d;
   Cov[i,i] = Variance[i];
```

```
end;

SampleMean = mean(x);
SampleCov = cov(x);
print SampleMean Mean, SampleCov Cov;
```

Figure 25.338 Estimated Mean and Covariance Matrix

SampleMean		Mean	
0.4992449	0.2485677	0.5	0.25

SampleCov		Cov	
0.0502652	-0.026085	0.05	-0.025
-0.026085	0.0393922	-0.025	0.0375

For further details about sampling from the Dirichlet distribution, see Kotz, Balakrishnan, and Johnson (2000); Gentle (2003); or Devroye (1986).

RANDFUN Function

RANDFUN(*N, "Distribution"* < *, param1*> < *, param2*> < *, param3*>**);**

The RANDFUN function is part of the IMLMLIB library. The RANDFUN function is a convenient interface to the RANDGEN subroutine. If *N* is a positive integer, the function returns an $N \times 1$ column vector of random numbers that are drawn from the *Distribution* family with the specified parameters. If *N* is a vector that contains a pair of integers, the function returns an $N[1] \times N[2]$ matrix of random numbers.

For simulation studies that generate matrices of random numbers within a DO loop, it is more efficient to use the RANDGEN subroutine.

The following example simulates data from three distributions:

- the Bernoulli distribution with probability $p = 1/2$
- the uniform distribution on the interval $(0, 1)$
- the normal distribution with mean 5 and standard deviation 2

```
call randseed(123);
b = randfun({4 2}, "Bernoulli", 0.5); /* 4 rows, 2 cols */
u = randfun(4, "Uniform");
x = randfun(4, "Normal", 5, 2);
print b u x;
```

Figure 25.339 Random Samples from Three Distributions

b		u	x
0	1	0.056701	4.956251
1	1	0.0798305	5.8587927
1	1	0.9233735	5.9284455
1	1	0.2258509	6.2806841

RANDGEN Call

CALL RANDGEN(*result, distname* < *, parm1* > < *, parm2*> < *, parm3*> **);**

The RANDGEN subroutine generates random numbers from a specified distribution.

The subroutine takes the following input arguments:

result	specifies a matrix that is to be filled with random samples from the specified distribution.
distname	specifies the name of a probability distribution.
parm1	specifies a distribution parameter.
parm2	specifies a distribution parameter.
parm3	specifies a distribution parameter.

For most distributions, the RANDGEN subroutine generates random numbers by using the same numerical method as the RAND function in Base SAS software, with the efficiency optimized for matrices. You can initialize the random number stream that is used by RANDGEN by calling the RANDSEED subroutine. The *result* parameter should be allocated to a size equal to the number of values that you want to generate. If *result* is not initialized, then it receives a single random value.

The following statements fill a vector with 1,000 random values from a standard normal distribution:

```
call randseed(12345);
x = j(1000,1);              /* allocate (1000 x 1) vector */
call randgen(x, "Normal"); /* fill it */
```

Vectors of Parameters

Except for the "Table" and "NormalMix" distributions, the distribution parameters are usually scalar values. However, the RANDGEN subroutine also accepts vectors of parameters. If *result* is an $n \times m$ matrix, then *parm1*, *parm2*, and *parm3* can contain 1, n, m, or nm elements. The different sizes are interpreted as follows:

- If the parameters are scalar quantities, each element of *result* is a sample value from the same distribution.
- Otherwise, if the parameters contain m elements, the jth column of the *result* matrix consists of random values drawn from the distribution with parameters *param1[j]*, *param2[j]*, and *param3[j]*.
- Otherwise, if the parameters contain n elements, the ith row of the *result* matrix consists of random values drawn from the distribution with parameters *param1[i]*, *param2[i]*, and *param3[i]*.
- Otherwise, if the parameters contain nm elements, the (i, j) element of the *result* matrix contains a random value drawn from the distribution with parameters *param1[s]*, *param2[s]*, and *param3[s]*, where $s = m(i-1) + j$.

All parameters must be the same length. You cannot specify a scalar for one parameter and a vector for another. If you pass in parameter vectors that do not satisfy one of the preceding conditions, then the first element of each parameter is used.

For example, the jth column of the following matrix is a sample drawn from a normal population with mean j and standard deviation $j/4$:

```
n = 5; m = 4;
x = j(n, m);
Mu = 1:m;
Sigma = (1:m) / m;
call randgen(x, "Normal", Mu, Sigma);
print x;
```

Figure 25.340 Columns Drawn from Different Distributions

x			
0.7953097	2.109807	2.5903507	4.567692
1.1153841	1.9143935	3.5193908	4.461049
1.1036757	2.6768648	3.3873821	4.5642427
1.1543757	1.6322845	2.6431948	4.2777107
0.8030879	1.4097247	3.0206292	2.6724841

The following sections describe the distributions that are supported.

Bernoulli Distribution

To generate random variates from the Bernoulli distribution, specify 'Bernoulli' for the *distname* argument. The subroutine generates random values of x in the discrete set $\{0, 1\}$. The values are drawn from the probability mass function:

$$f(x) = \begin{cases} 1 & \text{for } p = 0, x = 0 \\ p^x(1-p)^{1-x} & \text{for } 0 < p < 1, x = 0, 1 \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

The parameter p, $0 \leq p \leq 1$, is the probability of a "success." A success means that x has the value 1.

Beta Distribution

To generate random variates from the beta distribution, specify 'Beta' for the *distname* argument. The subroutine generates random values of x in the bounded interval $(0, 1)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

The shape parameters $a > 0$ and $b > 0$ are required. Be aware that small values of the parameters $(a, b < 1)$ correspond to a U-shaped density that has singularities. Very small parameter values $(a, b \ll 1)$ can lead to loss of precision because of numerical underflow.

Binomial Distribution

To generate random variates from the binomial distribution, specify 'Binomial' for the *distname* argument. The subroutine generates values of x in the discrete set $\{0, 1, \ldots, n\}$. The values of x are drawn from the probability mass function:

$$f(x) = \begin{cases} 1 & \text{for } p = 0, x = 0 \\ \binom{n}{x} p^x(1-p)^{n-x} & \text{for } 0 < p < 1, x = 0, \ldots, n \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

The parameter p is the success probability, with range $0 \leq p \leq 1$. The parameter n specifies an integer number of independent trials, $n \geq 0$.

Intuitively, x is the number of successes in n independent Bernoulli trials with probability p of success.

Cauchy Distribution

To generate random variates from the Cauchy distribution, specify 'Cauchy' for the *distname* argument. The subroutine generates random values of x in the interval $(-\infty, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{1}{\pi(1 + x^2)}$$

Chi-Square Distribution

To generate random variates from the chi-square distribution, specify 'ChiSquare' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{2^{-d/2}}{\Gamma(\frac{d}{2})} x^{d/2-1} e^{-x/2}$$

The parameter d represents degrees of freedom, with $d > 0$. Often d is an integer, but it does not have to be. Very small parameter values ($d \ll 1$) can lead to loss of precision because of numerical underflow.

Conway-Maxwell-Poisson Distribution

The Conway-Maxwell-Poisson distribution (also called the COM-Poisson distribution) is a generalization of the Poisson distribution that enables you to model underdispersed and overdispersed data. To generate random variates from the Conway-Maxwell-Poisson distribution, specify 'ConMaxPoi' for the *distname* argument. The subroutine generates random values of x in the discrete set $\{0, 1, 2, \ldots\}$. The probability mass function for the COM-Poisson distribution is defined by

$$f(x) = \frac{1}{Z(\lambda, \nu)} \frac{\lambda^x}{(x!)^\nu}$$

where the normalization factor is $Z(\lambda, \nu) = \Sigma_{n=0}^{\infty} \lambda^n / (n!)^\nu$.

The parameters are a location parameter $\lambda > 0$ and a dispersion parameter $\nu \geq 0$. When $\nu = 1$, the distribution is Poisson with rate parameter λ. When $\nu = 0$ and $\lambda < 1$, the distribution is geometric with probability of success $1 - \lambda$.

The Conway-Maxwell-Poisson distribution is explained further in the documentation for the PDF function in the *SAS Language Reference: Dictionary*.

Erlang Distribution

To generate random variates from the Erlang distribution, specify 'Erlang' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{1}{\lambda^a \Gamma(a)} x^{a-1} e^{-x/\lambda}$$

The Erlang distribution is a gamma distribution with an integer value for the shape parameter $a = 1, 2, \ldots$. The optional scale parameter $\lambda > 0$ has the default value $\lambda = 1$.

Exponential Distribution

To generate random variates from the exponential distribution, specify 'Exponential' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{e^{-x/\sigma}}{\sigma}$$

The optional scale parameter $\sigma > 0$ has the default value $\sigma = 1$.

Extreme Value Distribution

The generalized extreme value distribution (for a maximum) contains three parameters: a location parameter μ, a scale parameter σ, and a shape parameter ξ. The default values are $\mu = 0$, $\sigma = 1$, and $\xi = 0$. To generate random variates from the extreme-value distribution, specify 'ExtremeValue' for the *distname* argument. The values of x are drawn from the cumulative distribution function,

$$F(x) = e^{-t(x)}$$

where $t(x) = (1 + \xi(x - \mu)/\sigma)^{-1/\xi}$ for $\xi \neq 0$ and where $t(x) = \exp(-(x - \mu)/\sigma)$ for $\xi = 0$.

When $\xi = 0$, the distribution is a Type 1 distribution, sometimes called a Gumbel-type distribution. The random values of x are in the interval $(-\infty, \infty)$. For $\xi = 0$, it is more efficient to use the Gumbel distribution, which is supported by the RANDGEN subroutine.

When $\xi > 0$, the distribution is a Type 2 distribution, sometimes called a Fréchet-type distribution. The random values of x are in the interval $(\mu - \sigma/\xi, \infty)$.

When $\xi < 0$, the distribution is a Type 3 distribution, sometimes called a Weibull-type distribution. The random values of x are in the interval $(-\infty, \mu - \sigma/\xi)$.

F Distribution ($F_{n,d}$)

1

To generate random variates from the F distribution, specify 'F' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{\Gamma(\frac{n+d}{2}) n^{\frac{n}{2}} d^{\frac{d}{2}} x^{\frac{n}{2}-1}}{\Gamma(\frac{n}{2})\Gamma(\frac{d}{2})(d + nx)^{\frac{n+d}{2}}}$$

The two parameters n and d are degrees of freedom, with values $n > 0$ and $d > 0$.

Gamma Distribution

To generate random variates from the gamma distribution, specify 'Gamma' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{x^{a-1}}{\lambda^a \Gamma(a)} e^{-x/\lambda}$$

The parameter a is a shape parameter, $a > 0$. The optional scale parameter $\lambda > 0$ has the default value $\lambda = 1$.

Generalized Poisson Distribution

To generate random variates from the generalized Poisson distribution, specify 'GenPoisson' for the *distname* argument. The subroutine generates random values of x in the discrete set $\{0, 1, 2, \ldots\}$.

The generalized Poisson distribution has two shape parameters: $\theta > 0$ and $0 \leq \eta < 1$. When $\eta = 0$, the distribution is the Poisson distribution with mean and variance of θ. When $\eta > 0$, the mean is $\theta/(1-\eta)$ and the variance is $\theta/(1-\eta)^3$. Very large values of θ can result in loss of precision. Similarly, numerical difficulties can occur when η is close to 1; $\eta \leq 0.95$ is recommended.

The values of x are drawn from the following probability mass function:

$$f(x) = \frac{\theta}{x!}(\theta + \eta x)^{x-1} \exp(-\theta - \eta x)$$

Geometric Distribution

The geometric distribution is the distribution of a random variable that represents the number of Bernoulli trials (with probability p) until the first success occurs. To generate random variates from the geometric distribution, specify 'Geometric' for the *distname* argument. The subroutine generates random values of x in the discrete set $\{1, 2, \ldots\}$. The values of x are drawn from the probability mass function:

$$f(x) = \begin{cases} (1-p)^{x-1} p & \text{for } 0 < p < 1, x = 1, 2, \ldots \\ 1 & \text{for } p = 1, x = 1 \end{cases}$$

The parameter p is the success probability, where $0 < p \leq 1$.

Gompertz Distribution

To generate random variates from the geometric distribution, specify 'Gompertz' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the following cumulative probability distribution:

$$F(x) = 1 - \exp(-\lambda \exp(x/\sigma) - 1)$$

The shape parameter λ has the default value $\lambda = 1$. The scale parameter $\theta > 0$ has the default value $\theta = 1$.

Gumbel Distribution

To generate random variates from the Gumbel distribution, specify 'Gumbel' for the *distname* argument. The subroutine generates random values of x in the interval $(-\infty, \infty)$. The values of x are drawn from the following cumulative probability distribution:

$$F(x) = \exp(-\exp(-(x-\mu)/\sigma))$$

The location parameter μ has the default value $\mu = 0$. The scale parameter $\sigma > 0$ has the default value $\sigma = 1$. The Gumbel distribution is an extreme value distribution for the maximum. The distribution is skewed to the right.

Hypergeometric Distribution

To generate random variates from the hypergeometric distribution, specify 'Hypergeometric' for the *distname* argument. The values of x are drawn from the probability mass function:

$$f(x) = \frac{\binom{R}{x}\binom{N-R}{k-x}}{\binom{N}{k}}$$

The parameter N is the population size, with range $N = 1, 2, \ldots$. The parameter R is the size of the category of interest, with range $R = 0, 1, \ldots, N$. The parameter k is the sample size, with range $k = 0, 1, \ldots, N$. The subroutine generates random integer values in the discrete set $\{a, a+1, \ldots, b-1, b\}$, where $a = \max(0, k-(N-R))$ and $b = \min(k, R)$.

Intuitively, x is obtained by the following experiment. Put R red balls and $N - R$ black balls into an urn. The value x is the number of red balls in a sample of size k that is drawn from the urn without replacement.

Integer Distribution

The integer distribution is also called the discrete uniform distribution. To generate random variates from the integer distribution, specify 'Integer' for the *distname* argument.

The subroutine generates random integer values in a finite set. If you specify one integer parameter, c, then x is drawn uniformly from the set $\{1, 2, \ldots, c-1, c\}$. If you specify two integer parameters, a and b with $a \leq b$, then x is drawn uniformly from the set $\{a, a+1, \ldots, b-1, b\}$.

Laplace Distribution

To generate random variates from the Laplace distribution, specify 'Laplace' for the *distname* argument. The subroutine generates random values of x in the interval $(-\infty, \infty)$.

The values of x are drawn from the probability density function:

$$f(x) = \frac{1}{2\lambda} \exp\left(-\frac{|x-\theta|}{\lambda}\right)$$

The optional location parameter θ has the default value $\theta = 0$. The optional scale parameter $\lambda > 0$ has the default value $\lambda = 1$.

Logistic Distribution

To generate random variates from the logistic distribution, specify 'Logistic' for the *distname* argument. The subroutine generates random values of x in the interval $(-\infty, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{\exp\left(-(x-\theta)/\lambda\right)}{\lambda\left(1+\exp\left(-(x-\theta)/\lambda\right)\right)^2}$$

The optional location parameter θ has the default value $\theta = 0$. The optional scale parameter $\lambda > 0$ has the default value $\lambda = 1$.

Lognormal Distribution

If a random variable X is lognormally distributed, then $\log(X)$ is normally distributed. To generate random variates from the lognormal distribution, specify 'Lognormal' for the *distname* argument. The subroutine generates random values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{1}{x\lambda\sqrt{2\pi}} \exp\left(-\frac{(\ln(x)-\theta)^2}{2\lambda^2}\right)$$

The optional log-scale parameter θ has the default value $\theta = 0$. The optional shape parameter $\lambda > 0$ has the default value $\lambda = 1$.

Negative Binomial Distribution

To generate random variates from the negative binomial distribution, specify 'NegBinomial' for the *distname* argument. The subroutine generates values of x in the discrete set $\{0, 1, \ldots, n\}$. The values of x are drawn from the probability mass function:

$$f(x) = \begin{cases} \binom{x+k-1}{k-1}(1-p)^x p^k & \text{for } 0 < p < 1, x = 0, 1, \ldots \\ 1 & \text{for } p = 1, x = 0 \end{cases}$$

The parameter p is the success probability with range $0 < p \leq 1$. The parameter k is an integer that counts the number of successes, with range $k = 1, 2, \ldots$.

Intuitively, x is the number of failures before the kth success during a series of Bernoulli trials with probability of success p.

Normal Distribution

To generate random variates from the normal distribution, specify 'Normal' for the *distname* argument. The subroutine generates random values of x in the interval $(-\infty, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{1}{\lambda\sqrt{2\pi}} \exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

The optional parameter θ $(-\infty < \theta < \infty)$ is the mean (location) parameter, which has the default value $\theta = 0$. The optional parameter $\lambda > 0$ is the standard deviation, which has the default value $\lambda = 1$.

Normal Mixture Distribution

To generate random variates from a finite mixture of normal distributions, specify 'NormalMix' for the *distname* argument. The subroutine generates random values of x in the interval $(-\infty, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \sum_{i=1}^{n} p_i \phi(x; \mu_i, \sigma_i)$$

where $\phi(x; \mu_i, \sigma_i)$ is the normal PDF with mean μ_i and standard deviation σ_i, and where p is a vector of probabilities such that

$$\sum_{i=1}^{n} p_i = 1$$

The parameters p, μ, and σ are vectors with n elements.

Pareto Distribution

To generate random variates from the Pareto distribution, specify 'Pareto' for the *distname* argument. The subroutine generates values of x in the interval (k, ∞). The values of x are drawn from the probability density function:

$$f(x) = \frac{a}{k}\left(\frac{k}{x}\right)^{a+1}$$

The shape parameter a is valid for $a > 0$. The optional scale parameter $k > 0$ has the default value $k = 1$.

Poisson Distribution

To generate random variates from the Poisson distribution, specify 'Poisson' for the *distname* argument. The subroutine generates values of x in the discrete set $\{0, 1, 2, \ldots\}$. The values of x are drawn from the probability density function:

$$f(x) = \frac{m^x e^{-m}}{x!}$$

The parameter m is a rate parameter with range $m > 0$.

Shifted Gompertz

To generate random variates from the shifted Gompertz distribution, specify 'ShGompertz' for the *distname* argument. The subroutine generates values of x in the interval $(0, \infty)$. The values of x are drawn from the following cumulative probability distribution:

$$F(x) = (1 - \exp(-\tau x)) \exp(-\eta \exp(-\tau x))$$

The shape parameter $\eta > 0$ has the default value $\eta = 1$. The inverse scale parameter $\tau > 0$ has the default value $\tau = 1$. As $\eta \to 0$, the shifted Gompertz distribution approaches the exponential distribution with shape parameter $1/\tau$.

t Distribution

To generate random variates from the *t* distribution, specify 'T' for the *distname* argument. The subroutine generates values of x in the interval $(-\infty, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{\Gamma\left(\frac{d+1}{2}\right)}{\sqrt{d\,\pi}\,\Gamma\left(\frac{d}{2}\right)}\left(1+\frac{x^2}{d}\right)^{-\frac{d+1}{2}}$$

The parameter d is the degrees of freedom, with the range $d > 0$.

Table Distribution

A table distribution (also called a tabled distribution) is a discrete distribution in which you specify the probability that a random variable takes on each of n values. To generate random variates from a table distribution, specify 'Table' for the *distname* argument. The values of x are drawn 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

$$\sum_{j=1}^{n} p_j \leq 1$$

If $\sum p_j = 1$, the subroutine generates values of x in the discrete set $\{0, 1, \ldots, n\}$.

Triangle Distribution

To generate random variates from the triangle distribution, specify 'Triangle' for the *distname* argument. The subroutine generates values of x in the bounded interval $(0, 1)$. The values of x are drawn from the piecewise linear probability density function:

$$f(x) = \begin{cases} \frac{2x}{h} & \text{for } 0 \leq x \leq h \\ \frac{2(1-x)}{1-h} & \text{for } h < x \leq 1 \end{cases}$$

The parameter h is the horizontal location of the peak of the triangle, with range $0 \leq h \leq 1$.

Tweedie Distribution

To generate random variates from the Tweedie distribution, specify 'Tweedie' for the *distname* argument. The subroutine generates values of x in the interval $(0, \infty)$.

The Tweedie distribution has three parameters: $p \geq 1$ is the power parameter, $\mu > 0$ is the mean of the distribution, and $\phi > 0$ is a scale parameter. The default values for the optional parameters are $\mu = 1$ and $\phi = 1$. The Tweedie distribution has the property that the variance of the distribution is equal to $\phi\mu^p$.

The density function is given by

$$f(x) = a(x,\phi)\exp\left[\frac{1}{\phi}\left(\frac{x\mu^{1-p}}{1-p} - \kappa(\mu,p)\right)\right]$$

where $\kappa(\mu,p) = \mu^{2-p}/(2-p)$ for $p \neq 2$ and $\kappa(\mu,p) = \log(\mu)$ for $p = 2$. The function $a(x,\phi)$ does not have an analytical expression, but is typically represented by an infinite series.

For most modeling tasks, $1 < p < 2$. For p in this range, the Tweedie distribution is a sum of N gamma random variables, where N is Poisson distributed. For more information, see the documentation for the SEVERITY procedure in the *SAS/ETS User's Guide*. The documentation for the PDF function in *SAS Language Reference: Dictionary* is also relevant.

Uniform Distribution

To generate random variates from the uniform distribution, specify 'Uniform' for the *distname* argument. The subroutine generates values of x in the bounded interval (a, b). The values of x are drawn from the constant probability density function:

$$f(x) = \begin{cases} 1 & \text{if } a = b \\ \frac{1}{|b-a|} & \text{if } a \neq b \end{cases}$$

If you do not specify any parameters, the range of x is $(0, 1)$. If you specify one parameter, c, then the range is $(0, c)$. If you specify two parameters, a and b, then $a < x < b$.

Wald (Inverse Gaussian) Distribution

To generate random variates from the Wald (inverse Gaussian) distribution, specify 'Wald' or 'IGauss' for the *distname* argument. The subroutine generates values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \left(\frac{\lambda}{2\pi x^3}\right)^{\frac{1}{2}} \exp\left(\frac{-\lambda(x-\theta)^2}{2\theta^2 x}\right)$$

The parameter $\lambda > 0$ is a shape parameter. The optional parameter θ has the default value $\theta = 1$.

Notice that many references, including the MCMC procedure in SAS/STAT software, list θ as the first parameter for the inverse Gaussian distribution. However, the θ parameter is listed last for the RAND, PDF, CDF, and QUANTILE functions because it an optional parameter.

The RAND function uses a slightly different algorithm for generating the Wald variates. You can specify the 'Wald2' distribution to obtain the same variates that are generated by the RAND function.

Weibull Distribution

To generate random variates from the Weibull distribution, specify 'Weibull' for the *distname* argument. The subroutine generates values of x in the interval $(0, \infty)$. The values of x are drawn from the probability density function:

$$f(x) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} \exp\left(-\left(\frac{x}{b}\right)^a\right)$$

The shape parameters a and b are have values $a > 0$ and $b > 0$. The default value of b is 1.

Summary of Distributions

Table 25.6 describes how parameters of the RANDGEN call correspond to the distribution parameters. Optional arguments are enclosed in angle brackets, along with the default value when the argument is not specified. For example, if you do not supply values for the parameters of the normal distribution, the default values of $\theta = 0$ and $\lambda = 1$ are used.

Table 25.6 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'	d		
COM-Poisson	'ConMaxPoi'	λ	ν	
Erlang	'Erlang'	a	$<\lambda = 1>$	
Exponential	'Exponential'	$<\sigma = 1>$		
Extreme value	'ExtremeValue'	$<\mu = 0>$	$<\sigma = 1>$	$<\xi = 0>$
$F_{n,d}$	'F'	n	d	
Gamma	'Gamma'	a	$<\lambda = 1>$	
Generalized Poisson	'GenPoisson'	θ	η	
Geometric	'Geometric'	p		
Gompertz	'Gompertz'	$<\lambda = 1>$	$<\theta = 1>$	
Gumbel	'Gumbel'	$<\mu = 0>$	$<\sigma = 1>$	
Hypergeometric	'Hypergeometric'	N	R	n
Integer	'Integer'	a	$<b>$	
Laplace	'Laplace'	$<\theta = 0>$	$<\lambda = 1>$	
Logistic	'Logistic'	$<\theta = 0>$	$<\lambda = 1>$	
Lognormal	'Lognormal'	$<\theta = 0>$	$<\lambda = 1>$	
Negative binomial	'NegBinomial'	p	k	
Normal	'Normal'	$<\theta = 0>$	$<\lambda = 1>$	
Normal mixture	'NormalMix'	p	μ	σ
Pareto	'Pareto'	a	$<k = 1>$	
Poisson	'Poisson'	m		
t	'T'	d		
Shifted Gompertz	'ShGompertz'	$\eta = 1$	$\tau = 1$	
Table	'Table'	p		
Triangle	'Triangle'	h		
Tweedie	'Tweedie'	p	$<\mu = 1>$	$<\phi = 1>$
Uniform	'Uniform'	$<a = 0>$	$<b = 1>$	
Wald	'Wald' or 'IGauss'	$<\lambda = 1>$	$<\theta = 1>$	
Weibull	'Weibull'	a	$<b = 1>$	

The *distname* argument can be in lowercase or uppercase, and you need to specify only enough letters to distinguish one distribution from the others, as shown by the following statements:

```
/* generate 10 samples from a Bernoulli distribution */
r = j(10, 1, .);         /* allocate room for samples */
call randgen(r, "Ber", 0.5);
```

The following example illustrates the RANDGEN call for various distributions:

```
call randseed(12345);
/* get four random observations from each distribution */
x = j(1, 4, .);
/* each row comes from a different distribution */
DiscreteDist = {'Bern','Binom','ConMaxPoi','GenPoi','Geom',
                'Hyper','Integer','NegB','Poisson','Table'};
D = j(nrow(DiscreteDist), 4, .);
i = 1;
call randgen(x, 'Bern', 0.75);        D[i, ] = x;  i = i+1;
call randgen(x, 'Binom', 0.75, 10);   D[i, ] = x;  i = i+1;
call randgen(x, 'ConMax', 2.3, 0.4);  D[i, ] = x;  i = i+1;
call randgen(x, 'GenPoi', 1, 0.7);    D[i, ] = x;  i = i+1;
call randgen(x, 'Geom', 0.1);         D[i, ] = x;  i = i+1;
call randgen(x, 'Hyper', 10, 3, 5);   D[i, ] = x;  i = i+1;
call randgen(x, 'Integer', 1, 10);    D[i, ] = x;  i = i+1;
call randgen(x, 'NegB', 0.8, 5);      D[i, ] = x;  i = i+1;
call randgen(x, 'Poisson', 6.1);      D[i, ] = x;  i = i+1;
p = {0.2 0.5 0.3};
call randgen(x, 'Table', p);          D[i, ] = x;  i = i+1;
print D[rowname=DiscreteDist label="Discrete"];

ContinDist = {'Beta','Cauchy','ChiSq','Erlang','Expo',
              'ExtremeVal','F','Gamma','Gompertz','Gumbel',
              'Laplace','Logistic','LogN','Normal',
              'NormalMix','Pareto','ShGompertz','T',
              'Triangle','Tweedie','Uniform','Wald','Weib'};
C = j(nrow(ContinDist), 4, .);
i = 1;
call randgen(x, 'Beta', 0.5, 0.4);    C[i, ] = x;  i = i+1;
call randgen(x, 'Cauchy');            C[i, ] = x;  i = i+1;
call randgen(x, 'ChiSq', 22);         C[i, ] = x;  i = i+1;
call randgen(x, 'Erlang', 7);         C[i, ] = x;  i = i+1;
call randgen(x, 'Expo');              C[i, ] = x;  i = i+1;
call randgen(x, 'Extreme', 0,1, 0.5); C[i, ] = x;  i = i+1;
call randgen(x, 'F', 12, 322);        C[i, ] = x;  i = i+1;
call randgen(x, 'Gamma', 7.25);       C[i, ] = x;  i = i+1;
call randgen(x, 'Gompertz', 1, 0.5);  C[i, ] = x;  i = i+1;
call randgen(x, 'Gumbel', 0, 2);      C[i, ] = x;  i = i+1;
call randgen(x, 'Laplace');           C[i, ] = x;  i = i+1;
call randgen(x, 'Logistic');          C[i, ] = x;  i = i+1;
call randgen(x, 'LogN');              C[i, ] = x;  i = i+1;
call randgen(x, 'Normal');            C[i, ] = x;  i = i+1;
p = {0.2 0.5 0.3};  mu = {0 5 10}; sig = {1 1 2};
call randgen(x, 'NormalMix',p,mu,sig); C[i,] = x;  i = i+1;
call randgen(x, 'Pareto', 3, 1);      C[i, ] = x;  i = i+1;
call randgen(x, 'ShGomp', 0.5, 1.2);  C[i, ] = x;  i = i+1;
call randgen(x, 'T', 4);              C[i, ] = x;  i = i+1;
```

```
call randgen(x, 'Triangle', 0.7);      C[i, ] = x;  i = i+1;
call randgen(x, 'Tweedie', 1.7);       C[i, ] = x;  i = i+1;
call randgen(x, 'Uniform');            C[i, ] = x;  i = i+1;
call randgen(x, 'Wald', 1, 2);         C[i, ] = x;  i = i+1;
call randgen(x, 'Weib', 0.25, 2.1);    C[i, ] = x;  i = i+1;
print C[rowname=ContinDist label="Continuous"];
```

Figure 25.341 Random Numbers from Various Distributions

Discrete				
Bern	1	0	1	0
Binom	6	8	7	8
ConMaxPoi	8	14	9	13
GenPoi	0	12	0	1
Geom	7	7	25	3
Hyper	2	2	1	2
Integer	7	2	6	10
NegB	2	0	3	3
Poisson	6	4	9	7
Table	2	3	3	2

Continuous				
Beta	0.9468434	0.4544648	0.6459307	0.011023
Cauchy	-0.825261	1.5411411	-4.139702	-0.619614
ChiSq	17.361673	26.761346	23.274385	10.866499
Erlang	6.6854394	2.4368303	7.8471153	8.4478236
Expo	3.49168	1.852013	0.6628163	0.9330359
ExtremeVal	-0.205707	-0.380827	5.0987735	2.2144298
F	1.3919449	0.562435	1.4313833	0.8932595
Gamma	9.0062446	9.9090029	11.629442	5.6664198
Gompertz	0.4878535	0.4083678	0.2269387	0.0566942
Gumbel	2.8602282	3.5817945	-0.046359	1.2502257
Laplace	-0.642796	-0.405355	0.0387922	-1.110559
Logistic	-0.330181	0.6248592	3.0345357	-0.822849
LogN	0.3848309	0.3067906	2.2291277	1.5444741
Normal	-0.596858	-0.123449	-0.319319	0.1869868
NormalMix	11.843406	-1.656049	12.668203	10.082624
Pareto	1.522683	1.103138	1.0237359	1.10953
ShGompertz	0.6579323	4.2929889	0.0768349	0.0196648
T	-1.186566	-2.7108	-4.127142	0.3334233
Triangle	0.8571725	0.7602454	0.5846363	0.6371195
Tweedie	0.0164902	2.8447271	0.3945128	0.48139
Uniform	0.9185988	0.7517717	0.0511648	0.5227403
Wald	1.570377	3.3594217	0.274323	1.3845168
Weib	16.812803	0.0001131	0.536375	0.5118368

RANDMULTINOMIAL Function

RANDMULTINOMIAL(*N, NumTrials, Prob***);**

The RANDMULTINOMIAL function is part of the IMLMLIB library. The RANDMULTINOMIAL function generates a random sample from a multinomial distribution, which is a multivariate generalization of the binomial distribution.

The input parameters are as follows:

N	is the number of observations to sample.
NumTrials	is the number of trials. $NumTrials[j] \geq 0$, for $j = 1 \ldots p$.
Prob	is a $1 \times p$ vector of probabilities with $0 < Prob[j] \leq 1$ and $\Sigma_{j=1}^{p}\text{Prob}[j] = 1$.

For each trial, Prob$[j]$ is the probability of event E_j, where the E_j are mutually exclusive and $\Sigma_{j=1}^{p}\text{Prob}[j] = 1$.

The RANDMULTINOMIAL function returns an $N \times p$ matrix that contains N observations of NumTrials random draws from the multinomial distribution. Each row of the resulting matrix is an integer vector $\{X_1\ X_2\ \ldots\ X_p\}$ with $\Sigma X_j = \text{NumTrials}$. That is, for each row, X_j indicates how many times event E_j occurred in NumTrials trials.

If $X = \{X_1\ X_2\ \ldots\ X_p\}$ follows a multinomial distribution with n trials and probabilities $\rho = \{\rho_1\ \rho_2\ \ldots\ \rho_p\}$, then

- the probability density function for x is

$$f(x; n, \rho) = \frac{n!}{\prod_{i=1}^{p} x_i!} \prod_{i=1}^{p} {\rho_i}^{x_i}$$

- the expected value of X_i is $n\rho_i$.
- the variance of X_i is $n\rho_i(1 - \rho_i)$.
- the covariance of X_i with X_j is $-n\rho_i\rho_j$.
- if $p = 1$ then X is constant.
- if $p = 2$ then X_1 is Binomial(n, ρ_1) and X_2 is Binomial(n, ρ_2).

The following example generates 1,000 samples from a multinomial distribution with three mutually exclusive events. For each sample, 10 events are generated. Each row of the returned matrix x represents the number of times each event is observed. The example also computes the sample mean and covariance and compares them with the expected values.

```
call randseed(1);
prob = {0.3,0.6,0.1};
NumTrials = 10;
N = 1000;
x = RandMultinomial(N,NumTrials,prob);
```

```
/* population mean and covariance */
Mean = NumTrials * prob`;
Cov = -NumTrials*prob*prob`;
/* replace diagonal elements of Cov with Variance */
Variance = NumTrials*prob#(1-prob);
do i = 1 to nrow(prob);
   Cov[i,i] = Variance[i];
end;

SampleMean = mean(x);
SampleCov = cov(x);
print SampleMean Mean, SampleCov Cov;
```

Figure 25.342 Estimated Mean and Covariance Matrix

SampleMean			Mean		
2.891	6.059	1.05	3	6	1

SampleCov			Cov		
2.0051241	-1.69126	-0.313864	2.1	-1.8	-0.3
-1.69126	2.3198388	-0.628579	-1.8	2.4	-0.6
-0.313864	-0.628579	0.9424424	-0.3	-0.6	0.9

For further details about sampling from the multinomial distribution, see Gentle (2003), or Fishman (1996).

RANDMVT Function

RANDMVT(*N, DF, Mean, Cov* **);**

The RANDMVT function is part of the IMLMLIB library. The RANDMVT function returns an $N \times p$ matrix that contains N random draws from the Student's t distribution with *DF* degrees of freedom, mean vector *Mean*, and covariance matrix *Cov*.

The inputs are as follows:

N	is the number of desired observations sampled from the multivariate Student's t distribution.
DF	is a scalar value that represents the degrees of freedom for the t distribution.
Mean	is a $1 \times p$ vector of means.
Cov	is a $p \times p$ symmetric positive definite variance-covariance matrix.

If X follows a multivariate t distribution with ν degrees of freedom, mean vector μ, and variance-covariance matrix Σ, then

- the probability density function for x is

$$f(x;\nu,\mu,\Sigma) = \frac{\Gamma((\nu+p)/2)}{|\Sigma|^{1/2}(\pi\nu)^{p/2}\Gamma(\nu/2)}\left(1+\frac{(x-\mu)\Sigma^{-1}(x-\mu)^T}{\nu}\right)^{-(\nu+p)/2}$$

- if $p = 1$, the probability density function reduces to a univariate Student's t distribution.
- the expected value of X_i is μ_i.
- the covariance of X_i and X_j is $\frac{\nu}{\nu-2}\Sigma_{ij}$ when $\nu > 2$.

The following example generates 1,000 samples from a two-dimensional t distribution with 7 degrees of freedom, mean vector (1, 2), and covariance matrix S. Each row of the returned matrix x is a row vector sampled from the t distribution. The example computes the sample mean and covariance and compares them with the expected values.

```
call randseed(1);
N = 1000;
DF = 4;
Mean = {1 2};
S = {1 1, 1 5};
Cov = DF/(DF-2) * S;  /* population covariance */
x = RandMVT( N, DF, Mean, S );
SampleMean = mean(x);
SampleCov = cov(x);
print SampleMean Mean, SampleCov Cov;
```

Figure 25.343 Estimated Mean and Covariance Matrix

SampleMean		Mean	
1.0109905	1.9372765	1	2

SampleCov		Cov	
1.9556572	2.2581732	2	2
2.2581732	10.437216	2	10

In the preceding example, the columns (marginals) of x do *not* follow univariate t distributions. If you want a sample whose marginals are univariate t, then you need to scale each column of the output matrix:

```
x = RandMVT( N, DF, Mean, S );
StdX = x / sqrt(T(vecdiag(S))); /* StdX columns are univariate t */
```

Equivalently, you can generate samples whose marginals are univariate t by passing in a correlation matrix instead of a general covariance matrix.

For further details about sampling from the multivariate t distribution, see Kotz and Nadarajah (2004).

RANDNORMAL Function

RANDNORMAL(*N, Mean, Cov* **);**

The RANDNORMAL function is part of the IMLMLIB library. The RANDNORMAL function returns an $N \times p$ matrix that contains N random draws from the multivariate normal distribution with mean vector *Mean* and covariance matrix *Cov*.

The inputs are as follows:

N	is the number of desired observations sampled from the multivariate normal distribution.
Mean	is a $1 \times p$ vector of means.
Cov	is a $p \times p$ symmetric positive definite variance-covariance matrix.

If X follows a multivariate normal distribution with mean vector μ and variance-covariance matrix Σ, then

- the probability density function for x is

$$f(x;\mu,\Sigma) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{(x-\mu)\Sigma^{-1}(x-\mu)^T}{2}\right)$$

- if $p = 1$, the probability density function reduces to a univariate normal distribution.
- the expected value of X_i is μ_i.
- the covariance of X_i and X_j is Σ_{ij}.

The following example generates 1,000 samples from a two-dimensional multivariate normal distribution with mean vector (1, 2) and a given covariance matrix. Each row of the returned matrix x is a row vector sampled from the multivariate normal distribution. The example computes the sample mean and covariance and compares them with the expected values.

```
call randseed(1);
N = 1000;
Mean = {1 2};
Cov = {2.4 3, 3 8.1};

x = RandNormal( N, Mean, Cov );
SampleMean = mean(x);
SampleCov = cov(x);
print SampleMean Mean, SampleCov Cov;
```

Figure 25.344 Estimated Mean and Covariance Matrix

SampleMean		Mean	
1.0619604	2.1156084	1	2

SampleCov		Cov	
2.5513518	3.2729559	2.4	3
3.2729559	8.7099585	3	8.1

For further details about sampling from the multivariate normal distribution, see Gentle (2003).

RANDWISHART Function

RANDWISHART(*N, DF, Sigma* **);**

The RANDWISHART function is part of the IMLMLIB library. The RANDWISHART function returns an

$N \times (p \times p)$ matrix that contains N random draws from the Wishart distribution with *DF* degrees of freedom. Each row of the returned matrix represents a $p \times p$ matrix.

The inputs are as follows:

N	is the number of desired observations sampled from the distribution.
DF	is a scalar value that represents the degrees of freedom, DF $\geq p$.
Sigma	is a $p \times p$ symmetric positive definite matrix.

The Wishart distribution is a multivariate generalization of the gamma distribution. (Note, however, that Kotz, Balakrishnan, and Johnson (2000) suggest that the term "multivariate gamma distribution" should be restricted to those distributions for which the marginal distributions are univariate gamma. This is not the case with the Wishart distribution.) A Wishart distribution is a probability distribution for nonnegative definite matrix-valued random variables. These distributions are often used to estimate covariance matrices.

If a $p \times p$ nonnegative definite matrix X follows a Wishart distribution with parameters ν degrees of freedom and a $p \times p$ symmetric positive definite matrix Σ, then

- the probability density function for x is

$$f(x; \nu, \Sigma) = \frac{|x|^{(\nu-p-1)/2} \exp(-\frac{1}{2}\operatorname{trace}(x\,\Sigma^{-1}))}{2^{p\nu/2}|\Sigma|^{\nu/2}\pi^{p(p-1)/4}\prod_{i=1}^{p}\Gamma(\frac{\nu-i+1}{2})}$$

- if $p = 1$ and $\Sigma = 1$, then the Wishart distribution reduces to a chi-square distribution with ν degrees of freedom.
- the expected value of X is $\nu\Sigma$.

The following example generates 1,000 samples from a Wishart distribution with 7 degrees of freedom and 2×2 matrix parameter S. Each row of the returned matrix x represents a 2×2 nonnegative definite matrix. (You can reshape the ith row of x with the SHAPE function.) The example computes the sample mean and compares it with the expected value.

```
call randseed(1);
N = 1000;
DF = 7;
S = {1 1, 1 5};
x = RandWishart( N, DF, S );
Mean = DF * S;
SampleMean = shape( mean(x), 2, 2);
print SampleMean Mean;
```

Figure 25.345 Estimated Mean of Matrices

SampleMean		Mean	
7.0518633	7.2402925	7	7
7.2402925	36.056848	7	35

For further details about sampling from the Wishart distribution, see Johnson (1987).

RANPERK Function

RANPERK(*n, k* < *, numperm* > **);**

RANPERK(*set, k* < *, numperm* > **);**

The RANPERK function generates a random permutation of *k* elements from a set of *n* elements. The random number seed is set by the RANDSEED subroutine. The RANPERK function is similar to the RANCOMB function. A combination is a sorted permutation of the *k* elements.

The first argument, *set*, can be a scalar or a vector. If *set* is a scalar, the function returns *k* indices in the range 1–*n*. If *set* is a vector, the number of elements of the vector determines *n*, and the RANPERK function returns *k* elements of *set*, which can be numeric or character.

By default, the RANPERK function returns a single random permutation with one row and *k* columns. If the *numperm* argument is specified, the function returns a matrix with *numperm* rows and *k* columns. Each row of the returned matrix represents a single random draw.

The following statements generate four random permutations that consist of two elements from the set 1, 2, 3:

```
call randseed(1234);
n = 3;
p = ranperk(n, 2, 4);
print p;
```

Figure 25.346 Two Elements of a Random Permutation

p	
3	1
1	2
3	2
1	3

Alternatively, the following statements compute random permutations that consist of two elements from an unsorted character vector:

```
q = ranperk({C B A}, 2, 4);
print q;
```

Figure 25.347 Random Permutation of a Character Vector

q	
C	A
C	A
B	C
A	C

RANPERM Function

RANPERM(*n*);

RANPERM(*set*, < , *numperm* >);

The RANPERM function generates random permutations of a set with *n* elements. The random number seed is set by the RANDSEED subroutine.

The first argument, *set*, can be a scalar or a vector. If *set* is a scalar, the function returns indices in the range 1–*n*. If *set* is a vector, the number of elements of the vector determines *n* and the RANPERM function returns elements of *set*, which can be numeric or character.

By default, the RANPERM function returns a single random combination with one row and *n* columns. If the *numperm* argument is specified, the function returns a matrix with *numperm* rows and *n* columns. Each row of the returned matrix represents a single permutation.

The following statements generate five random permutations of the set $\{1, 2, 3\}$:

```
call randseed(1234);
n = 3;
p = ranperm(n, 5);
print p;
```

Figure 25.348 Random Permutations of Three Items

p		
1	2	3
3	2	1
3	2	1
3	1	2
3	1	2

Alternatively, the following statements compute five random permutations of an unsorted character vector:

```
a = ranperm({C B A}, 5);
print a;
```

Figure 25.349 Random Permutations of a Character Vector

a		
B	C	A
B	A	C
C	B	A
B	C	A
B	C	A

RANDSEED Call

CALL RANDSEED(*seed* < , *reinit* > **);**

CALL RANDSEED(*RNG* < , *seed* > < , *reinit* > **);**

The RANDSEED subroutine sets the initial state for the RANDGEN subroutine and the RANDFUN function. It also sets the seed for routines that generate random combinations (RANCOMB), permutations (RANPERK and RANPERM), and samples (SAMPLE).

RNG is a string that specifies a valid random-number generator (RNG).

seed is a number to be used to initialize the RANDGEN random number generator.

reinit specifies whether the random number stream should be reinitialized after the first initialization, within the same PROC IML session.

Table 25.7 shows valid values for the *RNG* argument. These random-number generators are described in the documentation for the STREAMINIT call in *SAS Functions and CALL Routines: Reference*.

Table 25.7 Random-Number Generators That Are Supported in SAS

Value	Description
MTHYBRID	(Default) Hybrid 1998/2002 32-bit Mersenne twister
MT1998	(Deprecated) 1998 32-bit Mersenne twister
MT32 | MT2002	2002 32-bit Mersenne twister
MT64	64-bit Mersenne twister
PCG | PCG64I	64-bit permuted congruential generator
TF2 | THREEFRY2X64	Threefry 2x64-bit counter-based RNG
TF4 | THREEFRY4X64	Threefry 4x64-bit counter-based RNG
HARDWARE	Any supported hardware-based random number generator
RDRAND	Intel hardware-based RdRand instructions

The RANDSEED subroutine enables you to specify an initial seed for subsequent RANDGEN calls. A seed is automatically generated if the RANDSEED subroutine is not called, if you specify 0 as a seed, or if you use a calling syntax that does not specify a seed. The initial seed is obtained from the Intel RdRand instruction on CPUs that support that instruction, or from the current datetime value otherwise.

Hardware-based RNGs do not use seeds. If you specify a seed for a hardware-based RNG, the seed is silently ignored. If you request a hardware-based RNG but it is not available on your CPU, then the default RNG is used.

The RANDSEED routine is used to generate a reproducible stream of pseudorandom numbers. The optional *reinit* parameter controls whether the random number stream is reinitialized after it has begun generating random numbers. If the second argument is 1, the stream is reinitialized; otherwise, subsequent calls to the RANDSEED subroutine are ignored. To ensure that you are working with an independent random number stream within your PROC IML session, set *reinit* to 0 or do not specify that argument.

In the third maintenance release of SAS 9.4 software, the MTHYBRID generator became the default RNG, instead of the earlier default, the MT1998 generator. The MTHYBRID generator is a hybrid between

the MT1998 and the MT2002 generators. When the seed is exactly divisible by 8192, the MTHYBRID algorithm uses the 2002 initialization algorithm (Matsumoto and Nishimura 2002). Otherwise, the algorithm is initialized by using the 1998 initialization algorithm (Matsumoto and Nishimura 1998). A SAS Note appears when the 2002 initialization is used.

```
proc iml;
call randseed(12345);            /* use default RNG; set seed       */
x1 = randfun(5, "uniform");      /* get 5 random uniform variates   */
call randseed(12345, 1);         /* reset current stream, same seed */
x2 = randfun(5, "uniform");      /* x2 equals x1                    */
call randseed(4321, 1);          /* reset current stream, different seed */
x3 = randfun(5, "uniform");      /* x3 is different from x1         */

/* switch to new RNG */
call randseed("PCG", 9876, 1);  /* use PCG RNG                    */
y1 = randfun(5, "uniform");     /* get 5 random uniform variates */
call randseed("PCG", 54321, 1); /* reset PCG with new seed       */
y2 = randfun(5, "uniform");     /* y2 is different from y1       */
call randseed("TF2", 54321, 1); /* use TF2 RNG; same seed        */
y3 = randfun(5, "uniform");     /* y3 is different from y2       */
r = x1 || x2 || x3 || y1 || y2 || y3;
print r[F=5.3 C={x1 x2 x3 y1 y2 y3} L="Random Uniform Variates"];
quit;
```

Figure 25.350 Random Uniform Variates from Different RNGs and Seeds

Random Uniform Variates					
X1	X2	X3	Y1	Y2	Y3
0.583	0.583	0.791	0.407	0.167	0.088
0.994	0.994	0.375	0.599	0.864	0.391
0.588	0.588	0.930	0.916	0.135	0.714
0.857	0.857	0.209	0.019	0.886	0.726
0.825	0.825	0.216	0.337	0.374	0.479

RANGE Function

RANGE(*matrix1* < , *matrix2, . . . , matrix15*> **);**

The RANGE function returns the range of values of a numerical matrix or set of matrices.

Missing values are excluded in the computation. When the arguments contain at least one nonmissing value, the range is defined as the maximum value minus the minimum value. If all arguments are missing, the RANGE function returns a missing value.

The following example uses the RANGE function:

```
c = {1 -123 13 56 128 -81 12};
r = range(c);
print r;
```

Figure 25.351 Range of Values

r
251

RANK Function

RANK(*matrix***);**

The RANK function creates a new matrix that contains elements that are the ranks of the corresponding elements of the numerical argument, *matrix*. The rank of a missing value is a missing value. The ranks of tied values are assigned arbitrarily. (See the description of the RANKTIE function for alternate approaches.)

For example, the following statements produce the ranks of a vector:

```
x = {2 2 1 0 5};
r = rank(x);
print r;
```

Figure 25.352 Ranks of a Vector

r
3 4 2 1 5

Provided that a vector, **x**, does not contain missing values, the RANK function can be used to sort the vector, as shown in the following statements:

```
b = x;
x[,rank(x)] = b;
print x;
```

Figure 25.353 Sorted Vector

x
0 1 2 2 5

You can also sort a matrix by using the SORT subroutine. The SORT subroutine handles missing values in the data.

The RANK function can also be used to find anti-ranks of **x**, as follows:

```
x = {2 2 1 0 5};
r = rank(x);
a = r;
a[,r] = 1:ncol(x);
print a;
```

Figure 25.354 Anti-Ranks of a Vector

a
4 3 1 2 5

Although the RANK function ranks only the elements of numerical matrices, you can rank the elements of a character matrix by using the UNIQUE function, as demonstrated by the following statements:

```
/* Create RANK-like functionality for character matrices */
start rankc(x);
   s = unique(x);              /* the unique function returns a sorted list */
   idx = j(nrow(x), ncol(x));
   ctr = 1;                    /* there can be duplicate values in x */
   do i = 1 to ncol(s);        /* for each unique value */
      t = loc(x = s[i]);
      nDups = ncol(t);
      idx[t] = ctr : ctr+nDups-1;
      ctr = ctr + nDups;
   end;
   return (idx);
finish;

/* call the RANKC module */
x = {every good boy does fine and good and well every day};
rc = rankc(x);
print rc[colnam=x];

/* Notice that ranking is in ASCII order, in which capital
   letters precede lower case letters.  To get case-insensitive
   behavior, transform the matrix before comparison */
x = {"a" "b" "X" "Y" };
asciiOrder = rankc(x);
alphaOrder = rankc(upcase(x));
print x, asciiOrder, alphaOrder;
```

Figure 25.355 Ranks of Character Matrices

	rc										
	EVERY	GOOD	BOY	DOES	FINE	AND	GOOD	AND	WELL	EVERY	DAY
ROW1	6	9	3	5	8	1	10	2	11	7	4

x			
a	b	X	Y

asciiOrder			
3	4	1	2

alphaOrder			
1	2	3	4

There is no SAS/IML function that directly computes the linear algebraic rank of a matrix. In linear algebra, the rank of a matrix is the maximal number of linearly independent columns (or rows). You can use the following technique to compute the numerical rank of matrix **a**:

```
/* Only four linearly independent columns */
A = {1 0 1 0 0,
     1 0 0 1 0,
     1 0 0 0 1,
```

```
     0 1 1 0 0,
     0 1 0 1 0,
     0 1 0 0 1 };
rank = round(trace(ginv(a)*a));
print rank;
```

Figure 25.356 Numerical Rank of a Matrix

rank
4

Another common technique used to examine the rank of a matrix is to look at the number of nonzero singular values in the singular value decomposition of a matrix (see the SVD call). However, keep in mind that numerical computations might result in singular values for a rank-deficient matrix that are small but nonzero.

RANKTIE Function

RANKTIE(*matrix* < , *method*> **);**

The RANKTIE function creates a new matrix that contains elements that are the ranks of the corresponding elements of *matrix*. The rank of a missing value is a missing value. The ranks of tied values are computed by using one of several methods.

The arguments to the function are as follows:

matrix specifies the data.

method specifies the method used to compute the ranking of tied values. These methods correspond to those defined by using the TIES= option in the RANK procedure. For details, see the "Concepts" section of the documentation for the RANK procedure in the *Base SAS Procedures Guide*.

The following values are valid:

"Mean" specifies that tied elements are assigned rankings equal to the mean of the tied elements. This is the default method. This method is known as a fractional competition ranking.

"Low" specifies that tied elements are assigned rankings equal to the minimum order rank of the tied elements. This method is known as a standard competition ranking.

"High" specifies that tied elements are assigned rankings equal to the maximum rank of the tied elements. This method is known as a modified competition ranking.

"Dense" specifies that ranks are consecutive integers that begin with 1 and end with the number of unique, nonmissing values. Tied values are assigned the same rank. This method is known as a dense ranking.

The RANKTIE function differs from the RANK function in that the RANK function breaks ties arbitrarily.

For example, the following statements produce ranks of a vector by using several different methods of breaking ties:

```
x = {4 4 0 6};
rMean = ranktie(x);   /* default is "Mean" */
rLow = ranktie(x, "Low");
rHigh = ranktie(x, "High");
rDense = ranktie(x, "Dense");
print rMean, rLow, rHigh, rDense;
```

Figure 25.357 Numerical Ranks of a Vector

rMean
2.5 2.5 1 4

rLow
2 2 1 4

rHigh
3 3 1 4

rDense
2 2 1 3

Although the RANKTIE function ranks only the elements of numerical matrices, you can rank the elements of a character matrix by using the UNIQUE function, as demonstrated by the following statements:

```
/* Create RANKTIE-like functionality for character matrices */
start ranktiec(x);
   s = unique(x);
   idx = j(nrow(x), ncol(x));
   ctr = 1;                      /* there can be duplicate values in x */
   do i = 1 to ncol(s);          /* for each unique value */
      t = loc(x = s[i]);
      nDups = ncol(t);
      idx[t] = ctr+(nDups-1)/2; /* =(ctr:ctr+nDups-1)[:] */
      ctr = ctr + nDups;
   end;
   return (idx);
finish;

/* call the RANKTIEC module */
x = {every good boy does fine and good and well every day};
rtc = ranktiec(x);
print rtc[colname=x];
```

Figure 25.358 Numerical Ranks of a Character Vector

	rtc										
	EVERY	GOOD	BOY	DOES	FINE	AND	GOOD	AND	WELL	EVERY	DAY
ROW1	6.5	9.5	3	5	8	1.5	9.5	1.5	11	6.5	4

RATES Function

RATES(*rates, oldfreq, newfreq***);**

The RATES function computes a column vector of (per-period, such as per-year) interest rates converted from one base to another. The arguments to the RATES function are as follows:

rates	is an $n \times 1$ column vector of rates that correspond to the old base. Elements should be positive.
oldfreq	is a scalar that represents the old base. If positive, it represents discrete compounding as the reciprocal of the number of compoundings per period. If zero, it represents continuous compounding. If –1, the rates represent discount factors. No other negative values are accepted.
newfreq	is a scalar that represents the new base. If positive, it represents discrete compounding as the reciprocal of the number of compoundings per period. If zero, it represents continuous compounding. If –1, the rates represent discount factors. No other negative values are accepted.

Let $D(t)$ be the discount function, which is the present value of a unit amount to be received t periods from now. The discount function can be expressed in the following ways:

- with per-unit-time-period discount factors d_t:

  $$D(t) = d_t^t$$

- with continuous compounding:

  $$D(t) = e^{-r_t t}$$

- with discrete compounding:

  $$D(t) = (1 + fr)^{-t/f}$$

  where $0 < f < 1$ is the frequency, the reciprocal of the number of compoundings per unit time period. The RATES function converts between these three representations.

For example, the following example produces the output shown in Figure 25.359:

```
rates = T(do(0.1, 0.3, 0.1));
oldfreq = 0;
newfreq = 0;
rates = rates(rates, oldfreq, newfreq);
print rates;
```

Figure 25.359 Interest Rates

rates
0.1
0.2
0.3

RATIO Function

RATIO(*ar, ma, terms* **<** *, dim* **>);**

The RATIO function divides matrix polynomials.

The arguments to the RATIO function are as follows:

ar is an $n \times (ns)$ matrix that represents a matrix polynomial generating function, $\Phi(B)$, in the variable B. The first $n \times n$ submatrix represents the constant term and must be nonsingular, the second $n \times n$ submatrix represents the first-order coefficients, and so on.

ma is an $n \times (mt)$ matrix that represents a matrix polynomial generating function, $\Theta(B)$, in the variable B. The first $n \times m$ submatrix represents the constant term, the second $n \times m$ submatrix represents the first-order term, and so on.

terms is a scalar that contains the number of terms to be computed, denoted by *r* in the following discussion. This value must be positive.

dim is a scalar that contains the value of *m*, a dimension of the matrix *ma*. The default value is 1.

The RATIO function multiplies a matrix of polynomials by the inverse of another matrix of polynomials. It is useful for expressing univariate and multivariate ARMA models in pure moving average or pure autoregressive forms.

The value returned is an $n \times (mr)$ matrix that contains the terms of $\Phi(B)^{-1}\Theta(B)$ considered as a matrix of rational functions in B that have been expanded as power series.

The RATIO function can be used to consolidate the matrix operators that are used in a multivariate time series model of the form

$$\Phi(B)\mathbf{Y}_t = \Theta(B)\epsilon_t$$

where $\Phi(B)$ and $\Theta(B)$ are matrix polynomial operators whose first matrix coefficients are identity matrices. The RATIO function can be used to compute a truncated form of $\Psi(B) = \Phi(B)^{-1}\Theta(B)$ for the equivalent infinite-order model

$$\mathbf{Y}_t = \Psi(B)\epsilon_t$$

The RATIO function can also be used for simple scalar polynomial division, giving a truncated form of $\theta(x)/\phi(x)$ for two scalar polynomials $\theta(x)$ and $\phi(x)$.

The cumulative sum of the elements of a column vector **x** can be obtained by using the following statement:

```
ratio({ 1 -1}, x, ncol(x));
```

The following example defines polynomial coefficients that are used in a multivariate ARMA(1,1) model and computes the ratio of the polynomials:

```
ar = {1 0 -0.5  2,
      0 1  3   -0.8};
ma = {1 0  0.9  0.7,
      0 1  2   -0.4};
psi = ratio(ar, ma, 4, 2);
print psi;
```

Figure 25.360 The Ratio of Polynomials

psi							
1	0	1.4	-1.3	2.7	-1.45	11.35	-9.165
0	1	-1	0.4	-5	4.22	-12.1	7.726

RCEPSTRUM Function

RCEPSTRUM(*x*);

The RCEPSTRUM function computes the real cepstrum of a time series. The real cepstrum of the time series x is defined as

$$y = \frac{1}{2\pi} \int_{-\pi}^{\pi} \log |X(e^{i\omega})| e^{i\omega n} d\omega$$

where $|X(e^{i\omega})|$ is the magnitude of the Fourier transform of x and $i = \sqrt{-1}$.

The input argument to the RCEPSTRUM function is as follows:

x specifies the input data vector. It can be either a column vector or a row vector. Any missing value in this vector is replaced with 0.

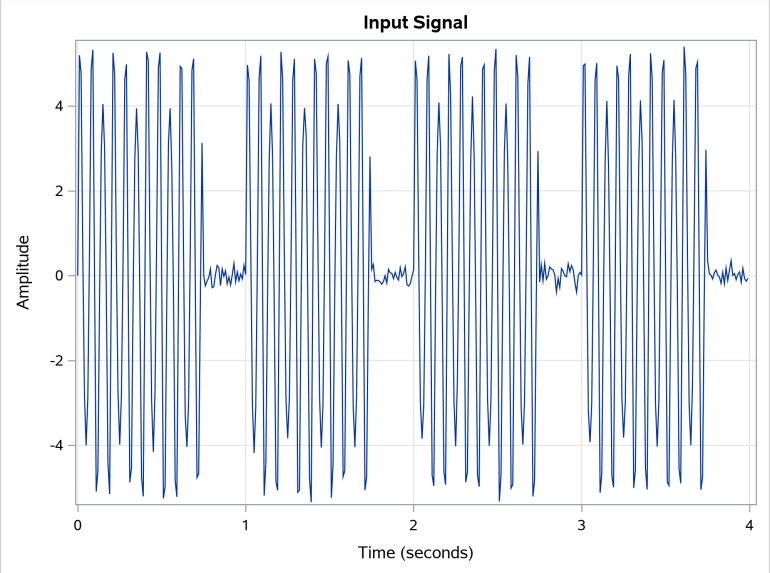
The RCEPSTRUM function returns a column vector that contains the real cepstrum of the input data. The length of the output is the same as the length of the input data.

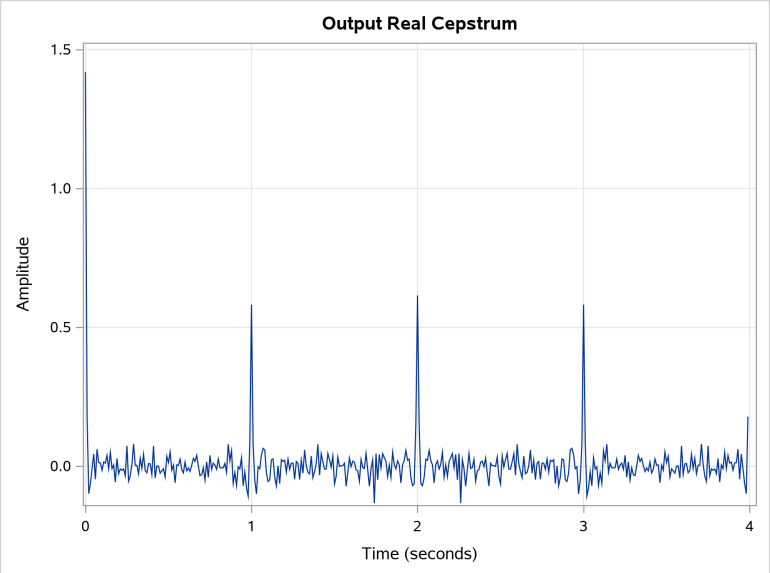
The following example shows how to use the RCEPSTRUM function to estimate the period of a periodic signal. First an input signal is generated that consists of four segments. Each segment is 1 second long; the first 0.75 seconds contain the sum of two sinusoidal signals, and the next 0.25 seconds contain all zeros. This segment is repeated four times, and Gaussian random noise is added to form a pseudoperiodic signal. The output complex spectrum in Figure 25.361 shows a spike around $t = 1$ second, which is a good estimation of the period.

```
sf  = 100;    /* sampling frequency */
T = 1.0/sf;   /* sampling period */
N = 400;
t = do(0, (N-1)*T, T);
t1 = t[, 1:N*3/16];

f1 = 15;
f2 = 25;
pi = constant("pi");
x1 = 5*sin(2*pi*f1*t1);
x2 = sin(2*pi*f2*t1);
x3 = x1 + x2;        /* sum of two sinusoidal signals */
x4 = j(1, N/16, 0);
x5 = repeat(x3 || x4, 1, 4);

call randseed(1);
noise_mean = 0;
noise_std  = 0.15;
```

```
xn = randfun(N, "Normal", noise_mean, noise_std);
x = x5` + xn;  /* input signal */

y = rcepstrum(x);

title "Input Signal";
call series(t,x) grid= {X Y}
                 label={"Time (seconds)" "Amplitude"};

title "Output Real Cepstrum";
call series(t,y) grid= {X Y}
                 label={"Time (seconds)" "Amplitude"};
```

Figure 25.361 Real Cepstrum Example





RDODT and RUPDT Calls

CALL RDODT(*def, rup, bup, sup, r, z* **<** *, b* **>** **<** *, y* **>** **<** *, ssq* **>** **);**

CALL RUPDT(*rup, bup, sup, r, z* **<** *, b* **>** **<** *, y* **>** **<** *, ssq* **>** **);**

If $\mathbf{A} = \mathbf{QR}$ is the QR decomposition of the matrix $\mathbf{A}$, the RUPDT subroutine enables you to efficiently recompute the $\mathbf{R}$ matrix when a new row is added to A. This is called an update. Similarly, the RDODT subroutine enables you to efficiently recompute the $\mathbf{R}$ matrix when an existing row is deleted from A. This is called a downdate. You can also use the RDODT and RUPDT subroutines to downdate and update Cholesky decompositions.

The RDODT and RUPDT subroutines return the values:

def is only used for downdating, and it specifies whether the downdating of matrix $\mathbf{R}$ by using the q rows in argument *z* has been successful. The result *def*=2 means that the downdating of $\mathbf{R}$ by at least one row of $\mathbf{Z}$ leads to a singular matrix and cannot be completed successfully (since the result of downdating is not unique). In that case, the results *rup*, *bup*, and *sup* contain missing values only. The result *def*=1 means that the residual sum of squares, *ssq*, could not be downdated successfully and the result *sup* contains missing values only. The result *def*=0 means that the downdating of $\mathbf{R}$ by $\mathbf{Z}$ was completed successfully.

rup is the $n \times n$ upper triangular matrix $\mathbf{R}$ that has been updated or downdated by using the q rows in $\mathbf{Z}$.

bup is the $n \times p$ matrix $\mathbf{B}$ of right-hand sides that has been updated or downdated by using the q rows in argument *y*. If the argument *b* is not specified, *bup* is not computed.

sup is a p vector of square roots of residual sum of squares that is updated or downdated by using the q rows of argument *y*. If *ssq* is not specified, *sup* is not computed.

The input arguments to the RDODT and RUPDT subroutines are as follows:

r specifies an $n \times n$ upper triangular matrix $\mathbf{R}$ 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 *z* is added to matrix $\mathbf{A}$.

- *downdate*: An n vector 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 z requires only $2n^2$ floating-point operations.

If the QR decomposition is used to solve the full-rank linear least squares problem

$$\min_x \|\mathbf{A}x - b\|^2 = \text{ssq}$$

by solving the nonsingular upper triangular system

$$x = \mathbf{R}^{-1}\mathbf{Q}'b$$

then the RUPDT and RDODT subroutines can be used to update or downdate the p-transformed right-hand sides $\mathbf{Q}'\mathbf{B}$ and the residual sum-of-squares p vector *ssq* provided that for each n vector z added to or deleted from $\mathbf{A}$ there is also a p vector 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 z_k (and y_k), $k = 1, \ldots, q$, in the order in which they are stored.

The QR decomposition of an $m \times n$ matrix $\mathbf{A}$, $m \geq n$, $\text{rank}(\mathbf{A}) = n$,

$$\mathbf{A} = \mathbf{QR}, \text{ where } \mathbf{Q}'\mathbf{Q} = \mathbf{Q}\mathbf{Q}' = \mathbf{I}_m$$

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 $\text{rank}(\mathbf{A}) = n$, the upper triangular matrix $\mathbf{R}$ computed by the QR decomposition (with positive diagonal elements) is the same as the one computed by Cholesky factorization except for numerical error,

$$\mathbf{A}'\mathbf{A} = (\mathbf{QR})'(\mathbf{QR}) = \mathbf{R}'\mathbf{R}$$

Adding a row vector z to matrix $\mathbf{A}$ corresponds to the rank-1 modification of the crossproduct matrix $\mathbf{C}$

$$\widetilde{\mathbf{C}} = \mathbf{C} + z'z, \text{ where } \widetilde{\mathbf{C}} = \widetilde{\mathbf{A}}'\widetilde{\mathbf{A}}$$

and the $(m+1) \times n$ matrix $\widetilde{\mathbf{A}}$ contains all rows of $\mathbf{A}$ with the row z added.

Deleting a row vector z from matrix $\mathbf{A}$ corresponds to the rank-1 modification

$$\mathbf{C}^* = \mathbf{C} - z'z, \text{ where } \mathbf{C}^* = \mathbf{A}^{*\prime}\mathbf{A}^*$$

and the $(m-1) \times n$ matrix $\mathbf{A}^*$ contains all rows of $\mathbf{A}$ with the row z deleted. Thus, you can also use the subroutines RUPDT and RDODT to update or downdate the Cholesky factor $\mathbf{R}$ of a positive definite crossproduct matrix $\mathbf{C}$ of $\mathbf{A}$.

The process of downdating an upper triangular matrix $\mathbf{R}$ (and eventually a residual sum-of-squares vector *ssq*) is not always successful. First of all, the downdated matrix $\mathbf{R}$ could be rank-deficient. Even if the downdated matrix $\mathbf{R}$ is of full rank, the process of downdating can be ill-conditioned and does not work well

if the downdated matrix is close (by rounding errors) to a rank-deficient one. In these cases, the downdated matrix **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

$$ssq_{\text{new}} = \sqrt{ssq_{\text{old}} - ssq_{\text{dod}}}$$

cannot always be computed because, due to rounding errors, the radicand can be negative. In this case, the result vector *sup* returns missing values, and *def* returns 1.

You can use various methods to compute the p columns x_k of the $n \times p$ matrix **X** that minimize the p linear least squares problems with an $m \times n$ coefficient matrix **A**, $m \geq n$, $\text{rank}(\mathbf{A}) = n$, and p right-hand-side vectors b_k (stored columnwise in the $m \times p$ matrix **B**).

The methods in this section use the following simple example:

```
a = { 1 3 ,
      2 2 ,
      3 1 };
b = { 1, 1, 1};
m = nrow(a);
n = ncol(a);
p = ncol(b);
```

- Cholesky decomposition of crossproduct matrix:

```
/* form and solve the normal equations */
aa = a` * a; ab = a` * b;
r  = root(aa);
x  = trisolv(2,r,ab);
x  = trisolv(1,r,x);
print x;
```

- QR decomposition by Householder transformations:

```
call qr(qtb, r, piv, lindep, a, , b);
x = trisolv(1, r[,piv], qtb[1:n,]);
```

- Stepwise update by Givens rotations:

```
r = j(n,n,0); qtb = j(n,p,0); ssq = j(1,p,0);
do i = 1 to m;
   z = a[i,];
   y = b[i,];
   call rupdt(rup,bup,sup,r,z,qtb,y,ssq);
   r   = rup;
   qtb = bup;
   ssq = sup;
```

```
end;
x = trisolv(1,r,qtb);
```

Or, equivalently:

```
r   = j(n,n,0); qtb = j(n,p,0); ssq = j(1,p,0);
call rupdt(rup,bup,sup,r,a,qtb,b,ssq);
x   = trisolv(1,rup,bup);
```

- Singular value decomposition:

```
call svd(u, d, v, a);
d = diag(1 / d);
x = v * d * u` * b;
```

For the preceding 3×2 example matrix **a**, each method obtains the unique LS estimator:

```
ss = ssq(a * x - b);
print ss x;
```

Figure 25.362 Least Squares Solution and Sum of Squared Residuals

ss	x
2.465E-31	0.25
	0.25

To compute the (transposed) matrix $\mathbf{Q}$, you can use the following technique:

```
r = repeat(0,n,n);
y = i(m);
qt = repeat(0,n,m);
call rupdt(rup, qtup, sup, r, a, qt, y);
print qtup;
```

Figure 25.363 Transposed Matrix

qtup		
0.2672612	0.5345225	0.8017837
-0.872872	-0.218218	0.4364358

READ Statement

READ <*range*> <**VAR** *operand*> <**WHERE**(*expression*)> <**INTO** *name* <[**ROWNAME**=*row-name* **COLNAME**=*column-name*]>> **;**

The READ statement reads observations from the current SAS data set. For example, the following statements read data from the Sashelp.Class data set:

```
use Sashelp.Class;
read all var {Sex Height}; /* creates vectors Sex and Height */
read all var _NUM_ into X[colname=varNames]; /* numerical data */
read all var {Weight} where(Sex='M'); /* vector of male weights */
read point 10 var {Name}; /* 10th name in data set */
close Sashelp.Class;
```

See Chapter 7 for further examples.

The arguments to the READ statement are as follows:

range specifies a range of observations. If *range* is not specified, the current observation is read. You can specify a range of observations by using the ALL, CURRENT, NEXT, AFTER, and POINT keywords, as described in the section "Process a Range of Observations" on page 104.

operand selects a set of variables. If the VAR clause is omitted, all variables are read into vectors whose names are identical to the names of the variables in the data set. As described in the section "Select Variables with the VAR Clause" on page 105, you can specify variable names by using a matrix literal, a character matrix, an expression, or the _ALL_, _CHAR_, or _NUM_ keywords.

expression specifies a criterion by which certain observations are selected. If the WHERE clause is omitted, no subsetting occurs. The optional WHERE clause conditionally selects observations that are contained within the *range* specification. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

name is the name of the target matrix.

row-name is a character matrix or quoted literal that contains descriptive row labels.

column-name is a character matrix or quoted literal that contains descriptive column labels.

The *range*, VAR, WHERE, and INTO clauses are all optional and can be specified in any order.

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_.

Reading Variables into Columns of a Matrix

When you use the INTO clause, the specified variables are read into the columns of a matrix.

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 single 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.

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.

REMOVE Function

REMOVE(*matrix, indices***);**

The REMOVE function discards elements from a matrix. The arguments to the REMOVE function are as follows:

matrix	is a numeric or character matrix or literal.
indices	specifies the indices of elements of *matrix* to remove.

The REMOVE function returns (as a row vector) a subset of the elements of the first argument. Elements that correspond to indices in the second argument are removed. The elements of the first argument are enumerated in row-major order, and the indices must be in the range 1 to np, where *matrix* is an $n \times p$ matrix. Nonintegral indices are truncated to their integer part. You can repeat the indices and give them in any order. If all elements are removed, the result is an empty matrix with zero rows and zero columns.

The following statements remove the third element, creating a row vector with three elements:

```
x = {5 6, 7 8};
a = remove(x, 3);       /* remove element 3 */
print a;
```

Figure 25.364 Result of Removing an Element

a
5 6 8

The following statements remove all but the fourth element:

```
r = {3 2 3 1};
b = remove(x, r);       /* equivalent to removing elements 1:3 */
print b;
```

Figure 25.365 Result of Removing Several Elements

b
8

The output shown in Figure 25.365 shows that repeated indices are ignored.

REMOVE Statement

REMOVE <**MODULE**=*(module-list)*> <*matrix-list*> **;**

The REMOVE statement removes modules and matrices from storage.

The arguments to the REMOVE statement are as follows:

module-list specifies a module or modules to remove from storage.

matrix-list specifies a matrix or matrices to remove from storage.

The REMOVE statement removes matrices and modules from the current library storage. For example, the following statement removes the three modules A, B, and C and the matrix X:

```
remove module=(A B C) X;
```

The special operand _ALL_ can be used to remove all matrices or all modules or both. For example, the following statement removes all stored items:

```
remove _all_ module=_all_;
```

For additional and related information, see Chapter 19, "Storage Features," and the descriptions of the LOAD, STORE, RESET, and SHOW statements.

RENAME Call

CALL RENAME(< *libref,* > *member-name, new-name***);**

The RENAME subroutine renames a SAS data set.

The arguments to the RENAME subroutine are as follows:

libref is a character matrix or quoted literal that contains the name of the SAS data library.

member-name is a character matrix or quoted literal that contains the current name of the data set.

new-name is a character matrix or quoted literal that contains the new data set name.

The RENAME subroutine renames a SAS data set in the specified library. All of the arguments can directly be specified in quotes, although quotes are not required. If a one-level data set name is specified, the libref specified by the RESET DEFLIB statement is used. Examples of valid statements follow:

```
call rename("a","b");
call rename(a,b);
call rename(work,a,b);
```

REPEAT Function

REPEAT(*x, nrow, ncol***);**

REPEAT(*x, freq***);**

The REPEAT function creates a matrix of repeated values. There are two ways to specify the syntax. The first syntax repeats the entire matrix *nrow***ncol* times. The arguments for this syntax are as follows:

x	is a numeric matrix or literal.
nrow	specifies the number of times *matrix* is repeated down rows.
ncol	specifies 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 form a new matrix that consists of two vertical and three horizontal copies of **x**:

```
x = {1 2,
     3 4};
y = repeat(x, 2, 3);
print y;
```

Figure 25.366 Repeated Values

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

A second way to call the REPEAT function is to provide an argument, *freq* that has the same number of elements as *x*. The return value is a row vector in which **x[1]** is repeated **freq[1]** times, **x[2]** is repeated **freq[2]** times, and so forth, where the elements of *x* are enumerated in row-major order. Each element of *freq* should be a nonnegative integer. The return value will have **sum(freq)** elements. This is shown in the following example:

```
z = repeat(x, {2 3 0 1});
print z;
```

Figure 25.367 Repeated Values from a Frequency Vector

z					
1	1	2	2	2	4

REPLACE Statement

REPLACE <*range*> <**VAR** *operand*> <**WHERE***(expression)*> **;**

The REPLACE statement replaces values of observations in a SAS data set.

The arguments to the REPLACE statement are as follows:

range	specifies a range of observations. You can specify a range of observations by using the ALL, CURRENT, NEXT, AFTER, and POINT keywords, as described in the section "Process a Range of Observations" on page 104.

operand specifies a set of variables. As described in the section "Select Variables with the VAR Clause" on page 105, you can specify variable names by using a matrix literal, a character matrix, an expression, or the _ALL_, _CHAR_, or _NUM_ keywords.

expression specifies a criterion by which certain observations are selected. If the WHERE clause is omitted, no subsetting occurs. The optional WHERE clause conditionally selects observations that are contained within the *range* specification. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

The REPLACE statement replaces the values of observations in a SAS data set with current values of matrices with the same name. Use the *range*, VAR, and WHERE arguments to limit replacement to specific variables and observations. Replacement matrices should be the same type as the data set variables. The REPLACE statement uses matrix elements in row order, replacing the value in the *i*th observation with the *i*th matrix element. If there are more observations in *range* than matrix elements, the REPLACE statement continues to use the last matrix element.

For example, the following statements increment the weights of all males in a data set:

```
data class;
   set Sashelp.Class;
run;

proc iml;
edit class;          /* open data set for edit */
read all var {weight} where(sex="M");
weight = weight + 5; /* add 5 to male weights */
replace all var {weight} where(sex="M");
close class;
```

RESET Statement

RESET < *options* > **;**

The RESET statement sets processing options. The options are described in the following list. Note that the prefix NO turns off the feature where indicated. For options that take operands, the operand should be a literal, a name of a matrix that contains the value, or an expression in parentheses. The SHOW OPTIONS statement displays the current settings of options.

AUTONAME | NOAUTONAME
specifies whether rows are automatically labeled ROW1, ROW2, and so on, and columns are labeled COL1, COL2, and so on, when a matrix is printed. Row-name and column-name attributes specified in the PRINT statement or associated via the MATTRIB statement override the default labels. The AUTONAME option causes the SPACES option to be reset to 4. The default is NOAUTONAME.

CENTER | NOCENTER
specifies whether output from the PRINT statement is centered on the page. The default is CENTER. This resets the global CENTER/NOCENTER option for the SAS session.

CLIP | NOCLIP
specifies whether SAS/IML graphs are automatically clipped outside the viewport; that is, any data falling outside the current viewport are not displayed. The default is NOCLIP.

DEFLIB=*operand*
specifies the default libref for SAS data sets when no other libref is given. This defaults to USER if a USER libref is set up, or WORK if not. The libref 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.

EIGEN93 | NOEIGEN93
specifies the method used to compute eigenvalues and eigenvectors. On Linux and 64-bit Windows computers that contain vendor-supplied linear algebra subroutines (such as the Intel Math Kernel Library), SAS/IML can call the vendor-supplied subroutines for eigenvalue computations. The vendor-supplied routines are called if the NOEIGEN93 option is set. If the EIGEN93 option is set, or if a vendor-supplied subroutine is not available, then SAS/IML will use the eigenvalue routines from SAS 9.3. The default value is NOEIGEN93.

FLOW | NOFLOW
specifies whether operations are shown as executed. It is used for debugging only. The default is NOFLOW.

FUZZ <=*number***> | NOFUZZ**
specifies whether very small numbers are printed as zero rather than in scientific notation. If the absolute value of the number is less than the value specified in *number*, it is printed as 0. The *number* argument is optional, and the default value varies across hosts but is typically around 1E–12. The default is NOFUZZ.

FW=*number*
sets the field width for printing numeric values. The default field width is 9.

LINESIZE=*n*
specifies the linesize for printing. The default value is usually 78. This resets the global LINESIZE option for the SAS session.

LOG | NOLOG
specifies whether output is routed to the SAS Log rather than to the LISTING destination. In the log, the results are interleaved with the statements and messages. The RESET LOG has no effect if the ODS LISTING destination is not active. The default value is NOLOG, which means that output is sent to the open ODS destinations, but not to the SAS Log.

NAME | NONAME
specifies whether the matrix name or label is printed with the value for the PRINT statement. The default is NAME.

PAGESIZE=*n*
specifies the pagesize for printing. The default value is inherited from the SAS environment. Changing the

PAGESIZE=
option also changes the global PAGESIZE option.

PRINT | NOPRINT
specifies whether the final results from assignment statements are printed automatically. The default is NOPRINT.

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=< *libref.*>*memname*;
specifies the file to be the current library storage for STORE and LOAD statements. The default library storage is WORK.IMLSTOR. The *libref* argument is optional and defaults to Sasuser. It can be specified with or without quotes.

WIDE | NOWIDE
specifies whether to suppress the column labels (COL1, COL2, and so on) and row labels (ROW1, ROW2, and so on) that are automatically produced for wide matrices. These labels are produced when the horizontal space needed to display the matrix in the ODS LISTING destination exceeds the current value of the LINESIZE option. If you are printing to an ODS destination for which horizontal space is unlimited (such as the HTML destination), you can use the RESET WIDE option to suppress the column and row labels. The default is NOWIDE.

RESUME Statement

RESUME ;

The RESUME statement enables you to continue execution from the line in a module where the most recent PAUSE statement was executed. PROC IML issues an automatic pause when an error occurs inside a module. If a module was paused because of an error, the RESUME statement resumes execution immediately after the statement that caused the error. The SHOW PAUSE statement displays the current state of all paused modules.

RETURN Statement

RETURN < *(operand)* > **;**

The RETURN statement causes a program to return to a previous calling point.

The RETURN statement with an *operand* is used in function modules that return a value. The *operand* can be a variable name or an expression. It is evaluated and the value is returned. Parentheses are optional. The RETURN statement without an argument is used to return from a user-defined subroutine.

You can also use the RETURN statement in conjunction with a LINK statement. If a LINK statement has been issued, the RETURN statement returns control to the statement that follows the LINK statement. See the description of the LINK statement. Also, see Chapter 6 for details.

If a RETURN statement is encountered outside a module, execution is stopped as with a STOP statement.

The following examples use the RETURN statement to exit from modules:

```
start sum1(a, b);
   sum = a+b;
   return(sum);
finish;
start sum2(s, a, b);
   s = a+b;
   return;
finish;

x = sum1(2, 3);
run sum2(y, 4, 5);
print x y;
```

Figure 25.368 Return from Module Calls

x	y
5	9

ROOT Function

ROOT(*matrix* < , *OnError*>**);**

The ROOT function performs the Cholesky decomposition of a symmetric and positive definite matrix. The arguments are as follows:

matrix	specifies a symmetric and positive definite matrix.
OnError	is an optional string that controls the behavior of the function when *matrix* is not positive definite. The default behavior is to stop with an error if *matrix* is not positive definite. If the string has the value “NoError”, the function returns a matrix of missing values but does not stop with an error.

The Cholesky decomposition factors the symmetric, positive definite matrix, $\mathbf{A}$, into the product

$$\mathbf{A} = \mathbf{U}'\mathbf{U}$$

where $\mathbf{U}$ is upper triangular.

For example, the following statements compute the upper-triangular matrix, `U`, in the Cholesky decomposition of a matrix:

```
A = {25  0  5,
       0  4  6,
       5  6 59};
U = root(A);
print U;
```

Figure 25.369 Cholesky Decomposition

U
5 0 1
0 2 3
0 0 7

If you need to solve a linear system and you already have a Cholesky decomposition of your matrix, then use the TRISOLV function as illustrated by the following statements:

```
b = {5, 2, 53};
/* Want to solve A * v = b.
   First solve U` z = b,
   then solve U v = z */
z = trisolv(2, U, b);
v = trisolv(1, U, z);
print v;
```

Figure 25.370 Solution to a Linear System

v
0
-1
1

The ROOT function performs most of its computations in the memory allocated for returning the Cholesky decomposition.

You can use the optional argument to test whether a matrix is positive definite, as shown in the following statements:

```
call randseed(12345);
count = 0;
x = j(3,3);
do i = 1 to 10;
   call randgen(x,"Normal");
   m = x` + x + 2*I(3); /* symmetric, but might not be pos. def. */
   g = root(m, "NoError");
   if all(g=.) then count = count + 1;
end;
msg = char(count) + " out of 10 matrices were not positive definite";
print msg;
```

Figure 25.371 Test Whether Matrices Are Positive Definite

msg
4 out of 10 matrices were not positive definite

ROW Function

ROW(*x*);

The ROW function is part of the IMLMLIB library. The ROW function returns a matrix that has the same dimensions as the *x* matrix and whose *i*th row has the value *i*. You can use the ROW and COL function to extract elements of a matrix. See the COL function for an example.

You can also use the ROW function to generate an ID variable when you convert data from a wide format to a long format. For example, the following statements show how to generate a column vector that has values $\{1, 1, 1, 2, 2, 2, \ldots, 5, 5, 5\}$:

```
NumSubjects = 5;         /* number of subjects */
NumRepeated = 3;         /* number of repeated obs per subject */
Z = row(j(NumSubjects, NumRepeated));
Subj = shape(Z, 0, 1);  /* {1, 1, 1, 2, 2, 2, ..., 5, 5, 5}   */
```

ROWCAT Function

ROWCAT(*matrix* < , *rows* > < , *columns* > **);**

The ROWCAT function concatenates rows of a character matrix without using blank compression. In particular, the function takes a character matrix or submatrix as its argument and creates a new matrix with one column whose elements are the concatenation of all row elements into a single string.

The arguments to the ROWCAT function are as follows:

matrix	is a character matrix or quoted literal.
rows	select the rows of *matrix*.
columns	select the columns of *matrix*.

If the input matrix has *n* rows and *m* columns, the result will have *n* rows and 1 column. The element length of the result is *m* times the element length of the argument. The optional rows and columns arguments can be used to select which rows and columns are concatenated.

For example, the following statements produce the 2×1 matrix shown:

```
b = {"ABC"  "D  "  "EF ",
     " GH"  " I "  " JK"};
a = rowcat(b);
print a;
```

Figure 25.372 Concatenation of Rows

a
ABCD EF
GH I JK

You can put quotes (") around elements of a character matrix in order to embed blanks or special characters, and to specify values that are lowercase or mixed case.

The syntax

ROWCAT(*matrix, rows, columns***);**

returns the same result as

ROWCAT(*matrix[rows, columns]***);**

The syntax

ROWCAT(*matrix, rows***);**

returns the same result as

ROWCAT(*matrix[rows,]***);**

ROWCATC Function

ROWCATC(*matrix* < , *rows* > < , *columns* > **);**

The ROWCATC function concatenates rows of a character matrix by using blank compression.

The arguments the ROWCATC function are as follows:

matrix	is a character matrix or quoted literal.
rows	select the rows of *matrix*.
columns	select the columns of *matrix*.

The ROWCATC function works the same way as the ROWCAT function except that blanks in element strings are moved to the end of the concatenation, as shown in the following example:

```
b = {"ABC"  "D  "  "EF ",
     " GH"  " I "  " JK"};
a = rowcatc(b);
print a (nleng(a))[label="NumChars"];
```

Figure 25.373 Concatenation of Rows

a	NumChars
ABCDEF	9
GHIJK	

ROWVEC Function

ROWVEC(*matrix*);

The ROWVEC function is part of the IMLMLIB library. 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 m elements in the vector correspond to the first row of the input matrix, the next m elements correspond to the second row, and so on, on, as shown in the following example.

```
x = {1 2 3,
     4 5 6};
y = rowvec(x);
print y;
```

Figure 25.374 A Row Vector

y
1 2 3 4 5 6

See the COLVEC function for converting a matrix into a column vector.

RSUBSTR Function

RSUBSTR(*x, p, l, r*);

The RSUBSTR function is part of the IMLMLIB library. The RSUBSTR function returns an $m \times n$ matrix with substrings of the input matrix with new strings.

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.

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 corresponding entry in the matrix R:

```
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, c;
```

Figure 25.375 Substring Replacement

x
ABC DEF GHI
JKL MNO PQR

c
ZC YF XI
WL VO UR

RUN Statement

RUN < *name* > < *(arguments)* > **;**

The RUN statement executes a user-defined module or invokes PROC IML's built-in subroutines.

The arguments to the RUN statement are as follows:

name	is the name of a user-defined module or a built-in subroutine.
arguments	are arguments to the subroutine.

If you define a module that has the same name as a built-in subroutine, the RUN statement can be used to call the user-defined subroutine.

If a RUN statement cannot be resolved at resolution time, a warning appears. If the RUN statement is still unresolved when executed and a storage library is open at the time, an attempt is made to load a module from that storage. If no module is found, an error message is generated.

If you do not supply a module name, the RUN statement tries to run the module named MAIN.

The following example defines and runs a module:

```
start MySum(y, x);
   y = sum(x);
finish;
run MySum(y, 1:5);
print y;
```

Figure 25.376 Run a User-Defined Module

y
15

See Chapter 6 and the CALL statement for further details.

RUPDT Call

CALL RUPDT(*rup, bup, sup, r, z* < *, b* > < *, y* > < *, ssq* > **);**

See the entry for the RDODT subroutine for details.

RZLIND Call

CALL RZLIND(*lindep, rup, bup, r* **<** *, sing* **>** **<** *, b* **>** **);**

The RZLIND subroutine computes rank-deficient linear least squares solutions, complete orthogonal factorizations, and Moore-Penrose inverses.

The RZLIND subroutine returns the following values:

lindep is a scalar that contains the number of linear dependencies that are recognized in **R** (number of zeroed rows in **`rup[n,n]`**).

rup is the updated $n \times n$ upper triangular matrix **R** that contains zero rows corresponding to zero recognized diagonal elements in the original **R**.

bup is the $n \times p$ matrix **B** of right-hand sides that is updated simultaneously with **R**. If *b* is not specified, *bup* is not accessible.

The input arguments to the RZLIND subroutine are as follows:

r specifies the $n \times n$ upper triangular matrix **R**. Only the upper triangle of *r* is used; the lower triangle can contain any information.

sing is an optional scalar that specifies a relative singularity criterion for the diagonal elements of **R**. The diagonal element r_{ii} is considered zero if $r_{ii} \leq \mathit{sing}\|r_i\|$, where $\|r_i\|$ is the Euclidean norm of column r_i of **R**. If the value provided for *sing* is not positive, the default value $\mathit{sing} = 1000\epsilon$ is used, where ϵ is the relative machine precision.

b specifies the optional $n \times p$ matrix **B** of right-hand sides that have to be updated or downdated simultaneously with **R**.

The singularity test used in the RZLIND subroutine is a relative test that uses the Euclidean norms of the columns r_i of **R**. The diagonal element r_{ii} is considered as nearly zero (and the *i*th row is zeroed out) if the following test is true:

$$r_{ii} \leq \mathrm{sing}\|r_i\|, \quad \text{where } \|r_i\| = \sqrt{r_i' r_i}$$

Providing an argument $\mathit{sing} \leq 0$ is the same as omitting the argument *sing* in the RZLIND call. In this case, the default is $\mathit{sing} = 1000\epsilon$, where ϵ is the relative machine precision. If **R** is computed by the QR decomposition $\mathbf{A} = \mathbf{QR}$, then the Euclidean norm of column *i* of **R** is the same (except for rounding errors) as the Euclidean norm of column *i* of **A**.

A Cholesky Root

Consider the following application of the RZLIND subroutine. Assume that you want to compute the upper triangular Cholesky factor **R** of the $n \times n$ positive semidefinite matrix $\mathbf{A}'\mathbf{A}$,

$$A'A = R'R \text{ where } A \in \mathcal{R}^{m \times n},\ \mathrm{rank}(A) = r,\ r \leq n \leq m$$

The Cholesky factor **R** of a positive definite matrix $\mathbf{A}'\mathbf{A}$ is unique (with the exception of the sign of its rows). However, the Cholesky factor of a positive semidefinite (singular) matrix $\mathbf{A}'\mathbf{A}$ can have many different forms.

In the following example, **A** is a 12×8 matrix with linearly dependent columns $a_1 = a_2 + a_3 + a_4$ and $a_1 = a_5 + a_6 + a_7$ with $r = 6$, $n = 8$, and $m = 12$.

```
proc iml;
call randseed(12345);
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};
RHS = randfun(nrow(a), "uniform");
a = a || RHS;
aa = a` * a;
m = nrow(a); n = ncol(a);
```

Applying the ROOT function to the coefficient matrix $\mathbf{A}'\mathbf{A}$ of the normal equations generates an upper triangular matrix $\mathbf{R}_1$ in which linearly dependent rows are zeroed out. The following statements verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_1'\mathbf{R}_1$:

```
r1 = root(aa);
ss1 = ssq(aa - r1` * r1);
print ss1 r1[format=best6.];
```

Figure 25.377 A Cholesky Root

ss1	r1							
7.011E-29	3.4641	1.1547	1.1547	1.1547	1.1547	1.1547	1.1547	2.264
	0	1.633	-0.816	-0.816	0.4082	-0.204	-0.204	0.2499
	0	0	1.4142	-1.414	39E-18	0.3536	-0.354	-0.195
	0	0	0	13E-16	11E-17	56E-18	39E-17	-3E-16
	0	0	0	0	1.5811	-0.791	-0.791	0.085
	0	0	0	0	0	1.3693	-1.369	-0.026
	0	0	0	0	0	0	89E-17	-5E-16
	0	0	0	0	0	0	0	0.6676

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.];
```

Figure 25.378 A QR Decomposition

ss2	r2							
4.417E-29	-3.464	-1.155	-1.155	-1.155	-1.155	-2.264	-1.155	-1.155
	0	-1.633	0.2041	-0.408	0.8165	-0.145	0.8165	0.2041
	0	0	1.6202	-0.772	0.3086	0.0624	-0.309	-0.849
	0	0	0	-1.38	-0.173	-0.288	0.1725	1.3801
	0	0	0	0	-1.369	-0.026	1.3693	18E-17
	0	0	0	0	0	0.6676	-1E-17	-2E-16
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

Using the RUPDT subroutine for stepwise updating of $\mathbf{R}$ by the m rows of $\mathbf{A}$ results in an upper triangular matrix $\mathbf{R}_3$ with $n - r$ nearly zero diagonal elements. However, other elements in rows with nearly zero diagonal elements can have significant values. The following statements verify that $\mathbf{A}'\mathbf{A} = \mathbf{R}_3'\mathbf{R}_3$:

```
r3 = shape(0,n,n);
call rupdt(rup,bup,sup,r3,a);
r3 = rup;
ss3 = ssq(aa - r3` * r3);
print ss3 r3[format=best6.];
```

Figure 25.379 An Updated Matrix

ss3	r3							
3.469E-29	3.4641	1.1547	1.1547	1.1547	1.1547	1.1547	1.1547	2.264
	0	-1.633	0.8165	0.8165	-0.408	0.2041	0.2041	-0.25
	0	0	-1.414	1.4142	-6E-17	-0.354	0.3536	0.1949
	0	0	0	5E-16	0.3739	0.3484	-0.722	0.0285
	0	0	0	0	-1.536	0.8984	0.6379	-0.08
	0	0	0	0	0	1.2536	-1.254	-0.032
	0	0	0	0	0	0	-4E-16	0.4507
	0	0	0	0	0	0	0	0.4921

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 to the upper triangular result $\mathbf{R}_3$ of the RUPDT subroutine generates a Cholesky factor $\mathbf{R}_4$ with rows of zeros that correspond to diagonal elements that are small. This gives 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.];
```

Figure 25.380 The Transformed Root Matrix

ss4	r4							
3.75E-29	3.4641	1.1547	1.1547	1.1547	1.1547	1.1547	1.1547	2.264
	0	-1.633	0.8165	0.8165	-0.408	0.2041	0.2041	-0.25
	0	0	-1.414	1.4142	-6E-17	-0.354	0.3536	0.1949
	0	0	0	0	0	0	0	0
	0	0	0	0	-1.581	0.7906	0.7906	-0.085
	0	0	0	0	0	1.3693	-1.369	-0.026
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0.6676

Rank-deficient Least Squares

Consider the rank-deficient linear least squares problem:

$$\min_x \|Ax - b\|^2 \text{ where } \mathbf{A} \in \mathcal{R}^{m \times n},\ \text{rank}(\mathbf{A}) = r,\ r \leq n \leq m$$

For $r = n$, the optimal solution, $\hat{x}$, is unique; however, for $r < n$, the rank-deficient linear least squares problem has many optimal solutions, each of which has the same least squares residual sum of squares:

$$\text{ss} = (\mathbf{A}\hat{x} - b)'(\mathbf{A}\hat{x} - b)$$

The solution of the full-rank problem, $r = n$, is illustrated in the section "The Full-Rank Linear Least Squares Problem" on page 903. The following example demonstrates how to compute several solutions to the singular problem. The example uses the 12×8 matrix from the preceding section and generates a new column vector b. The vector b and the matrix $\mathbf{A}$ are shown in the output.

```
b = randfun(m, "uniform");
ab = a` * b;
print b a[format=best6.];
```

Figure 25.381 Singular Data

b	a						
0.2783823	1	1	0	0	1	0	0	0.5833
0.9335446	1	1	0	0	1	0	0	0.9936
0.187392	1	1	0	0	0	1	0	0.5879
0.3687583	1	1	0	0	0	0	1	0.8575
0.1010699	1	0	1	0	1	0	0	0.8247
0.7411925	1	0	1	0	0	1	0	0.2806
0.1506827	1	0	1	0	0	1	0	0.6474
0.8406209	1	0	1	0	0	0	1	0.3819
0.7504291	1	0	0	1	1	0	0	0.449
0.9937212	1	0	0	1	0	1	0	0.8758
0.0935358	1	0	0	1	0	0	1	0.5184
0.7384078	1	0	0	1	0	0	1	0.8427

The rank-deficient linear least squares problem can be solved in the following ways. Although each method minimizes the residual sum of squares, not all of the given solutions are of minimum Euclidean length.

An SVD Solution

You can solve the rank-deficient least squares problem by using 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.];
```

Figure 25.382 SVD Solution

ss1	len1
1.2163596	0.1588224

x1							
0.2579	-0.004	0.0396	0.2224	0.1018	0.1082	0.0478	0.1301

The solution $\hat{x}_1$ obtained by singular value decomposition, $\hat{x}_1 = \mathbf{VD}^{-}\mathbf{U}'b/4$, is of minimum Euclidean length.

QR with Column Pivoting

You can solve the rank-deficient least squares problem by using the QR decomposition with column pivoting:

$$\mathbf{A\Pi} = \mathbf{QR} = \begin{bmatrix} \mathbf{Y} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \\ \mathbf{0} & \mathbf{0} \end{bmatrix} = \mathbf{Y} \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 \end{bmatrix}$$

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 $\hat{x}_2$:

$$\mathbf{R}^{-} = \begin{bmatrix} \mathbf{R}_1^{-1} \\ \mathbf{0} \end{bmatrix} \hat{x}_2 = \mathbf{\Pi R}^{-}\mathbf{Y}'b$$

```
ord = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr = n - lindqr;
r = r2[1:nr,1:nr];
x2 = shape(0,n,1);
x2[pivqr] = trisolv(1,r,qtb[1:nr]) // j(lindqr,1,0.);
len2 = x2` * x2;
ss2 = ssq(a * x2 - b);
x2 = x2`;
print ss2 len2, x2 [format=best6.];
```

Figure 25.383 QR Solution

ss2	len2
1.2163596	0.220502

x2							
0.4057	-0.044	0	0.1828	-0.006	0	-0.06	0.1301

Notice that the residual sum of squares is minimal, but the solution $\hat{x}_2$ is not of minimum Euclidean length.

Cholesky Root

You can solve the rank-deficient least squares problem by using the result $\mathbf{R}_1$ of the ROOT function to obtain the vector *piv* which indicates the zero rows in the upper triangular matrix $\mathbf{R}_1$. The following statements define a function that returns an index vector:

```
start PivotR(R);
   n = ncol(R);
   piv = j(n,1,.);
   z = j(n,1,.);   /* temp storage for index of zero diagonals */
   j1 = 0; j2 = 0;
   /* detect diagonal elements that are numerically zero */
   do i = 1 to n;
      small = 1000*constant("maceps")*norm(R[,i]);
      if R[i,i] > small then do;
         j1 = j1 + 1; piv[j1] = i;
      end;
      else do;
         j2 = j2 + 1; z[j2] = i;
      end;
   end;
   if j2>0 then piv[(n-j2+1):n] = z[1:j2];
   return( piv );
finish;

r1 = root(aa);
nr = n - lind;
piv = PivotR(r1);
```

Now compute $\hat{x}_3$ by solving the equation $\hat{x}_3 = \mathbf{R}^{-1}\mathbf{R}^{-\prime}\mathbf{A}'b$.

```
r  = r1[piv[1:nr],piv[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.];
```

Figure 25.384 Cholesky Root Solution

ss3	len3
1.2163596	0.3870311

x3							
0.5281	-0.226	-0.183	0	0.054	0.0604	0	0.1301

Note that the residual sum of squares is minimal, but the solution $\hat{x}_3$ is not of minimum Euclidean length.

Update of Cholesky Root

You can solve the rank-deficient least squares problem by using the result $\mathbf{R}_3$ of the RUPDT call on page 961 and the vector *piv* (obtained in the previous solution), which indicates the zero rows of upper triangular matrices $\mathbf{R}_1$ and $\mathbf{R}_3$. After zeroing out the rows of $\mathbf{R}_3$ belonging to small diagonal pivots, solve the system $\hat{x}_4 = \mathbf{R}^{-1}\mathbf{Y}'b$.

```
r3 = shape(0,n,n);
qtb = shape(0,n,1);
call rupdt(rup,bup,sup,r3,a,qtb,b);
r3 = rup; qtb = bup;
call rzlind(lind,r4,bup,r3,,qtb);
qtb = bup[piv[1:nr]];
x  = trisolv(1,r4[piv[1:nr],piv[1:nr]],qtb);
x4 = shape(0,n,1);
x4[piv] = x // j(lind,1,0.);
len4 = x4` * x4;
ss4 = ssq(a * x4 - b);
x4 = x4`;
print ss4 len4, x4[format=best6.];
```

Figure 25.385 Cholesky Update Solution

ss4	len4
1.2163596	0.3870311

x4							
0.5281	-0.226	-0.183	0	0.054	0.0604	0	0.1301

Because the matrices $\mathbf{R}_4$ and $\mathbf{R}_1$ are the same (except for the signs of rows), the solution $\hat{x}_4$ is the same as $\hat{x}_3$.

RZLIND Method

You can solve the rank-deficient least squares problem by using the result $\mathbf{R}_4$ of the RZLIND subroutine in the previous solution, which is the result of the first step of *complete QR decomposition*, and perform the second step of complete QR decomposition. The rows of matrix $\mathbf{R}_4$ can be permuted to the upper trapezoidal form

$$\begin{bmatrix} \widehat{\mathbf{R}} & \mathbf{T} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $\widehat{\mathbf{R}}$ is nonsingular and upper triangular and $\mathbf{T}$ is rectangular. Next, perform the second step of complete QR decomposition with the lower triangular matrix

$$\begin{bmatrix} \widehat{\mathbf{R}}' \\ \mathbf{T}' \end{bmatrix} = \bar{\mathbf{Y}} \begin{bmatrix} \bar{\mathbf{R}} \\ \mathbf{0} \end{bmatrix}$$

which leads to the upper triangular matrix $\bar{\mathbf{R}}$.

```
r = r4[piv[1:nr],]`;
call qr(q,r5,piv2,lin2,r);
y  = trisolv(2,r5,qtb);
x5 = q * (y // j(lind,1,0.));
len5 = x5` * x5;
ss5 = ssq(a * x5 - b);
x5 = x5`;
print ss5 len5, x5[format=best6.];
```

Figure 25.386 RZLIND Solution

ss5	len5
1.2163596	0.1588224

x5							
0.2579	-0.004	0.0396	0.2224	0.1018	0.1082	0.0478	0.1301

The solution $\hat{x}_5$ obtained by complete QR decomposition has minimum Euclidean length.

Complete QR Decomposition

You can solve the rank-deficient least squares problem by performing both steps of complete QR decomposition. The first step performs the pivoted QR decomposition of $\mathbf{A}$,

$$\mathbf{A\Pi} = \mathbf{QR} = \mathbf{Y} \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} = \mathbf{Y} \begin{bmatrix} \widehat{\mathbf{R}}\mathbf{T} \\ \mathbf{0} \end{bmatrix}$$

where $\widehat{\mathbf{R}}$ is nonsingular and upper triangular and $\mathbf{T}$ is rectangular. The second step performs a QR decomposition as described in the previous method. This results in

$$\mathbf{A\Pi} = \mathbf{Y} \begin{bmatrix} \bar{\mathbf{R}}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \bar{\mathbf{Y}}'$$

where $\bar{\mathbf{R}}'$ is lower triangular.

```
ord  = j(n,1,0);
call qr(qtb,r2,pivqr,lindqr,a,ord,b);
nr   = n - lindqr;
r    = r2[1:nr,]`;
call qr(q,r5,piv2,lin2,r);
y    = trisolv(2,r5,qtb[1:nr]);
x6   = shape(0,n,1);
x6[pivqr] = q * (y // j(lindqr,1,0.));
len6 = x6` * x6;
```

```
ss6  = ssq(a * x6 - b);
x6   = x6`;
print ss6 len6, x6[format=best6.];
```

Figure 25.387 Complete QR Solution

ss6	len6
1.2163596	0.1588224

x6							
0.2579	-0.004	0.0396	0.2224	0.1018	0.1082	0.0478	0.1301

The solution $\hat{x}_6$ obtained by complete QR decomposition has minimum Euclidean length.

Complete QR Decomposition with LUPDT

You can solve the rank-deficient least squares problem by performing a 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.];
```

Figure 25.388 Complete QR Solution with Update

ss7	len7
1.2163596	0.1588224

x7							
0.2579	-0.004	0.0396	0.2224	0.1018	0.1082	0.0478	0.1301

The solution $\hat{x}_7$ obtained by complete QR decomposition has minimum Euclidean length.

Complete QR Decomposition with RUPDT

You can solve the rank-deficient least squares problem by performing a 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.];
```

Figure 25.389 Complete QR Solution with Updates

ss8	len8
1.2163596	0.1588224

x8							
0.2579	-0.004	0.0396	0.2224	0.1018	0.1082	0.0478	0.1301

The solution $\hat{x}_8$ obtained by complete QR decomposition has minimum Euclidean length. The same result can be obtained with the APPCORT call or the COMPORT call.

Moore-Penrose Inverse

You can use various orthogonal methods to compute the Moore-Penrose inverse $\mathbf{A}^-$ of a rectangular matrix $\mathbf{A}$. The following examples find the Moore-Penrose inverse of the matrix $\mathbf{A}$ shown in section "A Cholesky Root" on page 962.

Generalized Inverse

You can find the Moore-Penrose inverse by using the GINV function. The GINV function 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.];
```

Figure 25.390 Moore-Penrose Inverse

ss1
6.06E-29

Figure 25.390 *continued*

ga											
0.2252	-0.136	0.1656	-0.055	-0.169	0.2546	-0.068	0.1821	0.2791	-0.152	0.1791	-0.106
0.25	0.044	0.2826	0.1568	-0.241	0.0668	-0.117	0.0254	0.0141	-0.165	0.0237	-0.139
-0.042	-0.058	-0.111	-0.055	0.2068	0.1592	0.1447	0.2226	-0.04	-0.126	-0.044	-0.057
0.0173	-0.121	-0.006	-0.157	-0.134	0.0286	-0.095	-0.066	0.3047	0.139	0.1996	0.09
0.2244	0.0711	-0.068	-0.161	0.1239	0.0369	-0.1	0.0061	0.314	-0.136	0.0048	-0.116
-0.004	-0.099	0.2471	-0.078	-0.174	0.2039	0.1189	-0.082	0.0104	0.1634	-0.016	-0.091
0.0046	-0.108	-0.014	0.1839	-0.118	0.0138	-0.087	0.2579	-0.045	-0.18	0.1903	0.1013
-0.447	0.4739	-0.295	0.2677	0.5573	-0.522	0.3013	-0.337	-0.584	0.5152	-0.329	0.3984

An SVD Solution

You can find the Moore-Penrose inverse by using the singular value decomposition. The singular decomposition $\mathbf{A} = \mathbf{UDV}'$ with $\mathbf{U}'\mathbf{U} = \mathbf{I}_m$, $\mathbf{D} = \mathrm{diag}(d_i)$, and $\mathbf{V}'\mathbf{V} = \mathbf{V}\mathbf{V}' = \mathbf{I}_n$, can be used to compute $\mathbf{A}^- = \mathbf{V}\mathbf{D}^\dagger\mathbf{U}'$, with $\mathbf{D}^\dagger = \mathrm{diag}(d_i^\dagger)$ and

$$d_i^\dagger = \begin{cases} 0 & \text{where } d_i \leq \epsilon \\ 1/d_i & \text{otherwise} \end{cases}$$

The result $\mathbf{A}^-$ should be the same as that given by the GINV function if the singularity criterion ϵ is selected correspondingly. Since you cannot specify the criterion ϵ for the GINV function, the singular value decomposition approach can be important for applications where the GINV function uses an unsuitable ϵ criterion. The slight discrepancy between the values of SS1 and SS2 is because of 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;
```

Figure 25.391 SVD Solution

ss2
6.388E-29

Complete QR Decomposition

You can find the Moore-Penrose inverse by using the complete QR decomposition. The complete QR decomposition

$$\mathbf{A} = \mathbf{Y}\begin{bmatrix} \bar{\mathbf{R}}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\bar{\mathbf{Y}}'\mathbf{\Pi}'$$

where $\bar{\mathbf{R}}'$ is lower triangular, yields the Moore-Penrose inverse

$$\mathbf{A}^{-} = \mathbf{\Pi}\bar{\mathbf{Y}}\begin{bmatrix} \bar{\mathbf{R}}^{-\prime} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\mathbf{Y}'$$

```
ord = j(n,1,0);
call qr(q1,r2,pivqr,lindqr,a,ord);
nr = n - lindqr;
q1 = q1[,1:nr]; r = r2[1:nr,]`;
call qr(q2,r5,piv2,lin2,r);
tt = trisolv(4,r5`,q1`);
ga = shape(0,n,m);
ga[pivqr,] = q2 * (tt // shape(0,n-nr,m));
t1 = a * ga; t2 = t1`;
t3 = ga * a; t4 = t3`;
ss3 = ssq(t1 - t2) + ssq(t3 - t4) +
      ssq(t1 * a - a) + ssq(t3 * ga - ga);
print ss3;
```

Figure 25.392 Complete QR Solution

ss3
1.592E-29

Complete QR Decomposition with LUPDT

You can find the Moore-Penrose inverse by using the 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;
```

Figure 25.393 Complete QR Solution with Update

ss4
2.985E-29

Complete QR Decomposition with RUPDT

You can find the Moore-Penrose inverse by using the complete QR decomposition with the RUPDT call and the LUPDT call:

```
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;
```

Figure 25.394 Complete QR Solution with Updates

ss5
1.021E-29

SAMPLE Function

SAMPLE(*x* < , *n*> < , *method*> < , *prob*> **);**

The SAMPLE function generates a random sample of the elements of *x*. The function can sample from *x* with replacement or without replacement. The function can sample from *x* with equal probability or with unequal probability.

The arguments are as follows:

x is a matrix that specifies the sample space. That is, the sample is drawn from the elements of *x*.

n specifies the number of times to sample. The argument can be a scalar or a two-element vector.

- If this argument is omitted, then the number of elements of *x* is used.
- If *n* is a scalar, then it represents the sample size, which is the number of independent draws from the population. This value determines the number of columns in the output matrix.
- If *n* is a two-element vector, the first element represents the sample size. The second element specifies the number of samples, which is the number of rows in the output matrix. If the sampling is without replacement, then *n[1]* must be less than or equal to the number of elements in *x*.

method is an optional argument that specifies how sampling is performed. The following are valid options:

"Replace"	specifies simple random sampling with replacement. This is the default value.
"NoReplace"	specifies simple random sampling without replacement. The elements in the samples might appear in the same order as in *x*.
"WOR"	specifies simple random sampling without replacement. After elements are randomly selected, their order is randomly permuted.

prob is a vector with the same number of elements as *x*. The vector specifies the sampling probability for the elements of *x*. The SAMPLE function internally scales the elements of *prob* so that they sum to unity.

The SAMPLE function uses the random seed that is set by the RANDSEED function.

The *prob* argument specifies the probabilities that are used when sampling from *x*. When *method* is "Replace," the probabilities do not change during the sampling. However, when *method* is "NoReplace," the probabilities are renormalized after each selection.

For example, suppose that the element $x_i, i = 1 \ldots n$ has probability p_i of being sampled, where $\Sigma_{i=1}^{n} p_i = 1$. If the element x_1 is selected in the first round of sampling, the remaining elements have the new probability q_i of being sampled during the second round, where $q_i = p_i / (\Sigma_{j=2}^{n} p_j)$ and $i = 2 \ldots n$.

The following statements use three different methods to choose a sample from the integers 1–5:

```
x = 1:5;
call randseed(12345);
s1 = sample(x);
s2 = sample(x, 5, "Replace", {0.6 0.1 0.0 0.1 0.2});
s3 = sample(x, 3, "NoReplace");
print s1, s2, s3;
```

Figure 25.395 Random Samples

s1
3 5 3 5 5

s2
1 5 1 1 2

s3
1 2 5

SAVE Statement

SAVE ;

The SAVE statement saves data to a SAS data set.

The SAVE statement flushes any data residing in output buffers for all active output data sets and files to ensure that the data are written to disk. This is equivalent to closing and then reopening the files.

SCATTER Call

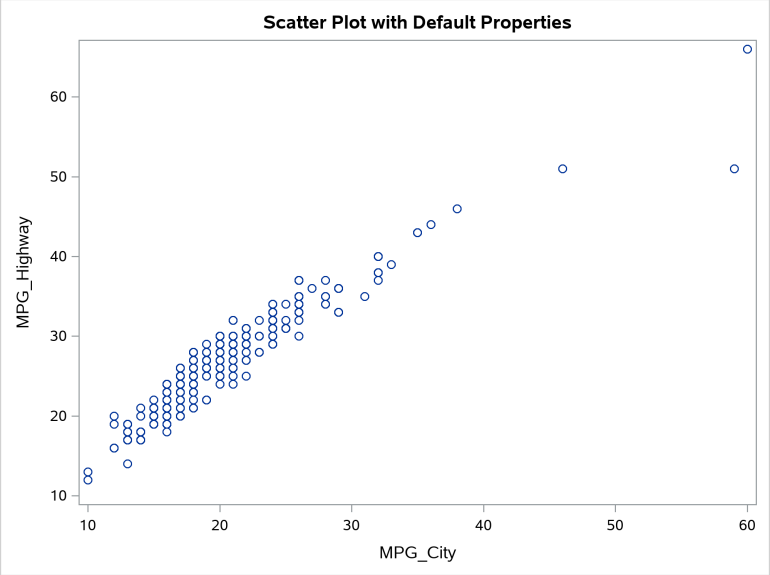
CALL SCATTER(*x,y*) **GROUP=***GroupVector*
DATALABEL=*LabelVector*
OPTION=*ScatterOption*
GRID={"X" <,"Y">}
LABEL={*XLabel* <,*YLabel*>**}**
XVALUES=*xValues*
YVALUES=*yValues*
PROCOPT=*ProcOption*
LINEPARM=*{x0 y0 slope}*
OTHER=*Stmts* **;**

The SCATTER subroutine displays a SCATTER plot by calling the SGPLOT procedure. The required arguments *x* and *y* are vectors that contain the data to plot. The remaining arguments are optional and can be specified in any order. The SCATTER subroutine is not a comprehensive interface to the SGPLOT procedure. It is intended for creating simple scatter plots for exploratory data analysis. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

A simple example follows:

```
use Sashelp.Cars;
read all var {MPG_City MPG_Highway};
close Sashelp.Cars;

title "Scatter Plot with Default Properties";
run Scatter(MPG_City, MPG_Highway) label={"MPG_City" "MPG_Highway"};
```

Figure 25.396 A Scatter Plot



Specify the required *x* vector inside parentheses and specify all optional arguments outside the parentheses. Use the global TITLE and FOOTNOTE statements to specify titles and footnotes. Each option corresponds

to a statement or option in the SGPLOT procedure.

The following options correspond to options in the SCATTER statement in the SGPLOT procedure:

GROUP= specifies a vector of values that determine groups in the plot. You can use a numeric or character vector. This option corresponds to the GROUP= option in the SCATTER statement.

DATALABEL= specifies a vector of values that label each marker in the plot. You can use a numeric or character vector.

OPTION= specifies a character matrix or string literal. The value is used verbatim to specify options in the SCATTER statement.

The SCATTER subroutine also supports the following options. The BAR subroutine documents these options and gives an example of their usage.

GRID= specifies whether to display grid lines for the X or Y axis.

LABEL= specifies axis labels for the X or Y axis.

XVALUES= specifies a vector of values for ticks for the X axis.

YVALUES= specifies a vector of values for ticks for the Y axis.

PROCOPT= specifies options in the PROC SGPLOT statement.

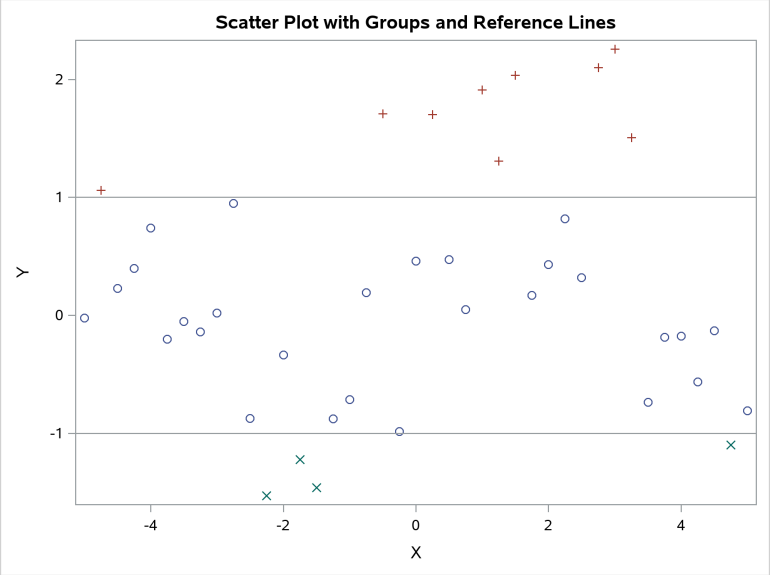
OTHER= specifies statements in the SGPLOT procedure.

In addition, the LINEPARM= option specifies a three-element vector whose elements specify the X=, Y= and SLOPE= options, respectively, on the LINEPARM statement.

The following example creates several scatter plots with various options. Each scatter plot is documented in the program comments.

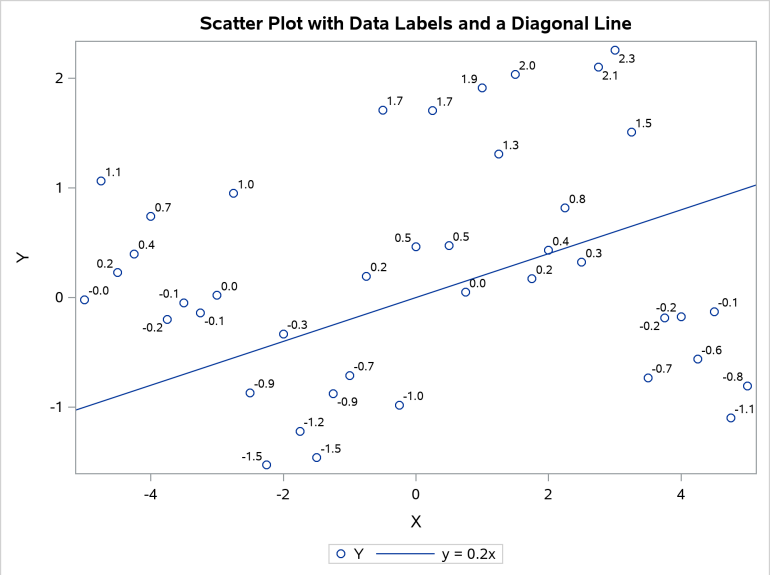
```
/* define data */
call randseed(1);
x = do(-5, 5, 0.25);
y = x/5 + sin(x) + RandFun(1||ncol(x), "Normal");

title "Scatter Plot with Groups and Reference Lines";
/* 1. Use the GROUP= option to assign a group to each observation
 * 2. Use the OTHER= option to add reference lines to the Y axis
 * 3. Use the PROCOPT= option to suppress the legend
 */
g = j(ncol(x), 1, 1);
g[ loc(y>=1) ] = 2;
g[ loc(y< -1) ] = 3;
run Scatter(x, y) group=g                       /* assign color/marker shape */
                  other="refline -1 1 / axis=y" /* add reference line */
                  procopt="noautolegend";       /* PROC option */
```

Figure 25.397 Group Attributes and Reference Lines



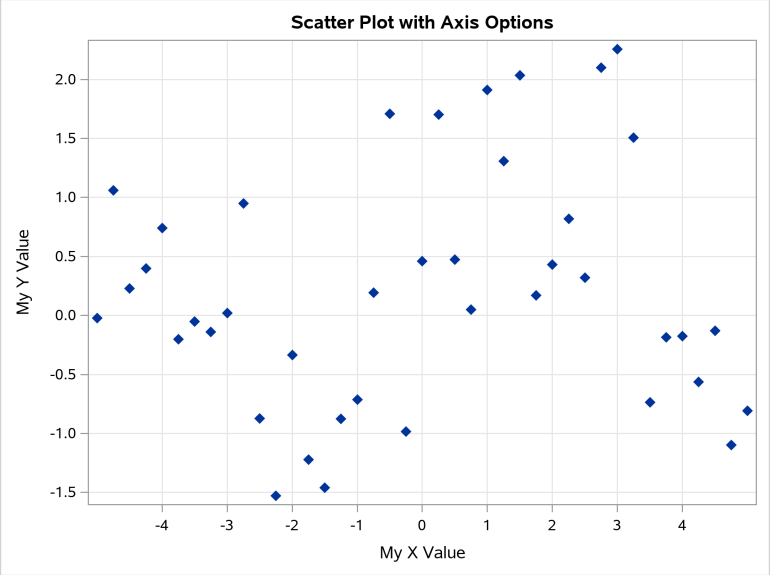

```
title "Scatter Plot with Data Labels and a Diagonal Line";
/* 1. Use the DATALABEL= option to label each marker
 * 2. Use the LINEPARM= option to add line passing through
 *    (0,0) with slope=0.2
 */
dlabels = putn(y, "4.1");
run Scatter(x, y) datalabel=dlabels    /* label each marker */
                  lineparm={0 0 0.2};  /* line through (0,0) with slope 0.2 */
```

Figure 25.398 Data Labels and Diagonal Line



```
title "Scatter Plot with Axis Options";
/* 1. Use the OPTION= option to specify marker attributes
 * 2. Use the GRID= option to add a reference grid
 * 3. Use the LABEL= option to specify axis labels
 * 4. Use the XVALUES= and YVALUES= options to specify tick positions
 */
call Scatter(x,y) option="markerattrs=(symbol=DiamondFilled)"
                  grid= {X Y}
                  label={"My X Value" "My Y Value"}
                  xvalues = -4:4
                  yvalues = do(-2,2,0.5);
```

Figure 25.399 Marker and Axis Attributes



SEQ, SEQSCALE, and SEQSHIFT Calls

CALL SEQ(*prob, domain* < , **TSCALE**=*tscale*> < , **EPS**=*eps*> < , **DEN**=*den*> **);**

CALL SEQSCALE(*prob, gscale, domain, level* < , **IGUESS**=*iguess*> < , **TSCALE**=*tscale*> < , **EPS**=*eps*> < , **DEN**=*den*> **);**

CALL SEQSHIFT(*prob, shift, domain, plevel* < , **IGUESS**=*iguess*> < , **TSCALE**=*tscale*> < , **EPS**=*eps*> < , **DEN=***den*> **);**

The SEQ, SEQSCALE, and SEQSHIFT subroutines perform discrete sequential tests.

The SEQSHIFT subroutine returns the following values:

prob is an $(m + 1) \times n$ matrix. The $[i, j]$ entry in the array contains the probability at the $[i, j]$ entry of the argument *domain*. Also, the probability at infinity at every level j is returned in the last entry ($[m + 1, j]$) of column j. Upon a successful completion of any routine, this variable is always returned.

gscale is a numeric variable that returns from the routine SEQSCALE and contains the scaling of the current geometry defined by *domain* that would yield a given significance level *level*.

shift is a numeric variable that returns from the routine SEQSHIFT and contains the shift of current geometry defined by *domain* that would yield a given power level *plevel*.

The input arguments to the SEQSHIFT subroutine are as follows:

domain specifies an $m \times n$ matrix that contains the boundary points separating the intervals of continuation/stopping of the sequential test. Each column k contains the boundary points at level k sorted in an ascending order. The values `.M` and `.P` represent $-\infty$ and $+\infty$, respectively. They must start on the first row, and any remaining entries must be filled with a missing value. Elements that follow the missing value in any column are ignored. The number of columns n is equal to the number of stages present in the sequential test. The row dimension m must be even, and it is equal to the maximum number of boundary points in a level. In fact, *domain* is the tabular form of the finite boundary points. Entries in *domain* with absolute values that exceed a standardized value of 8 at any level are internally reset to a standardized value of 8 or -8, depending on the sign of the entry. This is reflected in the results returned for the probabilities and the densities.

tscale specifies an optional $n - 1$ vector that describes the time intervals between two consecutive stages. In the absence of *tscale*, these time intervals are internally set to 1. The keyword for *tscale* is TSCALE.

eps specifies an optional numeric parameter for controlling the absolute precision of the computation. In the absence of *eps*, the precision is internally set to 1E–7. The keyword for *eps* is EPS.

den specifies an optional character string to describe the name of an $m \times n$ matrix. The $[i, j]$ entry in the matrix returns the density of the distribution at the $[i, j]$ entry of the matrix specified by the *domain* argument. The keyword for *den* is DEN.

iguess specifies an optional numeric parameter that contains an initial guess for the variable *gscale* in the SEQSCALE subroutine or for the variable *mean* in the SEQSHIFT subroutine. In general, very good estimates for these initial guesses can be provided by an iterative process, and these estimates become extremely valuable near convergence. The keyword for *iguess* is IGUESS.

level specifies a numeric parameter in the SEQSCALE subroutine that contains the required significance level to be achieved through scaling the *domain* (see the description of SEQSCALE).

plevel specifies a numeric parameter in the SEQSHIFT subroutine that provides the required power level to be achieved through shifting the *domain* (see the description of SEQSHIFT).

SEQ Call

To compute the probability from a sequential test, you must specify a matrix that contains the boundaries. With the optional additional information concerning the time intervals and the target accuracy, or their default values, the SEQ subroutine returns the matrix that contains the probability and optionally returns the density from a sequential test evaluated at each given point of the boundary. Let C_j denote the continuation set at each level j. C_j is defined to be the union at the jth level of all the intervals bounded from below by the points with even indices $0, 2, 4, \ldots$ and from above by the points with odd indices $1, 3, \ldots$.

The SEQ subroutine computes, with $\mu = 0$, the densities

$$f_j(s,\mu) = \int_{C_{j-1}} \phi(s-y,\mu,t_{j-1})\, f_{j-1}(y,\mu)\, dy, \text{ for } j = 2,3,\ldots$$

with

$$f_1(s,\mu) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{(s-\mu)^2}{2}\right]$$

and

$$\phi(s,\mu,t) = \frac{1}{\sqrt{2\pi t}} \exp\left[-\frac{(s-\mu)^2}{2t}\right]$$

with the associated probability at any point a at level j to be

$$P_j(a,\mu) = \int_{C_{j-1}} \Phi(a-y,\mu,t_j)\, f_{j-1}(y,\mu)\, dy, \text{ for } j = 2,3,\ldots$$

with

$$\Phi(b,\mu,t) = \int_{-\infty}^{b} \phi(s,\mu,t)\, ds$$

The notation τ denotes the vector of time intervals $t_1,\ldots,t_{n-1}$, and $P_j(g,\mu,\tau)$ denotes the probability of continuation at the jth level for a given domain g, a given mean μ, and a given time vector τ. The variance at the jth level can be computed from τ.

$$\begin{aligned} \sigma_1^2 &= 1 \\ \sigma_{j+1}^2 &= \sigma_j^2 + \tau_j, \text{ for } j = 1,2,\ldots \end{aligned}$$

It is important to understand the limitations that are imposed internally on the domain by the numerical method. Any element g_{ij} will always be limited within a symmetric interval with standardized values not to exceed 8. That is,

$$g_{ij} = \max[\min(g_{ij}, 8\sigma_j), -8\sigma_j]$$

SEQSCALE Call

Given a domain g, an optional time vector τ, and a probability level p_s, the SEQSCALE subroutine finds the amount of scaling s that would solve the problem

$$P_n(gs,0) = p_s$$

The result for the amount of scaling s is returned as the second argument of the SEQSCALE subroutine, *scale*. Note that because of the complexity of the problem, the SEQSCALE subroutine will not attempt to scale a domain with multiple intervals of continuation.

For a significance level of α, set $p_s = 1 - \alpha$.

SEQSHIFT Call

Given a geometry g, an optional time vector τ, and a power level $1-\beta$, the SEQSHIFT subroutine finds the mean μ that solves $\mu \geq 0$ such that $P_n(g,\mu) = \beta$.

Actually, a simple transformation of the variables in the sequential problem yields the following result:

$$P_j(g^{\mu},0) = P_j(g,\mu), \text{ for } j = 1,2,\ldots,n$$

where g^{μ} is given by $g^{\mu}_{ij} = g_{ij} - \mu j$.

Many options are available with the NLP family of optimization routines, which are described in Chapter 4, "Nonlinear Optimization Subroutines."

Example 1

Consider the following continuation intervals:

$$\begin{aligned} C_1 &= \{-6,2\} \\ C_2 &= \{-6,3\} \\ C_3 &= \{-6,4,5,6\} \\ C_4 &= \{-6,4\} \end{aligned}$$

The following statements computes the probability from the sequential test at each boundary point specified in the geometry.

```
/* function to insert a into the kth column of m */
start table(m,a,k);
   if ncol(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, prob, density;
```

Figure 25.400 Sequential Test Probabilities and Densities

m			
-6	-6	-6	-6
2	3	4	4
.	.	5	.
.	.	6	.

Figure 25.400 *continued*

prob			
9.866E-10	0.000011	0.0002592	0.0011748
0.9772499	0.9665354	0.9621691	0.9497676
1	0.9772499	0.9661587	0.9622611
1	0.9772499	0.96651	0.9622611
1	0.9772499	0.9665244	0.9622611

DENSITY			
6.0759E-9	0.0000348	0.0005668	0.0021223
0.053991	0.0226042	0.0092853	0.0194903
0	0	0.0010391	0
0	0	0.0000524	0

Figure 25.400 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*$[4, 3] = 0.0000524$.

Example 2

Consider the continuation intervals

$$\begin{aligned} C_1 &= \{-20, 2\} \\ C_2 &= \{-20, 20\} \\ C_3 &= \{-3, 3\} \end{aligned}$$

Note that the continuation at level 2 can be effectively considered infinite, and it does not numerically affect the results of the computation at level 3. The following statements verify this by using the *tscale* parameter to compute this problem.

```
free m;
call table(m, {-20,2}, 1);
call table(m, {-20,20}, 2);
call table(m, {-3, 3}, 3);

/************************************/
/* TSCALE has the default value of 1  */
/************************************/
call seq(prob1,m) eps = 1.e-8 den="density";
print m[format=f5.]  prob1[format=e12.5];

call table(mm,{-20,2},1);
call table(mm,{-3,3},2);
/* You can use 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];
```

Figure 25.401 Sequential Test Probabilities

m			
-6	-6	-6	-6
2	3	4	4
.	.	5	.
.	.	6	.

prob			
9.866E-10	0.000011	0.0002592	0.0011748
0.9772499	0.9665354	0.9621691	0.9497676
1	0.9772499	0.9661587	0.9622611
1	0.9772499	0.96651	0.9622611
1	0.9772499	0.9665244	0.9622611

DENSITY			
6.0759E-9	0.0000348	0.0005668	0.0021223
0.053991	0.0226042	0.0092853	0.0194903
0	0	0.0010391	0
0	0	0.0000524	0

Figure 25.401 shows the values returned for the variables **m**, **prob1**, **mm** and **prob2**.

Some internal limitations are imposed on the geometry. Consider the three-level case with geometry m in the preceding statements. Since the *tscale* variable is not specified, it is set to its default value, (1, 1). The variance at the jth level is $\sigma_j^2 = j$ for $j = 1, 2, 3$. The first level has a lower boundary point of -20, as represented by the value of $m[1, 1]$. Since the absolute standardized value is larger than 8, this point is replaced internally by the value -8. Hence, the densities and the probabilities reported for the first level at this point are not for the given value -20; instead, they are for the internal value of -8. For practical purposes, this limitation is not severe since the absolute error introduced is of the order of 10^{-16}.

The computations performed by the first call of the SEQ subroutine can be simplified since the second level is large enough to be considered infinite. The matrix MM contains the first and third columns of the matrix M. However, in order to specify the two-step separation between the levels, you must specify *tscale*=2.

Example 3

This example verifies some of the results published in Table 3 of Pocock (1982). That is, the following statements verify for the given domain that the significance level is 0.05 and that the power is $1 - \beta$ under the alternative hypothesis:

```
proc iml;
/* 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];
```

Figure 25.402 A Group Sequential Analysis

summary						
0.94997	0.95000	0.50000	0.50000	3.18225	15.91123	14.27319
0.95002	0.95000	0.25000	0.25000	6.05489	30.27447	22.64256
0.94998	0.95000	0.10000	0.10000	9.70370	48.51850	28.63182
0.94996	0.95000	0.05000	0.05000	12.29344	61.46720	31.29225

Notice that the variables *eps* and *tscale* have been internally set to their default values. Figure 25.402 shows the computed values, which compare well with the values shown in Table 3 of Pocock (1982). Differences are of the order of 10^{-5}.

Example 4

This example shows how to verify the results in Table 1 of Wang and Tsiatis (1987). For a given δ, the following program finds Γ that yields a symmetric continuation interval given by

$$-\Gamma j^{\delta} \leq C_j \leq \Gamma j^{\delta}$$

with a given significance level of α:

```
proc iml;
start func(delta,k) global(level);
   m       = ((1:k))##delta;
   mm      = (-m//m);
   /* meet the significance level by scaling */
   call seqscale(prob,scale,mm,level);
   return(scale);
finish;
```

```
/* alpha levels of 0.05 and 0.01 */
blevel        = {0.95 0.99};
do i = 1 to 2;
   level      = blevel[i];
   free summary;
   do delta   = 0 to .7 by .1;
      free row;
      do k=2 to 5;
         x    = func(delta,k);
         row  = row || x;
      end;
      summary = summary //row;
   end;
   print summary[format=10.5];
end;
```

Figure 25.403 Sequential Analysis

summary			
2.79651	3.47109	4.04857	4.56177
2.63138	3.14419	3.56921	3.93711
2.48773	2.86390	3.16426	3.41735
2.36514	2.62969	2.83067	2.99432
2.26248	2.43945	2.56507	2.66243
2.17827	2.28942	2.36129	2.41318
2.11096	2.17504	2.21128	2.23475
2.05897	2.09172	2.10680	2.11495

summary			
3.64806	4.49446	5.21782	5.86135
3.41360	4.04953	4.57518	5.03019
3.20589	3.66178	4.02728	4.33492
3.02838	3.33454	3.57007	3.76293
2.88369	3.07085	3.20639	3.31248
2.77170	2.87291	2.93864	2.98659
2.69054	2.73668	2.76152	2.77721
2.63633	2.65284	2.65923	2.66222

Figure 25.403 shows the value of SUMMARY for the 0.95 and 0.99 levels. Notice that since *eps* and *tscale* are not specified, they are internally set to their default values.

Example 5

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;
proc iml;
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   = 0.95;
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];
```

Figure 25.404 Sequential Analysis

summary				
0.99359	1.31083	1.59229	1.75953	2.07153

Figure 25.404 shows the results, which agree with Table 2 of Pocock (1977).

Example 6

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 Table 2 and Table 3 of Wang and Tsiatis (1987):

```
%let nl = 5;
proc iml;
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 *((&nl-1):0)`);
   return(asn);
finish;

/* set up the global variables needed by func */
level   = 0.95;
plevel  = 0.01;

/* set up options used to control the optimization routine */
opt     = {0 1 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 call nlpdd */
delta   = 0.5;
ods select IterStop ConvergenceStatus;
call nlpdd(rc,rx,"func",delta) opt=opt tc=tc par=par;
```

Figure 25.405 optimal Boundary

Optimization Results			
Iterations	3	**Function Calls**	6
Gradient Calls	5	**Active Constraints**	0
Objective Function	34.877416816	**Max Abs Gradient Element**	0.0001256295
Slope of Search Direction	-0.000010001	**Radius**	1

FCONV convergence criterion satisfied.

Figure 25.405 displays the results. 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 $1 - \beta = 0.99$. Notice that the variables *eps* and *tscale* are internally set to their default values. For more information about the NLPDD subroutine, see the section "NLPDD Call" on page 826. For details about the *opt*, *tc*, and *par* arguments in the NLPDD call, see the section "Options Vector" on page 335, the section "Termination Criteria" on page 339, and the section "Control Parameters Vector" on page 346, 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.

Example 7

This example illustrates how to find the boundaries that minimize ASN given the required significance level and the required power. It replicates some of the results published in Table 3 of Pocock (1982). The program computes the domain that

- minimizes the ASN

- yields a given significance level of 0.05
- yields a given power $1 - \beta$ under the alternative hypothesis

The last two nonlinear conditions on the optimization process can be incorporated as a penalty applied on the error in these nonlinear conditions. The following program does the computations for a power of 0.9.

```
%let nl=5;
proc iml;
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 *((&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 options used to control 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 to ensure 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 call nlpdd */
m       = { 1  1  1  1  1  };
call nlpdd(rc,rx,"func",m) opt=opt blc = con tc=tc par=par;
print stgeom;
```

Figure 25.406 Boundaries That Minimize ASN

Optimization Results			
Iterations	3	**Function Calls**	6
Gradient Calls	5	**Active Constraints**	0
Objective Function	34.877416816	**Max Abs Gradient Element**	0.0001256295
Slope of Search Direction	-0.000010001	**Radius**	1

FCONV convergence criterion satisfied.

Although *eps* has been set to *eps*=10^{-8}, *tscale* has been internally set to its default value. You can choose to run the program with and without the specification of the keyword IGUESS to see the effect on the execution time.

Notice the following about the optimization process:

- Different levels of precision are imposed on different modules. In this example, *epss*, which is used as the precision for the sequential tests, is 1E–8. The absolute and relative function criteria for the objective function are set to *par*[7]=1E–5 and *tc*[7]=1E–4, respectively. Since finite differences are used to compute the first and second derivatives, the sequential test should be more precise than the optimization routine. Otherwise, the finite difference estimation is worthless. Optimally, if the precision of the function evaluation is $O(\epsilon)$, the first- and second-order derivatives should be estimated with perturbations $O(\epsilon^{\frac{1}{2}})$ and $O(\epsilon^{\frac{1}{3}})$, respectively. For example, if all three precision levels are set to 1E–5, the optimization process does not work properly.
- Line search techniques that do not depend on the computation of the derivative are preferable.
- The amount of printed information from the optimization routines is controlled by *opt[2]* and can be set to any value between 0 and 3. Larger numbers produce more output.

SEQSCALE Call

CALL SEQSCALE(*prob, gscale, domain, level* **<,** **IGUESS=***iguess***>** **<,** **TSCALE=***tscale***>** **<,** **EPS=***eps***>** **<,** **DEN=***den***>** **);**

The SEQSCALE subroutine computes estimates of scales associated with discrete sequential tests.

See the entry for the SEQ subroutine for details.

SEQSHIFT Call

CALL SEQSHIFT(*prob, shift, domain, plevel* **<,** **IGUESS=***iguess***>** **<,** **TSCALE=***tscale***>** **<,** **EPS=***eps***>** **<,** **DEN=***den***>** **);**

The SEQSHIFT subroutine computes estimates of means associated with discrete sequential tests.

See the entry for the SEQ subroutine for details.

SERIES Call

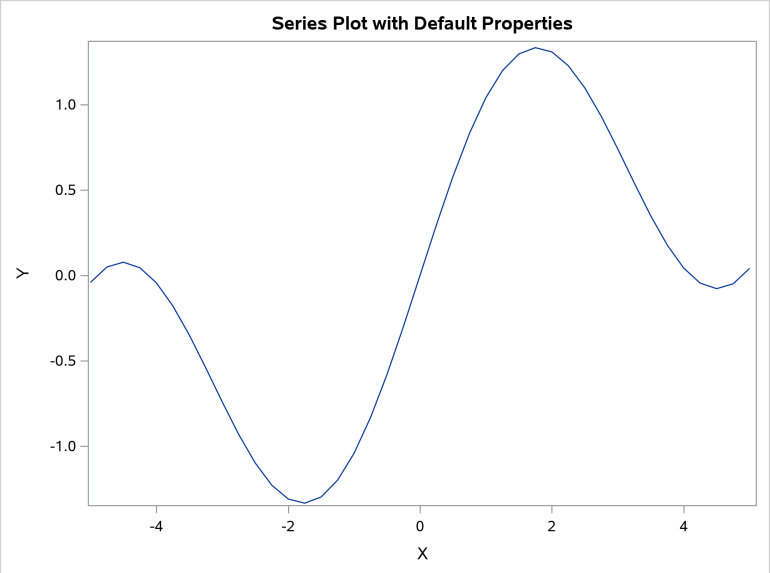
CALL SERIES(*x,y***) GROUP=***GroupVector***
OPTION=***SeriesOption***
GRID={"X" <,"Y">}
LABEL={***XLabel*** <,***YLabel***>}
XVALUES=***xValues***
YVALUES=***yValues***
PROCOPT=***ProcOption***
OTHER=***Stmts*** ;**

The SERIES subroutine displays a SERIES plot by calling the SGPLOT procedure. The required arguments *x* and *y* are vectors that contain the data to plot. The remaining arguments are optional and can be specified in any order. The SERIES subroutine is not a comprehensive interface to the SGPLOT procedure. It is intended for creating simple line plots for exploratory data analysis. The ODS statistical graphics subroutines are described in Chapter 18, "Statistical Graphics."

A simple example follows:

```
x = do(-5, 5, 0.25);
y = x/5 + sin(x);

title "Series Plot with Default Properties";
run Series(x, y);
```

Figure 25.407 A Series Plot



Specify the required *x* vector inside parentheses and specify all optional arguments outside the parentheses. Use the global TITLE and FOOTNOTE statements to specify titles and footnotes. Each option corresponds to a statement or option in the SGPLOT procedure.

The SERIES subroutine also supports the following options. The BAR subroutine documents these options and gives an example of their usage.

GRID= specifies whether to display grid lines for the X or Y axis.

LABEL= specifies axis labels for the X or Y axis.

XVALUES= specifies a vector of values for ticks for the X axis.

YVALUES= specifies a vector of values for ticks for the Y axis.

PROCOPT= specifies options in the PROC SGPLOT statement.

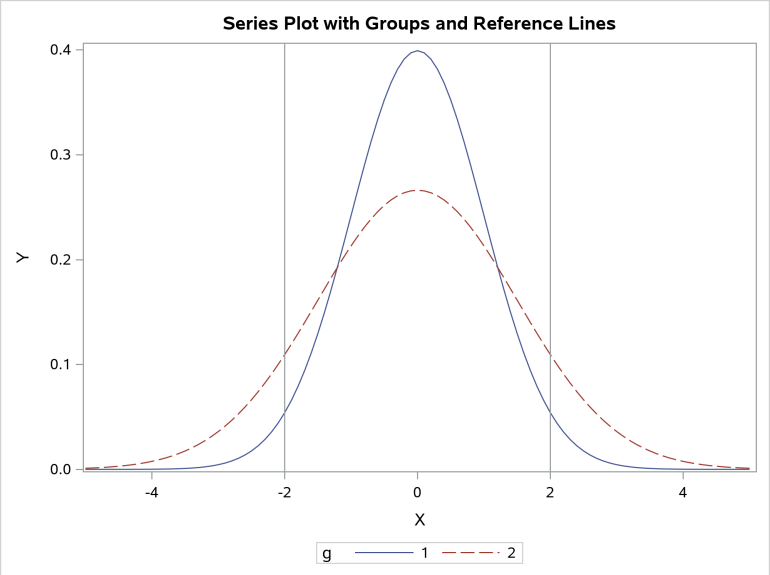
OTHER= specifies statements in the SGPLOT procedure.

In addition, you can use the OPTION= option to specify a character matrix or string literal. The value is used verbatim to specify options in the SERIES statement.

The following example creates several series plots with various options. Each series plot is documented in the program comments.

```
/* assign a group to each observation */
x = do(-5, 5, 0.1);
y1 = pdf("Normal", x, 0, 1);
y2 = pdf("Normal", x, 0, 1.5);
g = repeat({1,2}, 1, ncol(x));
x = x  || x ;
y = y1 || y2;

title "Series Plot with Groups and Reference Lines";
/* 1. Use the GROUP= option to assign a group to each observation
 * 2. Use the OTHER= option to add reference lines to the X axis
 */
call Series(x, y) group=g                  /* assign color/marker shape */
                  other="refline -2 2 / axis=x"; /* add reference line */
```

Figure 25.408 Group Attributes and Reference Lines



```
title "Series Plot with Axis Options";
/* 1. Use the OPTION= option to display markers
 * 2. Use the GRID= option to add a reference grid
 * 3. Use the LABEL= option to specify axis labels
 * 4. Use the XVALUES= and YVALUES= options to specify tick positions
 */
x = do(-5, 5, 0.25);
y = x/5 + 2*sin(x);
call Series(x,y) option="markers"
                 grid= {X Y}
                 label={"My X Value" "My Y Value"}
                 xvalues = -4:4
                 yvalues = do(-2,2,0.5);
```

Figure 25.409 Marker and Axis Attributes



SETDIF Function

SETDIF(*A, B***);**

The SETDIF function returns as a row vector the sorted set (without duplicates) of all element values present in *A* but not in *B*. If the resulting set is empty, the SETDIF function returns an empty matrix with zero rows and zero columns.

The arguments to the SETDIF function are as follows:

A	is a reference matrix. It can be either numeric or character.
B	is the comparison matrix. It must be the same type (numeric or character) as *A*.

For character matrices, the element length of the result is the same as the element length of the *A*. Shorter elements in the second argument are padded on the right with blanks for comparison purposes.

The following statements produce the matrix **C**, which contains the elements of **A** that are not contained in **B**:

```
A = {1 2 4 5};
B = {3 4};
C = setdif(A, B);
print C;
```

Figure 25.410 Difference of Sets

C
1 2 5

SETIN Statement

SETIN *SAS-data-set* <**NOBS** *name*> <**POINT** *value*> **;**

The SETIN data set makes a data set the current input data set.

The arguments to the SETIN statement are as follows:

SAS-data-set can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). You can also specify an expression (enclosed in parentheses) that resolves to the name of a SAS data set. See the example for the CLOSE statement.

name is the name of a variable to contain the number of observations in the data set. The NOBS option is optional.

value specifies the current observation. If the POINT option is not specified, the current observation does not change.

The SETIN statement chooses the specified data set from among the data sets that are open for input by the EDIT or USE statement. (The SHOW DATASETS command lists these data sets.) This data set becomes the current input data set for subsequent data management statements.

If specified, the NOBS option returns the number of observations in the data set in the scalar variable *name*. The POINT option points the data set to a particular observation and makes it the current observation.

In the following example, the data set WORK.A has 20 observations. The SETIN statement sets the variable SIZE to 20 and sets the current observation to 10.

```
proc iml;
x = T(1:20);
create A var {x}; append; close A;

use A;
setin A nobs size point 10;
list;                /* lists observation 10 */
```

Figure 25.411 Result of SETIN Statement

OBS	X
10	10.0000

SETOUT Statement

SETOUT *SAS-data-set* <**NOBS** *name*> <**POINT** *value*> **;**

The SETOUT data set makes a data set the current output data set.

The arguments to the SETOUT statement are as follows:

SAS-data-set can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). You can also specify an expression (enclosed in parentheses) that resolves to the name of a SAS data set. See the example for the CLOSE statement.

name specifies the name of a variable to contain the number of observations in the data set. The NOBS option is optional.

value specifies the observation to be made the current observation. If the POINT option is not specified, the current observation does not change.

The SETOUT statement chooses the specified data set from among those data sets that are already opened for output by the EDIT or CREATE statement. (The SHOW DATASETS command lists these data sets.) 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 following example, the data set WORK.A has 20 observations. The SETOUT statement sets the variable SIZE to 20 and sets the current observation to 5.

```
proc iml;
x = T(1:20);
create A var {x}; append;
setout A nobs size point 5;
list;                /* lists observation 10 */
```

Figure 25.412 Result of SETOUT Statement

```
   OBS          X

------ ---------

     5    5.0000
```

SHAPE Function

SHAPE(*matrix, nrow* < *, ncol* > < *, pad-value* > **);**

The SHAPE function reshapes and repeats values in a matrix.

The arguments to the SHAPE function are as follows:

matrix	is a numeric or character matrix or literal.
nrow	specifies the number of rows for the new matrix.
ncol	specifies the number of columns for the new matrix.
pad-value	specifies a value to use for elements of the new matrix if the quantity *nrow*×*ncol* is greater than the number of elements in *matrix*.

The SHAPE function creates a new matrix from data in *matrix*. The values *nrow* and *ncol* specify the number of rows and columns, respectively, in the new matrix. The function can reshape both numeric and character matrices.

There are three ways of using the function:

- If only *nrow* is specified, the number of columns is determined as the number of elements in the object matrix divided by *nrow*. The number of elements must be exactly divisible; otherwise, a conformability error occurs.
- If both *nrow* and *ncol* are specified, but not *pad-value*, the result is achieved by 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 first copies the elements of *matrix* into the result. If the number of elements in the result matrix is larger than the number of elements in *matrix*, the *pad-value* value is used for the remaining elements.

If *nrow* or *ncol* is specified as 0, then the number of rows or columns, respectively, becomes the number of values divided by *ncol* or *nrow*.

For example, the following statements create constant matrices of a given size:

```
r = shape(12, 3, 4);        /* 3 x 4 matrix with constant value 12 */
s = shape({99 31}, 3, 3);   /* 3 x 3 matrix with alternating values */
print r, s;
```

Figure 25.413 Constant and Repeated Matrices

r			
12	12	12	12
12	12	12	12
12	12	12	12

s		
99	31	99
31	99	31
99	31	99

The SHAPE function produces the result matrix by traversing the argument matrix in row-major order until the specified number of elements is reached. If necessary, the SHAPE function reuses elements.

You can also use the SHAPE function to reshape an existing matrix, as shown in the following statements:

```
t = shape(1:6, 2);
print t;
```

Figure 25.414 Reshaped Matrix

t	
1	2	3
4	5	6

SHAPECOL Function

SHAPECOL(*matrix, nrow* < , *ncol* > < , *pad-value* > **);**

The SHAPECOL function reshapes and repeats values in a matrix. It is similar to the SHAPE function except that the SHAPECOL function produces the result matrix by traversing the argument matrix in column-major order.

The following statements demonstrate the SHAPECOL function:

```
A = {1 2 3, 4 5 6};
c = shapecol(A, 3);
v = shapecol(A, 0, 1);
print c v;
```

Figure 25.415 Reshaped Matrices

c		v
1	5	1
4	3	4
2	6	2
		5
		3
		6

The vector **v** in the example is called the "vec of **A**" and is written vec(**A**). Uses of the vec operator in matrix algebra are described in Harville (1997). One important property is the relationship between the vec operator and the direct product operator (Kronecker product operator). If **A**, **B**, and **X** have the appropriate dimensions, then

$$\text{vec}(\mathbf{AXB}) = (\mathbf{B}' \otimes \mathbf{A})\text{vec}(\mathbf{X})$$

There is also a relationship between the SHAPECOL function and the SHAPE function. If **A** is a matrix, then the following two computations are equivalent:

```
b = shapecol(A, m, n, padVal);
c = T(shape(A`, n, m, padVal));
```

See the VECH function for a similar function that is useful for computing with symmetric matrices.

SHOW Statement

SHOW *operands* **;**

The SHOW statement displays system information. The following *operands* are available:

ALL	displays all the information included by the OPTIONS, SPACE, DATASETS, FILES, and MODULES options.
ALLNAMES	behaves like NAMES, but also displays names without values.
CONTENTS	displays the names and attributes of the variables in the current SAS data set.
DATASETS	displays all open SAS data sets.
FILES	displays all open files.
MEMORY	returns the size of the largest chunk of main memory available.
MODULES	displays all modules that exist in the current PROC IML environment. A module already referenced but not yet defined is listed as undefined.
name	displays 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 attributes of the matrix are shown.
NAMES	displays attributes of all matrices having values. Attributes include number of rows, number of columns, data type, and size.

OPTIONS	displays current settings of all PROC IML options (see the RESET statement).
PAUSE	displays the status of all paused modules that are waiting to resume.
SPACE	displays the workspace and symbolspace size and their current usage.
STORAGE	displays the modules and matrices in the current PROC IML library storage.

An example of a valid SHOW statement follows:

```
a = {1 2, 3 4};
b = 1:5;
free c;
start MyMod(x);
   return(2*x);
finish;
create Temp;

show modules allnames datasets memory;
```

Figure 25.416 System Information

```
Modules:

  MYMOD


 SYMBOL   ROWS   COLS TYPE   SIZE

 ------ ------ ------ ---- ------

 a           2      2 num       8

 b           1      5 num       8

 c           0      0 ?         0

  Number of symbols = 3  (includes those without values)


LIBNAME  MEMNAME                         OPEN MODE   STATUS

-------- -------------------------------- ---------   --------

WORK     TEMP                            Update      Current Input/Output


Memory Usage (in bytes):
```

SKEWNESS Function

SKEWNESS(*x*);

The SKEWNESS function is part of the IMLMLIB library. The SKEWNESS function returns the sample skewness for each column of a matrix. The sample skewness measures the asymmetry of a data distribution. Observations that are symmetrically distributed should have a skewness near 0.

The SKEWNESS function returns the same sample skewness as the UNIVARIATE procedure. For a formula, see the section "Descriptive Statistics" in the chapter "The UNIVARIATE Procedure" in *Base SAS Procedures Guide: Statistical Procedures*.

The following example computes the skewness for each column of a matrix:

```
x = {1 0,
     2 1,
     4 2,
     8 3,
    16 . };
skew = skewness(x);
print skew;
```

Figure 25.417 Sample Skewness of Two Columns

skew	
1.3253147	0

SOLVE Function

SOLVE(*A, B*);

The SOLVE function solves a system of linear equations.

The arguments to the SOLVE function are as follows:

A is an $n \times n$ nonsingular matrix.

B is an $n \times p$ matrix.

The SOLVE function solves the set of linear equations $\mathbf{AX} = \mathbf{B}$ for $\mathbf{X}$. The matrix $\mathbf{A}$ must be square and nonsingular.

The expression `X = SOLVE(A,B)` is mathematically equivalent to using the INV function in the expression `X = INV(A)*B`. However, the SOLVE function is recommended over the INV function because it is more efficient and more accurate.

The following example uses the SOLVE function:

```
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};

/* solve linear system */
x = solve(A,b);
print x;
```

Figure 25.418 Solving a Linear System

x
3
1
4
1
5

The solution method that is used is discussed in Forsythe, Malcom, and Moler (1967). The SOLVE function uses a criterion to determine whether the input matrix is singular. See the INV function for details.

If **A** is an $n \times n$ matrix, the SOLVE function temporarily allocates an n^2 array in addition to the memory allocated for the return matrix.

SOLVELIN Call

CALL SOLVELIN(*x, status, A, b, method***);**

The SOLVELIN subroutine uses direct decomposition to solve sparse symmetric linear systems.

The SOLVELIN subroutine returns the following values:

x	is the solution to $Ax = b$.
status	is the final status of the solution.

The input arguments to the SOLVELIN subroutine are as follows:

A	is the sparse coefficient matrix in the equation $Ax = b$. You can use SPARSE function to convert a matrix from dense to sparse storage.
b	is the right side of the equation $Ax = b$.
method	is the name of the decomposition to be used.

The input matrix A represents the coefficient matrix in sparse format; it is an n by 3 matrix, where n is the number of nonzero elements. The first column contains the nonzero values, while the second and third columns contain the row and column locations for the nonzero elements, respectively. Since A is assumed to be symmetric, only the elements on and below the diagonal should be specified, and it is an error to specify elements above the diagonal.

The solution to the system is returned in x. Your program should also check the returned *status* to make sure that a solution was found.

status = 0 indicates success.

status = 1 indicates the matrix A is not positive-definite.

status = 2 indicates the system ran out of memory.

If the SOLVELIN subroutine is unable to solve your system, you can try the iterative method ITSOLVER subroutine.

Two different factorization methods are available from the call, Cholesky and Symbolic LDL, specified as 'CHOL' or 'LDL' with the *method* parameter. Both these factorizations are applicable only to positive-definite symmetric systems; if your system is not positive-definite or not symmetric, you can use an ITSOLVER call.

The following example uses SOLVELIN to solve the system:

$$\begin{bmatrix} 3 & 1.1 & 0 & 0 \\ 1.1 & 4 & 1 & 3.2 \\ 0 & 1 & 10 & 0 \\ 0 & 3.2 & 0 & 3 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

```
/* value     row column */
A = {  3        1      1,
       1.1      2      1,
       4        2      2,
       1        3      2,
       3.2      4      2,
      10        3      3,
       3        4      4};

/* right hand side */
b = {1, 1, 1, 1};

call solvelin(x, status, A, b, 'LDL');
print status x;
```

Figure 25.419 Solving a Sparse Linear System

status	x
0	2.68
	-6.4
	0.74
	7.16

SORT Call

CALL SORT(*matrix* < , *by* < , *descend* > > **);**

The SORT subroutine sorts a matrix by the values of one or more columns.

The arguments to the SORT subroutine are as follows:

matrix is the input matrix. It is sorted in place by the call. If you want to preserve the original order of the data, make a copy of *matrix*.

by specifies the columns used to sort the matrix. The argument *by* is either a numeric matrix that contains column numbers, or a character matrix that contains the names of columns assigned to *matrix* by a MATTRIB statement or READ statement. If *by* is not specified, then the first column is used.

descend specifies which columns, if any, should be sorted in descending order. Any *by* columns not specified as descending will be ascending. If *descend* = *by*, then all *by* columns will be descending; if *descend* is skipped or is a null matrix, then all *by* columns will be ascending.

The SORT subroutine is used to sort a matrix according to the values in the columns specified by the *by* and *descend* arguments. Because the sort is done in place, very little additional memory space is required. The SORT subroutine is not as fast as the SORTNDX call for matrices with a large number of rows. After a matrix has been sorted, the unique combinations of values in the *by* columns can be obtained from the UNIQUEBY function.

For example, the following statements sort a matrix:

```
m = { 1 1 0,
      2 2 0,
      1 1 1,
      2 2 2};
call sort(m, {1 3}, 3); /* ascending by col 1; descending by col 3 */
print m;
```

Figure 25.420 Sorted Matrix

m		
1	1	1
1	1	0
2	2	2
2	2	0

SORT Statement

SORT < **DATA**=*SAS-data-set*> < **OUT**=*SAS-data-set*> **BY** < *variables*> < **DESCENDING** *variables*> **;**

The SORT statement sorts a SAS data set. You can use the following clauses with the SORT statement:

DATA=*SAS-data-set* names the SAS data set to be sorted. It can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). You can also specify an expression (enclosed in parentheses) that resolves to the name of a SAS data set. (See the example for the CLOSE statement.) 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 replaced by the sorted version.

BY *variables* specifies the variables to be sorted. A BY clause *must* be used with the SORT statement.

DESCENDING specifies the variables are to be sorted in descending order.

The SORT statement sorts the observations in a SAS data set by one or more variables, stores the resulting sorted observations in a new SAS data set, or replaces the original.

In contrast with other data processing statements, it is *mandatory* that the data set to be sorted be closed prior to the execution of the SORT statement. The SORT statement gives an error if you try to sort a data set that is open.

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 retains the same relative positions of the observations with identical BY variable values.

For example, the following statement sorts data from the Sashelp.Class data set by the variables Age and Height, where Age is sorted in descending order, and all observations with the same Age value are sorted by Height in ascending order:

```
sort Sashelp.Class out=sortClass by descending age height;
```

The output data set sortClass 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.

Notice that all the clauses of the SORT statement must be specified in the order given in the syntax.

SORTNDX Call

CALL SORTNDX(*index, matrix* < *, by*> < *, descend*> **);**

The SORTNDX subroutine creates an index to reorder a matrix by specified columns.

The SORTNDX subroutine returns the following value:

index is a vector such that *index*[*i*] is the row index of the *i*th element of *matrix* when sorted according to *by* and *descend*. Consequently, *matrix*[*index*,] is the sorted matrix.

The arguments to the SORTNDX subroutine are as follows:

matrix is the input matrix, which is not modified by the call.

by specifies the columns used to sort the matrix. The argument *by* is either a numeric matrix that contains column numbers, or a character matrix that contains the names of columns assigned to *matrix* by a MATTRIB statement or READ statement. If *by* is not specified, then the first column is used.

descend specifies which columns, if any, should be sorted in descending order. Any *by* columns not specified as descending will be ascending. If *descend* = *by*, then all *by* columns will be descending; if *descend* is skipped or is a null matrix, then all *by* columns will be ascending.

The SORTNDX subroutine can be used to process the rows of a matrix in a sorted order, without having to actually modify the matrix.

For example, the following statements return a vector that specifies the order of the rows in a matrix:

```
m = { 1 1 0,
      2 0 0,
      1 3 1,
      2 2 2 };
call sortndx(ndx, m, {1 3}, 3);
print ndx;
```

Figure 25.421 Sort Index

ndx
3
1
4
2

The output is shown in Figure 25.421. The SORTNDX subroutine returns the vector **ndx** that indicates how rows of **m** will appear if you sort **m** in ascending order by column 1 and in descending order by column 3. The values of the vector **ndx** indicate that row 3 of **m** will be the first row in the sorted matrix. Row 1 of **m** will become the second row. Row 4 will become the third row, and row 2 will become the last row.

The matrix can be physically sorted as follows:

```
sorted = m[ndx,];
```

The SORTNDX subroutine can be used with the UNIQUEBY function to extract the unique combinations of values in the *by* columns. If you want to replace a matrix with a sorted copy of itself, use the SORT call.

SOUND Call

CALL SOUND(*freq* < , *dur*> **);**

The SOUND subroutine generates a tone with a frequency (in hertz) given by the *freq* parameter and a duration (in seconds) given by the *dur* parameter.

The arguments to the SOUND subroutine are as follows:

freq is a numeric matrix or literal that contains the frequency in hertz.

dur is a numeric matrix or literal that contains the duration in seconds. Note that the *dur* argument differs from that in the DATA step.

Matrices can be specified for frequency and duration to produce multiple tones, but if both arguments are nonscalar, then the number of elements must match. The duration argument is optional and defaults to 0.25 (one quarter second).

For example, the following statements produce tones from an ascending musical scale, all with a duration of 0.2 seconds:

```
notes = 400#(2##do(0, 1, 1/12));
call sound(notes, 0.2);
```

SPARSE Function

SPARSE(*x* < , *type*> **);**

The SPARSE function converts an $n \times p$ matrix that contains many zeros into a matrix stored in a sparse format which suitable for use with the ITSOLVER call or the SOLVELIN call.

The arguments to the SPARSE function are as follows:

x specifies an $n \times p$ numerical matrix. Typically, *x* contains many zeros and only k nonzeros, where k is much smaller than np.

type specifies whether the *x* matrix is symmetric. The following values are valid:

- "symmetric" specifies that only the lower triangular nonzero values of the *x* matrix are used.
- "unsymmetric" specifies that all nonzero values of the *x* matrix are used. This is the default value.

The *type* argument is not case-sensitive. The first three characters are used to determine the value. For example, "SYM" and "symmetric" specify the same option.

The matrix returned by the SPARSE function is a $k \times 3$ matrix that contains the following values:

- The first column contains the nonzero values of the *x* matrix.
- The second column contains the row numbers for each value.
- The third column contains the column numbers for each value.

For example, the following statements compute a sparse representation of a dense matrix with many zeros:

```
x = {3   1.1  0   0  ,
     1.1 4    0   3.2,
     0   1   10   0  ,
     0   3.2  0   3  };
a = sparse(x, "sym");
print a[colname={"Value" "Row" "Col"}];
```

Figure 25.422 Sparse Data Representation

a		
Value	Row	Col
3	1	1
1.1	2	1
4	2	2
1	3	2
10	3	3
3.2	4	2
3	4	4

SPECTROGRAM Call

CALL SPECTROGRAM(*x***) WINDOW=***Window***
OVERLAP=***R***
FFTLEN=***K***
TITLE=***Title***
COLORRAMP=***ColorRamp***
SAMPRATE=***f***
TRUNCATE=***alpha***
SCALE='LINEAR' | 'LOG' | 'DB'
PANEL=0 | 1 ;**

The SPECTROGRAM subroutine displays a heat map that shows the power (squared magnitude) of the short-time Fourier transform (STFT) of the signal for various bins in the time and frequency domain. Internally, the STFT is computed by using the TFSTFT function. For more information about the STFT and spectrograms, see Rabiner and Schafer (2007).

The first four arguments of the SPECTROGRAM subroutine are the same as for the TFSTFT function. The first argument, *x*, is a required column vector of size N that contains the time series signal. The signal is assumed to be sampled at N evenly spaced time intervals. The remaining arguments are optional and can be specified in any order.

If the SAMPRATE= argument is not specified, the sampling frequency is assumed to be 1 Hz. So by default the time points are assumed to be $t = 0, 1, \ldots, N-1$ seconds. If you specify a value f for the SAMPRATE= argument, then the signal is assumed to be sampled f times per unit of time. For example, if the signal is sampled at 400 Hz (that is, 400 times per second), then $f = 400$ and the time points are evenly spaced values in $[0, (N-1)/f]$.

The following optional arguments are supported:

WINDOW=*Window*

specifies the windowing function for the STFT. If *Window* is a positive integer, L, then a Hanning window of length L is used. If *Window* is a vector that contains L values, such as is produced by the TFWINDOW function, then the values specify the shape of the windowing function.

The relative length of the windowing function determines the resolution of the STFT. A relatively wide window provides good resolution in the frequency domain at the expense of poorer resolution in the time domain. In contrast, a narrow window provides good resolution in the time domain but poorer resolution in the frequency domain.

By default, L is the lesser of N and 256. See also the *window* argument for the TFSTFT function.

OVERLAP=*R*

specifies the amount of overlap between consecutive windows. This value must be less than L (which is specified by the WINDOW= argument).

In practice, many analysts use an overlap value that is 50% of the window length (Harris 1978; Trethewey 2000). Heinzel, Rüdiger, and Schilling (2002) provide a table of recommended overlaps for many common window shapes. The recommended overlap for many nonrectangular windows is between 50% and 75%.

By default, $R = \lfloor L/2 \rfloor$. See also the *overlap* argument for the TFSTFT function.

FFTLEN=*K*

specifies a positive integer to be used to calculate the K-point fast Fourier transform (FFT) of the windowed time series data. By default, *K* is chosen to be the greater of L and 256. See also the *fftlen* argument for the TFSTFT function.

TITLE=*Title*

specifies a character string to use as the title for the spectrogram plot. By default, the global TITLE1 value is used. If the TITLE1 value is null, then the spectrogram does not include a title.

COLORRAMP=*Colorramp*

specifies a color ramp that assigns colors to cells in the spectrogram. For more information about how to specify a color ramp, see the HEATMAPCONT call. By default, the color ramp is the "TwoColorRamp" in the current ODS style.

SAMPRATE=*f*

specifies the sampling frequency in samples per second, where *f* must be positive. By default, *f*=1 Hz.

TRUNCATE=*alpha*

specifies a truncation value, which must be in the interval $[0, 1)$. If a is the minimum value in the spectrogram and b is the maximum value, then values that are less than $a + \textit{alpha} * (b - a)$ are truncated to a. This can be useful when the energy is a signal span many orders of magnitude. By default, *alpha*=0.

SCALE='LINEAR' | 'LOG' | 'DB'

specifies whether the spectrogram plots the power, the logarithm of the power, or the power in decibels. You can specify the following values:

'LINEAR' requests that the spectrogram display the power, which is the squared magnitude of the STFT.

'LOG' requests that the spectrogram display $\log_{10}(P)$, where P is the power.

'DB' requests that the spectrogram display the power in decibels, which is the quantity $10\log_{10}(P)$, where P is the power.

By default, SCALE='LINEAR'.

PANEL=0 | 1

specifies whether to plot the spectrogram by itself or as part of a panel of graphs. You can specify the following values:

0 displays only the spectrogram and a color ramp.

1 displays three graphs in a panel:

- The spectrogram, which has axes for frequency and time, and a color ramp.
- Above the spectrogram is a plot of the original signal. The time axis of the signal is parallel to the time axis of the spectrogram.
- To the left of the spectrogram is a plot of the average power in each frequency bin.

By default, PANEL=0.

The relationship between the arguments to the spectrogram must satisfy $R < L \leq K$ and $L \leq N$, where R is the overlap length, L is the window length, K is the FFT length, and N is the signal length.

The spectrogram is one way to visualize a signal in time and frequency domains simultaneously. The following DATA step constructs a signal that consists of different frequencies. The signal is sampled 100 times per second. For the first second, the frequency of the signal is 5 Hz. For the second second, the frequency of the signal is 10 Hz. The frequency is 20 Hz and 40 Hz for the third and fourth seconds, respectively.

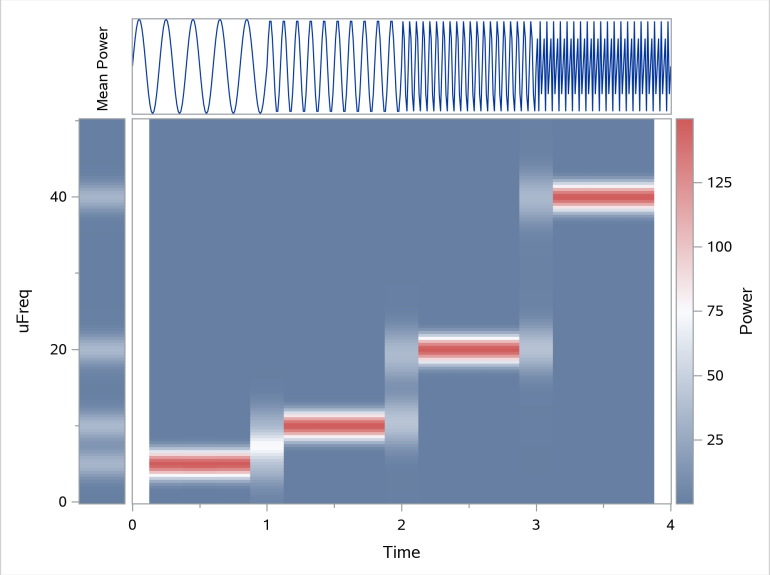
```
%let SampRate=100;
data Signal;
keep t x;
pi = constant("pi");
/* sample at SampRate Hz */
do t = 0 to 4 by 1/&SampRate;
   if t<1 then x = sin(2*pi*5*t);
   else if t<2 then x = sin(2*pi*10*t);
   else if t<3 then x = sin(2*pi*20*t);
   else if t<4 then x = sin(2*pi*40*t);
   output;
end;
run;
```

The power spectrum of the FFT of the signal shows peaks at frequencies 5, 10, 20, and 40. However, the FFT does not provide information when these frequencies occur. In contrast, a spectrogram indicates the frequencies in a signal and the time intervals during which those frequencies are dominant.

The following SAS/IML statements read the signal and create a spectrogram, which is shown in Figure 25.423. The SPECTROGRAM call specifies a window of length 50, which corresponds to 500 ms of this signal. By default, a Hanning window is used. Also by default, the overlap is 50% of the window size, which is a length of 25 time units.

```
proc iml;
use Signal;  read all var {"t" "x"};  close;
winLen = (500 / 1000) * &SampRate;    /* 500 ms = 50 time units */
call spectrogram(x, winLen) panel=1
                sampRate=&SampRate colorramp="ThreeColor";
```

Figure 25.423 Spectrogram, 500 ms Window



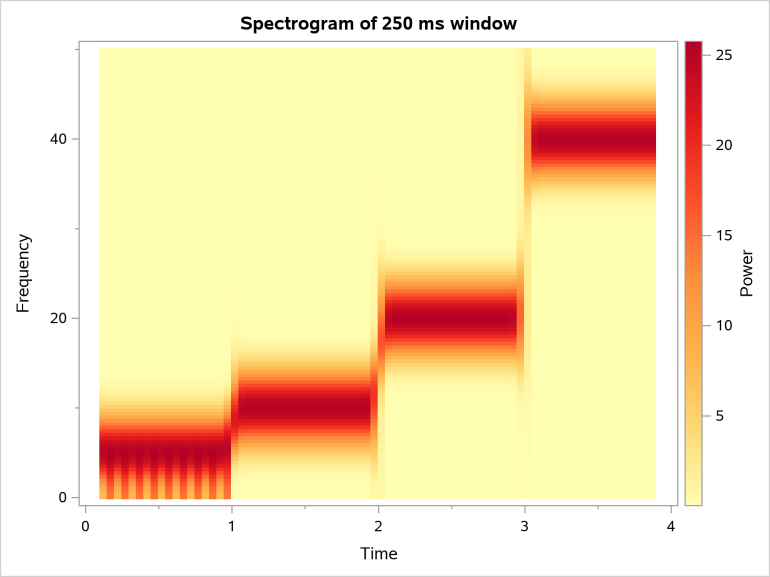
The SPECTROGRAM call displays a panel of graphs. The panel contains four graphical components:

- At the top of the panel is a plot of the time series. You can see that the signal consists of four sinusoidal curves and that the changes of frequency occur at $t = 1, 2$, and 3 seconds.
- Directly below the signal is the spectrogram, which is a heat map of the power in the time-frequency domain. The spectrogram shows four red horizontal line segments, which correspond to the maximum power of the signal. From the vertical axis, the maximum power appears to be located near the frequencies 5, 10, 20, and 40 on the vertical axis. The power appears to transition between frequencies near $t = 1, 2$, and 3 seconds, but the spectrogram is blurry near the transition times. A shorter window would improve the resolution of the graph in time, but at the cost of worsening the resolution for frequency.
- The legend on the far right shows the color ramp that encodes the power values. The range of the power appears to be in the interval [0, 150]. The ThreeColorRamp color ramp for the ODS style in this documentation is a blue-white-red blend.
- The heat map on the far left indicates the average power for each frequency. The average is taken over time and indicates that the average power of this signal is greatest near the frequencies 5, 10, 20, and 40.

You can specify options that control the appearance of the spectrogram. The following statements specify a Blackman window that is 25 time units wide (250 ms). The overlap parameter is set to 80% of the window width, or 20 time units. The COLORRAMP= option calls the PALETTE function to obtain a yellow-orange-red color ramp. Only the spectrogram is shown because the PANEL=1 option is not specified. The spectrogram displays the title that was set by using the TITLE statement. The resulting spectrogram is shown in Figure 25.424.

```
title "Spectrogram of 250 ms window";
winLen = (250 / 1000) * &SampRate;
window = tfwindow(winLen, "Blackman");
call spectrogram(x, window, int(winLen*0.8))
                sampRate=&SampRate colorramp=palette("YLORRD",7);
```

Figure 25.424 Spectrogram, 250 ms Window

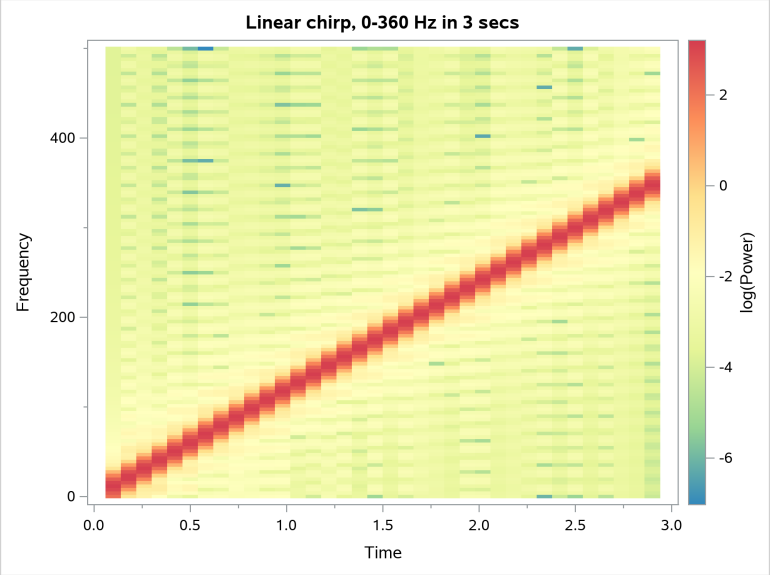


The spectrogram in Figure 25.424 shows features that are similar to Figure 25.423, but Figure 25.424 provides a finer resolution in time because of the narrower window and the increased overlap parameter. This enables you to better estimate the times at which the signal changes frequencies.

The next example is a spectrogram for a linear chirp signal. A linear *chirp* is a sinusoidal wave whose frequency increases at a linear rate with time. The signal for the following example is $f(t) = \sin(120\pi t^2)$ for t in the interval [0, 3]. During that time period, the frequency of the signal increases at a linear rate from 0 to 360 Hz. The following statements define the signal and create a spectrogram, which is shown in Figure 25.425:

```
/* Linear chirp on interval [0,3] */
t = T( do(0, 3, 0.001) );     /* sampRate = 1000 Hz */
pi = constant('pi');
w = pi*120*t;                 /* frequency linear in time */
x = sin(w#t);
title "Linear chirp, 0-360 Hz in 3 secs";
winLen = 200;
window = tfwindow(winLen, "Gaussian");
noverlap = floor(0.6*winLen);
nfft = 256;
ramp = palette("Spectral",7)[7:1]; /* reverse "spectral" color ramp */
call spectrogram(x,window,noverlap,nfft) scale="log"
                 colorramp=ramp sampRate=1000;
```

Figure 25.425 Spectrogram of Linear Chirp



The spectrogram displays the power on a logarithmic (base 10) scale, so that the red cells indicate a power of approximately $10^2 = 100$ whereas the blue cells indicate a power of approximately 10^{-6}. You can see that the power increases linearly in time, from zero frequency at $t = 0$ to a frequency of 360 Hz at $t = 3$.

A few remarks about the spectrogram:

- If the time series contains N points, then an accurate graph of the signal requires at least N pixels in the horizontal direction. Because typical signals contains thousands of points, a plot of the time series might not be practical. Consequently, the SPECTROGRAM call displays only the spectrogram by default. For a short time series, you can specify the PANEL=1 option to display the time series above the spectrogram.

- The NXYBINSMAX= option in the ODS GRAPHICS statement determines the maximum number of cells that are displayed in a heat map. By default, NXYBINSMAX=100,000. For long signals, the SPECTROGRAM call might produce an empty plot, in which case the SAS log displays the following:

  WARNING: The number of bins in HEATMAPPARM statement is *x*. It exceeds the NXYBINSMAX threshold in the ODS GRAPHICS statement. The plot will not be drawn.

  To display the graph, submit the statement **`ODS GRAPHICS / NXYBINSMAX=`*`n`***, where ***`n`*** is larger than the number of bins that is displayed in the log.

- The SPECTROGRAM subroutine creates the data set WORK._STFT. The data set contains the time signal and the values from the short-time Fourier transform. The WORK._STFT data set contains the following variables:

Signal	The time series values x.
n	The evenly spaced time points. The signal can be graphed by plotting the pairs (n, x).
Power	The power (squared magnitude) of the short-time Fourier transform (STFT) of the signal for each time-frequency bin in the spectrogram.

logPower	The base-10 logarithm of the Power variable.
dBPower	The logPower variable multiplied by 10.
Freq	The frequency value of a cell in the spectrogram.
Time	The time value of a cell in the spectrogram.
Z	The response value for a bin in the spectrogram. This is equal to either Power, logPower, or dBPower.
uFreq	The unique values of the Freq variable. These are the values of each row in the spectrogram. These values are the vertical positions of the cells in the leftmost heat map in a spectrogram panel such as Figure 25.423.
EnergyPower	The mean values of the rows of the spectrogram for each value of the Frequency variable. These values are represented by the colors in the leftmost heat map in a spectrogram panel.
x0 and x1	Constants for the ODS template that defines the spectrogram panel. These values define the left and right edges of the leftmost heat map in a spectrogram panel.

You can use the WORK._STFT data to modify the spectrogram. For example, the following call to PROC SGPLOT uses the ThreeColorRamp color ramp to display a spectrogram of the most recently created signal. You can use the WORK._STFT data set to create additional graphs without recomputing the STFT.

```
title "Heat Map from Spectrogram Data";
proc sgplot data=_STFT;
heatmapparm x=Time y=Freq colorresponse=Z /
            colormodel=ThreeColorRamp;
run;
```

SPLINE and SPLINEC Calls

CALL SPLINE(*fitted, data* **<** *, smooth* **>** **<** *, delta* **>** **<** *, nout* **>** **<** *, type* **>** **<** *, slope* **>** **);**

CALL SPLINEC(*fitted, coeff, endSlopes, data* **<** *, smooth* **>** **<** *, delta* **>** **<** *, nout* **>** **<** *, type* **>** **<** *, slope* **>** **);**

The SPLINE and SPLINEC subroutines fit cubic splines to data. The SPLINE subroutine is the same as SPLINEC but does not return the matrix of spline coefficients needed to call SPLINEV, nor does it return the slopes at the endpoints of the curve.

The SPLINEC subroutine returns the following values:

fitted	is an $n \times 2$ matrix of fitted values.
coeff	is an $n \times 5$ (or $n \times 9$) matrix of spline coefficients. The matrix contains the cubic polynomial coefficients for the spline for each interval. Column 1 is the left endpoint of the x-interval for the regular (nonparametric) spline or the left endpoint of the parameter for the parametric spline. Columns $2-5$ are the constant, linear, quadratic, and cubic coefficients, respectively, for the x-component. If a parametric spline is used, then columns $6-9$ are the constant, linear, quadratic, and cubic coefficients, respectively, for the y-component. The coefficients for each interval are with respect to the variable $x - x_i$ where x_i is the left endpoint of the interval and x is the point of interest. The matrix *coeff* can be processed to yield the integral or the

derivative of the spline. This, in turn, can be used with the SPLINEV function to evaluate the resulting curves. The SPLINEC subroutine returns *coeff*.

endSlopes is a 1×2 matrix that contains the slopes of the two ends of the curve expressed as angles in degrees. The SPLINEC subroutine returns the *endSlopes* argument.

The input arguments to the SPLINEC subroutine are as follows:

data specifies a $n \times 2$ (or $n \times 3$) matrix of (x, y) points on which the spline is to be fit. The optional third column is used to specify a weight for each data point. If *smooth* > 0, the weight column is used in calculations. A weight ≤ 0 causes the data point to be ignored in calculations.

smooth is an optional scalar that specifies the degree of smoothing to be used. If *smooth* is omitted or set equal to 0, then a cubic interpolating spline is fit to the data. If *smooth* > 0, then a cubic spline is used. Larger values of *smooth* generate more smoothing.

delta is an optional scalar that specifies the resolution constant. If *delta* is specified, the fitted points are spaced by the amount *delta* on the scale of the first column of *data* if a regular spline is used or on the scale of the curve length if a parametric spline is used. If both *nout* and *delta* are specified, *nout* is used and *delta* is ignored.

nout is an optional scalar that specifies the number of fitted points to be computed. The default is *nout*=200. If *nout* is specified, then *nout* equally spaced points are returned. The *nout* argument overrides the *delta* argument.

type is an optional 1×1 (or 1×2) character matrix or quoted literal that contains the type of spline to be used. The first element of *type* should be one of the following:

- "periodic", which requests periodic endpoints
- "zero", which sets second derivatives at endpoints to 0

The *type* argument controls the endpoint constraints unless the *slope* argument is specified. If "periodic" is specified, the response values at the beginning and end of column 2 of *data* must be the same unless the smoothing spline is being used. If the values are not the same, an error message is printed and no spline is fit. The default value is "zero". The second element of *type* should be one of the following.

- "nonparametric", which requests a nonparametric spline
- "parametric", which requests a parametric spline

If "parametric" is specified, a parameter sequence $\{t_i\}$ is formed as follows: $t_1 = 0$ and

$$t_i = t_{i-1} + \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Splines are then fit to both the first and second columns of *data*. The resulting values are paired to form the output. Changing the relative scaling of the first two columns of *data* changes the output because the sequence $\{t_i\}$ assumes Euclidean distance.

Note that if the points are not arranged in strictly ascending order by the first columns of *data*, then a parametric method must be used. An error message results if the nonparametric spline is requested.

slope is an optional 1×2 matrix of endpoint slopes given as angles in degrees. If a parametric spline is used, the angle values are used modulo 360. If a nonparametric spline is used, the tangent of the angles is used to set the slopes (that is, the effective angles range from -90 to 90 degrees).

See Stoer and Bulirsch (1980), Reinsch (1967), and Pizer (1975) for descriptions of the methods used to fit the spline. For simplicity, the following explanation assumes that the *data* matrix does not contain a weighting column.

Nonparametric splines can be used to fit data for which you believe there is a functional relationship between the X and Y variables. The unique values of X (stored in the first column of *data*) form a partition $\{a = x_1 < x_2 < \ldots < x_n = b\}$ of the interval $[a, b]$. You can use a spline to interpolate the data (produce a curve that passes through each data point) provided that there is a single Y value for each X value. The spline is created by constructing cubic polynomials on each subinterval $[x_i, x_{i+1}]$ so that the value of the cubic polynomials and their first two derivatives coincide at each x_i.

Interpolating Splines

An interpolating spline is not uniquely determined by the set of Y values. To achieve a unique interpolant, S, you must specify two constraints on the endpoints of the interval $[a, b]$. You can achieve uniqueness by specifying one of the following conditions:

- $S''(a) = 0, S''(b) = 0$. The second derivative at both endpoints is zero. This is the default condition, but can be explicitly set by using `type='zero'`.
- Periodic conditions. If your data are periodic so that x_1 can be identified with x_n, and if $y_1 = y_n$, then the interpolating spline already satisfies $S(a) = S(b)$. Setting `type='periodic'` further requires that $S'(a) = S'(b)$ and $S''(a) = S''(b)$.
- Fixed slopes at endpoints. Setting slope=$\{y'_1, y'_n\}$ requires that $S'(a) = y'_1$ and $S'(b) = y'_n$.

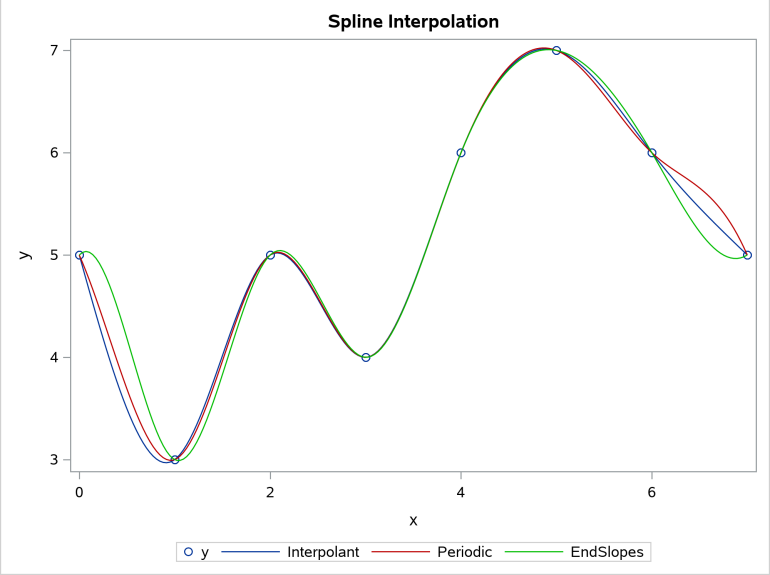
The following statements give three examples of computing an interpolating spline for data. Note that the first and last Y values are the same, so you can ask for a periodic spline.

```
proc iml;
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};

/* write data */
create SplineData from data[colname={"x" "y"}];
append from data;
close SplineData;
```

```
/* write fitted interpolants */
fit = fitted || periodicFitted[,2] || slopeFitted[,2];
varNames = {"t" "Interpolant" "Periodic" "EndSlopes"};
create SplineFit from fit[colname=varNames];
append from fit;
close SplineFit;
quit;

/* merge data and plot */
data Spline;
merge SplineData SplineFit;
run;

title "Spline Interpolation";
proc sgplot data=Spline;
   scatter x=x y=y;
   series x=t y=Interpolant;
   series x=t y=Periodic / lineattrs=(color="dark red");
   series x=t y=EndSlopes / lineattrs=(color="dark green");
run;
```

As shown in Figure 25.426, the interpolants pass through each point of the data. They differ from each other by the derivatives at the boundary points, $x = 0$ and $x = 7$. The generic interpolant has second derivatives that vanish at the boundary points. The periodic curve has a derivative at $x = 0$ that matches the derivative at $x = 7$. The third curve has derivatives that match the given slopes at the boundary points.

Figure 25.426 Three Spline Interpolants with Different Boundary Conditions



Smoothing Splines

You can also use a spline to smooth data. In general, a smoothing spline does not pass through any data pair exactly. A very small value of the *smooth* smoothing parameter approximates an interpolating polynomial for data in which each unique X value is assigned the mean of the Y values that correspond to that X value. As the *smooth* parameter gets very large, the spline approximates a linear regression.

The following statements compute two smoothing splines for the same data as in the previous example. The spline coefficients are passed to the SPLINEV function, which evaluates the smoothing spline at the original X values. The smoothing spline does not pass through the original Y values. Also, 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.

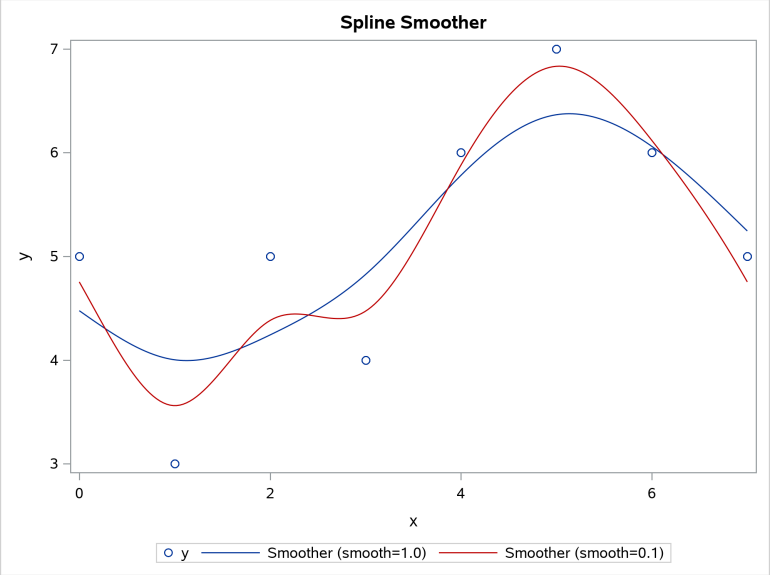
```
proc iml;
data = { 0 5, 1 3, 2 5, 3 4, 4 6, 5 7, 6 6, 7 5 };

/* 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(periodicFitted,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 */
print smoothFit smoothPeriodicFit;
```

Figure 25.427 Two Spline Smoothers

	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

You can write the smoothers to a SAS data set and merge them with the data, as shown in the previous example. Figure 25.428 shows the resulting graph.

Figure 25.428 Graph of Two Spline Smoothers



Parametric Splines

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 are shaped like an "S." The variable *fitted* is returned as a *numParam* $\times 2$ matrix that contains the ordered pairs that correspond to the parametric spline. These ordered pairs correspond to *numParam* evenly spaced points in the domain of the parameter t.

The purpose of the SPLINEV function is to evaluate (*score*) an interpolating or smoothing spline at an arbitrary set of points. The following program shows how to construct the parameters that correspond to the original data by using the formula specified in the documentation for the *type* argument. These parameters are used to construct the evenly spaced parameters that correspond to the data in the *fitted* matrix.

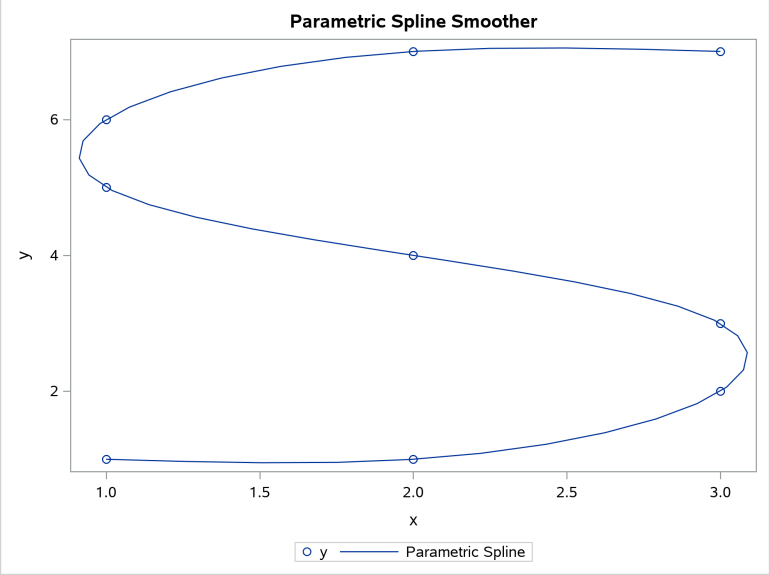
```
proc iml;
data = {3 7, 2 7, 1 6, 1 5, 2 4, 3 3, 3 2, 2 1, 1 1};

/* Compute parametric spline interpolant */
numParam = 40;
call splinec(fitted,coeff,endSlopes,data)
             nout=numParam type={"zero" "parametric"};

/* write data */
create SplineData from data[colname={"x" "y"}];
append from data;
close SplineData;

/* write parametric spline values */
create SplineFit from fitted[colname={"xt" "yt"}];
append from fitted;
close SplineFit;
```

```
/* Manually reproduce/verify the "fitted" values */
/* (1) Form the parameters mapped onto the 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;

/* (2) Construct numParam evenly spaced parameters
   between 0 and t[nrow(t)] */
params = do(0, t[nrow(t)], t[nrow(t)]/(numParam-1))`;

/* (3) Evaluate the parametric spline at these points */
fit = splinev(coeff, params);
maxDiff = max(abs(fitted-fit));
print maxDiff; /* should be very small or zero */
quit;

/* merge data and plot */
data Spline;
merge SplineData SplineFit;
run;

title "Parametric Spline Smoother";
proc sgplot data=Spline;
   scatter x=x y=y;
   series x=xt y=yt / legendlabel="Parametric Spline";
run;
```

Figure 25.429 Verification of Parametric Spline Properties

maxDiff
0

Figure 25.430 Parametric Smoothing Spline



The Domain of the Spline Functions

Attempting to evaluate a spline outside its domain of definition results in a missing value. For example, the following statements define a spline on the interval [0, 7]. Attempting to evaluate the spline at points outside this interval (−1 or 20) results in missing values.

```
proc iml;
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;
```

Figure 25.431 Extrapolation of a Spline

	v
-1	.
1	3
2	5
3	4
3.5	4.7073171
4	6
20	.

Integration of Spline Functions

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 points. In the trapezoid rule, the integral is estimated by the exact integral of a piecewise linear function that connects the data points. In Simpson's rule, the integral is estimated as the exact integral of a piecewise quadratic function between the data points.

Because 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 (**SpleinEval**) uses a helpful trick to evaluate a spline at a single point. If you pass in a scalar argument to the SPLINEV function, you get back a vector that represents the evaluation of the spline along evenly spaced points, rather than the spline evaluated at the argument. To avoid this, the following statements evaluate the spline at the vector **x // x** and then extract the entry in the first row, second column. This number is the value of the spline evaluated at *x*.

```
proc iml;
x = { 0, 2, 5, 7, 8, 10 };
y = x + 0.1*sin(x);
a = x || y;
call splinec(fit,coeff,endSlopes,a);

start SplineEval(x) global(coeff,power);
   /* The first column of v contains the points of evaluation;
      the second column contains the evaluation. */
   v = x##power # splinev(coeff, x//x);
```

```
   return(v[1,2]); /* return spline(x) */
finish;

/* Evaluate the "moment" of a function.
   moment(0) = integral of f(x) dx
   moment(1) = integral of x*f(x) dx
   moment(2) = integral of x##2 *f(x) dx, etc
   Use QUAD to integrate */
start moment(pow) global(coeff,power);
   power = pow;
   intervals = coeff[,1]; /* left endpts of x intervals */
   call quad(z,"SplineEval", intervals) eps = 1.e-10;
   return( sum(z) );
finish;

mass  = moment(0);   /* to compute the mass */
m     = mass //
       (moment(1)/mass) // /* to compute the mean */
       (moment(2)/mass) ;  /* to compute the meansquare */
print m;


/* Check the previous computation by using Gauss-Legendre
   integration, which is valid 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); /* = 6 */
maxng  =  2*ngauss-4;

start momentGL(pow) global(coeff,gauss,ngauss,maxng);
   if pow >= maxng  then return(.);
   nrow    = nrow(coeff);
   left    = coeff[1:nrow-1,1]; /* left endpt of interval */
   right   = coeff[2:nrow,1];   /* right endpt */
   mid     = 0.5*(left+right);
   interv  = 0.5*(right - left);
   /* scale the weights on each interval */
   wgts    = rowvec( interv*gauss[2,] );
   /* scale the points on each interval */
   pts     = rowvec( interv*gauss[1,] + mid );
   /* evaluate the function */
   eval    = splinev(coeff,pts)[,2]`;
   return( sum( wgts#pts##pow # eval) );
finish;

mass  = momentGL(0);       /* to compute the mass */
m2    = mass // (momentGL(1)/mass) //  (momentGL(2)/mass) ;
print m2; /* should agree with earlier result */
```

Figure 25.432 Integral and Other Moments of the Spline Function

m
50.204224
6.658133
49.953307

m2
50.204224
6.658133
49.953307

SPLINEV Function

SPLINEV(*coeff* < , *delta*> < , *nout*> **);**

The SPLINEV function evaluates a cubic spline at a set of points. The function returns a two-column matrix that contains the points of evaluation in the first column and the corresponding fitted values of the spline in the second column.

The arguments to the SPLINEV function are as follows:

coeff is an $n \times 5$ (or $n \times 9$) matrix of spline coefficients, as returned by the SPLINEC call. The *coeff* argument should not contain missing values.

delta is an optional vector that specifies evaluation points. If *delta* is a scalar, the spline is evaluated at equally spaced points *delta* apart. If *delta* is a vector arranged in ascending order, the spline is evaluated at each of these values. Evaluation at a point outside the support of the spline results in a missing value in the output. If you specify the *delta* argument, you cannot specify the *nout* argument.

nout is an optional scalar that specifies the desired number of fitted points. The default is *nout*=200. If you specify the *nout* argument, you cannot specify the *delta* argument.

If you want to evaluate a spline at a single point, you need to use a trick. The trick is to evaluate the spline at *two* points but use only the fitted value at the first point. For example, if **x** is a scalar value, then the following statement evaluates the spline fit at **x**:

```
y = splinev(coeff, x//x)[1,2]; /* first row is (x, spline(x)) */
```

See the section "SPLINE and SPLINEC Calls" on page 1013 for details and examples.

SPOT Function

SPOT(*times, forward_rates***);**

The SPOT function returns an $n \times 1$ vector of (per-period) spot rates, given vectors of forward rates and times.

The arguments to the SPOT function are as follows:

times	is an $n \times 1$ column vector of times in consistent units. Elements should be nonnegative.
forward_rates	is an $n \times 1$ column vector of corresponding (per-period) forward rates. Elements should be positive.

The SPOT function transforms the given spot rates as

$$s_1 = f_1$$

$$s_i = \left(\Pi_{j=1}^{j=i}(1 + f_j)^{t_j - t_{j-1}}\right)^{\frac{1}{t_i}} - 1; \quad i = 2, \ldots, n$$

where, by convention, $t_0 = 0$.

For example, the following statements produce the output shown in Figure 25.433:

```
time = T(do(1, 5, 1));
forward = T(do(0.05, 0.09, 0.01));
spot = spot(time, forward);
print spot;
```

Figure 25.433 Spot Rates

spot
0.05
0.0549882
0.0599686
0.0649413
0.0699065

SQRSYM Function

SQRSYM(*matrix***);**

The SQRSYM function takes a packed-symmetric matrix (such as generated by the SYMSQR function) and transforms it back into a dense square matrix.

The argument to the SQRSYM function is a symmetric matrix. The elements of the argument are unpacked (in row-major order) into the lower triangle of the result and reflected across the diagonal into the upper triangle. If you want the lower-triangular elements to be stacked in column-major order, use the VECH function.

For example, the following statements return a symmetric matrix:

```
v = T(1:6);
sqr = sqrsym(v);
print sqr;
```

Figure 25.434 Symmetric Matrix

sqr
1 2 4
2 3 5
4 5 6

The SQRSYM function and the SYMSQR function are inverse operations on the set of symmetric matrices. See also the SQRVECH function, which unpacks elements in column-major order.

SQRT Function

SQRT(*matrix***);**

The SQRT function returns the positive square roots of each element of the argument matrix.

An example of a valid statement follows:

```
a = {1 2 3 4};
c = sqrt(a);
print c;
```

Figure 25.435 Square Roots

c
1 1.4142136 1.7320508 2

SQRVECH Function

SQRVECH(*matrix***);**

The SQRVECH function transforms a packed-symmetric matrix into a dense square matrix.

The elements of the argument are unpacked (columnwise) into the lower triangle of the result and reflected across the diagonal into the upper triangle. The argument *matrix* should be a column-stacked, packed-symmetric matrix, such as generated by the VECH function.

For example, the following statements return a symmetric matrix:

```
v = T(1:6);
sqr = sqrvech(v);
print sqr;
```

Figure 25.436 Symmetric Matrix

sqr
1 2 3
2 4 5
3 5 6

The SQRVECH function and the VECH function are inverse operations on the set of symmetric matrices. See also the SQRSYM function, which unpacks elements in row-major order.

SSQ Function

SSQ(*matrix1 < , matrix2, . . . , matrix15* **>);**

The SSQ function returns as a single numeric value the (uncorrected) sum of squares for all the elements of all arguments. You can specify as many as 15 numeric argument matrices.

The SSQ function checks for missing arguments and does not include them in the accumulation. If all arguments are missing, the result is 0.

An example of a valid statement follows:

```
a = {1 2 3, 4 5 6};
x = ssq(a);
print x;
```

Figure 25.437 Sums of Squares

x
91

STANDARD Function

STANDARD(*matrix***);**

The STANDARD function is part of the IMLMLIB library. The STANDARD function standardizes each column of an $n \times m$ matrix. Each column of the input matrix is standardized to have a mean of zero and unit standard deviation, as shown in the following example:

```
use Sashelp.Class;
read all var _NUM_ into X[colname=varNames];
close Sashelp.Class;
stdx = standard( x );
print "Standardized Data", stdx[colname=varnames];
```

Figure 25.438 Standardized Data

Standardized Data

Figure 25.438 *continued*

stdx		
Age	Height	Weight
0.4583796	1.2996021	0.5477176
-0.21156	-1.138435	-0.703713
-0.21156	0.5779431	-0.088975
0.4583796	0.0903357	0.1086191
0.4583796	0.2268658	0.1086191
-0.881499	-0.982401	-0.747623
-0.881499	-0.494793	-0.681758
1.1283191	0.0318228	0.5477176
-0.21156	0.0318228	-0.703713
-0.881499	-0.650828	-0.02311
-1.551439	-2.152658	-2.174693
0.4583796	0.3829002	-0.440254
-0.881499	-1.177444	-1.011082
1.1283191	0.8119947	0.5257627
1.7982586	1.884731	2.194337
-0.881499	0.4804216	1.2283203
1.1283191	0.9095162	1.4478695
-1.551439	-0.943392	-0.659803
1.1283191	0.8119947	0.5257627

START Statement

START < *name* > < *(arguments)* > < **GLOBAL***(arguments)* > **;**

language statements

FINISH < *name* > **;**

The START statement defines the beginning of a module definition. Subsequent statements are not executed immediately, but are instead parsed for later execution. The FINISH statement signals the end of a module definition.

The arguments to the START statement are as follows:

name	is the name of a user-defined module.
arguments	are names of variable arguments to the module. Arguments can be either input variables or output (returned) variables. Arguments listed in the GLOBAL clause are treated as global variables. Otherwise, the arguments are local.
language statements	are statements making up the body of the module.

If a parsing error occurs during module compilation, the module is not defined. See Chapter 6 for details.

The following example defines a function module that has one argument. It returns a matrix that is the same dimensions as the input argument. Each element of the output matrix is twice as large as the corresponding element of the input matrix:

```
start MyFunc(x);
   return(2*x);
finish;

c = {1 2, 3 4};
d = MyFunc(c);
print d;
```

Figure 25.439 A Function Module

d
2 4
6 8

The next example defines a module subroutine that has two input arguments (A and B) and two output arguments (X and Y). Notice that the arguments sent into the module are changed by the module:

```
start MyMod(x, y, a, b);
   x = a + b;
   y = a - b;
finish;

a = 1:3;
b = {1 0 -3};
run MyMod(p, q, a, b);
print p, q;
```

Figure 25.440 A Module Subroutine

p
2 2 0

q
0 2 6

The last example defines a module that has a GLOBAL clause. The global variables Z and W can be read and modified by the module:

```
start MyGlobal(a,b) global(z,w);
   z = a*w + b;
finish;

w = 1:4;
call MyGlobal(2, 1);
print z;
```

Figure 25.441 Results of Calling a Module with a GLOBAL Statement

z
3 5 7 9

STD Function

STD(*x*);

The STD function computes a sample standard deviation of data. The sample standard deviation of a column vector is computed as the square root of the sample variance. See the VAR function for details.

When *x* is a matrix, the sample variance is computed for each column, as the following example shows:

```
x = {5 1 10,
     6 2 3,
     6 8 .,
     6 7 9,
     7 2 13};

std = std(x);
print std;
```

Figure 25.442 Standard Deviation of Columns

std		
0.7071068	3.2403703	4.1932485

The STD function returns a missing value for columns with fewer than two nonmissing observations.

STOP Statement

STOP < *error-message* > **;**

The STOP statement stops the program, and no further matrix statements are executed. However, PROC IML does not exit, and continues to execute if more statements are submitted. See also the descriptions of the RETURN statement and the ABORT statement.

If execution was interrupted by a PAUSE statement or by a break, the STOP statement clears all the paused states and returns to immediate mode. For more information, see the section "Termination Statements" on page 79.

If you specify the optional *error-message*, the message is written to the SAS Log.

STORAGE Function

STORAGE();

The STORAGE function returns a matrix of the names of all the matrices and modules in the current storage library. The result is a character vector in which each matrix or module name occupies a row. Matrices are listed before modules. The SHOW STORAGE command separately lists all the modules and matrices in storage.

For example, the following statements print a list of the matrices and modules in the current storage library. Use the RESET STORAGE statement to change the current storage directory.

```
x = 1:5;
y = {A B C};
start MyMod(x);
   return(2*x);
finish;
store x y module=MyMod;

a = storage();
print a;
```

Figure 25.443 Contents of Storage Library

a
X
Y
MYMOD

STORE Statement

STORE <**MODULE**=*(module-list)*> <*matrix-list*> **;**

The STORE statement stores matrices and modules in a storage library.

The arguments to the STORE statement are as follows:

module-list is a list of module names. You can use the _ALL_ keyword to store all modules.

matrix-list is a list of matrix names. You can use the _ALL_ keyword to store all matrices.

See the STORAGE function for an example of the STORE statement.

The following statement stores the modules A, B, and C and the matrix X:

```
store module=(A B C) X;
```

To store all matrices or all modules, use the _ALL_ keyword, as follows:

```
store _all_ module=_all_;
```

Similarly, the following statement stores all matrices:

```
store;
```

The storage library can be specified by using the RESET STORAGE statement and defaults to WORK.IMLSTOR. The SHOW STORAGE statement lists the current contents of the storage library, and the STORAGE function returns the names of all stored items.

See Chapter 19, "Storage Features," and the descriptions of the LOAD, REMOVE, RESET, and SHOW statements for related information.

SUB2NDX Function

SUB2NDX(*dim, subscripts* **);**

The SUB2NDX function is part of the IMLMLIB library. The SUB2NDX function module converts subscripts of a matrix into indices for the matrix. The arguments are as follows:

dim specifies the dimensions of the matrix. For example, the value of this argument might be the 1×2 vector that is returned from the DIMENSION function.

subscripts is a matrix with k columns that specifies the elements of a matrix. The first column of *subscripts* specifies the first subscript dimension, the second column specifies the second subscript dimension, and so forth. When $k = 2$, the first column of *subscripts* specifies the row subscripts and the second column specifies the column subscripts.

The SUB2NDX function converts subscripts to indices. For a two-dimensional matrix, subscripts are pairs (i, j) such that $1 \le i \le n$ and $1 \le j \le p$. The indices of an $n \times p$ matrix are the elements $1, 2, \ldots, np$. The indices enumerate the elements in row-major order: the first p indices enumerate the first row, the next p indices enumerate the second row, and so forth.

The following statements construct a tridiagonal matrix that contains 2s on the diagonal and 1s on the sub- and superdiagonals. The DIAG function is used to construct a diagonal matrix. The subscripts of the superdiagonal (which, for this example, are (1,2), (2,3), and (3,4)) and the subdiagonal (which are (2,1), (3,2), and (4,3)) are then enumerated. The SUB2IND module converts these subscripts to indices, and the value 1 is assigned to all off-diagonal elements of the matrix.

```
/* construct a tridiagonal matrix */
y = diag({2,2,2,2}); /* assign diagonal */
p = ncol(y);
supDiag = T(1:p-1) || T(2:p); /* subscripts for superdiagonal */
subDiag = T(2:p) || T(1:p-1); /* subscripts for subdiagonal */
/* find index of all super- and subdiagonal elements */
dim = dimension(y);
idx = sub2ndx(dim, supDiag//subDiag);
y[idx] = 1; /* assign sub- and superdiagonal to 1 */
print y;
```

Figure 25.444 Subscripts That Correspond to Indices

y			
2	1	0	0
1	2	1	0
0	1	2	1
0	0	1	2

You can also use the SUB2NDX function to store the results of a multidimensional array in a matrix. For an array with d dimensions, a subscript is a d-dimensional vector $(i_1, i_2, \ldots, i_k)$, where $1 \le i_j \le d_j$ for $j = 1 \ldots k$. For example, suppose you store the values of a $4 \times 3 \times 3$ array in a 12×3 matrix. The following program computes the indices that correspond to the first, middle, and last elements in the matrix:

```
dim = {4 3 3};/* a 12x3 matrix can store values from 4x3x3 array */
s = {1 1 1,
     2 3 3,
     4 3 3};
ndx = sub2ndx(dim, s);
print ndx;
```

To convert from indices to subscripts, see the NDX2SUB function.

SUBMIT Statement

SUBMIT <*parameters*> </ *options*> **;**

language statements

ENDSUBMIT ;

The SUBMIT statement enables you to submit SAS statements for processing from within a SAS/IML program. You can use the SUBMIT statement to call SAS procedures, DATA steps, and macros. All text between the SUBMIT statement and the ENDSUBMIT statement are referred to as a *SUBMIT block*. The SUBMIT block is processed by the SAS language processor.

If you use the R option, the SUBMIT statement enables you to submit statements to the R language for processing.

The SUBMIT statement must appear on a line by itself. All SAS/IML matrices that are defined prior to the SUBMIT statement remain defined after the ENDSUBMIT statement.

parameters specifies one or more optional SAS/IML matrices whose values are substituted into the language statements in the SUBMIT block. To reference a parameter in the SUBMIT block, prefix the name of the parameter with an ampersand (&). If you do not specify the *parameters* argument, the SUBMIT block is sent without modification to the SAS (or R) language processor.

The following options are available in the SUBMIT statement after a slash (/).

OK=*ok-matrix* specifies the name of a matrix. The matrix is set to 1 if the SUBMIT block executes without error, and to 0 otherwise.

R specifies that statements in the SUBMIT block are processed by the R statistical software. You can use the R option to call functions in the R language, provided that the following statements are true:

1. the R statistical software is installed on the SAS workspace server.
2. The SAS system administrator at your site has enabled the RLANG SAS system option. (See the section "The RLANG System Option" on page 244.)

The following example calls a SAS procedure from a PROC IML program. The example passes in a parameter which is used by the FREQ procedure:

```
proc iml;
VarName = "Sex";
submit VarName;
proc freq data=Sashelp.Class;
   table &VarName / out=OutFreq;
run;
endsubmit;
```

Prior to the SUBMIT statement, the program defines the **VarName** matrix. The matrix contains the name of a variable in the Sashelp.Class data set. The **VarName** matrix is listed in the SUBMIT statement, which means that the contents of the matrix is available for substitution into the SUBMIT block. The SUBMIT block references the contents of the matrix by preceding the matrix name by an ampersand (&). Consequently, the FREQ procedure carries out a one-way frequency analysis for the Sex variable. The output from PROC FREQ is shown in Figure 25.445.

Figure 25.445 Result of Calling a SAS Procedure

The FREQ Procedure

Sex	Frequency	Percent	Cumulative Frequency	Cumulative Percent
F	9	47.37	9	47.37
M	10	52.63	19	100.00

The preceding statements also create output data set, OutFreq. The following statements read the data into SAS/IML matrices:

```
use OutFreq;
read all var VarName into Levels;
read all var {Count};
close OutFreq;

print Count[rowname=Levels];
```

Notice that the **VarName** matrix is still defined, even after the FREQ procedure has finished execution. The statements read portions of the PROC FREQ output data set into two SAS/IML vectors. The output from the program is shown in Figure 25.446.

Figure 25.446 Result of Calling a SAS Procedure

	COUNT
F	9
M	10

Chapter 13, "Submitting SAS Statements," provides details and further examples of submitting SAS statements. Chapter 14, "Calling Functions in the R Language," describes how to submit R statements and provides examples.

You cannot use the SUBMIT statement in code that is pushed to the input command queue with the EXECUTE, PUSH, or QUEUE subroutines. A SUBMIT block cannot be executed from a SAS macro.

SUBSTR Function

SUBSTR(*matrix, position* < *, length* > **);**

The SUBSTR function takes a character matrix as an argument (along with starting positions and lengths) and produces a character matrix with the same dimensions as the argument. Elements of the result matrix are substrings of the corresponding argument elements.

The arguments to the SUBSTR function are as follows:

matrix	is a character matrix or quoted literal.
position	is a numeric matrix or scalar that contains the starting position.
length	is a numeric matrix or scalar that contains the length of the substring.

Each substring is constructed by using the starting *position* supplied. If a *length* is supplied, this length is the length of the substring. If no *length* is supplied, the remainder of the argument string is the substring.

The arguments can be scalars or numeric matrices. If more than one argument is a matrix, all matrix arguments must have the same dimensions. If *matrix* is a matrix, its dimensions determine the dimensions of the output of the function. If *matrix* is a scalar, the dimensions of the *position* or *length* determine the dimensions of the output of the function.

If *length* is supplied, the element length of the result is MAX(*length*); otherwise, the element length of the result is

$$\text{NLENG(matrix)} - \text{MIN(position)} + 1$$

The following statements return the output shown:

```
m = {abc def ghi, jkl mno pqr};
a = substr(m, 3, 2);
print a;

s = "ABCDE";
b = substr(s, 1:4, 5:2);
print b;
```

Figure 25.447 Substrings of a Character Matrix and String

a
C F I
L O R

b
ABCDE BCDE CDE DE

In the example output, the element size of matrix **a** is 2; the elements are padded with blanks. Matrix **b** is a 1×4 matrix that contains various substrings of the string **s**.

SUM Function

SUM(*matrix1* < *, matrix2, . . . , matrix15* > **);**

The SUM function returns as a single numeric value the sum of all the elements in all arguments. There can be as many as 15 argument matrices. The SUM function checks for missing values and does not include them in the summation. It returns 0 if all values are missing.

For example, following statements compute the sum of all elements in the matrix A:

```
a = {2 1 ., 0 -1 0};
b = sum(a);
print b;
```

Figure 25.448 Sum of Matrix Elements

b
2

If you want to compute the sum for each row or for each column of a matrix, you can use the subscript reduction operator, as follows:

- `a[+,]` computes a 1×3 row vector that contains the sum of each column.
- `a[,+]` computes a 2×1 column vector that contains the sum of each row.
- `a[+]` computes a scalar value that is equivalent to `sum(a)`.

See the section "Subscript Reduction Operators" on page 55 for more information about subscript reduction operators.

SUMMARY Statement

SUMMARY < **CLASS** *operand* > < **VAR** *operand* > < **WEIGHT** *operand* > < **STAT** *operand* > < **OPT** *operand* > < **WHERE***(expression)* > **;**

The SUMMARY statement computes statistics for numeric variables for an entire data set or a subset of observations in the data set. The statistics can be stratified by the use of CLASS variables. The computed statistics are displayed in tabular form and optionally can be saved in matrices. Like most other data processing statements, the SUMMARY statement works on the current data set.

You can specify the following options:

CLASS *operand*
specifies the variables in the current input SAS data set to be used to group the summaries. The *operand* is a character matrix that contains the names of the variables. For example:

```
summary class {age sex} ;
```

Both numeric and character variables can be used as CLASS variables.

VAR *operand*
computes statistics for a set of numeric variables from the current input data set. The *operand* is a character matrix that contains the names of the variables. Also, the special keyword _NUM_ can be used as a VAR operand to specify all numeric variables. If the VAR clause is missing, the SUMMARY statement produces only the number of observations in each classification group.

WEIGHT *operand*
specifies a character value that contains the name of a numeric variable in the current data set whose values are to be used to weight each observation. Only one variable can be specified. If you specify a weight variable, variances are computed according to the formula as used by the VARDEF=WDF option in PROC SUMMARY. If a weight variable has a missing or nonpositive weight, the observation is excluded from the analysis.

STAT *operand*
computes the specified statistics. The *operand* is a character matrix that contains the names of statistics. For example, to get the mean and standard deviation, specify the following:

```
summary stat{mean std};
```

You can specify the following keywords 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 that are used in the computation of the various statistics for the corresponding analysis variable. If there is a weight variable, the result is the same as using the EXCLNPWGT option in PROC SUMMARY.
NMISS	computes the number of observations in the subgroup that have missing values for the analysis variable.
STD	computes the standard deviation. If there is a weight variable, the computation is the same as using the VARDEF=WDF option in PROC SUMMARY.
SUM	computes the sum.
SUMWGT	computes the sum of the WEIGHT variable values if WEIGHT is specified; otherwise, computes the number of observations used in the computation of statistics.
USS	computes the uncorrected sum of squares.
VAR	computes the variance.

When the STAT clause is omitted, the SUMMARY statement computes the MIN, MEAN, MAX, and STD statistics for each variable in the VAR clause.

NOBS, the number of observations in each CLASS group, is always displayed.

OPT *operand*

sets the PRINT or NOPRINT and SAVE or NOSAVE options. The NOPRINT option suppresses the printing of the results from the SUMMARY statement. The SAVE option requests that the SUMMARY statement save the resultant statistics in matrices. The *operand* is a character matrix that contains one or more of the options.

When the SAVE option is set, the SUMMARY statement creates a CLASS vector for each CLASS variable, a statistic matrix for each analysis variable, and a column vector named _NOBS_. The CLASS vectors are named by the corresponding CLASS variable and have an equal number of rows. There are as many rows as there are subgroups defined by the interaction of all CLASS variables. The statistic matrices are named by the corresponding analysis variable. Each column of the statistic matrix corresponds to a requested statistic, and each row corresponds to the statistics of the subgroup that is defined by the CLASS variables. If no CLASS variable is specified, each matrix has one row that contains the statistics. The _NOBS_ vector contains the number of observations for each subgroup.

The default is PRINT NOSAVE.

WHERE *expression*

conditionally selects observations according to conditions given in *expression*. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

The following example demonstrates the use of the SUMMARY statement:

```
proc iml;
use Sashelp.Class;
summary class {sex}
        var {height weight}
        opt {noprint save};

/* print vectors that contain the stats */
print sex _NOBS_;
print height[r=sex c={Min Max Mean Std}],
      weight[r=sex c={Min Max Mean Std}];
```

Figure 25.449 Summary Statistics

Sex	_NOBS_
F	9
M	10

	Height			
	MIN	MAX	MEAN	STD
F	51.3	66.5	60.588889	5.0183275
M	57.3	72	63.91	4.937937

	Weight			
	MIN	MAX	MEAN	STD
F	50.5	112.5	90.111111	19.383914
M	83	150	108.95	22.727186

See Chapter 7 for further details.

SVD Call

CALL SVD(*u, q, v, a***);**

The SVD subroutine computes the singular value decomposition for a numerical matrix.

The input to the SVD subroutine is as follows:

a	is the $m \times n$ input matrix that is factored as described in the following discussion.
	The SVD subroutine returns the following output arguments:
u	is an $m \times n$ orthonormal matrix
q	is an $n \times 1$ vector that contains the singular values
v	is an $n \times n$ orthonormal matrix

If $m \geq n$, the SVD subroutine factors a real $m \times n$ matrix $\mathbf{A}$ into the form

$$\mathbf{A} = \mathbf{U}\text{diag}(\mathbf{Q})\mathbf{V}'$$

where

$$\mathbf{U}'\mathbf{U} = \mathbf{V}'\mathbf{V} = \mathbf{V}\mathbf{V}' = \mathbf{I}_n$$

and $\mathbf{Q}$ contains the singular values of $\mathbf{A}$. The columns of $\mathbf{U}$ contains of the orthonormal eigenvectors of $\mathbf{A}\mathbf{A}'$, and $\mathbf{V}$ contains the orthonormal eigenvectors of $\mathbf{A}'\mathbf{A}$. $\mathbf{Q}$ contains the square roots of the eigenvalues of $\mathbf{A}'\mathbf{A}$ and $\mathbf{A}\mathbf{A}'$, except for some zeros.

If $m < n$, a corresponding decomposition is done where $\mathbf{U}$ and $\mathbf{V}$ switch roles:

$$\mathbf{A} = \mathbf{U}\text{diag}(\mathbf{Q})\mathbf{V}'$$

where

$$\mathbf{U}'\mathbf{U} = \mathbf{U}\mathbf{U}' = \mathbf{V}'\mathbf{V} = \mathbf{I}_w$$

The singular values are sorted in descending order.

For information about the method used in the SVD subroutine, see Wilkinson and Reinsch (1971).

The following example is taken from Wilkinson and Reinsch (1971):

```
a = {22  10   2   3   7,
     14   7  10   0   8,
     -1  13  -1 -11   3,
     -3  -2  13  -2   4,
      9   8   1  -2   4,
      9   1  -7   5  -1,
      2  -6   6   5   1,
      4   5   0  -2   2};
call svd(u, q, v, a);
print u, q, v;

/* check correctness of factors */
```

```
zero = ssq(a - u*diag(q)*v`);
reset fuzz;           /* print small numbers as zero */
print zero;
```

The matrix is rank-3 with exact singular values $\sqrt{1248}$, 20, $\sqrt{384}$, 0, and 0. Because of the repeated singular values, the last two columns of the **U** matrix are not uniquely determined. A valid result is shown in Figure 25.450:

Figure 25.450 Singular Value Decomposition

u				
0.7071068	0.1581139	-0.176777	-0.328209	-0.328056
0.5303301	0.1581139	0.3535534	0.5309976	0.0489362
0.1767767	-0.790569	0.1767767	-0.413567	0.1307398
0	0.1581139	0.7071068	-0.266418	0.0321656
0.3535534	-0.158114	0	0.0253566	-0.041441
0.1767767	0.1581139	-0.53033	-0.19666	0.3666144
0	0.4743416	0.1767767	-0.500944	0.4145131
0.1767767	-0.158114	0	0.2793571	0.7509412

q
35.327043
20
19.595918
1.1E-15
5.501E-16

v				
0.8006408	0.3162278	-0.288675	-0.419095	0
0.4803845	-0.632456	0	0.4405091	0.4185481
0.1601282	0.3162278	0.8660254	-0.052005	0.3487901
0	0.6324555	-0.288675	0.6760591	0.244153
0.3202563	0	0.2886751	0.4129773	-0.802217

zero
0

The SVD routine performs most of its computations in the memory allocated for returning the singular value decomposition.

SWEEP Function

SWEEP(*matrix, index-vector***);**

The SWEEP function sweeps *matrix* on the pivots indicated in *index-vector* to produce a new matrix.

The arguments the SWEEP function are as follows:

matrix is a numeric matrix or literal. For most applications, the matrix is symmetric, although the SWEEP function does not enforce that requirement.

index-vector is a numeric vector that indicates the pivots.

The values of the index vector must be less than or equal to the number of rows or the number of columns in *matrix*, whichever is smaller.

For example, suppose that **A** is symmetric and positive definite and is partitioned into

$$\begin{bmatrix} \mathbf{R} & \mathbf{S} \\ \mathbf{S}' & \mathbf{U} \end{bmatrix}$$

such that **R** is $q \times q$ and **U** is $(m-q) \times (n-q)$. Let $I = \{123 \ldots q\}$. Then, the statement **`B=sweep(A,I)`** becomes

$$\begin{bmatrix} \mathbf{R}^{-1} & \mathbf{R}^{-1}\mathbf{S} \\ -\mathbf{S}'\mathbf{R}^{-1} & \mathbf{U} - \mathbf{S}'\mathbf{R}^{-1}\mathbf{S} \end{bmatrix}$$

The index vector can be omitted. In this case, the function sweeps the matrix on all pivots on the main diagonal 1:MIN(*nrow*,*ncol*).

The SWEEP function has sequential and reversibility properties when the submatrix swept is positive definite:

- SWEEP(SWEEP(**A**,1),2)=SWEEP(**A**,{ 1 2 })
- SWEEP(SWEEP(**A**,**I**),**I**)=**A**

See Beaton (1964) for more information about these properties.

To use the SWEEP function for regression, suppose the matrix **A** contains

$$\begin{bmatrix} \mathbf{X}'\mathbf{X} & \mathbf{X}'\mathbf{Y} \\ \mathbf{Y}'\mathbf{X} & \mathbf{Y}'\mathbf{Y} \end{bmatrix}$$

where $\mathbf{X}'\mathbf{X}$ is $k \times k$.

Then $\mathbf{B} = \text{SWEEP}(\mathbf{A}, 1 \ldots k)$ contains

$$\begin{bmatrix} (\mathbf{X}'\mathbf{X})^{-1} & (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y} \\ -\mathbf{Y}'\mathbf{X}(\mathbf{X}'\mathbf{X})^{-1} & \mathbf{Y}'(\mathbf{I} - \mathbf{X}(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}')\mathbf{Y} \end{bmatrix}$$

The partitions of **B** form the beta values, SSE, and a matrix proportional to the covariance of the beta values for the least squares estimates of **B** in the linear model

$$\mathbf{Y} = \mathbf{X}\mathbf{B} + \epsilon$$

If any pivot becomes very close to zero (less than or equal to 1E–12), the row and column for that pivot are zeroed. See Goodnight (1979) for more information.

The following example uses the SWEEP function for regression:

```
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;
```

Figure 25.451 Results of a Linear Regression

cov		
0.9323716	-0.436247	0.0427693
-0.436247	0.2423596	-0.025662
0.0427693	-0.025662	0.0028513

beta
5.0693393
-1.109935
0.5396369

sse
2.395083

The SWEEP function performs most of its computations in the memory allocated for the result matrix.

SYMSQR Function

SYMSQR(*matrix***);**

The SYMSQR function takes an $n \times n$ matrix and packs the elements from the lower triangular portion into a column vector that contains $n(n+1)/2$ rows. The matrix is not checked for symmetry, but usually *matrix* is a symmetric numeric matrix. Character matrices are also supported.

The following statement produces the output shown in Figure 25.452:

```
sym = symsqr({1 2, 3 4});
print sym;
```

Figure 25.452 Elements of Lower Triangular Matrix

sym
1
3
4

Notice that the (1, 2) element is lost since it is only present in the upper triangular portion of the input matrix.

The SYMSQR function and the SQRSYM function are inverse operations on the set of symmetric matrices. See also the VECH function, which unpacks elements in column-major order.

T Function

T(*matrix***);**

The T (transpose) function returns the transpose of its argument. You can also use the transpose operator (‘) to transpose a matrix.

For example, the following statements transpose a matrix:

```
x = {1 2, 3 4, 5 6};
y = t(x);
print y;
```

Figure 25.453 Matrix Transpose

y		
1	3	5
2	4	6

TABLEADDVAR Call

CALL TABLEADDVAR(*table, colnames, matrix***);**

CALL TABLEADDVAR(*table1, table2***);**

The TableAddVar subroutine adds new columns to an existing table. The source of the columns can be a numeric or character matrix. The new columns are appended after the existing columns.

You can use the TableAddVar subroutine to horizontally concatenate two tables. The columns of the second table are appended to the first table. The second table is unchanged.

The subroutine takes the following input arguments:

table	specifies an existing table.
colnames	specifies a character vector that contains *p* strings. The strings specify the names for the new columns in the table. It is an error to specify a column name that already exists in the table.
matrix	specifies a numeric or character matrix that has *p* columns.

The following example calls the TableCreate function to create an empty table and then calls the TableAddVar subroutine twice: first to add character columns, and then to add numeric columns. Each row in the table contains data about historic hurricanes that struck the United States. Figure 25.454 shows the names and types of the columns of the table.

```
tbl = TableCreate();   /* create empty table */

Hurr  = {"Katrina", "Ike",       "Andrew", "Wilma"};
Month = {"August",  "September", "August", "October"};
call TableAddVar(tbl, {"Name" "Month"}, Hurr||Month); /* add char cols */

Yr      = {2005, 2008, 1992, 2005};
Wind = { 175,  145,  175,  185};   /* mph */
call TableAddVar(tbl, {"Year" "MaxWind"}, Yr||Wind);  /* add numeric cols */

colNames = TableGetVarName(tbl);
colTypes = TableGetVarType(tbl);
print colTypes[c=colNames L="Table Names and Types"];
```

Figure 25.454 Names and Types of Table Columns

Table Names and Types			
Name	Month	Year	MaxWind
C	C	N	N

You can use the TableAddVar subroutine to horizontally concatenate two tables. You can also use the horizontal concatenation operator (||) to horizontally concatenate two or more tables. The following statements show two ways to concatenate tables. The number of rows in the tables do not have to be equal, although usually they will be.

```
/* Method 1: Concatenation operator */
tbl1 = TableCreate("Letters", {"A", "B", "C"});   /* create a character table */
tbl2 = TableCreate("x1":"x2", {1 2, 3 4, 5 6} );  /* create a numeric table */
concatTbl = tbl1 || tbl2;

/* Method 2: Copy the first table; append columns of the second table */
concatTbl = tbl1;
call TableAddVar(concatTbl, tbl2);
```

TABLECREATE Function

TABLECREATE();

TABLECREATE(*matrix***);**

TABLECREATE(*colnames, matrix***);**

TABLECREATE(*colnames1, matrix1, colnames2, matrix2, . . .***);**

The TableCreate function creates a table from a matrix. Each column of the matrix becomes a new column in the table.

The function takes the following input arguments:

colnames	specifies a character vector that contains *p* strings. The strings specify the names for the columns in the table. If this argument is omitted, the names Col1 through Col*p* are used.
matrix	specifies a numeric or character matrix that has *p* columns.

If no arguments are supplied, an empty table is created. An empty table has no variables and no observations.

The following example creates three tables. The first is an empty table. The next table has one column that contains character data. The third table contains three numeric columns named Col1, Col2, and Col3. Figure 25.454 shows the dimensions of each table.

```
tbl0 = TableCreate();                        /* create an empty table */
tbl1 = TableCreate("Letters", T("A":"Z"));   /* create a character table */
tbl2 = TableCreate( {1 2 3, 4 5 6} );        /* create a numeric table */

dim = dimension(tbl0) //
      dimension(tbl1) //
      dimension(tbl2);
print dim[c={"nrow" "ncol"} r={"tbl0" "tbl1" "tbl2"}];
```

Figure 25.455 Dimensions of Three Tables

	dim	
	nrow	ncol
tbl0	0	0
tbl1	26	1
tbl2	2	3

The TABLECREATE function supports a syntax that enables you to create a matrix from multiple matrices, both numeric and character. You have to specify the names of columns, followed by the data for the columns. You can supply up to seven pairs of arguments. The following example show how to create a table from multiple matrices. Notice that the odd-numbered arguments contain the names of columns whereas the even-numbered arguments contain data.

```
/* create a table from four matrices */
C = {"Alonzo"  "M",
     "Juanita" "F",
     "Wei"     "M"};
C2 = {A B C,
      C D E,
      E F G};
X  = {1 2,
      3 4,
      5 6};
X2 = {14, 16, 15};
```

```
tbl3 = TableCreate({"Name" "Sex"}, C,     /* character data */
                   "X1":"X2", X,          /* numeric data */
                   "Age", X2,             /* more numeric data */
                   {"A" "B" "C"}, C2);    /* more character data */
```

TABLECREATEFROMDATASET Function

TABLECREATEFROMDATASET(*member***);**

TABLECREATEFROMDATASET(*libref, member***);**

TABLECREATEFROMDATASET(*libref, member, dsOptions***);**

The TableCreateFromDataSet function creates a table from a SAS data set. Each variable in the data set becomes a new column in the table.

The function takes the following input arguments, which are all character strings:

libref	specifies the name of a SAS libref. If this argument is omitted, the default libref is used. You can use the DEFLIB= option in the RESET statement to specify the default libref.
member	specifies the name of a SAS data set.
dsOptions	specifies data set options, such as DROP=, KEEP=, and WHERE= options.

The function returns a table. If you omit the *dsOptions* argument, the table contains all the variables and all the observations in the *member* data set.

The following program creates a table from the Sashelp.Class data set. It also creates a table from a data set (Cars) in the Work library.

```
data work.cars;
   set Sashelp.Cars;
run;

proc iml;
tbl1 = TableCreateFromDataSet("Sashelp", "Class");
reset deflib=work;
tbl2 = TableCreateFromDataSet("Cars");    /* read from WORK */
dsOpt = "drop=Invoice rename=(Wheelbase=Width) " +
        "obs=100 where=(Origin='Asia')";
tbl3 = TableCreateFromDataSet("work", "Cars", dsOpt);
```

TABLEGETVARDATA Function

TABLEGETVARDATA(*table, cols***);**

The TableGetVarData function creates a matrix from the columns of a table. The specified columns must be the same type: either all numeric or all character.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}).

The following statements create a table. The Name and Sex variables are extracted to a character matrix, and the third and fifth variables are extracted to a numeric matrix.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
c = TableGetVarData(tbl, {"Name" "Sex"});
m = TableGetVarData(tbl, {3 5});  /* 3rd and 5th col are numeric */
```

TABLEGETVARFORMAT Function

TABLEGETVARFORMAT(*table* < , *cols* >**);**

The TableGetVarFormat function returns a row vector that contains the formats of the specified columns. Blank strings are returned for columns that do not have formats.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If this argument is omitted, the formats for all columns are returned.

The following statements create a table from the Sashelp.Cars data. The TableGetVarFormat function retrieves the formats for four columns, and the names are shown in Figure 25.456. A blank space is displayed for each variable that does not have a format.

```
tbl = TableCreateFromDataSet("Sashelp", "Cars");
colNames = {"MSRP" "Invoice" "EngineSize" "Weight"};
formats = TableGetVarFormat(tbl, colNames);
print formats[colname=colNames];
```

Figure 25.456 Column Formats

formats			
MSRP	Invoice	EngineSize	Weight
DOLLAR8.	DOLLAR8.		

TABLEGETVARINDEX Function

TABLEGETVARINDEX(*table, colnames***);**

The TableGetVarIndex function returns a row vector that contains the column numbers (indices) for the specified column names.

The function takes the following input arguments:

table specifies an existing table.

colnames specifies a character vector of names (such as {“X” “Y”}).

The following statements obtain the column numbers of the Sex and Height variables. Figure 25.457 shows that Sex is the second variable and Height is the fourth variable.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
colNames = {"Sex" "Height"};
idx = TableGetVarIndex(tbl, colNames);
print idx[colname=colNames];
```

Figure 25.457 Indices of Column Names

idx	
Sex	Height
2	4

TABLEGETVARINFORMAT Function

TABLEGETVARINFORMAT(*table* < , *cols* >**);**

The TableGetVarInformat function returns a row vector that contains the informats of the specified columns. Blank strings are returned for columns that do not have informats.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If this argument is omitted, the informats for all columns are returned.

The following statements create a SAS data set whose variables contain informats. The subsequent PROC IML statements create a table from the data set. The TableGetVarInformat retrieves the informats, which are displayed in Figure 25.458.

```
data Statisticians;
informat First $8. Last $CHAR10. BirthDate YYMMDD8.;
format BirthDate DATE10.;
input First $ Last $ BirthDate;
datalines;
Karl     Pearson   18570327
William  Gossett   18760613
Ronald   Fisher    18900217
;

proc iml;
tbl = TableCreateFromDataSet("Statisticians");
Name = TableGetVarName(tbl);
informats = TableGetVarInformat(tbl);
All = Name` || informats`;
```

```
cnames = {"VarName" "Informats" "Length(informat)"};
print All[L="All Informats" c=cnames];
```

Figure 25.458 Column Informats

All Informats	
VarName	**Informats**
First	$8.
Last	$CHAR10.
BirthDate	YYMMDD8.

TABLEGETVARLABEL Function

TABLEGETVARLABEL(*table* < , *cols* >**);**

The TableGetVarLabel function returns a row vector that contains the labels of the specified columns.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If this argument is omitted, the labels for all columns are returned.

The following statements create a table from the Sashelp.Cars data. Figure 25.459 shows the labels for the specified columns.

```
tbl = TableCreateFromDataSet("Sashelp", "Cars");
colNames = {"Length" "MPG_City" "MPG_Highway"};
labels = TableGetVarLabel(tbl, colNames);
print labels[colname=colNames];
```

Figure 25.459 Column Labels

labels		
Length	**MPG_City**	**MPG_Highway**
Length (IN)	MPG (City)	MPG (Highway)

TABLEGETVARNAME Function

TABLEGETVARNAME(*table* < , *cols* >**);**

The TableGetVarName function returns a row vector that contains the names of the specified columns.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument is a numeric vector of column numbers such as {2 4 7}. If this value argument is not specified, all column names are returned.

The following statements create a table from the Sashelp.Class data set. Figure 25.460 shows the names of the first and fifth columns.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
names = TableGetVarName(tbl, {1 5});
print names;
```

Figure 25.460 Column Names for Indices

names
Name Weight

TABLEGETVARTYPE Function

TABLEGETVARTYPE(*table* < , *cols* >**);**

The TableGetVarType function returns a row vector that contains the types (numeric or character) of the specified columns.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If this argument is omitted, the types for all columns are returned.

The following statements obtain the types of the Sex and Height variables. Figure 25.461 shows that the Sex variable is character and the Height variable is numeric.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
colNames = {"Sex" "Height"};
types = TableGetVarType(tbl, colNames);
print types[c=colNames];
```

Figure 25.461 Types of Columns

types	
Sex	Height
C	N

TABLEISEXISTINGVAR Function

TABLEISEXISTINGVAR(*table, colnames***);**

The TableIsExistingVar function returns a binary row vector that indicates whether the specified column names exist in a table. The return value is 1 for each variable that exists and 0 for each variable that does not exist.

The function takes the following input arguments:

table specifies an existing table.

colnames specifies a character vector of names (such as {“X” “Y”}).

The following statements indicate whether the specified column names are in the table. Figure 25.462 shows that the table contains columns that are named Sex and Height. Figure 25.462 displays a 1 for the Sex and Height variables and 0 for the variables that do not exist.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
colNames = {"Sex" "ABC" "Height" "X"};
isVar = TableIsExistingVar(tbl, colNames);
print isVar[colname=colNames];
```

Figure 25.462 Indicator Variable for Column Names

isVar			
Sex	ABC	Height	X
1	0	1	0

TABLEISVARNUMERIC Function

TABLEISVARNUMERIC(*table* < , *cols* >**);**

The TableIsVarNumeric function returns a binary row vector that indicates whether the specified columns are numeric. The value 1 indicates a numeric variable; the value 0 indicates that a variable is not numeric.

The function takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If this argument is omitted, a binary vector is returned that indicates whether each variable is numeric.

The following statements indicate whether the Sex and Height variables are numeric. Figure 25.463 shows that the Sex variable is not numeric and the Height variable is numeric.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
colNames = {"Sex" "Height"};
isNumeric = TableIsVarNumeric(tbl, colNames);
print isNumeric[c=colNames];
```

Figure 25.463 Numeric and Character Columns

isNumeric	
Sex	Height
0	1

TABLEPRINT Call

CALL TABLEPRINT(*table***) VAR=***cols*
ID=*IDName*
LABEL=*label*
FIRSTOBS=*firstObs*
NUMOBS=*numObs*
COLHEADER="Names" | "Labels" | "None"
JUSTIFY=*just*
TEMPLATE=*template*
COLTEMPLATE=*colTemplates*
DYNAMIC=*dynValues*
COLDYNAMIC=*colDynValues*
;

The TablePrint subroutine displays a table in the open ODS destinations. The subroutine takes the following required input argument:

table specifies an existing table.

The remaining arguments are optional and can be specified as keyword-value pairs. Specify these options outside the parentheses, as shown in the example in this section.

VAR=*cols*
specifies the columns of the table, where *cols* can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If you omit this option, all variables appear in the table.

ID=*IDName*
specifies the name of a column. The column (sometimes called a *row header*) appears on the left side of the table. You can specify a blank string to suppress the row headers. If you omit this argument or specify the special string “#”, then row numbers are used for the row headers.

LABEL=*label*
specifies a string for the name of the table. If you omit this option, the symbol name is used as the label. You can specify a blank string to suppress the label.

FIRSTOBS=*firstObs*
specifies the first row of the table to display. If you omit this option, the table is displayed beginning with the first row.

NUMOBS=*numObs*
specifies the total number of rows to display. If you omit this option, all rows of the table are displayed.

COLHEADER="Names" | "Labels" | "None"
specifies the form of the *column headers* that appear above columns. If you omit this option or specify the string "Names", then the column names are displayed. The string "Labels" displays column labels, if they exist. Column headers are not displayed if you specify the string "None" or a blank string.

JUSTIFY=*just*
specifies the horizontal alignment for each column in a table, where *just* is a character vector with p elements that specifies the alignment for the first p columns. Each element of *just* is one of the following:

- To center a column, use "C" or "Center".
- To left-align a column, use "L" or "Left".
- To right-align a column, use "R" or "Right".
- To use the default alignment, use a space character (" "), "D", or "Default". By default, character columns are left-aligned and numeric columns are right-aligned.

If you specify a vector that has fewer elements than the number of columns, default values are used for the unspecified columns.

TEMPLATE=*template*
specifies the name of an ODS table template to be used to display the SAS/IML table. For example, TEMPLATE=MyTemplate causes ODS to search for a template named MyTemplate and use that template to display the table.

COLTEMPLATE=*colTemplates*
specifies the name of column templates that are used to display columns in the SAS/IML table. The *colTemplates* argument is a character vector with p elements that specifies template names for the first p columns. You can use the same template for multiple columns provided that the template has the GENERIC attribute. If you specify a vector that has fewer elements than the number of columns, default attributes are used to display the remaining columns. For example, the syntax COLTEMPLATE={CT1 CT2} looks for column templates named CT1 and CT2. It uses the CT1 definition to format the first column in the table and uses the CT2 definition to format the second column.

DYNAMIC=*dynValues*
specifies values for dynamic variables in a template, where *dynValues* is a character matrix with k elements that specifies the values of k dynamic variables. Each element has the form "*DynVar=IMLSym*", where *DynVar* is the name of a dynamic variable in the template and *IMLSym* is the name of a SAS/IML scalar matrix that contains the value to use for the dynamic variable. The *IMLSym* symbol can be a character matrix or a numeric matrix. If the SAS/IML symbol has the same name as the dynamic variable, then you can use an alternate syntax in which an element has the form "DynVar". For examples, see Chapter 9, "Mixed-Type Tables."

COLDYNAMIC=*colDynValues*
specifies values for dynamic variables in a column template, where *dynValues* is a character matrix with k elements that specifies the values for k columns. One template can have multiple dynamic variables, so each element has the form "*DynVar1=IMLSym1 DynVarD2=IMLSym* ...". For more information, see the DYNAMIC= option. For examples, see Chapter 9, "Mixed-Type Tables."

The following statements show how to print a table. The first call displays the table by using default options. The result is shown in Figure 25.464. The second call specifies several options. The result is shown in Figure 25.465.

```
proc iml;
tbl = TableCreateFromDataSet("Sashelp", "Class", "WHERE=(sex='M')");
call TablePrint(tbl);
call TablePrint(tbl) VAR={"Height" "Age"}
                     ID="Name"
                     LABEL="Heights and Ages for Five Boys"
                     FIRSTOBS=3
                     NUMOBS=5
                     JUSTIFY={C L};
```

Figure 25.464 Display a Table by Using Default Options

tbl

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69	112.5
2	Henry	M	14	63.5	102.5
3	James	M	12	57.3	83
4	Jeffrey	M	13	62.5	84
5	John	M	12	59	99.5
6	Philip	M	16	72	150
7	Robert	M	12	64.8	128
8	Ronald	M	15	67	133
9	Thomas	M	11	57.5	85
10	William	M	15	66.5	112

Figure 25.465 Display a Table

Heights and Ages for Five Boys

Name	Height	Age
James	57.3	12
Jeffrey	62.5	13
John	59	12
Philip	72	16
Robert	64.8	12

The section "Advanced Printing of Tables" on page 136 contains advanced examples of using the TABLEPRINT subroutine. Examples include the following:

- Using a custom template to display a table
- Headers that span multiple columns
- Cells that are colored according to data values (traffic lighting)
- Dynamic variables in templates whose values are specified at run time.

TABLERENAMEVAR Call

CALL TABLERENAMEVAR(*table, cols, newNames***);**

The TableRenameVar subroutine changes the names of specified columns. The subroutine takes the following input arguments:

table	specifies an existing table.
cols	specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}).
newNames	specifies a character vector of valid SAS variable names, which will become the new names for the specified columns.

The following statements rename the Sex and Height variables of the Sashelp.Class data. Figure 25.466 shows the original and the new names for the columns.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
oldNames = TableGetVarName(tbl);
call TableRenameVar(tbl, {"Sex" "Height"},  /* old names */
                         {"Gender" "Hgt"}); /* new names */
newNames = TableGetVarName(tbl);
print (oldNames // newNames)[rowname={"Old" "New"}];
```

Figure 25.466 Old and New Column Names

Old	Name	Sex	Age	Height	Weight
New	Name	Gender	Age	Hgt	Weight

TABLESETVARFORMAT Call

CALL TABLESETVARFORMAT(*table, cols, formats***);**

The TableSetVarFormat subroutine sets the formats of the specified columns. You can use a blank string to remove a format.

The subroutine takes the following input arguments:

table	specifies an existing table.
cols	specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}).
formats	specifies a character vector that contains formats for the specified columns.

The following statements set formats for three columns of a table. The blank cells in Figure 25.467 show that the columns did not originally have formats.

```
tbl = TableCreateFromDataSet("Sashelp", "Cars");
varNames = {"Model" "EngineSize" "Weight"};
OldFormats = TableGetVarFormat(tbl, varNames);

formats =  {"$25."  "4.1"  "COMMA9.2"};
call TableSetVarFormat(tbl, varNames, formats);
NewFormats = TableGetVarFormat(tbl, varNames);

cnames = {"VarName" "OrigFormats" "NewFormats"};
All = varNames` || OldFormats` || NewFormats`;
print All[L="New Formats" c=cnames];
```

Figure 25.467 Column Formats

New Formats		
VarName	**OrigFormats**	**NewFormats**
Model		$25.
EngineSize		4.1
Weight		COMMA9.2

TABLESETVARINFORMAT Call

CALL TABLESETVARINFORMAT(*table, cols, informats***);**

The TableSetVarInformat subroutine sets the informats of the specified columns. You can use a blank string to remove an informat.

The subroutine takes the following input arguments:

table	specifies an existing table.
cols	specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}).
informats	specifies a character vector that contains informats for the specified columns.

The following statements set informats for two columns of a table:

```
tbl = TableCreateFromDataSet("Sashelp", "Class");
varNames = {"Height" "Weight"};
informats = {"4.1"  "5.1"};
call TableSetVarInformat(tbl, varNames, informats);
```

TABLESETVARLABEL Call

CALL TABLESETVARLABEL(*table, cols, labels***);**

The TableSetVarLabel subroutine sets the labels of the specified columns.

The subroutine takes the following input arguments:

table specifies an existing table.

cols specifies the columns of the table. The *cols* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}).

labels specifies a character vector that contains labels for the specified columns.

The following statements set the labels for two columns of a table. The new labels are shown in Figure 25.468.

```
/* the Sashelp.Class data set does not contain labels */
tbl = TableCreateFromDataSet("Sashelp", "Class");
colNames = {"Height" "Weight"};
call TableSetVarLabel(tbl, colNames,
                           {"Height (in)" "Weight (lbs)"});
newLabels = TableGetVarLabel(tbl, colNames);
print newLabels[colname=colNames];
```

Figure 25.468 Column Labels

newLabels	
Height	**Weight**
Height (in)	Weight (lbs)

TABLESORT Call

CALL TABLESORT(*table***);**

CALL TABLESORT(*table, by***);**

CALL TABLESORT(*table, by, descend***);**

The subroutine takes the following input arguments:

table specifies the table to be sorted.

by specifies the columns to be sorted. The *by* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}). If this argument is not specified or is a null matrix, the table is sorted by the first column.

descend specifies which columns, if any, should be sorted in descending order. The remaining *by* columns are ascending. If *descend* is not specified or is a null matrix, the table is sorted by all *by* columns in ascending order. The *descend* argument can be a numeric vector of column numbers (such as {2 4 7}) or a character vector of names (such as {“X” “Y”}).

The TABLESORT subroutine, which sorts a table, is similar to the SORT subroutine, which sorts a matrix. One difference is that the TABLESORT subroutine performs linguistic sorting by using Unicode characters, whereas the SORT subroutine sorts character data by comparing the bytes of the data. To compare two character strings, the TABLESORT subroutine converts both strings from SAS session encoding to Unicode and then performs a linguistic comparison. A linguistic comparison sorts ‘a’ before ‘A’, whereas an ASCII-based byte comparison sorts ‘A’ before ‘a’. Both the TABLESORT and SORT subroutines sort numeric missing values as follows: `._` < `.` < `.A` < `.Z`.

You can call the TABLESORT subroutine by using keyword-value pairs syntax, where the keywords are **TABLE**, **BY**, and **DESCEND**. For example, `call tablesort(T) by={1 2};`.

The following program creates a table that has three columns and then sorts the table by the first two columns:

```
x = {  3,   2,   1,   1  };
y = { 'C', 'B', 'A', 'a' };
z = { 10,  20,  30,  40  };

tbl = TableCreate( "X", x, "Y", y, "Z", z );
call TableSort( tbl, {"X","Y"} );
call TablePrint( tbl );
```

Figure 25.469 A Sorted Table

newLabels	
Height	**Weight**
Height (in)	Weight (lbs)

TABLEWRITETODATASET Call

CALL TABLEWRITETODATASET(*table, member***);**

CALL TABLEWRITETODATASET(*table, libref, member***);**

CALL TABLEWRITETODATASET(*table, libref, member, dsOptions***);**

The TableWriteToDataSet subroutine creates a SAS data set from a table. Each column in the table becomes a new variable in the data set.

The subroutine takes the following input arguments. The first argument is a table; the other arguments are character strings.

table	specifies an existing table.
libref	specifies the name of a SAS libref. If this argument is omitted, the default libref is used. You can use the DEFLIB= option in the RESET statement to specify the default libref.
member	specifies the name of a SAS data set.
dsOptions	specifies data set options, such as DROP=, KEEP=, and WHERE= options.

The subroutine creates a data set. The location of the data set is determined by the value of the *libref* argument; the name of the data set is determined by the value of the *member* argument. If you do not specify the *dsOptions* argument, the data set contains all the variables and all the observations in the *member* data set.

The following program creates a table from the Sashelp.Class data set. The table is written to the default libref, which is usually Work. A second call to the TableWriteToDataSet subroutine writes a subset of the data to Work.Boys.

```
tbl = TableCreateFromDataSet("Sashelp", "Class");

call TableWriteToDataSet(tbl, "Class");  /* write to default libref */
dsOpt = "drop=Weight where=(Sex='M')";
call TableWriteToDataSet(tbl, "work", "Boys", dsOpt); /* work.Boys */
```

TABULATE Call

CALL TABULATE(*levels, freq, x* < *, method*> **);**

The TABULATE subroutine counts the number of elements in each of the unique categories of the *x* argument.

The output arguments are as follows:

levels	contains the unique sorted elements of the *x* argument. See also the UNIQUE function.
freq	contains the number of elements of *x* that match each element of *levels*.

The input arguments are as follows:

x	specifies a vector of values.
method	specifies whether missing values are included in the analysis. The following values are valid:
	"nomissing" specifies that missing values are excluded from the analysis. This is the default value for the option.
	"missing" specifies that missing values are counted as a valid separate level.

The *method* argument is not case-sensitive. The first two characters are used to determine the value. For example, "MISS" and "missing" specify the same option.

The following statements demonstrate the TABULATE subroutine:

```
x = {C, A, B, A, C, A};
call tabulate(labels, freq, x);
print freq[colname=labels];

x = {C, A, B, " ", A, C, A, " "};
call tabulate(labels, freq, x, "Missing");
labels = "Missing" || remove(labels, 1);
print freq[colname=labels];
```

Figure 25.470 Frequencies of Levels

freq		
A	B	C
3	1	2

freq			
Missing	A	B	C
2	3	1	2

TFHILBERT Function

TFHILBERT(*x***);**

The TFHILBERT function returns the analytic signal that corresponds to the input vector *x*. The analytic signal that corresponds to a continuous time series $x(t)$ is the complex time series $z(t) = x(t) + \mathbf{i}\hat{x}(t)$, where $\hat{x}(t)$ is the Hilbert transform of $x(t)$ and $\mathbf{i} = \sqrt{-1}$. In many applications, replacing the original time series by its analytic transform produces better results (Marple 1999).

If the input vector has *n* elements, then the output is an $n \times 2$ matrix whose first column represents the real part of the analytic signal and whose second column represents the imaginary part of the analytic signal.

For a continuous time series $x(t)$ with Fourier transform $X(f) = \int x(t) \exp(-2\pi \mathbf{i} f t)\, dt$, the Hilbert transform is defined by inverting $X(f)$ over the positive frequencies (Cohen 1995, p. 30):

$$\hat{x}(t) = 2 \int_0^{\infty} X(f) \exp(2\pi i f t)\, df$$

The spectrum of $\hat{x}(t)$ is identical to the spectrum of $x(t)$ for positive frequencies, and the spectrum of $\hat{x}(t)$ is 0 for negative frequencies.

The Hilbert transform of a discrete time series is similarly constructed as a time series whose discrete Fourier transform coincides with that of the input time series for positive spectra and vanishes otherwise. The implementation here is based on the method described in Marple (1999).

The following example demonstrates the TFHILBERT function:

```
x = 1:10;
s = tfhilbert(x);
print s;
```

Figure 25.471 Hilbert Transform

	s
1	5.5055277
2	-0.649839
3	-0.649839
4	-2.102924
5	-2.102924
6	-2.102924
7	-2.102924
8	-0.649839
9	-0.649839
10	5.5055277

TFPWV Function

TFPWV(*x, window* **<** *, overlap* **>** **<** *, fftlen* **>** **<** *, center* **>** **<** *, nthreads* **>** **<** *, hilbert_tsf* **>** **);**

The TFPWV function computes the pseudo-Wigner-Ville (PWV) transform of the input vector *x*. The function returns a three-column matrix that has fftlen $\times\, k$ rows, where

$$k = \left\lfloor \frac{\text{series\_length} - \textit{overlap}}{\text{window\_length} - \textit{overlap}} \right\rfloor$$

Here series_length is the dimension of the argument *x*, and window_length is the dimension of the argument *window*.

The three columns of the resulting output are as follows:

1. The first column stores the normalized frequency index. The frequency index starts at 0 and increments in multiples of $1/(2\textit{fftlen})$.
2. The second column stores the time index. The time index starts at 0 and increments in multiples of window_length – *overlap*.
3. The third column stores the value of the pseudo-Wigner-Ville distribution that corresponds to the time index and the normalized frequency index.

The input arguments are as follows:

x specifies the input vector. If the SAS/HPF product is not installed, the input vector is restricted to $2^{20} = 1048576$ elements.

window specifies a vector that contains the window values to be used for computation of pseudo-Wigner-Ville distribution. The *window* must have an odd number of elements. See also the TFWINDOW function.

overlap specifies the overlap between consecutive windows, where *overlap* must be an integer that is strictly less than window_length. The default value is $\lfloor \text{window\_length}/2 \rfloor$.

fftlen specifies the length of the vector on which to perform a finite Fourier transform. The value of *fftlen* must be a positive integer and must be at least as large as window_length. The computation runs fastest when *fftlen* is a power of two. The default value is the greater of 256 and window_length.

center specifies whether to center the input during computations. The default value is 0. The following values are valid:

- **0** does not center the input.
- **1** subtracts the mean of the input vector from each term of the input vector before calculating the PWV distribution. Missing values of the input are excluded in the calculation of the mean.

nthreads suggests the number of threads to use during computation. This option is not available unless you have a license for the SAS/HPF product.

hilbert_tsf specifies whether to replace the input signal by an analytical signal. The default value is 1. The following values are valid:

0 does not replace the input series.

1 replaces the input series. If the value of *center* is 1, the input series is replaced by the analytic signal for the centered signal. Otherwise the analytical signal for the input series is used.

The pseudo-Wigner-Ville distribution is a modification of the Wigner-Ville distribution. The Wigner-Ville distribution of a continuous time series $x(t)$ is obtained by computing the Fourier transform of $x(t+\tau/2)\overline{x(t-\tau/2)}$ for fixed t as τ varies. So the Wigner-Ville distribution of a continuous, possibly complex-valued, time series $x(t)$ is given by $W_x(t, f)$:

$$\begin{aligned} W_x(t, f) &= \int_{-\infty}^{\infty} x(t+\tau/2)\overline{x(t-\tau/2)}\exp(-2\pi \mathbf{i} f\tau)\, d\tau \\ &= 2\int_{-\infty}^{\infty} x(t+\tau)\overline{x(t-\tau)}\exp(-4\pi \mathbf{i} f\tau)\, d\tau. \end{aligned}$$

It can be shown that the $W_x(t, f)$ is real even when $x(t)$ takes complex values.

The pseudo-Wigner-Ville distribution is obtained by adding an additional term in the defining integral. The pseudo-Wigner-Ville distribution of a continuous time series $x(t)$ is given by

$$W_x(t, f) = \int w(\tau)x(t+\tau/2)\overline{x(t-\tau/2)}\exp(-2\pi \mathbf{i} f\tau)\, d\tau$$

where $w(\tau)$ is a window function.

The Wigner distribution of discrete time series $x[k]$ is defined as follows (Claasen and Mecklenbräuker 1980b, a; Debnath 2002):

$$W_f(n, f) = 2\sum_{k=-\infty}^{\infty} x[n+k]\overline{x[n-k]}\exp(-4\pi \mathbf{i} f k)$$

From the preceding formula, it follows that $W_n(n, f/2)$ is the discrete-time Fourier transform of $x[n+k]\overline{x[n-k]}$ and provides the basis for the computation here.

Given an input time series $x(t)$, the computation can be considered to be the evaluation of a function $U_x(n, f)$, which measures the value of the pseudo-Wigner-Ville distribution at time n and frequency $f/2$ for different values of n and f. Now $U_x(n, f)$ can be defined: given a possibly complex-valued time series $x = (x[0], x[2], \ldots, x[L-1])$ and a window of odd length $2m+1$, where *window*=$(w[0], \ldots, w[2m])$, define

$$U_x(n, f) = \sum_{k=-m}^{m} w[m+k]x[n+k]\overline{x[n-k]}\exp(-2\pi \mathbf{i} k f)$$

The preceding summation is performed with the following convention: any term in the summation for which both $n+k$ and $n-k$ do not lie between 0 and series_length – 1 is replaced with 0.

If $S = \text{window\_length} - \textit{overlap}$ and $k = \left\lfloor \dfrac{\text{series\_length} - \textit{overlap}}{S} \right\rfloor$, the output of the PWV function is determined by the evaluation of the complex value $U_x(n, f)$ for $n = 0, S, 2S, \ldots, kS$ and for $f = 0, 1/\textit{fftlen}, \ldots, (\textit{fftlen}-1)/\textit{fftlen}$. If u denotes the real part of $U_x(n, f)$, then a row of output consists of the triplet of values $(f/2, n, u)$ as f varies over $1/\textit{fftlen}, \ldots, (\textit{fftlen}-1)/\textit{fftlen}$ and n varies over $0, S, 2S, \ldots, kS$.

The pseudo-Wigner-Ville distribution has some undesirable properties. It displays annoying artifacts for multicomponent time series (Cohen 1995), but you can get better results by replacing the input with the analytic signal that corresponds to the input (Boashash 1988). For this reason, the TFPWV function provides the *hilbert_tsf* option, which replaces the original series with its analytic signal before computation, and the *center* option, which removes the mean from the series so that an overall mean effect does not show up in the output. Before any computation, the input series is first transformed depending on the value of the *hilbert_tsf* and *center* parameters. First the value of the *center* parameter is checked; if it is 1, then the input series is replaced by the centered series that is obtained by subtracting the series mean from each term of the series. If the value of the *hilbert_tsf* parameter is also 1, this possibly centered series is replaced by the analytic signal that corresponds to the centered input.

The following statements demonstrate the TFPWV function:

```
x = 1:10;
window_len=3; overlap=0; fftlen=4;
window = tfwindow(window_len);              /* Hanning window */
output = tfpwv(x, window, overlap, fftlen);
print output[colname={"f" "t" "pwv"}];
```

Figure 25.472 Pseudo-Wigner-Ville Distribution of a Signal

output		
f	t	pwv
0	0	4.4222912
0.125	0	4.4222912
0.25	0	4.4222912
0.375	0	4.4222912
0	3	29.422291
0.125	3	29.422291
0.25	3	29.422291
0.375	3	29.422291
0	6	64.422291
0.125	6	64.422291
0.25	6	64.422291
0.375	6	64.422291

TFSTFT Function

TFSTFT(*x* < , *window*> < , *overlap*> < , *fftlen*> < , *center*> < , *nthreads*> **);**

The TFSTFT function computes the short-time Fourier transform of *x*. The function returns a seven-column matrix that has *fftlen* $\times\, k$ rows, where

$$k = (1 + \lfloor \mathit{fftlen}/2 \rfloor) \left\lfloor \frac{\text{series\_length} - \mathit{overlap}}{\text{window\_length} - \mathit{overlap}} \right\rfloor$$

Here series_length is the dimension of the argument *x* and window_length is the dimension of the argument *window*.

The seven columns of *output* from left to right are as follows:

1. The first column contains values of the normalized frequency index. The frequency index starts at 0 and increases in increments of $1/$*fftlen*.
2. The second column contains the time index. The time index starts at 0 and increases in increments of window_length – *overlap*.
3. The third column contains the power of *x* that corresponds to the current normalized frequency index and the current time index. The value in this column is given by $u^2 + v^2$, where u is the real part of the Fourier coefficient (the value stored in the sixth column) and v is the imaginary part of the Fourier-coefficient (the value stored in the seventh column).
4. The fourth column contains the amplitude that corresponds to the current normalized frequency index and the current time index. The amplitude is given by $\sqrt{\text{power}}$, where power is the value stored in the third column.
5. The fifth column contains the phase of *x* that corresponds to the current normalized frequency index and the current time index. The phase is computed as ATAN2(v,u), where u is the real part of the Fourier-coefficient (stored in the sixth column) and v is the imaginary part of the Fourier-coefficient (stored in the seventh column) of *x*.
6. The sixth column contains the real part of the Fourier coefficient.
7. The seventh column contains the imaginary part of the Fourier coefficient.

The input arguments are as follows:

x specifies the input vector. Any missing value in this vector is replaced with 0. If the SAS/HPF product is not installed, the input vector is restricted to $2^{20} = 1048576$ elements.

window specifies the vector to use as a windowing function when computing the short-time Fourier transform. The default is a Hanning window whose length is the lesser of 256 and the dimension of *x*. Usually the vector is created by calling the TFWINDOW function.

overlap specifies the overlap between consecutive windows, where *overlap* must be an integer that is strictly less than window_length. The default value is $\lfloor \text{window\_length}/2 \rfloor$.

fftlen specifies the length of the vector on which to perform a finite Fourier transform. The value of *fftlen* must be a positive integer and must be at least as large as window_length. The computation runs fastest when *fftlen* is a power of two. The default value is the greater of 256 and window_length.

center specifies whether to center the input before any computation. The default value is 0. The following values are valid:

- **0** does not perform centering.
- **1** performs centering by subtracting the mean of the entire series from each term of the input series before performing the short-time Fourier transform. Missing values are ignored during the computation of the mean.

nthreads suggests the number of threads to use during computation. This option is not available unless you have a license for the SAS/HPF product.

The short-time Fourier transform (STFT) computations consist of multiple "local" discrete Fourier transform computations. The input time series is divided into multiple overlapping contiguous blocks, and their discrete Fourier transforms are computed in succession. The use of window functions makes the spectra smooth.

Given an input vector $x[0], x[1], \ldots, x[L-1]$ and a window $w[0], \ldots, w[m-1]$, the computation of STFT can be considered to be the evaluation of a function, $S_x(n, f)$, that measures the strength of the frequency f at time n for different values of n and f, where $S_x(n, f)$ is defined as

$$S_x(n, f) = \sum_{k=0}^{m-1} x[n+k]w[k]\exp(-\mathrm{i}2\pi kf)$$

The return value is a matrix whose rows describe the strength of a set of frequencies at different time points. Let $k = 1 + \left\lfloor \frac{\text{series\_length} - \textit{overlap}}{\text{window\_length} - \textit{overlap}} \right\rfloor$ and $S = \text{window\_length} - \textit{overlap}$. Then STFT is determined by the computation of $S_x(n, f)$ for $n = 0, S, 2S, \ldots, kS$ and for $f = 0, 1/\textit{fftlen}, \ldots, \lfloor \textit{fftlen}/2 \rfloor / \textit{fftlen}$. The complex number $S_x(n, f)$ is the Fourier coefficient that corresponds to the time index n and the frequency index f. If u represents its real component and v represents its imaginary component so that $S_x(n, f) = u + \mathrm{i}v$, then each row of the output consists of the following septuplet

$$\left(f, n, u^2 + v^2, \sqrt{u^2 + v^2}, \mathrm{ATAN2}(v, u), u, v\right)$$

The following statements demonstrate the TFSTFT function:

```
x = 1:10;
window_length=3; overlap=0; fftlen=4;
window = tfwindow(window_length);
output = tfstft(x,window,overlap,fftlen);
print output[colname={"f" "t" "Power" "Amplitude" "Phase" "Coef_Re" "Coef_Im"}];
```

Figure 25.473 Output from a Short-Time Fourier Transform

output						
f	t	Power	Amplitude	Phase	Coef_Re	Coef_Im
0	0	4	2	0	2	0
0.25	0	4	2	-1.570796	4.441E-16	-2
0.5	0	4	2	3.1415927	-2	0
0	3	25	5	0	5	0
0.25	3	25	5	-1.570796	1.11E-15	-5
0.5	3	25	5	3.1415927	-5	0
0	6	64	8	0	8	0
0.25	6	64	8	-1.570796	1.776E-15	-8
0.5	6	64	8	3.1415927	-8	0

TFWINDOW Function

TFWINDOW(*len* < , *type* > < , *params* > **);**

In signal processing, a window function is useful for smoothing spectra. The TFWINDOW function returns a window of a requested shape and length. For more information about the window functions that are implemented by the TFWINDOW function, see Harris (1978).

The input arguments are as follows:

len specifies the length of the window.

type specifies the type of the window to be created. The default is a HANNING window. The value of the *type* argument is case-insensitive, and you can abbreviate it. For the "Bartlett_Hahn" and "Blackman_Harris" windows, you must type the first nine characters. For all other windows, you must type at least the first four characters. For example, "Chebyshev" and "CHEB" specify the same window.

params specifies parameters to be used for creating windows. Not all windows require parameters. When *params* is missing, the default values of *params* are used.

The TFWINDOW function supports the following (case-insensitive) values for the *type* argument. In the following descriptions, N denotes the length of the window, and the N values that define a window are given by $w[0], \ldots, w[N-1]$. Some windows need additional parameters.

BARTLETT specifies a Bartlett window, which does not require any parameters. This window function is defined as

$$w[i] = 1 - \left| \frac{2i}{N-1} - 1 \right|, \; 0 \le i \le N-1$$

BARTLETT_HANN specifies a Bartlett-Hann window, which does not require any parameters. This window function is defined as

$$w[i] = 0.62 - 0.48 \left| \frac{i}{N-1} - 0.5 \right| - 0.38 \cos \frac{2\pi i}{N-1}, \; 0 \le i \le N-1$$

BLACKMAN specifies a Blackman window, which does not require any parameters. This window is defined as

$$w[i] = 0.42 - 0.5 \cos \frac{2\pi i}{N-1} + 0.08 \cos \frac{4\pi i}{N-1}, \; 0 \le i \le N-1$$

BLACKMAN_HARRIS specifies a Blackman-Harris window, which does not require any parameters. This window function is defined as

$$\begin{aligned} w[i] =& 0.35875 - 0.48829 \cos \frac{2\pi i}{N-1} + 0.14128 \cos \frac{4\pi i}{N-1} \\ & - 0.01168 \cos \frac{6\pi i}{N-1}, \; 0 \le i \le N-1 \end{aligned}$$

BOHMAN specifies a Bohman window, which does not require any parameters. This window function is defined as

$$\begin{aligned} w[0] =& w[N-1] = 0 \\ w[i] =& \left(1 - \left|1 - \frac{2i}{N-1}\right|\right) \cos\left(\pi \left|1 - \frac{2i}{N-1}\right|\right) \\ & + \frac{1}{\pi} \sin\left(\pi \left|1 - \frac{2i}{N-1}\right|\right), \text{ for } 1 \le i \le N-2 \end{aligned}$$

CHEBYSHEV specifies a Chebyshev window, which requires one parameter, *att*, whose default value is 100. This window is defined in terms of the nth-degree Chebyshev polynomial, T_n, which is the unique polynomial such that $T_n(\cos\theta) = \cos n\theta$ for all values of θ. $T_n(x)$ can be computed as

$$T_n(x) = \begin{cases} \cos(n\ \cos^{-1} x), & |x| \le 1 \\ \cosh(n\ \operatorname{acosh}(x)), & x > 1 \\ (-1)^n T_n(-x), & x < -1 \end{cases}$$

For odd N (say, $N = 2M + 1$) with $M > 0$, the Chebyshev window of length N can be defined as

$$w[i] = c\left(1 + \frac{2}{T_{2M}(\beta)} \sum_{k=1}^{M} T_{2M}(\beta \cos(k\pi/N)) \cos(2\pi k(i - M)/N)\right),\ 0 \le i \le N-1$$

where $\beta = \cosh(\operatorname{acosh}(10^{att/20})/(N-1))$ and c is chosen to make the largest term of w equal to 1.

For even N, the Chebyshev window of length N can be defined as

$$w[i] = c \sum_{k=0}^{N-1} (-1)^k T_{N-1}(\beta \cos(k\pi/N)) \cos(\pi k(2i + 1)/N),\ 0 \le i \le N-1$$

where c is chosen to make the largest term of w equal to 1.

FLAT_TOP specifies a flat-top window, which does not require any parameters. This window is function defined as

$$\begin{aligned} w[i] =& 0.21557895 - 0.41663158\cos\frac{2\pi i}{N-1} + 0.277263158\cos\frac{4\pi i}{N-1} \\ & - 0.083578947\cos\frac{6\pi i}{N-1} + 0.006947368\cos\frac{8\pi i}{N-1},\ 0 \le i \le N-1 \end{aligned}$$

GAUSSIAN specifies a Gaussian window, which requires one parameter, c, whose default value is 2.5. This window function is defined as

$$w[i] = \exp\left(-\frac{c^2}{2}\left(\frac{i - (N-1)/2}{(N-1)/2}\right)^2\right),\ 0 \le i \le N-1$$

HAMMING specifies a Hamming window, which does not require any parameters. This window function is defined as

$$w[i] = 0.54 - 0.46\cos\frac{2\pi i}{N-1},\ 0 \le i \le N-1$$

HANNING specifies a Hanning window, which does not require any parameters. This window function is defined as

$$w[i] = \frac{1}{2}\left(1 - \cos\frac{2\pi i}{N-1}\right),\ 0 \le i \le N-1$$

KAISER specifies a Kaiser window, which requires one parameter, β, whose default value is 0.5. This window function is defined as

$$w[i] = \frac{I_0\left(\beta\sqrt{1-(1-2i/(N-1))^2}\right)}{I_0(\beta)},\ 0 \le i \le N-1$$

where $I_0(\cdot)$ is the modified Bessel function of the first kind of order 0, which is defined as

$$I_0(x) = \sum_{m=0}^{\infty} \frac{(x/2)^{2m}}{m!^2}$$

PARZEN specifies a Parzen window, which does not require any parameters. This window function is defined as

$$w[i] = \begin{cases} 2\left(1-\dfrac{|2i-(N-1)|}{N}\right)^3, & 0 \le i < \dfrac{N-1}{4} \\ 1-6\left(\dfrac{|2i-(N-1)|}{N}\right)^2+6\left(\dfrac{|2i-(N-1)|}{N}\right)^3, & \dfrac{N-1}{4} \le i \le \dfrac{N-1}{2} \\ W[N-i-1], & \dfrac{N-1}{2} < i \le N-1 \end{cases}$$

The last half of the window is defined by symmetry, which implies $w[i] = w[N-i-1]$ for $0 \le i \le N-1$.

RECTANGULAR specifies a rectangular window, which does not require any parameters. This window function is defined as

$$w[i] = 1,\ 0 \le i \le N-1$$

TUKEY specifies a Tukey window, which requires one parameter, α, whose default value is 0.5. Let $\epsilon = 10^{-12}$. If $\alpha \ge 1$, then a Hanning window is returned; if $\alpha \le \epsilon$, a rectangular window is returned. For $\epsilon < \alpha < 1$, this window function is defined as

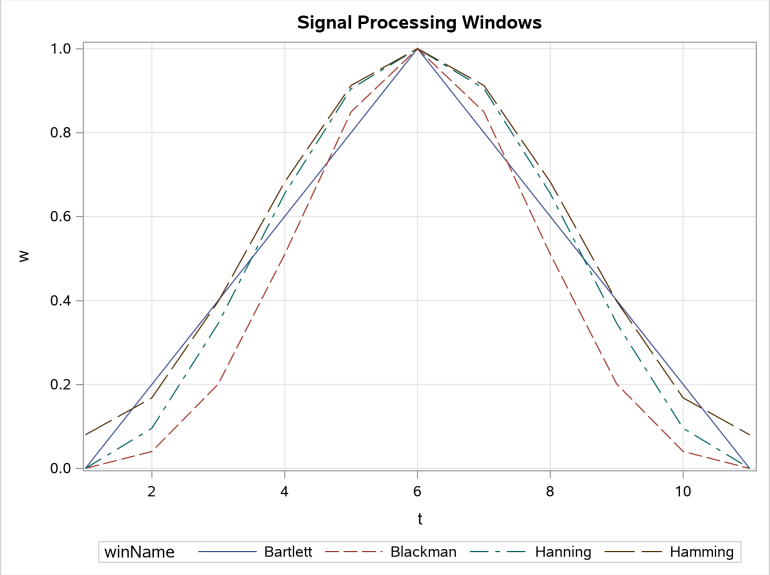
$$w[i] = \begin{cases} \dfrac{1}{2}\left(1+\cos\left(\dfrac{\pi}{\alpha}\left(\dfrac{2i}{N-1}-\alpha\right)\right)\right), & 0 \le i < \alpha(N-1)/2 \\ 1, & \alpha(N-1)/2 \le i \le (N-1)/2 \\ w[N-i-1], & (N-1)/2 < i <= N-1 \end{cases}$$

The last half of the window is defined by symmetry.

The following statements demonstrate the TFWINDOW function by generating and plotting four different windows:

```
names = {"Bartlett" "Blackman" "Hanning" "Hamming"};
len = 11;
window = j(len, ncol(names));
do j = 1 to ncol(names);
   window[,j] = tfwindow(len, names[j]);
end;
run WideToLong(t, w, winName,
               window, , names);
title "Signal Processing Windows";
call series(t, w) group=winName grid={X Y};
```

Figure 25.474 Window Functions for Signal Processing



TOEPLITZ Function

TOEPLITZ(*a*);

The TOEPLITZ function generates a Toeplitz matrix from a vector, or a block Toeplitz matrix from a matrix. A block Toeplitz matrix has the property that all matrices on the diagonals are the same. The argument *a* is an $(np) \times p$ or $p \times (np)$ matrix; the value returned is the $(np) \times (np)$ result.

The TOEPLITZ function uses the first $p \times p$ submatrix, $\mathbf{A}_1$, of the argument matrix as the blocks of the main diagonal. The second $p \times p$ submatrix, $\mathbf{A}_2$, of the argument matrix forms one secondary diagonal, with the transpose $\mathbf{A}_2'$ forming the other. The remaining diagonals are formed accordingly. If the first $p \times p$ submatrix of the argument matrix is symmetric, the result is also symmetric. If $\mathbf{A}$ is $(np) \times p$, the first p columns of the returned matrix, $\mathbf{R}$, are the same as $\mathbf{A}$. If $\mathbf{A}$ is $p \times (np)$, the first p rows of $\mathbf{R}$ are the same as $\mathbf{A}$.

The TOEPLITZ function is especially useful in time series applications, where the covariance matrix of a set of variables with its lagged set of variables is often assumed to be a block Toeplitz matrix.

If

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2|\mathbf{A}_3|\cdots|\mathbf{A}_n]$$

and if $\mathbf{R}$ is the matrix formed by the TOEPLITZ function, then

$$\mathbf{R} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2' & \mathbf{A}_1 & \mathbf{A}_2 & \cdots & \mathbf{A}_{n-1} \\ \mathbf{A}_3' & \mathbf{A}_2' & \mathbf{A}_1 & \cdots & \mathbf{A}_{n-2} \\ \vdots & & & & \\ \mathbf{A}_n' & \mathbf{A}_{n-1}' & \mathbf{A}_{n-2}' & \cdots & \mathbf{A}_1 \end{bmatrix}$$

If

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

and if **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:

```
r1 = toeplitz(1:5);
r2 = toeplitz({1 2 ,
               3 4 ,
               5 6 ,
               7 8});
r3 = toeplitz({1 2 3 4,
               5 6 7 8});
print r1, r2, r3;
```

Figure 25.475 Toeplitz Matrices

r1
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

r2
1 2 5 7
3 4 6 8
5 6 1 2
7 8 3 4

r3
1 2 3 4
5 6 7 8
3 7 1 2
4 8 5 6

TPSPLINE Call

CALL TPSPLINE(*fitted, coeff, adiag, gcv, x, y* **<** *, lambda* **>** **);**

The TSPLINE subroutine fits a thin-plate smoothing spline (TPSS) to data. The generalized cross validation (GCV) function is used to select the smoothing parameter.

The TPSPLINE subroutine returns the following values:

fitted is an $n \times 1$ vector of fitted values of the TPSS fit evaluated at the design points x. The n is the number of observations. The final TPSS fit depends on the optional *lambda*.

coeff is a vector of spline coefficients. The vector contains the coefficients for basis functions in the null space and the representer of evaluation functions at unique design points. (see Wahba (1990) for more detail on reproducing kernel Hilbert space and representer of evaluation functions.) The length of *coeff* vector depends on the number of unique design points and the number of variables in the spline model. In general, let *nuobs* and k be the number of unique rows and the number of columns of x respectively. The length of *coeff* equals to k + nuobs + 1. The *coeff* vector can be used as an input to the TPSPLNEV subroutine to evaluate the resulting TPSS fit at new data points.

adiag is an $n \times 1$ vector of diagonal elements of the "hat" matrix. See the "Details" section.

gcv If *lambda* is not specified, then *gcv* is the minimum value of the GCV function. If *lambda* is specified, then *gcv* is a vector (or scalar if *lambda* is a scalar) of GCV values evaluated at the *lambda* points. It provides you with both the ability to study the GCV curves by plotting *gcv* against *lambda* and the chance to identify a possible local minimum.

The input arguments to the TPSPLINE subroutine are as follows:

x is an $n \times k$ matrix of design points on which the TPSS is to be fit. The k is the number of variables in the spline model. The columns of x need to be linearly independent and contain no constant column.

y is the $n \times 1$ vector of observations.

lambda is a optional $q \times 1$ vector that contains λ values in $\log_{10}(n\lambda)$ scale. If *lambda* is not specified (or *lambda* is specified and $q > 1$) the GCV function is used to choose the "best" λ and the returning *fitted* values are based on the λ that minimizes the GCV function. If *lambda* is specified and $q = 1$, no minimization of the GCV function is involved and the *fitted*, *coeff* and *adiag* values are all based on the TPSS fit that uses this particular *lambda*.

Aside from the values returned, the TPSPLINE subroutine also prints other useful information such as the number of unique observations, the dimensions of the null space, the number of parameters in the model, a GCV estimate of λ, the smoothing penalty, the residual sum of square, the trace of $(I - A(\lambda))$, an estimate of σ^2, and the sum of squares for replication.

No missing values are accepted within the input arguments. Also, you should use caution if you want to specify small *lambda* values. Since the true $\lambda = (10^{\log_{10} lambda})/n$, a very small value for *lambda* can cause λ to be smaller than the magnitude of machine error and usually the returned *gcv* values from such a λ cannot be trusted. Finally, when using TPSPLINE be aware that TPSS is a computationally intensive method.

Therefore a large data set (that is, a large number of unique design points) will take a lot of computer memory and time.

For convenience, the TPSS method is illustrated with a two-dimensional independent variable $\mathbf{X} = (x^1, x^2)$. More details can be found in Wahba (1990), or in Bates et al. (1987).

Assume that the data are from the model

$$y_i = f(x_i) + \epsilon_i,$$

where $(x_i, y_i), i = 1, \ldots, n$ are the observations. The function f is unknown and you assume that it is reasonably smooth. The error terms $\epsilon_i, i = 1, \ldots, n$ are independent zero-mean random variables.

You measure the smoothness of f by the integral over the entire plane of the square of the partial derivatives of f of total order 2, that is

$$J_2(f) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left[\frac{\partial^2 f}{\partial {x_1}^2}\right]^2 + 2\left[\frac{\partial^2 f}{\partial x_1 \partial x_2}\right]^2 + \left[\frac{\partial^2 f}{\partial {x_2}^2}\right]^2 dx_1 dx_2$$

Using this as a smoothness penalty, the thin-plate smoothing spline estimate f_λ of f is the minimizer of

$$S_\lambda(f) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda J_2(f).$$

Duchon (1976) derived that the minimizer f_λ can be represented as

$$f_\lambda(x) = \sum_{i=1}^{3}\beta_i \phi_i(x) + \sum_{i=1}^{n}\delta_i E_2(x - x_i),$$

where $(\phi_1(x), \phi_2(x), \phi_3(x)) = (1, x^1, x^2)$ and $E_2(s) = \frac{1}{2^3\pi}\|s\|^2 ln(\|s\|)$.

Let matrix $\mathbf{K}$ have entries $(\mathbf{K})_{ij} = E_2(x_i - x_j)$ and matrix $\mathbf{T}$ have entries $(\mathbf{T})_{ij} = \phi_j(x_i)$. Then the minimization problem can be rewritten as finding coefficients β and δ to minimize

$$S_\lambda(\beta, \delta) = \frac{1}{n}\|y - \mathbf{T}\beta - \mathbf{K}\delta\|^2 + \lambda\delta^T\mathbf{K}\delta$$

The final TPSS fits can be viewed as a type of generalized ridge regression estimator. The λ is called the smoothing parameter, which controls the balance between the goodness of fit and the smoothness of the final estimate. The smoothing parameter can be chosen by minimizing the generalized cross validation function (GCV). If you write

$$\hat{y} = \mathbf{A}(\lambda)y$$

and call $\mathbf{A}(\lambda)$ as the "hat" matrix, the GCV function $V(\lambda)$ is defined as

$$V(\lambda) = \frac{(1/n)\|(\mathbf{I} - \mathbf{A}(\lambda)y\|^2}{[(1/n)\mathrm{tr}(\mathbf{I} - \mathbf{A}(\lambda))]^2}$$

The returned values from this function call provide the $\hat{y}$ as *fitted*, the (β, δ) as *coeff*, and $\mathrm{diag}(A(\lambda))$ as *adiag*.

To evaluate the TPSS fit $f_\lambda(x)$ at new data points, you can use the TPSPLNEV call.

Suppose $\mathbf{X}^{\text{new}}$, a $m \times k$ matrix, contains the m new data points at which you want to evaluate f_λ. Let $(\mathbf{T}^{\text{new}}_{ij}) = \phi_j(x^{\text{new}}_i)$ and $(\mathbf{K}^{\text{new}}_{ij}) = E_2(x^{\text{new}}_i - x_j)$ be the (i, j) elements of $\mathbf{T}^{\text{new}}$ and $\mathbf{K}^{\text{new}}$ respectively. The prediction at new data points $\mathbf{X}^{\text{new}}$ is

$$y_{\text{pred}} = T^{\text{new}}\beta + K^{\text{new}}\delta$$

Therefore, the y_{pred} can be easily evaluated by using the coefficient (β, δ) obtained from the TPSPLINE call.

An example is given in the documentation for the TPSPLNEV call.

TPSPLNEV Call

CALL TPSPLNEV(*pred, xpred, x, coeff***);**

The TPSPLNEV subroutine evaluates the thin-plate smoothing spline (TPSS) at new data points. It is used after the TPSPLINE subroutine fits a thin-plate spline model to data.

The TPSPLNEV subroutine returns the following value:

pred is an $m \times 1$ vector of the predicated values of the TPSS fit evaluated at m new data points.

The input arguments to the TPSPLNEV subroutine are as follows:

xpred is an $m \times k$ matrix of data points at which the f_λ is evaluated, where m is the number of new data points and k is the number of variables in the spline model.

x is an $n \times k$ matrix of design points that is used as an input of TPSPLINE call.

coeff is the coefficient vector returned from the TPSPLINE call.

See the previous section on the TPSPLINE call for details about the TSPLNEV subroutine.

The following example contains two independent variables and one response variable. The first panel of Figure 25.477 shows a plot of the data. The following statements define the data and a sequence of λ values within the interval $(-3.8, -3.3)$. The TPSPLINE call fits the thin-plate smoothing spline on those design points and computes the GCV function for each value of λ within the interval.

```
x={ -1.0 -1.0,    -1.0 -1.0,    -0.5 -1.0,    -0.5 -1.0,
     0.0 -1.0,     0.0 -1.0,     0.5 -1.0,     0.5 -1.0,
     1.0 -1.0,     1.0 -1.0,    -1.0 -0.5,    -1.0 -0.5,
    -0.5 -0.5,    -0.5 -0.5,     0.0 -0.5,     0.0 -0.5,
     0.5 -0.5,     0.5 -0.5,     1.0 -0.5,     1.0 -0.5,
    -1.0  0.0,    -1.0  0.0,    -0.5  0.0,    -0.5  0.0,
     0.0  0.0,     0.0  0.0,     0.5  0.0,     0.5  0.0,
     1.0  0.0,     1.0  0.0,    -1.0  0.5,    -1.0  0.5,
    -0.5  0.5,    -0.5  0.5,     0.0  0.5,     0.0  0.5,
     0.5  0.5,     0.5  0.5,     1.0  0.5,     1.0  0.5,
    -1.0  1.0,    -1.0  1.0,    -0.5  1.0,    -0.5  1.0,
     0.0  1.0,     0.0  1.0,     0.5  1.0,     0.5  1.0,
     1.0  1.0,     1.0  1.0 };
```

```
y = {15.54, 15.76, 18.67, 18.50, 19.66, 19.80, 18.60, 18.52,
     15.87, 16.04, 10.92, 11.14, 14.81, 14.83, 16.56, 16.44,
     14.91, 15.06, 10.92, 10.94,  9.61,  9.65, 14.03, 14.03,
     15.77, 16.00, 14.00, 14.03,  9.56,  9.58, 11.21, 11.09,
     14.84, 14.99, 16.55, 16.51, 14.98, 14.72, 11.15, 11.17,
     15.83, 15.96, 18.64, 18.56, 19.54, 19.81, 18.57, 18.61,
     15.87, 15.90 };

lambda = T( do(-3.8, -3.3, 0.1) );
call tpspline(fit, coef, adiag, gcv, x, y, lambda);
```

Figure 25.476 Output from the TPSPLINE Subroutine

SUMMARY OF TPSPLINE CALL

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	6.6006262E-6
Smoothing Penalty	2558.7692004
Residual Sum of Squares	0.2434454159
Trace of (I-A)	25.402044435
Sigma^2 Estimate	0.0095836938
Sum of Squares for Replication	0.23965

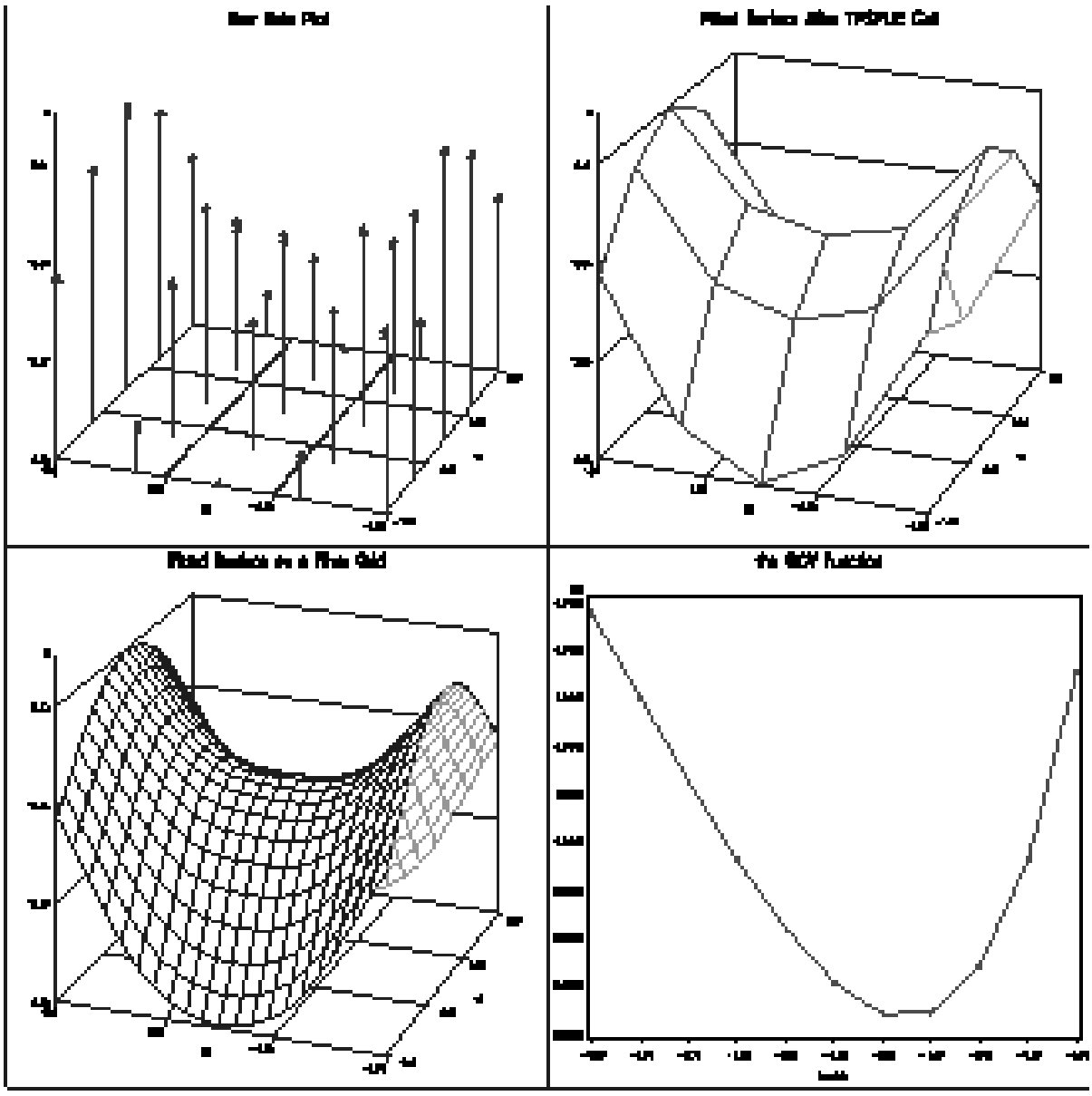
The TPSPLINE call returns the fitted values at each design point. The fitted surface is plotted in the second panel of Figure 25.477. The fourth panel shows a plot of the GCV function values against *lambda*.

You can use the TPSPLNEV call to score the thin-plate spline at a new set of points. The following statements generate a dense grid on $[-1, 1] \times [-1, 1]$. The **x** and **coef** matrices are used to evaluate the thin-plate spline on the new grid of points:

```
xpred = ExpandGrid( do(-1,1,0.1), do(-1,1,0.1));
call tpsplnev(pred, xpred, x, coef);
```

The third panel of Figure 25.477 shows the thin-plat spline evaluated on the grid of points.

Figure 25.477 Plots of Fitted Surface



TRACE Function

TRACE(*matrix***);**

The TRACE function returns the sum of the diagonal elements of *matrix*, as shown in the following example:

```
a = trace({5 2,
           1 3});
print a;
```

Figure 25.478 Trace of a Matrix



TRISOLV Function

TRISOLV(*form, R, b* **<** *, piv* **>);**

The TRISOLV function efficiently solves linear systems that involve a triangular matrix.

The TRISOLV function returns the $n \times p$ matrix $\mathbf{X}$ that contains p solutions of the p linear systems specified by *form*, *R*, and *b*.

The arguments to the TRISOLV function are as follows:

form specifies which of the following form of a triangular linear system is to be solved:

form=1	solve $\mathbf{R}x = b$, $\mathbf{R}$ upper triangular
form=2	solve $\mathbf{R}'x = b$, $\mathbf{R}$ upper triangular
form=3	solve $\mathbf{R}'x = b$, $\mathbf{R}$ lower triangular
form=4	solve $\mathbf{R}x = b$, $\mathbf{R}$ lower triangular

R specifies the $n \times n$ nonsingular upper (*form*=1,2) or lower (*form*=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 $b_k, k = 1 \ldots p$.

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 $\mathbf{A}_{\text{piv}[1]}, \ldots, \mathbf{A}_{\text{piv}[n]}$.

For *form*=1 and *form*=3, the solution is obtained by backward elimination. For *form*=2 and *form*=4, the solution is obtained by forward substitution.

If TRISOLV recognizes the upper or lower triangular matrix $\mathbf{R}$ as a singular matrix (that is, one that contains at least one zero diagonal element), it exits with an error message.

Consider the following example:

```
R = { 1  0  0  0,
      3  2  0  0,
      1 -3  5  0,
      2  7  9 -1 };

b = {1, 1, 4, -6 };
x = trisolv(4, R, b);
print x;
```

Figure 25.479 Solution of a Triangular System

x
1
-1
0
1

Also see the example in section "The Full-Rank Linear Least Squares Problem" on page 903.

TSBAYSEA Call

CALL TSBAYSEA(*trend, season, series, adjust, abic, data* < ,*order*> < , *sorder*> < , *rigid*> < , *npred*> < , *opt*> < , *cntl*> < , *print*> **);**

The TSBAYSEA subroutine performs Bayesian seasonal adjustment modeling.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSDECOMP Call

CALL TSDECOMP(*comp, est, aic, data,* < , *xdata*> < , *order*> < , *sorder*> < , *nar*> < , *npred*> < , *init*> < , *opt*> < , *icmp*> < , *print*> **);**

The TSDECOMP subroutine analyzes nonstationary time series by using smoothness priors modeling.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSMLOCAR Call

CALL TSMLOCAR(*arcoef, ev, nar, aic, start, finish, data* < , *maxlag*> < , *opt*> < , *missing*> < , *print*> **);**

The TSMLOCAR subroutine analyzes nonstationary or locally stationary time series by using the minimum AIC procedure.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSMLOMAR Call

CALL TSMLOMAR(*arcoef, ev, nar, aic, start, finish, data* < , *maxlag*> < , *opt*> < , *missing*> < , *print*> **);**

The TSMLOMAR subroutine analyzes nonstationary or locally stationary multivariate time series by using the minimum AIC procedure.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSMULMAR Call

CALL TSMULMAR(*arcoef, ev, nar, aic, data* < **,** *maxlag*> < **,** *opt*> < **,** *missing*> < **,** *print*> **);**

The TSMULMAR subroutine estimates VAR processes by using the minimum AIC procedure.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSPEARS Call

CALL TSPEARS(*arcoef, ev, nar, aic, data* < **,** *maxlag*> < **,** *opt*> < **,** *missing*> < **,** *print*> **);**

The TSPEARS subroutine analyzes periodic AR models with the minimum AIC procedure.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSPRED Call

CALL TSPRED(*forecast, impulse, mse, data, coef, nar, nma* < **,** *ev*> < **,** *npred*> < **,** *start*> < **,** *constant*> **);**

The TSPRED subroutine provides predicted values of univariate and multivariate ARMA processes when the ARMA coefficients are input.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSROOT Call

CALL TSROOT(*matout, matin, nar, nma,* < **,** *qcoef*> < **,** *print*> **);**

The TSROOT subroutine computes AR and MA coefficients from the characteristic roots of the model or computes the characteristic roots of the model from the AR and MA coefficients.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSTVCAR Call

CALL TSTVCAR(*arcoef, variance, est, aic, data* < **,** *nar*> < **,** *init*> < **,** *opt*> < **,** *outlier*> < **,** *print*> **);**

The TSTVCAR subroutine analyzes time series that are nonstationary in the covariance function.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TSUNIMAR Call

CALL TSUNIMAR(*arcoef, ev, nar, aic, data* < , *maxlag*> < , *opt*> < , *missing*> < , *print*> **);**

The TSUNIMAR subroutine determines the order of an AR process with the minimum AIC procedure and estimates the AR coefficients.

This function is deprecated. See the SAS/IML 9.3 documentation for syntax and examples.

TYPE Function

TYPE(*matrix***);**

The TYPE function returns a single character value that represents the type of a matrix. The value is 'N' if the type of the matrix is numeric; it is 'C' if the type of the matrix is character; it is 'U' if the matrix does not have a value.

The following statements determine the type for three different matrices:

```
cMat = {"Rick" "Nancy"};
t1 = type(cMat);
nMat = {3.14159 2.71828};
t2 = type(nMat);
free uMat;
t3 = type(uMat);
print t1 t2 t3;
```

Figure 25.480 The Types of Matrices

t1	t2	t3
C	N	U

UNION Function

UNION(*matrix1* < , *matrix2, . . . , matrix15*> **);**

The UNION function returns a row vector that contains the sorted set of unique values of the arguments. If the matrices are thought of as sets, the return value is the union of the sets. If you call the UNION function with a single argument, the function returns the sorted elements with no duplicates.

There can be up to 15 arguments, which can be either all character or all numeric. For character arguments, the element length of the result is the longest element length of the arguments. Shorter character elements are padded on the right with blanks.

This function is identical to the UNIQUE function.

The following statements compute the union of the elements in two matrices:

```
a = {1 2 4 5};
b = {3 4};
c = union(a, b);
print c;
```

Figure 25.481 Union of Elements

c
1 2 3 4 5

UNIQUE Function

UNIQUE(*matrix1* < *, matrix2, . . . , matrix15* > **);**

The UNIQUE function returns a row vector that contains the sorted set of unique values of the arguments. If you call the UNIQUE function with a single argument, the function returns the sorted elements with no duplicates.

This function is identical to the UNION function, the description of which includes an example.

UNIQUEBY Function

UNIQUEBY(*matrix* < *, by* > < *, index* > **);**

The UNIQUEBY function returns the locations of the unique BY-group combinations for a sorted or indexed matrix. The arguments to the UNIQUEBY function are as follows:

matrix is the input matrix, which must be sorted or indexed according to the *by* columns.

by is either a numeric matrix of column numbers, or a character matrix that contains the names of columns that correspond to column labels assigned to *matrix* by a MATTRIB statement or READ statement. If *by* is not specified, then the first column is used.

index is a vector such that *index*[*i*] is the row index of the *i*th element of *matrix* when sorted according to *by*. Consequently, *matrix*[*index*,] is the sorted matrix. *index* can be computed for a matrix and a given set of *by* columns with the SORTNDX call. If the matrix is known to be sorted according to the *by* columns already, then *index* should be 1:nrow(*matrix*). In this case, you can also omit the *index* argument.

The UNIQUEBY function returns a column vector whose *i*th row is the row in *index* whose value is the row in *matrix* of the *i*th unique combination of values in the *by* columns.

For example, the following statements use the SORTNDX subroutine to create a sort index for a matrix. The UNIQUEBY function is then used to determine the unique combinations of the columns of the matrix:

```
m = { 1 0,
      2 0,
      2 2,
      2 0,
```

```
      1 0,
      2 0,
      1 1 };
cols = 1:2;
call sortndx(ndx, m, cols);

sorted = m[ndx,];
unique_rows = uniqueby(m, cols, ndx);
unique_vals = m[ndx[unique_rows], cols ];
print sorted, unique_rows unique_vals;
```

Figure 25.482 Unique Values of the Sort Variables

sorted	
1	0
1	0
1	1
2	0
2	0
2	0
2	2

unique_rows	unique_vals
1	1 0
3	1 1
4	2 0
7	2 2

In addition, the following statements compute the number of unique values and the number of elements in each BY-group:

```
n = nrow(unique_rows);
size = j(n,1);
do i = 1 to n-1;
   size[i] = unique_rows[i+1] - unique_rows[i];
end;
size[n] = nrow(m) - unique_rows[n] + 1;
print n, size;
```

Figure 25.483 Number of BY Groups and Number of Elements in Each Group

n
4

size
2
1
3
1

If *matrix* is already sorted according to the *by* columns (see the SORT call), then UNIQUEBY can be called with 1:nrow(*matrix*) for the *index* argument, or the last argument can be omitted as shown in the following

statement:

```
unique_loc = uniqueby(sorted, cols);
print unique_loc;
```

Figure 25.484 Position of Unique Rows for a Sorted Matrix

unique_loc
1
3
4
7

USE Statement

USE *SAS-data-set* < **VAR** *operand* > < **WHERE***(expression)* > < **NOBS** *name* > **;**

The USE statement opens a SAS data set for reading.

The arguments to the USE statement are as follows:

SAS-data-set can be specified with a one-level name (for example, A) or a two-level name (for example, Sasuser.A). You can also specify an expression (enclosed in parentheses) that resolves to the name of a SAS data set.

operand specifies a set of variables. As described in the section "Select Variables with the VAR Clause" on page 105, you can specify variable names by using a matrix literal, a character matrix, an expression, or the _ALL_, _CHAR_, or _NUM_ keywords.

expression specifies a criterion by which certain observations are selected. If the WHERE clause is omitted, no subsetting occurs. The optional WHERE clause conditionally selects observations that are contained within the *range* specification. For details about the WHERE clause, see the section "Process Data by Using the WHERE Clause" on page 106.

name specifies a variable to contain the number of observations. The NOBS clause returns the total number of observations in the data set in the variable *name*.

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 and WHERE clauses are optional, and you can specify them in any order. If a data set was previously open, all the data set options are still in effect. To override any old options, the new USE statement must explicitly specify new options.

The following examples demonstrate various options of the USE statement:

```
use Sashelp.Class;
use Sashelp.Class var{name sex age};
use Sashelp.Class var{name sex age} where(age>10);
```

The data sets can be specified with a literal value as in the previous example, or with an expression (enclosed in parentheses) that resolves to the name of a SAS data set, as shown in the following statements:

```
f = "Sashelp.Class";
use (f);   /* expression */
read all var _NUM_ into X;
close (f);
```

VALSET Call

CALL VALSET(*name, value***);**

The VALSET subroutine performs indirect assignment. The subroutine takes the name of a matrix and assigns a value to that matrix. Calling the VALSET subroutine is useful for assigning values to a matrix whose name is not known until run time.

The C programming language has the concept of a "pointer," which enables you to assign values to allocated memory. The VALSET subroutine is similar. The *matrix* argument contains the name of the matrix to which the *value* is to be assigned.

The arguments to the VALSET subroutine are as follows:

matrix	is a character matrix or literal that specifies the name of a matrix.
value	is a value to which the matrix is set.

For example, the following statements assign the string "A" to the value of the matrix **B**. The VALSET subroutine assigns a vector to the matrix **A**.

```
B = "A";
call valset(B, 1:5);
print A;
```

Figure 25.485 Indirect Assignment

A
1 2 3 4 5

The following statement redefines the contents of **B**; it does not change the value of **A**.

```
b = 99;
```

See also the VALUE function, which retrieves the value that is contained in a matrix.

VALUE Function

VALUE(*name***);**

The VALUE function assigns values by indirect reference. The function takes the name of a matrix and returns the value of that matrix. The VALUE function is useful for retrieving values from a matrix whose name is not known until run time.

The C programming language has the concept of a "pointer," which enables you to assign values to allocated memory. The VALUE function is similar. The *name* argument contains the name of the matrix from which the *value* is to be retrieved.

For example, the following statements return the values that are contained in the variable **A**:

```
a = {1 2 3};
b = "A";      /* points to A */
c = value(b); /* returns the value of A */
print c;
```

Figure 25.486 Value of an Indirect Reference

c
1 2 3

You can use the VALUE function in a loop to extract the values of several matrices that have different sizes and shapes, as shown in the following example:

```
x = {1 2 3};
y = {9 6 10 5};
z = {5 5, 10 0};
name = {"x" "y" "z"};
sums = j(1, ncol(name)); /* allocate space for result */
do i = 1 to ncol(name);
   sums[i] = sum( value(name[i]) ); /* sum(x), sum(y), and sum(z) */
end;
print sums;
```

Figure 25.487 Sums of Matrices

sums
6 30 20

See also the VALSET subroutine, which performs indirect assignment of matrices.

VAR Function

VAR(*x***);**

The VAR function computes a sample variance of data.

The arguments to the VAR function are as follows:

x specifies an $n \times p$ numerical matrix. The VAR function computes the variance of the p columns of this matrix.

The VAR function computes the sample variance of a column vector *x* as $\Sigma_{i=1}^{n}(x_i - \bar{x})^2/(n-1)$ where n is the number of nonmissing values of *x* and any missing values have been excluded. When *x* is a matrix, the sample variance is computed for each column, as the following example shows:

```
x = {5 1 10,
     6 2 3,
     6 8 5,
     6 7 9,
     7 2 13};
var = var(x);
print var;
```

Figure 25.488 Variance of Columns

var		
0.5	10.5	16

The following statement computes the standard deviation of each column:

```
sd = sqrt(var(x));
```

The VAR function returns a missing value for columns with fewer than two nonmissing observations.

VARMACOV Call

CALL VARMACOV(*cov, phi, theta, sigma* < , *p*> < , *q*> < , *lag*> **);**

The VARMACOV subroutine computes the theoretical cross-covariance matrices for a stationary VARMA(p, q) model.

The input arguments to the VARMACOV subroutine are as follows:

phi specifies a $km_p \times k$ matrix, Φ, that contains the autoregressive coefficient matrices, where m_p is the number of elements in the subset of the AR order and $k \geq 2$ is the number of variables. All the roots of $|\Phi(B)| = 0$ should be greater than one in absolute value, where $\Phi(B)$ is the finite order matrix polynomial in the backshift operator B, such that $B^j \mathbf{y}_t = \mathbf{y}_{t-j}$. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

sigma specifies a $k \times k$ symmetric positive-definite covariance matrix of the innovation series. If *sigma* is not specified, then an identity matrix is used.

p specifies the subset of the AR order. The quantity m_p is defined as the ratio nrow(*phi*) / ncol(*phi*) where nrow(*phi*) is the number of rows of the matrix *phi* and ncol(*phi*) is the number of columns.

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 the ratio nrow(*theta*) / ncol(*theta*) where nrow(*theta*) is the number of rows of matrix *theta* and ncol(*theta*) is the number of columns.

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 $k(\text{lag} + 1) \times k$ matrix that contains the theoretical cross-covariance matrices of the VARMA(p, q) model.

Consider the following bivariate ($k = 2$) VARMA(1,1) model:

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t - \Theta \boldsymbol{\epsilon}_{t-1}$$

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To compute the cross-covariance matrices of this model, you can use the following statements:

```
phi  = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
sigma= { 1.0  0.5, 0.5 1.25};
call varmacov(cov, phi, theta, sigma) lag=3;
Lag = {"0", "", "1", "", "2", "", "3", ""};
print Lag cov;
```

Figure 25.489 Cross-Covariance Matrix

Lag	cov	
0	12.403036	8.4702334
	8.4702334	9.0377769
1	11.098527	9.3828916
	5.5703916	6.8934731
2	8.626786	9.4739834
	3.2377334	5.4102769
3	5.6151515	8.0182666
	1.1801416	3.5657231

VARMALIK Call

CALL VARMALIK(*lnl, series, phi, theta, sigma* < , *p*> < , *q*> < , *opt*> **);**

The VARMALIK subroutine computes the log-likelihood function for a VARMA(p, q) model.

The input arguments to the VARMALIK subroutine are as follows:

series specifies an $n \times k$ matrix that contains the vector time series (assuming mean zero), where n is the number of observations and $k \geq 2$ is the number of variables.

phi specifies a $km_p \times k$ matrix that contains the autoregressive coefficient matrices, where m_p is the number of the elements in the subset of the AR order. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

sigma specifies a $k \times k$ covariance matrix of the innovation series. If you do not specify *sigma*, an identity matrix is used.

p specifies the subset of the AR order. See the VARMACOV subroutine.

q specifies the subset of the MA order. See the VARMACOV subroutine.

opt specifies the method of computing the log-likelihood function:

- *opt=0* requests the multivariate innovations algorithm. This algorithm requires that the time series is stationary and does not contain missing observations.
- *opt=1* requests the conditional log-likelihood function. This algorithm requires that the number of the observations in the time series must be greater than *p*+*q* and that the series does not contain missing observations.
- *opt=2* requests the Kalman filtering algorithm. This is the default and is used if the required conditions in *opt=0* and *opt=1* are not satisfied.

The VARMALIK subroutine returns the following value:

lnl is a 3×1 matrix that contains the log-likelihood function, the sum of log determinant of the innovation variance, and the weighted sum of squares of residuals. The log-likelihood function is computed as $-0.5\times$ *(the sum of last two terms)*.

The options *opt=0* and *opt=2* are equivalent for stationary time series without missing values. Setting *opt=0* is useful for a small number of the observations and a high order of *p* and *q*; *opt=1* is useful for a high order of *P* and *q*; *opt=2* is useful for a low order of *p* and *q*, or for missing values in the observations.

Consider the following bivariate ($k = 2$) VARMA(1,1) model:

$$\mathbf{y}_t = \Phi \mathbf{y}_{t-1} + \boldsymbol{\epsilon}_t - \Theta \boldsymbol{\epsilon}_{t-1}$$

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To compute the log-likelihood function of this model, you can use the following statements:

```
phi  = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
sigma= { 1.0  0.5, 0.5 1.25};
call varmasim(yt, phi, theta) sigma=sigma seed=123;
call varmalik(lnl, yt, phi, theta, sigma);
labl = {"LogLik", "SumLogDet", "SSE"};
print lnl[rowname=labl];
```

Figure 25.490 Log-Likelihood Components

	lnl
LogLik	-85.50804
SumLogDet	4.8529601
SSE	166.16313

VARMASIM Call

CALL VARMASIM(*series, phi, theta, mu, sigma, n* < , *p*> < , *q*> < , *initial*> < , *seed*> **);**

The VARMASIM subroutine generates a VARMA(p,q) time series.

The input arguments to the VARMASIM subroutine are as follows:

phi specifies a $km_p \times k$ matrix that contains the autoregressive coefficient matrices, where m_p is the number of the elements in the subset of the AR order and $k \geq 2$ is the number of variables. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

mu specifies a $k \times 1$ (or $1 \times k$) mean vector of the series. If *mu* is not specified, a zero vector is used.

sigma specifies a $k \times k$ covariance matrix of the innovation series. If *sigma* is not specified, an identity matrix is used.

n specifies the length of the series. If *n* is not specified, $n = 100$ is used.

p specifies the subset of the AR order. See the VARMACOV subroutine.

q specifies the subset of the MA order. See the VARMACOV subroutine.

initial specifies the initial values of random variables. You can specify this argument in two ways:

- A scalar value, a_0, assigns $\mathbf{y}_{-p+1} = \ldots = \mathbf{y}_0 = a_0$ and $\boldsymbol{\epsilon}_{-q+1} = \ldots = \boldsymbol{\epsilon}_0 = a_0$.
- A $(p+q) \times k$ matrix value, $\mathbf{M}$, assigns values as follows. The first p rows of $\mathbf{M}$ specify the values for $\mathbf{y}_{-p+1}, \ldots, \mathbf{y}_0$. The next q rows of $\mathbf{M}$ specify the values for $\boldsymbol{\epsilon}_{-q+1}, \ldots, \boldsymbol{\epsilon}_0$. That is, the first row of $\mathbf{M}$ sets the value for $\mathbf{y}_{-p+1}$, the second row sets the value for $\mathbf{y}_{-p+2}$, and so on. Similarly, the $(p+1)$st row of $\mathbf{M}$ sets the value for $\boldsymbol{\epsilon}_{-q+1}$, the $(p+2)$nd row sets the value for $\boldsymbol{\epsilon}_{-q+2}$, and so on.

By default, the initial values are estimated for the stationary vector time series and the initial values are assumed to be zero for the nonstationary vector time series.

seed is a scalar that contains the random number seed. At the first execution of the subroutine, the seed variable is used as follows:

- If *seed* > 0, the input seed is used for generating the series.
- If *seed* = 0, the system clock is used to generate the seed.
- If *seed* < 0, the value $(-1)\times(\textit{seed})$ is used for generating the series.
- If the seed is not supplied, the system clock is used to generate the seed.

On subsequent calls to the subroutine, if *seed* > 0, the seed remains unchanged. Otherwise, the current seed is updated internally after each call.

The VARMASIM subroutine returns the following value:

series is an $n \times k$ matrix that contains the generated VARMA(p, q) time series. When either the *initial* option is specified or zero initial values are used, these initial values are not included in *series*.

Consider the following bivariate ($k = 2$) stationary VARMA(1,1) time series:

$$\mathrm{y}_t - \boldsymbol{\mu} = \Phi(\mathrm{y}_{t-1} - \boldsymbol{\mu}) + \boldsymbol{\epsilon}_t - \Theta\boldsymbol{\epsilon}_{t-1}$$

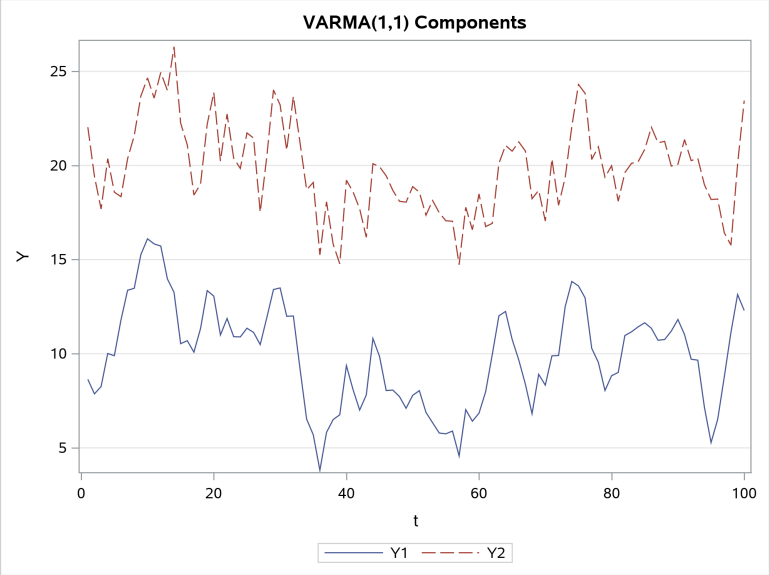
$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix} \quad \boldsymbol{\mu} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \quad \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To generate this series, you can use the following statements:

```
phi  = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
mu   = { 10, 20 };
sigma= { 1.0  0.5, 0.5 1.25};
call varmasim(yt, phi, theta, mu, sigma, 100) seed=123;
```

Each column of the matrix **yt** is plotted in Figure 25.491. The first series oscillates about a mean value of 10; the second series oscillates about a mean value of 20.

Figure 25.491 Time Series Components



You can also simulate a nonstationary VARMA(1,1) time series with the same μ, Σ, and Θ as in the previous example and with the following AR coefficient:

$$\Phi = \begin{bmatrix} 1.0 & 0 \\ 0 & 0.3 \end{bmatrix}$$

To generate this series, you can use the following statements:

```
phi  = { 1.0 0.0, 0.0 0.3 };
call varmasim(yt, phi, theta, mu, sigma, 100) initial=3 seed=123;
```

VECDIAG Function

VECDIAG(*matrix***);**

The VECDIAG function creates a column vector whose elements are the elements on the main diagonal of *matrix*. For example, the following statements produce the column vector shown in Figure 25.492:

```
a = {2 1, 0 -1};
d = vecdiag(a);
print d;
```

Figure 25.492 Diagonal of a Matrix

d
2
-1

VECH Function

VECH(*matrix***);**

The VECH function creates a column vector whose elements are the stacked columns of the lower triangular elements of *matrix*. Often, the argument is a symmetric matrix, in which case the VECH function has the effect of discarding the "duplicate" elements that are above the matrix diagonal. Notice that the lower triangular elements are returned in column-major order; use the SYMSQR function if you want the elements in row-major order.

Uses of the VECH function in matrix algebra are described in Harville (1997). "Vech" is an abbreviation for "vector-half."

The following statements produce the column vector shown in Figure 25.493:

```
a = {1 2 3, 4 5 6, 7 8 9};
v = vech(a);
print v;
```

Figure 25.493 Stacked Columns of Lower Triangular Matrix

v
1
4
7
5
8
9

The SQRVECH function and the VECH function are inverse operations on the set of symmetric matrices.

VNORMAL Call

CALL VNORMAL(*series, mu, sigma, n* < *, seed*> **);**

The VNORMAL subroutine generates a multivariate normal random series.

This function is deprecated. Instead, you should use the RANDNORMAL function to generate random values. The RANDNORMAL function calls the RANDGEN subroutine, which has excellent statistical properties. Consequently, the RANDNORMAL function is preferred when you need to generate millions of random numbers.

The input arguments to the VNORMAL subroutine are as follows:

mu specifies a $k \times 1$ (or $1 \times k$) mean vector, where $k \geq 2$ is the number of variables. You must specify either *mu* or *sigma*. If *mu* is not specified, a zero vector is used.

sigma specifies a $k \times k$ symmetric positive-definite covariance matrix. By default, *sigma* is an identity matrix with dimension k. You must specify either *mu* or *sigma*. If *sigma* is not specified, an identity matrix is used.

n specifies the length of the series. If *n* is not specified, $n = 100$ is used.

seed is a scalar that contains the random number seed. At the first execution of the subroutine, the seed variable is used as follows:

If *seed* > 0, the input seed is used for generating the series.

If *seed* = 0, the system clock is used to generate the seed.

If *seed* < 0, the value $(-1)\times(seed)$ is used for generating the series.

If the seed is not supplied, the system clock is used to generate the seed.

On subsequent calls of the subroutine in the DO loop like environment the seed variable is used as follows: If seed > 0, the seed remains unchanged. In other cases, after each execution of the subroutine, the current seed is updated internally.

The VNORMAL subroutine returns the following value:

series is an $n \times k$ matrix that contains the generated normal random series.

Consider a bivariate ($k = 2$) normal random series with mean $\boldsymbol{\mu}$ and covariance matrix Σ, where

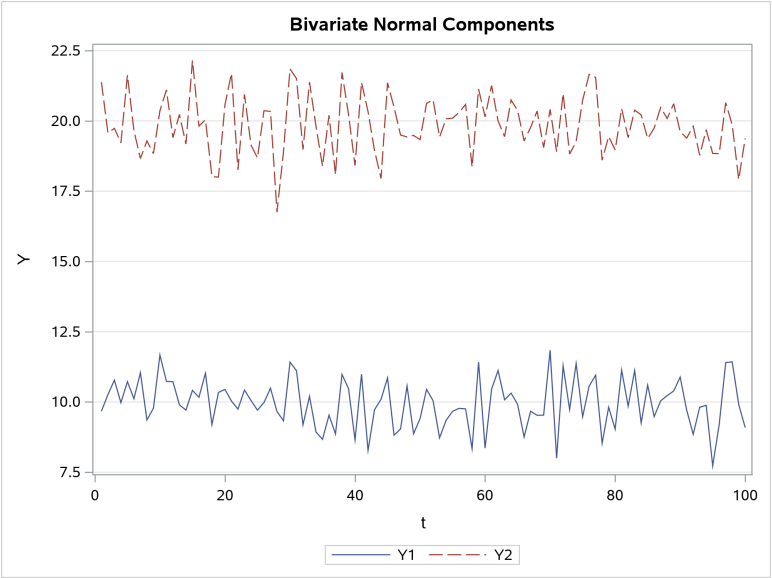
$$\boldsymbol{\mu} = \begin{bmatrix} 10 \\ 20 \end{bmatrix} \text{ and } \Sigma = \begin{bmatrix} 1.0 & 0.5 \\ 0.5 & 1.25 \end{bmatrix}$$

To generate this series, you can use the following statements:

```
mu    = { 10, 20 };
sigma= { 1.0  0.5, 0.5 1.25};
call vnormal(et, mu, sigma, 100) seed=123;
```

Each column of the matrix **et** is plotted in Figure 25.494. The first series oscillates about a mean value of 10; the second series oscillates about a mean value of 20.

Figure 25.494 Bivariate Normal Series



VTSROOT Call

CALL VTSROOT(*root, phi, theta* **<** *, p>* **<** *, q>* **);**

The VTSROOT subroutine computes the characteristic roots of the model from AR and MA characteristic functions.

The input arguments to the VTSROOT subroutine are as follows:

phi specifies a $km_p \times k$ matrix that contains the autoregressive coefficient matrices, where m_p is the number of the elements in the subset of the AR order and $k \geq 2$ is the number of variables. You must specify either *phi* or *theta*.

theta specifies a $km_q \times k$ matrix that contains the moving average coefficient matrices, where m_q is the number of the elements in the subset of the MA order. You must specify either *phi* or *theta*.

p specifies the subset of the AR order. See the VARMACOV subroutine.

q specifies the subset of the MA order. See the VARMACOV subroutine.

The VTSROOT subroutine returns the following value:

root is a $k(p_{\max} + q_{\max}) \times 5$ matrix, where $p_{\max}$ is the maximum order of the AR characteristic function and $q_{\max}$ is the maximum order of the MA characteristic function. The first $kp_{\max}$ rows refer to the results of the AR characteristic function; the last $kq_{\max}$ rows refer to the results of the MA characteristic function.

The first column contains the real parts, x, of eigenvalues of companion matrix associated with the AR($p_{\max}$) or MA($q_{\max}$) characteristic function; the second column contains the imaginary parts, y, of the eigenvalues; the third column contains the moduli of the eigenvalues, $\sqrt{x^2 + y^2}$; the fourth column contains the arguments $(\arctan(y/x))$ of the eigenvalues, measured in radians from the positive real axis. The fifth column contains the arguments expressed in degrees rather than radians.

Consider the roots of the characteristic functions, $\Phi(B) = I - \Phi B$ and $\Theta(B) = I - \Theta B$, where I is an identity matrix with dimension 2 and

$$\Phi = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix} \quad \Theta = \begin{bmatrix} -0.6 & 0.3 \\ 0.3 & 0.6 \end{bmatrix}$$

To compute these roots, you can use the following statements:

```
phi  = { 1.2 -0.5, 0.6 0.3 };
theta= {-0.6  0.3, 0.3 0.6 };
call vtsroot(root, phi, theta);
cols = {"Real" "Imag" "Modulus" "Radians" "Degrees"};
print root[colname=cols];
```

Figure 25.495 Characteristic Roots

root				
Real	Imag	Modulus	Radians	Degrees
0.75	0.3122499	0.8124038	0.3945069	22.603583
0.75	-0.31225	0.8124038	-0.394507	-22.60358
0.6708204	0	0.6708204	0	0
-0.67082	0	0.6708204	3.1415927	180

WAVFT Call

CALL WAVFT(*decomp, data, opt* < *, levels* > **);**

The fast wavelet transform (WAVFT) subroutine computes a specified discrete wavelet transform of the input data by using the algorithm of Mallat (1989). This transform decomposes the input data into sets of detail and scaling coefficients defined at a number of scales or "levels."

The input data are used as scaling coefficients at the top level in the decomposition. The fast wavelet transform then recursively computes a set of detail and a set of scaling coefficients at the next lower level by respectively applying "low pass" and "high pass" conjugate mirror filters to the scaling coefficients at the current level. The number of coefficients in each of these new sets is approximately half the number of scaling coefficients at the level above them. Depending on the filters being used, a number of additional scaling coefficients, known as boundary coefficients, can be involved. These boundary coefficients are obtained by using a specified method to extend the sequence of interior scaling coefficients

Details of the discrete wavelet transform and the fast wavelet transformation algorithm are available in many references, including Mallat (1989), Daubechies (1992), and Ogden (1997).

The input arguments to the WAVFT subroutine are as follows:

data specifies the data to transform. These data must be in either a row or column vector.

opt refers to an options vector with the following components:

opt[1] specifies the boundary handling used in computing the wavelet transform. At each level of the wavelet decomposition, necessary boundary scaling coefficients are obtained by extending the interior scaling coefficients at that level as follows:

- 0 specifies extension by zero.
- 1 specifies periodic extension.
- 2 specifies polynomial extension.
- 3 specifies extension by reflection.
- 4 specifies extension by anti-symmetric reflection.

opt[2] specifies the polynomial degree that is used for polynomial extension. (The value of *opt[2]* is ignored if *opt[1]*$\neq$ 2.)

0 specifies constant extension.

1 specifies linear extension.

2 specifies quadratic extension.

opt[3] specifies the wavelet family.

1 specifies the Daubechies Extremal phase family (Daubechies 1992).

2 specifies the Daubechies Least Asymmetric family (also known as the Symmlet family) (Daubechies 1992).

opt[4] specifies the wavelet family member. Valid values are

1 through 10, if *opt[3]*=1

4 through 10, if *opt[3]*=2

Some examples of wavelet specifications are

opt={1 . 1 1}; specifies the first member (more commonly known as the Haar system) of the Daubechies extremal phase family with periodic boundary handling.

opt={2 1 2 5}; specifies the fifth member of the Symmlet family with linear extension boundary handling.

levels is an optional scalar argument that specifies the number of levels from the top level to be computed in the decomposition. If you do not specify this argument, then the decomposition terminates at level 0. Usually, you do not need to specify this optional argument. You use this option to avoid unneeded computations in situations where you are interested in only the higher level detail and scaling coefficients.

The WAVFT subroutine returns

decomp a row vector that encapsulates the specified wavelet transform. The information that is encoded in this vector includes:

- the options specified for computing the transform
- the number of detail coefficients at each level of the decomposition
- all detail coefficients
- the scaling coefficients at the bottom level of the decomposition
- boundary scaling coefficients at all levels of the decomposition

NOTE: *decomp* is a private representation of the specified wavelet transform and is not intended to be interpreted in its raw form. Rather, you should use this vector as an input argument to the WAVIFT, WAVPRINT, WAVGET, and WAVTHRSH subroutines

The following program shows an example that uses wavelet calls to estimate and reconstruct a piecewise constant function:

```
/* define a piecewise constant step function */
start blocky(t);
   /* positions (p) and magnitudes (h) of jumps */
   p = {0.1 0.13 0.15 0.23 0.25 0.4 0.44 0.65 0.76 0.78 0.81};
   h  = {4 -5 3 -4 5 -4.2 2.1 4.3 -3.1 2.1 -4.2};

   y=j(1, ncol(t), 0);
   do i=1 to ncol(p);
      diff = ( (t-p[i])>=0 );
      y = y + h[i]*diff;
   end;
   return (y);
finish blocky;

n = 2##8;
x = 1:n;
x = (x-1)/n;
y = blocky(x);

opt = { 2, /* polynomial extension at boundary */
        1, /* using linear polynominal */
        1, /* Daubechies Extremal phase */
        3  /* family member 3 */
         };

call wavft(decomp, y, opt);
call wavprint(decomp,1); /* print summary information */

/* perform permanent thresholding */
threshOpt = { 2, /* soft thresholding */
              2, /* global threshold */
              ., /* ignored */
             -1  /* apply to all levels */
              };
call wavthrsh(decomp, threshOpt);

/* request detail coefficients at level 4 */
call wavget(detail4,decomp,2,4);

/* reconstruct function by using wavelets */
call wavift(estimate,decomp);

errorSS=ssq(y-estimate);
print errorSS;
```

Figure 25.496 Summary of Wavelet Analysis

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
1.81E-25

You can improve the readability of the options and threshold options if you use the macro variables that are defined in the WAVINIT autocall macro. See Chapter 21, "Wavelet Analysis," for details and an example. The WAVINIT macro also defines several graphical subroutines that enable you to visualize wavelet analyses, as follows:

CALL COEFFICIENTPLOT creates a graph of the detail coefficients arranged by level.

CALL MRAAPPROX creates a multiresolution approximation plot, which is a graph that shows reconstructions of the input signal by level.

CALL MRADECOMP creates a multiresolution decomposition plot, which is a graph that shows how the detail coefficients combine to reconstruct the signal.

CALL SCALOGRAM creates a wavelet scalogram, which shows the relative magnitudes of detail coefficients and the energy in each level.

WAVGET Call

CALL WAVGET(*result, decomp, request* < *, options* > **);**

The WAVGET subroutine is used to return information that is encoded in a wavelet decomposition.

The required input arguments are as follows:

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

request specifies a scalar that indicates what information is to be returned.

You can specify different optional arguments depending on the value of *request*:

request=1 requests the number of points in the input data vector.

result returns as a scalar that contains this number.

request=2 requests the detail coefficients at a specified level. Valid syntax is

CALL WAVGET(*result, decomp, 2, level* < *, opt* > **);**

The arguments are as follows:

level is the level at which the detail coefficients are requested.

opt is an optional vector which specifies the thresholding to be applied to the returned detail coefficients. See the WAVIFT subroutine for details. If you omit this argument, no thresholding is applied.

result returns as a column vector that contains the specified detail coefficients.

request=3 requests the scaling coefficients at a specified level. Valid syntax is

CALL WAVGET(*result, decomp, 3, level* < , *opt*> **);**

The arguments are as follows:

level is the level at which the scaling coefficients are requested.

opt is an optional vector that specifies the thresholding to be applied. See the WAVIFT subroutine for a description of this vector. The scaling coefficients at the requested level are obtained by using the inverse wavelet transform, after applying the specified thresholding. If you omit this argument, no thresholding is applied.

result returns as a column vector that contains the specified scaling coefficients.

request=4 requests the thresholding status of the detail coefficients in *decomp*.

result returns as a scalar whose value is

0, if thresholding has not been applied to the detail coefficients

1, otherwise

request=5 requests the wavelet options vector that you specified in the WAVFT subroutine to compute *decomp*.

result returns as a column vector with 4 elements that contains the specified options vector. See the WAVFT subroutine for the interpretation of the vector entries.

request=6 requests the index of the top level in *decomp*.

result returns as a scalar that contains this number.

request=7 requests the index of the lowest level in *decomp*.

result returns as a scalar that contains this number.

request=8 requests a vector evaluating the father wavelet used in *decomp*, at an equally spaced grid spanning the support of the father wavelet. The number of points in the grid is specified as a power of 2 times the support width of the father wavelet. For wavelets in the Daubechies extremal phase and least asymmetric families, the support width of the father wavelet is $2m - 1$, where m is the family member. Valid syntax is

CALL WAVGET(*result, decomp, 8* < , *power*> **);**

The optional argument has the following meaning:

power is the exponent of 2 that determines the number of grid points used. *power* defaults to 8 if you do not specify this argument.

result returns as a column vector that contains the specified evaluation of the father wavelet.

An example is available in the documentation for the WAVFT subroutine.

WAVIFT Call

CALL WAVIFT(*result, decomp* **<** *, opt* **>** **<** *, level* **>** **);**

The Inverse Fast Wavelet Transform (WAVIFT) subroutine computes the inverse wavelet transform of a wavelet decomposition computed by using the WAVFT subroutine. Details of this algorithm are available in many references, including Mallat (1989), Daubechies (1992), and Ogden (1997).

The inverse transform yields an exact reconstruction of the original input data, provided that no smoothing is specified. Alternatively, a smooth reconstruction of the input data can be obtained by thresholding the detail coefficients in the decomposition prior to applying the inverse transformation. Thresholding, also known as shrinkage, replaces the detail coefficient $d_j^{(i)}$ at level i by $\delta_{T_i}(d_j^{(i)})$, where the $\delta_T(x)$ is a shrinkage function and T_i is the threshold value used at level i. The wavelet subroutines support hard and soft shrinkage functions (Donoho and Johnstone 1994) and the nonnegative garrote shrinkage function (Breiman 1995). These functions are defined as follows:

$$\delta_T^{\text{hard}}(x) = \begin{cases} 0 & |x| \le T \\ x & |x| > T \end{cases}$$

$$\delta_T^{\text{soft}}(x) = \begin{cases} 0 & |x| \le T \\ x - T & x > T \\ x + T & x < -T \end{cases}$$

$$\delta_T^{\text{garrote}}(x) = \begin{cases} 0 & |x| \le T \\ x - T^2/x & |x| > T \end{cases}$$

You can specify several methods for choosing the threshold values. Methods in which the threshold T_i varies with the level i are called adaptive. Methods where the same threshold is used at all levels are called global.

The input arguments to the WAVIFT subroutine are as follows:

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

opt refers to an options vector that specifies the thresholding algorithm. If this optional argument is not specified, then no thresholding is applied.

The options vector has the following components:

opt[1] specifies the thresholding policy.

0 specifies that no thresholding be done. If *opt[1]*=0 then all other entries in the options vector are ignored.

1 specifies hard thresholding.

2 specifies soft thresholding.

3 specifies garrote thresholding.

opt[2] specifies the method for selecting the threshold.

0 specifies a global user-supplied threshold.

1 specifies a global threshold chosen by using the minimax criterion of Donoho and Johnstone (1994).

2 specifies a global threshold defined by using the universal criterion of Donoho and Johnstone (1994).

3 specifies an adaptive method where the thresholds at each level i are chosen to minimize an approximation of the L^2 risk in estimating the true data values by using the reconstruction with shrunken coefficients (Donoho and Johnstone 1995).

4 specifies a hybrid method of Donoho and Johnstone (1995). The universal threshold as specified by *opt[2]*=2 is used at levels where most of the detail coefficients are essentially zero. The risk minimization method as specified by *opt[2]*=4 is used at all other levels.

opt[3] specifies the value of the global user-supplied threshold if *opt[2]*=1. It is ignored if *opt[2]* $\neq$ 1.

opt[4] specifies the number of levels starting at the highest detail coefficient level at which thresholding is to be applied. If this value is negative or missing, thresholding is applied at all levels in *decomp*.

Some common examples of threshold options specifications are:

opt={1 3 . -1}; specifies hard thresholding with a minimax threshold applied at all levels in the decomposition. This threshold is named "*RiskShrink*" in Donoho and Johnstone (1994).

opt={2 2 . -1}; specifies soft thresholding with a universal threshold applied at all levels in the decomposition. This threshold is named "*VisuShrink*" in Donoho and Johnstone (1994).

opt={2 4 . -1}; specifies soft thresholding with level dependent thresholds which minimize the Stein Unbiased Estimate of Risk (SURE). This threshold is named "*SureShrink*" in Donoho and Johnstone (1995).

level is an optional scalar argument that specifies the level at which the reconstructed data are to be returned. If this argument is not specified then the reconstructed data are returned at the top level defined in *decomp*.

The WAVIFT subroutine returns

result a vector obtained by inverting, after thresholding the detail coefficients, the discrete wavelet transform encoded in *decomp*. The row or column orientation of *result* is the same as that of the input data specified in the corresponding WAVFT subroutine. If you specify the optional *level* argument, *result* contains the reconstruction at the specified level, otherwise the reconstruction corresponds to the top level in the decomposition.

An example is available in the documentation for the WAVFT subroutine and in Chapter 21, "Wavelet Analysis."

WAVPRINT Call

CALL WAVPRINT(*decomp, request* < *, options*> **);**

The WAVPRINT subroutine is used to display the information that is encoded in a wavelet decomposition.

The required input arguments are as follows:

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

request specifies a scalar that indicates what information is to be displayed.

You can specify different optional arguments depending on the value of *request*:

request=1 displays information about the wavelet family used to perform the wavelet transform. No additional arguments need to be specified.

request=2 displays the detail coefficients by level. Valid syntax is

CALL WAVPRINT(*decomp, 2* < *, lower*> < *, upper*> **);**

The optional arguments are as follows:

lower specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp.*

upper specifies the upper level to be displayed. The default value of *upper* is the highest detail level in *decomp*.

request=3 displays the scaling coefficients by level. Valid syntax is

CALL WAVPRINT(*decomp,3* < *, lower*> < *, upper*> **);**

The optional arguments are as follows:

lower and specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp.*

upper specifies the upper level to be displayed. The default value of *upper* is the top level in *decomp.*

request=4 displays the shrunken detail coefficients by level. Valid syntax is

CALL WAVPRINT(*decomp, 4* < *, opt*> < *, lower*> < *, upper*> **);**

The optional arguments are as follows:

opt specifies the thresholding to be applied to the displayed detail coefficients. See the WAVIFT subroutine for details. If you omit this argument, no thresholding is applied.

lower specifies the lowest level to be displayed. The default value of *lower* is the lowest level in *decomp*.

upper specifies the upper level to be displayed. The default value of *upper* is the highest detail level in *decomp*.

An example is available in the documentation for the WAVFT subroutine.

WAVTHRSH Call

CALL WAVTHRSH(*decomp, opt* **);**

The wavelet threshold (WAVTHRSH) subroutine applies a thresholding algorithm to the detail coefficients in a wavelet decomposition.

The required input arguments are as follows:

decomp specifies a wavelet decomposition that has been computed by using a call to the WAVFT subroutine.

opt refers to an options vector that specifies the thresholding algorithm used. See the WAVIFT subroutine for a description of this options vector.

On return, the detail coefficients encoded in *decomp* are replaced by the result of applying the threshold algorithm. Note that this action is not reversible. If you want to retain the original detail coefficients, you should not use the WAVTHRSH subroutine to do thresholding. Rather, you should supply the thresholding argument where appropriate in the WAVIFT, WAVGET, and WAVPRINT subroutines.

An example is available in the documentation for the WAVFT subroutine.

WIDETOLONG Call

CALL WIDETOLONG(*LongX, LongY, Group, Y* **<** *, X* **>** **<** *, YLabels* **>** **);**

The WIDETOLONG subroutine converts time series data from a wide form (an $N \times p$ matrix) into a long form, which consists of an $Np \times 1$ vector and an ID variable.

If the input matrix *Y* contains p variables, the output vector *LongY* is a column vector that contains the columns of *Y* stacked on top of each other. The vector *Group* contains an ID variable that enables you to determine which rows of *LongY* correspond to which columns of *Y*.

The output arguments are as follows:

LongX contains repeated values of the *X* vector.

LongY contains a stacked copy of the time series data.

Group contains an ID variable that identifies each series.

The input arguments are as follows:

Y specifies an $N \times p$ matrix of time series data.

X specifies an N-element vector that specifies values of the independent (time) variable. If this argument is not specified, the vector **T(1:N)** is used by default.

YLabels specifies a p-element vector that contains the values to be used for the output variable *Group*. If this argument is not specified, the vector **"Y1":"Yp"** is used by default.

In the following example, the matrix **Y** contains data in two columns and three rows. The rows of **X** contain the times (or positions) that correspond to each row of **Y**. A call to the WIDETOLONG subroutine converts the data into three vectors. The **OutY** vector contains the columns of **Y** (stacked on top of each other), the **OutX** vector contains repeated values of **X**, and the **Group** vector contains an ID variable, as shown in Figure 25.497.

```
X = {0, 1.5, 4};
Y = {6 2,
     5 4,
     2 9 };
call WideToLong(OutX, OutY, Group,
                Y, X);
print OutX OutY Group;
```

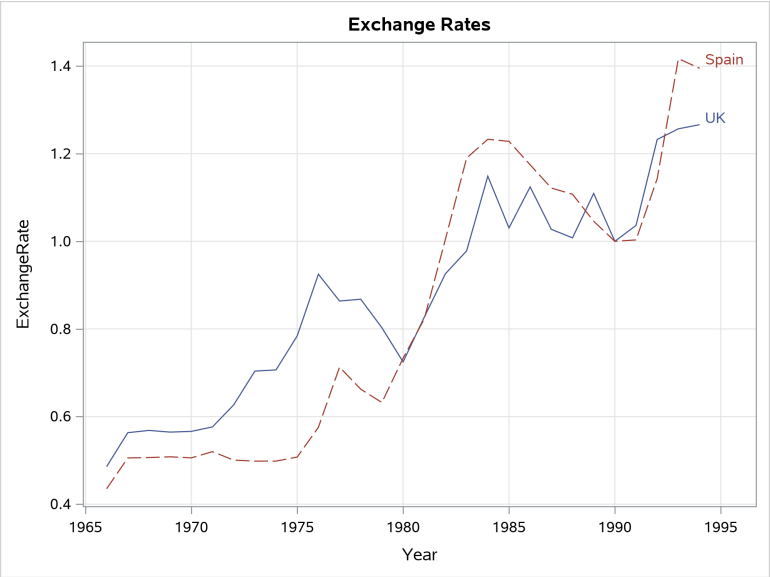
Figure 25.497 Convert Wide Data to Long Data

OutX	OutY	Group
0	6	Y1
1.5	5	Y1
4	2	Y1
0	2	Y2
1.5	4	Y2
4	9	Y2

A common use of the WIDETOLONG subroutine is to convert multiple columns of data into a form that can be easily graphed by using the SERIES subroutine. This is illustrated by the following statements, which create Figure 25.498:

```
/* exchange rates of UK pound and Spanish pesetas to the US dollar */
X = T(1966:1994);
Y = {0.485709 0.435193, 0.5632   0.505549, 0.568364 0.506419,
     0.564515 0.50816,  0.566155 0.505694, 0.576441 0.51991,
     0.626625 0.500617, 0.703709 0.498295, 0.706515 0.498295,
     0.784033 0.50758,  0.924894 0.57547,  0.863692 0.712845,
     0.867812 0.662508, 0.802721 0.632045, 0.724715 0.733154,
     0.826736 0.822732, 0.925951 1.004932, 0.978113 1.189962,
     1.14864  1.232828, 1.030533 1.228113, 1.124233 1.174585,
     1.027294 1.121564, 1.00786  1.107348, 1.109298 1.04584,
     1        1,        1.036266 1.003191, 1.232419 1.143178,
     1.256732 1.416842, 1.266178 1.394937 };

run WideToLong(Year, ExchangeRate, ID,     /* output (long) */
               Y, X, {"UK" "Spain"});      /* input (Y is wide) */

title "Exchange Rates";
call series(Year, ExchangeRate) group=ID grid={X Y}
     option="curvelabel curvelabelpos=end";
```

Figure 25.498 A Graph of Multiple Time Series



XMULT Function

XMULT(*matrix1, matrix2***);**

The XMULT function computes a matrix product by using extended-precision calculations. For most matrices, the XMULT function is numerically equivalent to the matrix multiplication operator (*). You should use the XMULT function on pathological examples for which extended-precision calculations are required to obtain an accurate product.

The following program demonstrates the use of 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];
```

Figure 25.499 Extended-Precision Multiplication

z
20000000000001

wrong
0

XSECT Function

XSECT(*matrix1 <, matrix2, . . . , matrix15>* **);**

The XSECT function returns as a row vector the sorted set (without duplicates) of the element values that are present in all of its arguments. This set is the intersection of the sets of values in its argument matrices. When the intersection is empty, the XSECT function returns an empty matrix with zero rows and zero columns. There can be up to 15 arguments, which must all be either character or numeric.

For characters, the element length of the result is the same as the shortest of the element lengths of the arguments. For comparison purposes, shorter elements are padded on the right with blanks.

For example, the following statements computes the intersection of two sets:

```
a = {1 2 4 5};
b = {3 4};
c = xsect(a,b);
print c;
```

Figure 25.500 Set Intersection

c
4

YIELD Function

YIELD(*times, flows, freq, value***);**

The YIELD function returns a scalar that contains yield-to-maturity of a cash-flow stream based on frequency and value specified.

The arguments to the YIELD function are as follows:

times	is an *n*-dimensional column vector of times. Elements should be nonnegative.
flows	is an *n*-dimensional column vector of cash flows.
freq	is a scalar that represents the base of the rates to be used for discounting the cash flows. If positive, it represents discrete compounding as the reciprocal of the number of compoundings. If zero, it represents continuous compounding. No negative values are accepted.
value	is a scalar that is the discounted present value of the cash flows.

The present value relationship can be written as

$$P = \sum_{k=1}^{K} c(k)D(t_k)$$

where P is the present value of the asset, $\{c(k)\}k = 1, ..K$ is the sequence of cash flows from the asset, t_k is the time to the kth cash flow in periods from the present, and $D(t)$ is the discount function for time t.

With continuous compounding:

$$D(t) = e^{-yt}$$

With discrete compounding:

$$D(t) = (1 + fy)^{-t/f}$$

where $f > 0$ is the frequency, the reciprocal of the number of compoundings per unit time period, and y is the yield-to-maturity. The YIELD function solves for y.

For example, the following statements produce the output shown in Figure 25.501:

```
timesn = T(do(1, 100, 1));
flows = repeat(10, 100);
freq = 50;
value = 682.31027;
yield = yield(timesn, flows, freq, value);
print yield;
```

Figure 25.501 Yield to Maturity

yield
0.01

Base SAS Functions Accessible from SAS/IML Software

You can call most functions available in Base SAS software from SAS/IML programs. If you call a Base SAS function with a matrix argument, the function will usually act elementwise on each element of the matrix.

Some DATA step functions are not applicable or not useful in the SAS/IML environment. For example, Base SAS functions that take a list of scalar arguments are often not very useful to the SAS/IML programmer. (The SAS/IML language does not support the OF keyword.) However, it is usually possible to use SAS/IML built-in functions to obtain the same results. For example, the SUMABS, EUCLID, and LPNORM functions in Base SAS are used to compute the vector norm of a list of arguments. Instead of using those functions, you should use the SAS/IML NORM function.

The following Base SAS functions are either not available from SAS/IML software, or behave differently from the Base SAS function of the same name.

Function	Comment
CHAR	conflicts with the SAS/IML CHAR function
CALL CATS	return variable must be preinitialized
DIF*n*	not supported; use the SAS/IML DIF function instead.
DIM	not supported, but see the DIMENSION function for similar functionality.
HBOUND	not supported
LAG*n*	not supported; use the SAS/IML LAG function instead.

LBOUND	not supported
MOD	base function performs “fuzzing”; the SAS/IML function does not
PUT	Use the PRINT statement instead
CALL PRXNEXT	return variables must be preinitialized
CALL PRXPOSN	return variables must be preinitialized
CALL PRXSUBSTR	return variables must be preinitialized
CALL SCAN	return variables must be preinitialized
VVALUE	not applicable: interrogates DATA step variables
VVALUEX	not applicable: interrogates DATA step variables
VNEXT	not applicable: interrogates DATA step variables

There are also some Base SAS features that are not supported by the SAS/IML language. For example, the DATA step permits N-literals (strings that end with ’N’) to be interpreted as the name of a variable, but the SAS/IML language does not.

The following Base SAS functions can be called from SAS/IML. The functions are documented in the *SAS Language Reference: Dictionary*. In some cases, SAS/IML does not accept all variations in the syntax. For example, SAS/IML does not accept the OF keyword as a way to generate an argument list in a function.

The functions displayed in italics are documented elsewhere in this user’s guide. These functions operate on matrices in addition to scalar values, as do many of the mathematical and statistical functions.

Bitwise Logical Operation Functions

BAND	returns the bitwise logical AND of two arguments
BLSHIFT	performs a bitwise logical left shift of an argument by a specified amount
BNOT	returns the bitwise logical NOT of an argument
BOR	returns the bitwise logical OR of two arguments
BRSHIFT	performs a bitwise logical right shift of an argument by a specified amount
BXOR	returns the bitwise logical EXCLUSIVE OR of two arguments

Character and Formatting Functions

ANYALNUM	searches a character string for an alphanumeric character and returns the first position at which it is found
ANYALPHA	searches a character string for an alphabetic character and returns the first position at which it is found
ANYCNTRL	searches a character string for a control character and returns the first position at which it is found

ANYDIGIT	searches a character string for a digit and returns the first position at which it is found
ANYFIRST	searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
ANYGRAPH	searches a character string for a graphical character and returns the first position at which it is found
ANYLOWER	searches a character string for a lowercase letter and returns the first position at which it is found
ANYNAME	searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
ANYPRINT	searches a character string for a printable character and returns the first position at which it is found
ANYPUNCT	searches a character string for a punctuation character and returns the first position at which it is found
ANYSPACE	searches a character string for a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found
ANYUPPER	searches a character string for an uppercase letter and returns the first position at which it is found
ANYXDIGIT	searches a character string for a hexadecimal character that represents a digit and returns the first position at which that character is found
BYTE	returns one character in the ASCII or EBCDIC collating sequence
CAT	concatenates character strings without removing leading or trailing blanks
CATQ	concatenates character or numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter
CATS	concatenates character strings and removes leading and trailing blanks
CALL CATS	concatenates character strings and removes leading and trailing blanks
CATT	concatenates character strings and removes trailing blanks
CALL CATT	concatenates character strings and removes trailing blanks
CATX	concatenates character strings, removes leading and trailing blanks, and inserts separators
CALL CATX	concatenates character strings, removes leading and trailing blanks, and inserts separators
CHOOSEC	returns a character value that represents the results of choosing from a list of arguments
CHOOSEN	returns a numeric value that represents the results of choosing from a list of arguments
COLLATE	returns an ASCII or EBCDIC collating sequence character string
COMPARE	returns the position of the left-most character by which two strings differ, or returns 0 if there is no difference
COMPBL	removes multiple blanks from a character string

CALL COMPCOST	sets the costs of operations for later use by the COMPGED function
COMPGED	compares two strings by computing the generalized edit distance
COMPLEV	compares two strings by computing the Levenshtein edit distance
COMPRESS	removes specific characters from a character string
COUNT	counts the number of times that a specific substring of characters appears within a character string that you specify
COUNTC	counts the number of specific characters that either appear or do not appear within a character string that you specify
COUNTW	counts the number of words in a character expression
FIND	searches for a specific substring of characters within a character string that you specify
FINDC	searches for specific characters that either appear or do not appear within a character string that you specify
FINDW	returns the character position of a word in a string, or returns the number of the word in a string
FIRST	returns the first character in a character string
FMTINFO	retrieves information about a format or informat
IFC	returns a character value that matches an expression
IFN	returns a numeric value that matches an expression
INDEX	searches a character expression for a string of characters
INDEXC	searches a character expression for specific characters
INDEXW	searches a character expression for a specified string as a word
INPUTC	applies a character informat at run time
INPUTN	applies a numeric informat at run time
LEFT	left aligns a character expression
LENGTH	returns the length of a character string
LENGTHC	returns the length of a character string, including trailing blanks
LENGTHM	returns the amount of memory (in bytes) that is allocated for a character string
LENGTHN	returns the length of a nonblank character string, excluding trailing blanks, and returns 0 for a blank character string
LOWCASE	converts all letters in an argument to lowercase
CALL MISSING	assigns a missing value to the specified character or numeric variable
NLITERAL	converts a character string that you specify to a SAS name literal (N-literal)
NOTALNUM	searches a character string for a nonalphanumeric character and returns the first position at which it is found
NOTALPHA	searches a character string for a nonalphabetic character and returns the first position at which it is found
NOTCNTRL	searches a character string for a character that is not a control character and returns the first position at which it is found
NOTDIGIT	searches a character string for any character that is not a digit and returns the first position at which that character is found
NOTFIRST	searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found

NOTGRAPH	searches a character string for a nongraphical character and returns the first position at which it is found
NOTLOWER	searches a character string for a character that is not a lowercase letter and returns the first position at which that character is found
NOTNAME	searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found
NOTPRINT	searches a character string for a nonprintable character and returns the first position at which it is found
NOTPUNCT	searches a character string for a character that is not a punctuation character and returns the first position at which it is found
NOTSPACE	searches a character string for a character that is not a white-space character (blank, horizontal and vertical tab, carriage return, line feed, form feed) and returns the first position at which it is found
NOTUPPER	searches a character string for a character that is not an uppercase letter and returns the first position at which that character is found
NOTXDIGIT	searches a character string for a character that is not a hexadecimal digit and returns the first position at which that character is found
NVALID	checks a character string for validity for use as a SAS variable name in a SAS statement
PROPCASE	converts all words in an argument to proper case
PUTC	applies a character format at run time
PUTN	applies a numeric format at run time
REPEAT	repeats a character expression
REVERSE	reverses a character expression
RIGHT	right aligns a character expression
SCAN	selects a given word from a character expression
CALL SCAN	returns the position and length of a given word from a character expression
ROUNDEX	encodes a string to facilitate searching
SPEDIS	determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words
STRIP	returns a character string with all leading and trailing blanks removed
SUBPAD	returns a substring that has specified length and is padded with blanks, if necessary
SUBSTRN	returns a substring, allowing a result with a length of zero
SUBSTR	extracts substrings of character expressions
TRANSLATE	replaces specific characters in a character expression
TRANSTRN	replaces or removes all occurrences of a substring in a character string
TRANWRD	replaces or removes all occurrences of a word in a character string
TRIM	removes trailing blanks from character expressions and returns one blank if the expression is missing
TRIMN	removes trailing blanks from character expressions and returns a null string (zero blanks) if the expression is missing
UPCASE	converts all letters in an argument to uppercase

UUIDGEN	returns the short or binary form of a Universal Unique Identifier (UUID)
VERIFY	returns the position of the first character that is unique to an expression
WHICHC	searches for a character value that is equal to the first argument, and returns the index of the first matching value
WHICHN	searches for a numeric value that is equal to the first argument, and returns the index of the first matching value

Character String Matching Functions and Subroutines

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

Combinatorial Functions

ALLCOMB	generates all combinations of *n* elements taken *k* at a time
ALLCOMBI	see the ALLCOMB function
ALLPERM	generates all permutations of *n* elements
COMB	returns the number of combinations of *n* items taken *r* at a time
FACT	returns the factorial of an integer
GRAYCODE	returns all subsets in a minimal change order
LCOMB	returns the logarithm of the COMB function
LEXCOMB	returns distinct combinations of *n* variables taken *k* at a time in lexicographic order
LEXCOMBI	returns combinations of the indices of *n* objects taken *k* at a time in lexicographic order
LEXPERK	returns distinct permutations *n* variables taken *k* at a time in lexicographic order
LEXPERM	returns distinct permutations of several variables in lexicographic order
LFACT	returns the logarithm of the FACT (factorial) function

LPERM	returns the logarithm of the PERM function
PERM	returns the number of permutations of *n* items taken *r* at a time
RANCOMB	returns random combinations of *n* elements taken *k* at a time
RANPERK	randomly permutes the values of the arguments, and returns a permutation of *k* out of *n* values
RANPERM	returns random permutations of *n* elements

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
HOLIDAY	returns the date of the specified holiday for the specified year. Other functions that begin with the HOLIDAY prefix are also supported.
HOUR	returns the hour from a SAS time or datetime value
INTCINDEX	returns the cycle index when a date, time, or datetime interval and value are specified
INTCK	returns the integer number of time intervals in a given time span
INTCYCLE	returns the date, time, or datetime interval at the next higher seasonal cycle
INTFIT	returns a time interval that is aligned between two dates
INTFMT	returns a recommended SAS format when a date, time, or datetime interval is specified
INTGET	returns a time interval based on three date or datetime values
INTINDEX	returns the seasonal index when a date, time, or datetime interval and value are specified
INTNX	advances a date, time, or datetime value by a given interval, and returns a date, time, or datetime value
INTSEAS	returns the length of the seasonal cycle when a date, time, or datetime interval is specified
INTSHIFT	returns the shift interval that corresponds to the base interval
INTEST	returns 1 if a time interval is valid, and returns 0 if a time interval is invalid
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
NWKDOM	returns the date for the nth occurrence of a weekday for the specified month and year
QTR	returns the quarter of the year from a SAS date value

SECOND	returns the second from a SAS time or datetime value
TIME	returns the current time of day
TIMEPART	extracts a time value from a SAS datetime value
TODAY	returns the current date as a SAS date value
WEEKDAY	returns the day of the week from a SAS date value
YEAR	returns the year from a SAS date value
YRDIF	returns the difference in years between two dates
YYQ	returns a SAS date value from the year and quarter

Descriptive Statistics Functions and Subroutines

CMISS	returns the number of nonmissing values
CSS	returns the corrected sum of squares
CV	returns the coefficient of variation
DIVIDE	returns the result of a division that handles special missing values for ODS output
GEOMEAN	returns the geometric mean
GEOMEANZ	returns the geometric mean without fuzzing the values of the arguments that are approximately 0
HARMEAN	returns the harmonic mean
HARMEANZ	returns the harmonic mean without fuzzing the values of the arguments that are approximately 0
IQR	returns the interquartile range
KURTOSIS	returns the kurtosis
LARGEST	returns the *k*th 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
ORDINAL	returns any specified order statistic
PCTL	computes percentiles
RANGE	returns the range of values
RMS	returns the root mean square
SKEWNESS	returns the skewness
SMALLEST	returns the *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
FCOPY	copies data from one fileref to another fileref
ENVLEN	returns the length of an environment variable
EXIST	verifies the existence of a SAS data library member
FAPPEND	appends the current record to the end of an external file and returns a value
FCLOSE	closes an external file, directory, or directory member, and returns a value
FCOL	returns the current column position in the File Data Buffer (FDB)
FDELETE	deletes an external file or an empty directory
FEXIST	verifies the existence of an external file associated with a fileref and returns a value
FGET	copies data from the File Data Buffer (FDB) into a variable and returns a value
FILEEXIST	verifies the existence of an external file by its physical name and returns a value
FILENAME	assigns or deassigns a fileref for an external file, directory, or output device and returns a value
FILEREF	verifies that a fileref has been assigned for the current SAS session and returns a value
FINFO	returns the value of a file information item
FNOTE	identifies the last record that was read and returns a value that FPOINT can use
FOPEN	opens an external file and returns a file identifier value
FOPTNAME	returns the name of an item of information about a file
FOPTNUM	returns the number of information items that are available for an external file
FPOINT	positions the read pointer on the next record to be read and returns a value
FPOS	sets the position of the column pointer in the File Data Buffer (FDB) and returns a value
FPUT	moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position, and returns a value
FREAD	reads a record from an external file into the File Data Buffer (FDB) and returns a value
FREWIND	positions the file pointer to the start of the file and returns a value

FRLEN	returns the size of the last record read, or, if the file is opened for output, returns the current record size
FSEP	sets the token delimiters for the FGET function and returns a value
FWRITE	writes a record to an external file and returns a value
MODEXIST	returns whether a software image exists in the version of SAS that you have installed
MOPEN	opens a file by directory identifier and member name, and returns the file identifier or a 0
MVALID	returns whether a character string is valid for use as a SAS member name
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
SYSPROD	returns whether a product is licensed
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

The SAS/IML language supports more than 50 functions in Base SAS that are applicable to finance, including the following:

BLACKCLPRC	calculates the call price for European options on futures, based on the Black model
BLACKPTPRC	calculates the put price for European options on futures, based on the Black model
BLKSHCLPRT	calculates the call price for European options, based on the Black-Scholes model
BLKSHPTPRT	calculates the put price for European options, based on the Black-Scholes model
COMPOUND	returns compound interest parameters
CONVX	returns the convexity for an enumerated cash flow
CONVXP	returns the convexity for a periodic cash flow stream
DACCDB	returns the accumulated declining balance depreciation
DACCDBSL	returns the accumulated declining balance with conversion to a straight-line depreciation
DACCSL	returns the accumulated straight-line depreciation
DACCSYD	returns the accumulated sum-of-years-digits depreciation
DACCTAB	returns the accumulated depreciation from specified tables
DEPDB	returns the declining balance depreciation
DEPDBSL	returns the declining balance with conversion to a straight-line depreciation
DEPSL	returns the straight-line depreciation
DEPSYD	returns the sum-of-years-digits depreciation
DEPTAB	returns the depreciation from specified tables
DUR	returns the modified duration for an enumerated cash flow
FINANCE	computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return

GARKHCLPRC	calculates the call price for European options on stocks, based on the Garman-Kohlhagen model
GARKHPTPRC	calculates the put price for European options on stocks, based on the Garman-Kohlhagen model
INTRR	returns the internal rate of return as a decimal
IRR	returns the internal rate of return as a percentage
MARGRCLPRC	calculates the call price for European options on stocks, based on the Margrabe model
MARGRPTPRC	calculates the put price for European options on stocks, based on the Margrabe model
MORT	returns amortization parameters
NETPV	returns the net present value as a decimal
NPV	returns the net present value as a percentage
PVP	returns the present value for a periodic cash flow stream
SAVING	returns the future value of a periodic saving
YIELDP	returns the yield-to-maturity for a periodic cash flow stream

Macro Functions and Subroutines

DOSUBL	Executes SAS code and exports macro variables
CALL RESOLVE	resolves the value of a text expression at execution time
SYMGET	returns the character value of a macro variable
SYMGETN	returns the numeric value of a macro variable
SYMEXIST	indicates the existence of a macro variable
CALL SYMPUT	sets the character value of a macro variable
CALL SYMPUTX	assigns a value to a macro variable and removes both leading and trailing blanks

Mathematical Functions and Subroutines

ABS	returns the absolute value
AIRY	returns the Airy function
BETA	returns the value of the beta function.
COALESCE	returns the first non-missing value from a list of numeric arguments
COALESCEC	returns the first non-missing value from a list of character arguments
COMPFUZZ	returns the result of a fuzzy comparison of numeric values
CONSTANT	returns some machine and mathematical constants
CNONCT	returns the noncentrality parameter from a chi-squared distribution
DAIRY	returns the derivative of the Airy function
DEVIANCE	returns the deviance from a specified distribution
DIGAMMA	returns the DIGAMMA function
ERF	returns the normal error function
ERFC	returns the complementary normal error function
EXP	returns the exponential function

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
LOG1PX	returns the log of 1 plus the argument
LOG2	returns the logarithm base 2
LOG10	returns the logarithm base 10
LOGISTIC	returns the logistic value of each argument
MOD	returns the remainder value
MSPLINT	returns the ordinate of a monotonicity-preserving interpolating spline
SIGN	returns the sign of a value
CALL SOFTMAX	returns the softmax value for each argument
SQRT	returns the square root of a value
TNONCT	returns the value of the noncentrality parameter from the Student's *t* distribution
TRIGAMMA	returns the value of the TRIGAMMA function

Probability Functions

You can call the following Base SAS functions for computing probabilities that are associated with statistical distributions. The functions that are indicated with an asterisk (*) are deprecated. You should use the CDF function instead.

CDF	computes cumulative distribution functions
LOGCDF	returns the logarithm of a left cumulative distribution function
LOGPDF	computes the logarithm of a probability function
LOGSDF	computes the logarithm of a survival function
PDF	computes probability density functions
POISSON*	returns the probability from a Poisson distribution
PROBBETA*	returns the probability from a beta distribution
PROBBNML*	returns the probability from a binomial distribution
PROBBNRM	returns the probability from the bivariate normal distribution
PROBCHI*	returns the probability from a chi-squared distribution
PROBF*	returns the probability from an F distribution
PROBGAM*	returns the probability from a gamma distribution
PROBHYPR*	returns the probability from a hypergeometric distribution
PROBMC	returns a probability or a quantile from various distributions for multiple comparisons of means
PROBNEGB*	returns the probability from a negative binomial distribution
PROBNORM*	returns the probability from the standard normal distribution
PROBT*	returns the probability from a *t* distribution
SDF	computes a survival function

Quantile Functions

You can call the following Base SAS functions for computing quantiles of statistical distributions. The functions that are indicated with an asterisk (*) are deprecated. You should use the QUANTILE function instead.

BETAINV*	returns a quantile from the beta distribution
CINV*	returns a quantile from the chi-squared distribution
FINV*	returns a quantile from the F distribution
GAMINV*	returns a quantile from the gamma distribution
PROBIT*	returns a quantile from the standard normal distribution
QUANTILE	returns the quantile from the specified distribution
SQUANTILE	returns the quantile when you specify the right probability (SDF)
TINV*	returns a quantile from the *t* distribution

Random Number Functions and Subroutines

You can call the following Base SAS functions to simulate random values from statistical distributions. However, the SAS/IML language supports the RANDGEN subroutine, which is a more efficient way to generate random values.

The functions that are indicated with an asterisk (*) are deprecated. You should use the RAND function instead.

RANBIN*	returns a random variate from a binomial distribution
RANCAU*	returns a random variate from a Cauchy distribution
RAND	returns a random variate from a specified distribution. (See the RANDGEN subroutine.)
RANEXP*	returns a random variate from an exponential distribution
RANGAM*	returns a random variate from a gamma distribution
RANNOR*	returns a random variate from a normal distribution
RANPOI*	returns a random variate from a Poisson distribution
RANTBL*	returns a random variate from a tabled probability
RANTRI*	returns a random variate from a triangular distribution
RANUNI*	returns a random variate from a uniform distribution
CALL STREAM	specifies a key value to use for subsequent random number generation by the RAND function.
CALL STREAMINIT	specifies a seed value to use for subsequent random number generation by the RAND function. (See the RANDSEED subroutine.)
CALL STREAMREWIND	rewinds a random number stream to the beginning

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
GEODISTANCE	returns the geodetic distance between two latitude and longitude coordinates
STFIPS	converts state postal codes to FIPS state codes
STNAME	converts state postal codes to uppercase state names
STNAMEL	converts state postal codes to mixed-case state names
ZIPCITY	returns a city name and the two-character postal code that corresponds to a zip code
ZIPCITYDISTANCE	returns the geodetic distance between two zip code locations
ZIPFIPS	converts zip codes to FIPS state codes
ZIPNAME	converts zip codes to uppercase state names
ZIPNAMEL	converts zip codes to mixed-case state names
ZIPSTATE	converts zip codes to state postal codes

Time Zone Functions

TZONEID	returns the current time zone ID
TZONENAME	returns the current standard or daylight savings time and the time zone name
TZONEOFF	returns the user time zone offset
TZONES2U	converts a SAS datetime value to a UTC datetime value
TZONEU2S	converts a UTC datetime value to a SAS datetime value

Trigonometric and Hyperbolic Functions

ARCOS	returns the arccosine
ARCOSH	returns the inverse hyperbolic cosine
ARSIN	returns the arcsine
ARSINH	returns the inverse hyperbolic cosine
ATAN	returns the arctangent
ARTANH	returns the inverse hyperbolic cosine
ATAN2	returns the arc tangent of two numeric variables
COS	returns the cosine
COSH	returns the hyperbolic cosine
COT	returns the cotangent
CSC	returns the cosecant
SIN	returns the sine
SINH	returns the hyperbolic sine
SEC	returns the secant

TAN	returns the tangent
CALL TANH	returns the hyperbolic tangent of each argument
TANH	returns the hyperbolic tangent

Truncation Functions

CEIL	returns the smallest integer ≥ the argument
CEILZ	returns the smallest integer that is greater than or equal to the argument, using zero fuzzing
FLOOR	returns the largest integer ≤ the argument
FLOORZ	returns the largest integer that is less than or equal to the argument, using zero fuzzing
FUZZ	returns the nearest integer if the argument is within 1E-12
INT	returns the integer portion of a value
INTZ	returns the integer portion of the argument, using zero fuzzing
MODZ	returns the remainder from the division of the first argument by the second argument, using zero fuzzing
ROUND	rounds a value to the nearest round-off unit
ROUNDE	rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples
ROUNDZ	rounds the first argument to the nearest multiple of the second argument, with zero fuzzing
TRUNC	returns a truncated numeric value of a specified length

Web Tools

HTMLDECODE	decodes a string that contains HTML numeric character references or HTML character entity references and returns the decoded string
HTMLENCODE	encodes characters by using HTML character entity references and returns the encoded string
URLDECODE	returns a string that was decoded by using the URL escape syntax
URLENCODE	returns a string that was encoded by using the URL escape syntax

References

Abramowitz, M., and Stegun, I. A., eds. (1972). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* 10th printing. New York: Dover.

Aiken, R. C. (1985). *Stiff Computation*. New York: Oxford University Press.

Al-Baali, M., and Fletcher, R. (1985). "Variational Methods for Nonlinear Least Squares." *Journal of the Operations Research Society* 36:405–421.

Al-Baali, M., and Fletcher, R. (1986). "An Efficient Line Search for Nonlinear Least Squares." *Journal of Optimization Theory and Applications* 48:359–377.

Ansley, C. F. (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.

Arthur, D., and Vassilvitskii, S. (2007). "k-Means++: The Advantages of Careful Seeding." In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1027–1035. Philadelphia: SIAM.

Baker, C. T. H. (1977). *The Numerical Treatment of Integral Equations*. Oxford: Clarendon Press.

Barnett, V., and Lewis, T. (1978). *Outliers in Statistical Data*. New York: John Wiley & Sons.

Barreto, H., and Maharry, D. (2006). "Least Median of Squares and Regression through the Origin." *Computational Statistics and Data Analysis* 50:1391–1397.

Barrodale, I., and Roberts, F. D. K. (1974). "Algorithm 478: Solution of an Overdetermined System of Equations in the L_1-Norm." *Communications of the ACM* 17:319–320.

Barry, D. A., Culligan-Hensley, P. J., and Barry, S. J. (1995). "Real Values of the W-Function." *ACM Transactions on Mathematical Software (TOMS)* 21:161–171.

Bates, D. M., Lindstrom, M. J., Wahba, G., and Yandell, B. S. (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*, edited by F. A. Lootsma, 39–43. London: Academic Press.

Beaton, A. E. (1964). *The Use of Special Matrix Operations in Statistical Calculus*. Princeton, NJ: Educational Testing Service.

Bickart, T. A., and Picel, Z. (1973). "High Order Stiffly Stable Composite Multistep Methods for Numerical Integration of Stiff Differential Equations." *BIT* 13:272–286.

Bishop, Y. M. M., Fienberg, S. E., and Holland, P. W. (1975). *Discrete Multivariate Analysis: Theory and Practice*. Cambridge, MA: MIT Press.

Boashash, B. (1988). "Note on the Use of the Wigner Distribution for Time-Frequency Signal Analysis." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36:1518–1521.

Box, G. E. P., and Jenkins, G. M. (1976). *Time Series Analysis: Forecasting and Control*. Rev. ed. San Francisco: Holden-Day.

Breiman, L. (1995). "Better Subset Regression Using the Nonnegative Garrote." *Technometrics* 37:373–384.

Brent, R. P. (1973). *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall. Chapter 5.

Brewer, C. A. (2013). "ColorBrewer 2.0: Color Advice for Cartography." Accessed June 4, 2013. http://colorbrewer.org/.

Brockwell, P. J., and Davis, R. A. (1991). *Time Series: Theory and Methods*. 2nd ed. New York: Springer-Verlag.

Brownlee, K. A. (1965). *Statistical Theory and Methodology in Science and Engineering*. New York: John Wiley & Sons.

Charnes, A., Frome, E. L., and Yu, P. L. (1976). "The Equivalence of Generalized Least Squares and Maximum Likelihood Estimation in the Exponential Family." *Journal of the American Statistical Association* 71:169–171.

Christensen, R. (1997). *Log-Linear Models and Logistic Regression*. 2nd ed. New York: Springer-Verlag.

Chung, C. F. (1996). "A Generalized Fractionally Integrated ARMA Process." *Journal of Time Series Analysis* 2:111–140.

Claasen, T. A. C. M., and Mecklenbräuker, W. F. G. (1980a). "The Wigner Distribution—a Tool for Time-Frequency Signal Analysis, Part I: Continuous Time Signals." *Philips Journal of Research* 35:217–250.

Claasen, T. A. C. M., and Mecklenbräuker, W. F. G. (1980b). "The Wigner Distribution—a Tool for Time-Frequency Signal Analysis, Part I: Discrete-Time Signals." *Philips Journal of Research* 35:276–300.

Cohen, L. (1995). *Time-Frequency Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Constantinides, A. G. (1970). "Spectral Transformations for Digital Filters." *Proceedings of the Institution of Electrical Engineers* 117:1585–1590.

Corless, R. M., Gonnet, G. H., Hare, D. E. G., Jeffrey, D. J., and Knuth, D. E. (1996). "On the Lambert W Function." *Advances in Computational Mathematics* 5:329–359.

Cox, D. R., and Hinkley, D. V. (1974). *Theoretical Statistics*. London: Chapman & Hall.

Daubechies, I. (1992). *Ten Lectures on Wavelets*. Vol. 61 of CBMS-NSF Regional Conference Series in Applied Mathematics. Philadelphia: SIAM.

Davies, L. (1992). "The Asymptotics of Rousseeuw's Minimum Volume Ellipsoid Estimator." *Annals of Statistics* 20:1828–1843.

De Boor, C. (1978). *A Practical Guide to Splines*. New York: Springer-Verlag.

De Jong, P. (1991). "Stable Algorithms for the State Space Model." *Journal of Time Series Analysis* 12:143–157.

Debnath, L. (2002). "Recent Developments in the Wigner-Ville Distribution and Time-Frequency Signal Analysis." *Proceedings of the Indian National Science Academy, Part A* 68:35–56.

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 and Applications* 28:453–482.

Devroye, L. (1986). *Non-uniform Random Variate Generation*. New York: Springer-Verlag. http://luc.devroye.org/rnbookindex.html.

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 espérances conditionnelles de champs gaussiens." *Annales scientifiques de l'Université de Clermont-Ferrand 2, Série Mathématique* 14:19–27.

Emerson, P. L. (1968). "Numerical Construction of Orthogonal Polynomials from a General Recurrence Formula." *Biometrics* 24:695–701.

Eskow, E., and Schnabel, R. B. (1991). "Algorithm 695: Software for a New Modified Cholesky Factorization." *ACM Transactions on Mathematical Software* 17:306–312.

Fishman, G. S. (1996). *Monte Carlo: Concepts, Algorithms, and Applications*. New York: John Wiley & Sons.

Fletcher, R. (1987). *Practical Methods of Optimization*. 2nd ed. Chichester, UK: John Wiley & Sons.

Fletcher, R., and Xu, C. (1987). "Hybrid Methods for Nonlinear Least Squares." *Journal of Numerical Analysis* 7:371–389.

Forsythe, G. E., Malcom, M. A., and Moler, C. B. (1967). *Computer Solution of Linear Algebraic Systems.* Chapter 17. Englewood Cliffs, NJ: Prentice-Hall.

Furnival, G. M., and Wilson, R. W. (1974). "Regression by Leaps and Bounds." *Technometrics* 16:499–511.

Gaffney, P. W. (1984). "A Performance Evaluation of Some FORTRAN Subroutines for the Solution of Stiff Oscillatory Ordinary Differential Equations." *ACM Transactions on Mathematical Software* 10:58–72.

Gay, D. M. (1983). "Subroutines for Unconstrained Minimization." *ACM Transactions on Mathematical Software* 9:503–524.

Gentle, J. E. (2003). *Random Number Generation and Monte Carlo Methods*. 2nd ed. Berlin: Springer-Verlag.

Gentleman, W. M., and Sande, G. (1966). "Fast Fourier Transforms for Fun and Profit." *AFIPS Proceedings of the Fall Joint Computer Conference* 19:563–578.

George, J. A., and Liu, J. W. (1981). *Computer Solutions of Large Sparse Positive Definite Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Geweke, J., and Porter-Hudak, S. (1983). "The Estimation and Application of Long Memory Time Series Models." *Journal of Time Series Analysis* 4:221–238.

Gill, P. E., 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*, edited by R. C. Milton and J. A. Nelder, 365–397. New York: Academic Press.

Golub, G. H., and Van Loan, C. F. (1989). *Matrix Computations*. 2nd ed. Baltimore: Johns Hopkins University Press.

Gonin, R., and Money, A. H. (1989). *Nonlinear L_p-Norm Estimation*. New York: Marcel Dekker.

Goodnight, J. H. (1979). "A Tutorial on the Sweep Operator." *American Statistician* 33:149–158.

Graybill, F. A. (1969). *Introduction to Matrices with Applications in Statistics*. Belmont, CA: Wadsworth.

Grizzle, J. E., Starmer, C. F., and Koch, G. G. (1969). "Analysis of Categorical Data by Linear Models." *Biometrics* 25:489–504.

Hadley, G. (1962). *Linear Programming*. Reading, MA: Addison-Wesley.

Harris, F. J. (1978). "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform." *Proceedings of the IEEE* 66:51–83.

Harvey, A. C. (1989). *Forecasting, Structural Time Series Models, and the Kalman Filter*. Cambridge: Cambridge University Press.

Harville, D. A. (1997). *Matrix Algebra from a Statistician's Perspective*. New York: Springer-Verlag.

Heinzel, G., Rüdiger, A., and Schilling, R. (2002). *Spectrum and Spectral Density Estimation by the Discrete Fourier Transform (DFT), Including a Comprehensive List of Window Functions and Some New Flat-Top Windows*. Tech. rep., Max-Planck-Institut für Gravitationsphysik (Albert-Einstein-Institut), Teilinstitut Hannover.

Hocking, R. R. (1985). *The Analysis of Linear Models*. Monterey, CA: Brooks/Cole.

Jenkins, M. A., and Traub, J. F. (1970). "A Three-Stage Algorithm for Real Polynomials Using Quadratic Iteration." *SIAM Journal on Numerical Analysis* 7:545–566.

Jennrich, R. I., and Moore, R. H. (1975). "Maximum Likelihood Estimation by Means of Nonlinear Least Squares." In *Proceedings of the Statistical Computing Section*, 57–65. Washington, DC: American Statistical Association.

Johnson, M. E. (1987). *Multivariate Statistical Simulation*. New York: John Wiley & Sons.

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* 15:107–115.

Kohn, R., and Ansley, C. F. (1982). "A Note on Obtaining the Theoretical Autocovariances of an ARMA Process." *Journal of Statistical Computation and Simulation* 15:273–283.

Korff, F. A., Taback, M. A. M., and Beard, J. H. (1952). "A Coordinated Investigation of a Food Poisoning Outbreak." *Public Health Reports* 67:909–913.

Kotz, S., Balakrishnan, N., and Johnson, N. L. (2000). *Continuous Multivariate Distributions*. 2nd ed. New York: Wiley-Interscience.

Kotz, S., and Nadarajah, S. (2004). *Multivariate* t *Distributions and Their Applications*. Cambridge: Cambridge University Press.

Kruskal, J. B. (1964a). "Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis." *Psychometrika* 29:1–27.

Kruskal, J. B. (1964b). "Nonmetric Multidimensional Scaling: A Numerical Method." *Psychometrika* 29:115–129.

Kwok, J. T., and Tsang, I. W. (2004). "The Pre-image Problem in Kernel Methods." *IEEE Transactions on Neural Networks* 15:1517–1525.

Lee, W., and Gentle, J. E. (1986). "The LAV Procedure." In *SUGI Supplemental Library User's Guide*, 257–260. Cary, NC: SAS Institute Inc.

Lewart, C. R. (1973). "Algorithm 463: Algorithms SCALE1, SCALE2, and SCALE3 for Determination of Scales on Computer Generated Plots." *Communications of the ACM* 16:639–640. https://dl.acm.org/citation.cfm?doid=362375.362417.

Lindström, P., and Wedin, P.-A. (1984). "A New Line-Search Algorithm for Nonlinear Least-Squares Problems." *Mathematical Programming* 29:268–296.

Madsen, K., and Nielsen, H. B. (1993). "A Finite Smoothing Algorithm for Linear L_1 Estimation." *SIAM Journal on Optimization* 3:223–235.

Mallat, S. (1989). "Multiresolution Approximation and Wavelets." *Transactions of the American Mathematical Society* 315:69–88.

Marple, L., Jr. (1999). "Computing the Discrete-Time 'Analytic' Signal via FFT." *IEEE Transactions on Signal Processing* 47:2600–2603.

Matsumoto, M., and Nishimura, T. (1998). "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator." *ACM Transactions on Modeling and Computer Simulation* 8:3–30.

Matsumoto, M., and Nishimura, T. (2002). "Mersenne Twister with Improved Initialization." Accessed April 10, 2015. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html.

McKean, J. W., and Schrader, R. M. (1987). "Least Absolute Errors Analysis of Variance." In *Statistical Data Analysis Based on L_1 Norm and Related Methods*, edited by Y. Dodge, 297–305. Amsterdam: North-Holland.

McLeod, A. I. (1975). "Derivation of the Theoretical Autocovariance Function of Autoregressive–Moving Average Time Series." *Journal of the Royal Statistical Society, Series C* 24:255–256.

Mittnik, S. (1990). "Computation of Theoretical Autocovariance Matrices of Multivariate Autoregressive Moving Average Time Series." *Journal of the Royal Statistical Society, Series B* 52:151–155.

Moler, C. B. (2004). *Numerical Computing with MATLAB*. Natick, MA: MathWorks. http://www.mathworks.com/moler/index_ncm.html.

Moler, C. B. (2011). *Experiments with MATLAB*. Natick, MA: MathWorks. Available as e-book only. http://www.mathworks.com/moler/exm/.

Monro, D. M., and Branch, J. L. (1977). "Algorithm AS 117: The Chirp Discrete Fourier Transform of General Length." *Journal of the Royal Statistical Society, Series C* 26:351–361.

Moré, J. J. (1978). "The Levenberg-Marquardt Algorithm: Implementation and Theory." In *Lecture Notes in Mathematics*, vol. 30, edited by G. A. Watson, 105–116. Berlin: Springer-Verlag.

Moré, J. J., and Sorensen, D. C. (1983). "Computing a Trust-Region Step." *SIAM Journal on Scientific and Statistical Computing* 4:553–572.

Nelder, J. A., and Wedderburn, R. W. M. (1972). "Generalized Linear Models." *Journal of the Royal Statistical Society, Series A* 135:370–384.

Nijenhuis, A., and Wilf, H. S. (1978). *Combinatorial Algorithms*. New York: Academic Press.

Nussbaumer, H. J. (1982). *Fast Fourier Transform and Convolution Algorithms*. 2nd ed. New York: Springer-Verlag.

Ogden, R. T. (1997). *Essential Wavelets for Statistical Applications and Data Analysis*. Boston: Birkhäuser.

Oppenheim, A. V., and Schafer, R. W. (2010). *Discrete-Time Signal Processing*. 3rd ed. New York: Pearson.

Osborne, M. R. (1985). *Finite Algorithms in Optimization and Data Analysis*. New York: John Wiley & Sons.

Parks, T. W., and Burrus, C. S. (1987). *Digital Filter Design*. New York: Wiley-Interscience.

Pizer, S. M. (1975). *Numerical Computing and Mathematical Analysis*. Chicago: Science Research Associates.

Pocock, S. J. (1977). "Group Sequential Methods in the Design and Analysis of Clinical Trials." *Biometrika* 64:191–199.

Pocock, S. J. (1982). "Interim Analyses for Randomized Clinical Trials: The Group Sequential Approach." *Biometrics* 38:153–162.

Powell, M. J. D. (1977). "Restart Procedures for the Conjugate Gradient Method." *Mathematical Programming* 12:241–254.

Powell, M. J. D. (1978a). "Algorithms for Nonlinear Constraints That Use Lagrangian Functions." *Mathematical Programming* 14:224–248.

Powell, M. J. D. (1978b). "A Fast Algorithm for Nonlinearly Constrained Optimization Calculations." In *Lecture Notes in Mathematics*, vol. 630, edited by G. A. Watson, 144–175. Berlin: Springer-Verlag.

Powell, M. J. D. (1982a). "Extensions to Subroutine VF02AD." In *Systems Modeling and Optimization, Lecture Notes in Control and Information Sciences*, vol. 38, edited by R. F. Drenick and F. Kozin, 529–538. Berlin: Springer-Verlag.

Powell, M. J. D. (1982b). *VMCWD: A Fortran Subroutine for Constrained Optimization*. Technical Report DAMTP 1982/NA4, Department of Applied Mathematics and Theoretical Physics, University of Cambridge.

Powell, M. J. D. (1992). *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*. Technical Report DAMTP 1992/NA5, Department of Applied Mathematics and Theoretical Physics, University of Cambridge.

Rabiner, L. R., and Schafer, R. W. (2007). *Introduction to Digital Speech Processing*. Boston: Now Publishers.

Ralston, A., and Rabinowitz, P. (1978). *A First Course in Numerical Analysis*. New York: McGraw-Hill.

Rao, C. R., and Mitra, S. K. (1971). *Generalized Inverse of Matrices and Its Applications*. New York: John Wiley & Sons.

Reinsch, C. H. (1967). "Smoothing by Spline Functions." *Numerische Mathematik* 10:177–183.

Reinsel, G. C. (1997). *Elements of Multivariate Time Series Analysis*. 2nd ed. New York: Springer-Verlag.

Rice, S. O. (1973). "Efficient Evaluation of Integrals of Analytic Functions by the Trapezoidal Rule." *Bell System Technical Journal* 52:707–722.

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*, edited by W. Grossmann, G. Pflug, I. Vincze, and W. Wertz, 283–297. Dordrecht, Netherlands: D. Reidel Publishing.

Rousseeuw, P. J., and Croux, C. (1993). "Alternatives to the Median Absolute Deviation." *Journal of the American Statistical Association* 88:1273–1283.

Rousseeuw, P. J., and Hubert, M. (1996). "Recent Development in PROGRESS." *Computational Statistics and Data Analysis* 21:67–85.

Rousseeuw, P. J., and Hubert, M. (1997). *Recent Developments in PROGRESS*. Hayward, CA: Institute of Mathematical Statistics.

Rousseeuw, P. J., and Leroy, A. M. (1987). *Robust Regression and Outlier Detection*. New York: John Wiley & Sons.

Rousseeuw, P. J., and Van Driessen, K. (1998). *Computing LTS Regression for Large Data Sets*. Technical report, University of Antwerp.

Rousseeuw, P. J., and Van Driessen, K. (1999). "A Fast Algorithm for the Minimum Covariance Determinant Estimator." *Technometrics* 41:212–223.

Rousseeuw, P. J., and Van Zomeren, B. C. (1990). "Unmasking Multivariate Outliers and Leverage Points." *Journal of the American Statistical Association* 85:633–639.

Roy, S. C. D. (2005). "A Simple Derivation of the Spectral Transformations for IIR Filters." *IEEE Transactions on Education* 48:274–278.

Schatzoff, M., Tsao, R., and Fienberg, S. (1968). "Efficient Calculation of All Possible Regressions." *Technometrics* 10:769–779.

Schölkopf, B., Smola, A., and Müller, K.-R. (1998). "Nonlinear Component Analysis as a Kernel Eigenvalue Problem." *Neural Computation* 10:1299–1319.

Shampine, L. (1978). "Stability Properties of Adams Codes." *ACM Transactions on Mathematical Software* 4:323–329.

Sikorsky, K. (1982). "Optimal Quadrature Algorithms in H_P Spaces." *Numerische Mathematik* 39:405–410.

Sikorsky, K., and Stenger, F. (1984). "Optimal Quadratures in H_P Spaces." *ACM Transactions on Mathematical Software* 3:140–151.

Singleton, R. C. (1969). "An Algorithm for Computing the Mixed Radix Fast Fourier Transform." *IEEE Transactions on Audio and Electroacoustics* 17:93–103.

Sowell, F. (1992). "Maximum Likelihood Estimation of Stationary Univariate Fractionally Integrated Time Series Models." *Journal of Econometrics* 53:165–188.

Squire, W. (1987). "Comparison of Gauss-Hermite and Midpoint Quadrature with Application to the Voigt Function." In *Numerical Integration: Recent Developments*, edited by P. Keast and G. Fairweather, 111–112. Dordrecht, Netherlands: D. Reidel Publishing.

Stenger, F. (1973a). "Integration Formulas Based on the Trapezoidal Formula." *Journal of the Institute of Mathematics and Its Applications* 12:103–114.

Stenger, F. (1973b). "Remarks on Integration Formulas Based on the Trapezoidal Formula." *Journal of the Institute of Mathematics and Its Applications* 19:145–147.

Stenger, F. (1978). "Optimal Convergence of Minimum Norm Approximations in H_P." *Numerische Mathematik* 29:345–362.

Stoer, J., and Bulirsch, R. (1980). *Introduction to Numerical Analysis*. New York: Springer-Verlag.

Thisted, R. A. (1988). *Elements of Statistical Computing: Numerical Computation*. London: Chapman & Hall.

Trethewey, M. W. (2000). "Window and Overlap Processing Effects on Power Estimates from Spectra." *Mechanical Systems and Signal Processing* 14:267–278.

Trotter, H. F. (1962). "Algorithm 115: PERM." *Communications of the ACM* 5:434–435.

Wahba, G. (1990). *Spline Models for Observational Data*. Philadelphia: SIAM.

Wang, S. K., and Tsiatis, A. A. (1987). "Approximately Optimal One Parameter Boundaries for Group Sequential Trials." *Biometrics* 43:193–199.

Wilkinson, J. H., and Reinsch, C. (1971). *Handbook for Automatic Computation: Linear Algebra.* Vol. 2. New York: Springer-Verlag.

Woodfield, T. J. (1988). "Simulating Stationary Gaussian ARMA Time Series." In *Computing Science and Statistics: Proceedings of the Twentieth Symposium on the Interface* (Fairfax, VA), edited by E. J. Wegman, D. T. Gantz, and J. J. Miller, 612–617. Alexandria, VA: American Statistical Association.

Young, F. W. (1981). "Quantitative Analysis of Qualitative Data." *Psychometrika* 46:357–388.

Zhang, K., Tsang, I. W., and Kwok, J. T. (2008). "Improved Nyström Low-Rank Approximation and Error Analysis." In *Proceedings of the 25th International Conference on Machine Learning*, 1232–1239. New York: ACM.

Chapter 26
Module Library

Contents

Overview **1129**
Contents of the IMLMLIB Library **1129**
IMLMLIB and the STORAGE library **1131**
Accessing the IMLMLIB Source Code **1132**
Order of Resolution for Functions and Subroutines **1132**
Error Diagnostics **1133**
Modules for Multivariate Random Sampling **1133**
Syntax for Demonstration Modules **1134**
GBXWHSKR Call 1134
GPROBCNT Call 1134
GXYPLOT Call 1134
QUADREG Call 1135
REGRESS Call 1136
TABPRT Call 1138
References **1138**

Overview

The IMLMLIB library contains modules written in the SAS/IML language. The library contains both functions and subroutines. You do not have to explicitly load these modules: they are automatically loaded at run time when they are called by a SAS/IML program.

Contents of the IMLMLIB Library

The IMLMLIB library contains the following computational modules:

CORR2COV function scales a correlation matrix into a covariance matrix

COV2CORR function scales a covariance matrix into a correlation matrix

CV function computes the sample coefficient of variation

EXPMATRIX function computes the exponential of a matrix

ISEMPTY function returns 1 if the argument is an empty matrix (zero rows and columns) and 1 otherwise

KURTOSIS function computes the sample kurtosis
MAGIC function returns a magic square of a given size
MAHALANOBIS function computes Mahalanobis distance
MEDIAN function returns the median of numeric data
QUADREG call performs quadratic regression
QUARTILE function computes quartiles
REGRESS call performs regression analysis
SKEWNESS function computes the sample skewness
STANDARD function standardizes numeric data

The IMLMLIB library contains the following utility modules:

BLANKSTR function returns a blank string of a specified length.
COL function returns a matrix, M, that is the same size as the input matrix and such that $M[i, j] = i$.
COLVEC function converts a matrix into a column vector
FEVAL function evaluates a function indirectly
EXPANDGRID function returns a matrix that contains all combinations of elements from specified vectors
MODULESTACK function returns the calling sequence for a module
NDX2SUB function converts matrix indices to subscripts
PALETTE function returns a discrete color palette that is suitable for visualizing categorical data
ROW function returns a matrix, M, that is the same size as the input matrix and such that $M[i, j] = j$.
ROWVEC function converts a matrix into a row vector
RSUBSTR function replaces substrings
SUB2NDX function converts matrix subscripts to indices
TABPRT call prints matrices in tabular format
WIDETOLONG call transforms data from wide to long form

The library contains the following functions for generating random samples from statistical distributions:

RANDDIRICHLET function generates a random sample from a Dirichlet distribution
RANDFUN function returns a matrix of random numbers from a specified distribution
RANDMULTINOMIAL function generates a random sample from a multinomial distribution
RANDMVT function generates a random sample from a multivariate Student's t distribution
RANDNORMAL function generates a random sample from a multivariate normal distribution
RANDWISHART function generates a random sample from a Wishart distribution

The library contains the following graphical subroutines that produce ODS graphics:

BAR call creates a bar chart

BOX call creates a box plot

HEATMAPCONT call creates a heat map with a continuous color ramp

HEATMAPDISC call creates a heat map with a discrete color ramp

HISTOGRAM call creates a histogram

SCATTER call creates a scatter plot

SERIES call creates a series plot

SPECTROGRAM call creates a spectrogram of a short-time Fourier transform

For compatibility with previous releases, the IMLMLIB library contains the following graphical subroutines that produce legacy graphics:

GBXWHSKR call draws a box-and-whiskers plot

GPROBCNT call draws a scatter plot with bivariate normal probability contours

GXYPLOT call draws scatter plots of *x*-*y* data

IMLMLIB and the STORAGE library

As described in Chapter 19, SAS/IML enables you to store and load matrices and modules in your own STORAGE library. The STORE, LOAD, REMOVE, and RESET STORAGE commands apply to the STORAGE library and enable you to store and load user-defined matrices and modules.

In contrast, the IMLMLIB library contains predefined read-only modules. You cannot store additional modules in IMLMLIB.

You can use the SHOW command to obtain information about both the IMLMLIB and the STORAGE libraries, as described in the following list:

- SHOW OPTIONS displays the settings of the STORAGE and IMLMLIB libraries and shows whether the libraries are open.
- 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 that are loaded in the current environment. These include modules loaded from either library and modules defined in the current session.

Accessing the IMLMLIB Source Code

The IMLMLIB library is a catalog in the SASHELP directory. The catalog contains an entry of type IMOD for each module. Each entry is a module stored in its compiled form.

The SAS/IML source code that defines a 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 the SAS windowing environment by using the COPY command and specifying the four-level name SASHELP.IML.*modulename*.SOURCE

The source code can be edited for customization or enhancements, and can be included in other SAS/IML applications. The modules also illustrate a variety of language features that can be used to solve statistical problems.

Order of Resolution for Functions and Subroutines

The SAS/IML language resolves functions in the following order:

1. User-defined SAS/IML modules that exist in the current environment
2. Function in the STORAGE library, if it is open
3. Functions in the IMLMLIB library
4. Functions built into SAS/IML software
5. SAS DATA step functions

Prior to SAS/IML 13.1, the order of resolution was different. The order was changed in order to make it easier for programmers to call user-defined functions.

When you call a module by using the CALL statement, the SAS/IML language resolves subroutines in the following order:

1. Subroutines built into SAS/IML software
2. User-defined SAS/IML modules that exist in the current environment
3. Subroutines in the STORAGE library, if it is open
4. Subroutines in the IMLMLIB library
5. SAS DATA step subroutines

If you want to be sure that user-defined subroutines take precedence over built-in subroutines, use the RUN statement. When you call a module by using the RUN statement, the SAS/IML language resolves subroutines in the following order:

1. User-defined SAS/IML modules that exist in the current environment
2. Subroutines in the STORAGE library, if it is open
3. Subroutines in the IMLMLIB library
4. Subroutines that are built into SAS/IML software
5. SAS DATA step subroutines

Error Diagnostics

When a run-time error occurs in a SAS/IML module, the program execution pauses inside the module environment. The SAS Log contains error diagnostics with a full traceback that can help to locate the problem. In the case of a loaded module, the traceback includes line offsets instead of the absolute SAS Log line numbers. The offsets can be used to track the problem into the source code that defined the module. The START statement at the beginning of the module definition always has an offset value of 1.

Offsets apply only to loaded modules. For modules that are explicitly defined in any given session, absolute line numbers are used in the traceback.

Modules for Multivariate Random Sampling

SAS/IML software includes pre-defined modules that generate random samples from common multivariate distributions. For univariate distributions, you can generate random samples from many distributions by using the RANDGEN subroutine.

RANDDIRICHLET generates a random sample from a Dirichlet distribution, which is a multivariate generalization of the beta distribution.

RANDFUN function returns a matrix of random numbers from a specified distribution

RANDMULTINOMIAL generates a random sample from a multinomial distribution, which is a multivariate generalization of the binomial distribution.

RANDMVT generates a random sample from a multivariate Student's *t* distribution

RANDNORMAL generates a random sample from a multivariate normal distribution

RANDWISHART generates a random sample from a Wishart distribution, which is a multivariate generalization of the gamma distribution.

All of the modules compute their results by using transformations of univariate random samples generated by the RANDGEN subroutine. Thus you can use the RANDSEED subroutine to set the seed for the modules.

Although you can sample from a multivariate normal distribution by using the built-in VNORMAL subroutine, the VNORMAL subroutine does not use the random number seed set in RANDSEED. To ensure independence and reproducibility of random number streams, the RANDNORMAL function is recommended.

For an overview of multivariate sampling, see Gentle (2003).

Syntax for Demonstration Modules

A few modules in the IMLMLIB are intended for demonstration purposes, rather than for serious analysis. Those modules, along with modules that produce legacy graphics, are documented in this section. The remaining modules in the IMLMLIB are described in the "Statements, Functions, and Subroutines" on page 528

GBXWHSKR Call

RUN GBXWHSKR(*matrix***);**

This subroutine is deprecated.

The GBXWHSKR module draws a box-and-whiskers plot for univariate numeric data contained in the specified $n \times m$ 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. You can edit the module source code in order to add viewports, text, or colors.

GPROBCNT Call

RUN GPROBCNT(*x, y*<*,p*>**);**

This subroutine is deprecated.

The GPROBCNT module draws probability contours for the bivariate normal distribution. One contour is drawn for each value in the matrix p, which must contain entries between zero and one.

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.

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 specify 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.

GXYPLOT Call

RUN GXYPLOT(*x, y***);**

This subroutine is deprecated.

The GXYPLOT draws a scatter plot of the data in the *x* and *y* arguments. 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 bivariate data, including axes with labeled tick marks. You cannot produce graphics until you specify the CALL GSTART statement. The plot remains open for further additions (such as a title and axis labels) until you specify the CALL GCLOSE statement, 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.

QUADREG Call

RUN QUADREG(*xopt, yopt, type, parms, x, y***);**

The QUADREG module fits a quadratic response surface to data. It is primarily used for demonstration purposes. The inputs to the QUADREG subroutine are as follows:

xopt is a returned value that contains $m \times 1$ critical factor values.

yopt is a returned value that contains the critical response value.

type is a returned character string that contains the solution type (maximum or minimum).

parms is a returned value that contains the parameter estimates for the quadratic model.

x is an $n \times m$ data 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 saddle point) by computing the eigenvalues of the estimated parameters.

```
x = { -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 quadreg( xopt, yopt, nature, parms, x, y );
print parms[rowname={c b1 b2 a11 a12 a22} label="Parameter Estimates"],
    xopt[rowname={x1 x2} label="Critical Factor Values"],
    nature[label=""] "Response" yopt [label=""];
```

Figure 26.1 Parameter Estimates and Optima

Parameter Estimates	
C	81.222222
B1	1.9666667
B2	0.2166667
A11	-3.933333
A12	-2.225
A22	-1.383333

Critical Factor Values	
X1	0.2949376
X2	-0.158881

Maximum Response	81.495032

REGRESS Call

RUN REGRESS(*x, y, name, <tval>, <l1>, <l2>, <l3>***);**

The REGRESS module performs ordinary least squares regression. It is primarily used for demonstration purposes.

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 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. An example follows:

```
/* U.S. Population for decades beginning 1790, in millions */
name = { "Intercept", "Decade", "Decade**2" };
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 };
/* 5 dof at 0.025 level to get 95% confidence interval */
```

```
tval = quantile("T", 1-0.025, 5);
l1 = { 0 1 0 };   /* test hypothesis lb=0 for linear coef */
l2 = { 0 1 0,     /* test hypothesis lb=0 for linear,quad */
       0 0 1 };
l3 = { 0 1 1 };   /* test hypothesis lb=0 for linear+quad */
run regress( x, y, name, tval, l1, l2, l3 );
```

Figure 26.2 Regression Analysis

name	b	stdb	t	probt
Intercept	5.0693393	0.9655939	5.2499702	0.0033263
Decade	-1.109935	0.4923003	-2.254588	0.0738509
Decade**2	0.5396369	0.0533975	10.10604	0.0001625

Covariance of Estimates

	Intercept	Decade	Decade**2
Intercept	0.9324	-0.436	0.0428
Decade	-0.436	0.2424	-0.026
Decade2**	0.0428	-0.026	0.0029

Correlation of Estimates

	Intercept	Decade	Decade**2
Intercept	1	-0.918	0.8295
Decade	-0.918	1	-0.976
Decade2**	0.8295	-0.976	1

Predicted values, Residuals, and Limits

y	yhat	resid	h	lowerm	upperm	lower	upper
3.929	4.499	-0.57	0.7083	3.0017	5.9964	2.1737	6.8244
5.308	5.008	0.3	0.2798	4.067	5.949	2.9954	7.0207
7.239	6.5963	0.6427	0.2321	5.7391	7.4535	4.6214	8.5711
9.638	9.2638	0.3742	0.2798	8.3228	10.205	7.2511	11.276
12.866	13.011	-0.145	0.2798	12.07	13.952	10.998	15.023
17.069	17.837	-0.768	0.2321	16.979	18.694	15.862	19.812
23.191	23.742	-0.551	0.2798	22.801	24.683	21.729	25.755
31.443	30.727	0.7164	0.7083	29.229	32.224	28.401	33.052

Test Hypothesis that l b = 0

	f	dfn	dfe	prob
for Linear Coef	5.0831686	1	5	0.0739

	f	dfn	dfe	prob
for Linear,Quad Coef	666.51095	2	5	<.0001

	f	dfn	dfe	prob
for Linear+Quad Coef	1.6774629	1	5	0.2518

TABPRT Call

RUN TABPRT(*matrix***);**

The TABPRT module is part of the IMLMLIB library. It is included for demonstration purposes. The TABPRT module prints a matrix in a tabular format. The module can be useful for printing large matrices. 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.

```
r = shape(1:50, 5, 10);  /* a 5 x 10 numeric matrix */
run tabprt( r );
```

Figure 26.3 Tabular Display

	X					
	COL1	COL2	COL3	COL4	COL5	COL6
ROW1	1.000	2.000	3.000	4.000	5.000	6.000
ROW2	11.000	12.000	13.000	14.000	15.000	16.000
ROW3	21.000	22.000	23.000	24.000	25.000	26.000
ROW4	31.000	32.000	33.000	34.000	35.000	36.000
ROW5	41.000	42.000	43.000	44.000	45.000	46.000

	X			
	COL7	COL8	COL9	COL10
ROW1	7.000	8.000	9.000	10.000
ROW2	17.000	18.000	19.000	20.000
ROW3	27.000	28.000	29.000	30.000
ROW4	37.000	38.000	39.000	40.000
ROW5	47.000	48.000	49.000	50.000

References

Gentle, J. E. (2003). *Random Number Generation and Monte Carlo Methods*. 2nd ed. Berlin: Springer-Verlag.

Subject Index

ABORT statement
    exiting PROC IML, 528
ABS function
    absolute value, 528
ADDITION operator
    adds corresponding matrix elements, 506
ALL function
    checking for nonzero elements, 529
ALLCOMB function
    generate combinations, 529
ALLPERM function
    generate permutations, 531
ANY function
    checking for nonzero elements, 532
APPEND statement
    SAS data sets, 534
APPLY function, 537
ARMACOV call
    autocovariance sequence, 539
ARMALIK call
    log likelihood and residuals, 540
ARMASIM function
    simulating univariate ARMA series, 542

BAR call
    create a bar chart, 543
Basic time series analysis
    autocovariance function of ARMA model, 289
    example, 288
    generating an ARMA process, 289
    log-likelihood function of ARMA model, 289
    overview, 288
Bessel function
    finding nonzero roots and derivatives of, 723
Biconjugate Gradient Algorithm, 465, 470
BIN function
    dividing numeric values into bins, 547
BLANKSTR function
    create blank strings, 549
BLOCK function
    forming block-diagonal matrices, 549
BOX call
    create box plot, 550
box-and-whiskers plot, 1134
BRANKS function
    computing bivariate ranks, 553
BSPLINE function
    computing B-spline basis, 554
BTRAN function
    computing the block transpose, 556
BYTE function
    returning values in a computer's character set, 557

CALL statement
    calling a subroutine or function, 558
Calling External Modules, 505
Calling R, 505
Calling SAS, 505
CCEPSTRUM call
    computes the complex cepstrum, 558
CHANGE call
    replacing text in an array, 560
CHAR function
    character representation of a numeric matrix, 561
Character Manipulation Functions, 494
CHOOSE function
    choosing and changing elements, 562
CLOSE statement
    closing SAS data sets, 563
CLOSEFILE statement
    closing a file, 564
COL function, 565
column vector, 566
COLVEC function
    reshaping matrices, 566
Combinatorial Functions, 494
COMPARISON operator
    compare matrix elements, 507
CONCAT function
    performing elementwise string concatenation, 569
CONCATENATION operator, horizontal
    concatenates matrices horizontally, 508
CONCATENATION operator, vertical
    concatenates matrices vertically, 509
Conjugate Gradient Algorithm, 465, 467
CONTENTS function
    obtaining the variables in SAS data sets, 570
CONTINUE statement
    goes to the next iteration of a DO loop, 571
Control Statements, 501
Convert indices to subscripts, 820
Convert subscripts to indices, 1030
CONVEXIT function
    calculating convexity of noncontingent cash flows, 571
CORR function

computing sample correlations, 572
CORR2COV function
convert correlation matrix, 574
COUNTMISS function
counting missing values, 574
COUNTN function
counting nonmissing values, 575
COUNTUNIQUE function
counting unique values, 576
COV function
computing sample covariances, 577
COV2CORR function
convert covariance matrix, 578
COVLAG function
computing autocovariance estimates, 578
CREATE statement
creating new SAS data sets, 579
CSHAPE function
reshaping and repeating character values, 582
CUPROD function
calculating cumulative products, 584
CUSUM function
calculating cumulative sums, 584
CV function
compute the coefficient of variation, 585
CVEXHULL function
finding a convex hull, 585

Data Set and File Statements, 501
DATASETS function
obtaining names of SAS data sets, 586
DELETE call
deleting SAS data sets, 586
DELETE statement
marking observations for deletion, 587
DESIGN function
creating a design matrix, 588
DESIGNF function
creating a full-rank design matrix, 589
DET function
computing determinants of a square matrix, 589
DFCONV function
computes the convolution of two vectors, 590
DFDESIGN call
FUNC DESCRIPTION HERE, 591
DFFILT function
filters a time series using a digital filter, 593
DFFREQZ function
generates a digital filter's frequency response by using transfer function coefficients, 596
DFFREQZZPK function
generates a digital filter's frequency response by using transfer function zeros, poles and gain, 597
DFMEDFILT function
filters a time series using a median filter, 598
DFORDER call
computes the minimum order required for a digital filter, 600
DFSOSFILT function
filters a time series using a digital filter's second-order section form, 605
DFSOSFREQZ function
generates a digital filter's frequency response corresponding to its second-order sections form, by using transfer function coefficients, 607
DFSOSFREQZZPK function
generates a digital filter's frequency response corresponding to its second-order sections form, by using transfer function zeros, poles and gain, 608
DIAG function
creating a diagonal matrix, 610
DIF function
computing difference of lagged values, 611
Digital Filtering Functions, 498
DIMENSION function
returns the dimensions of a matrix, 612
DIRECT PRODUCT operator
takes the direct product of two matrices, 510
DISTANCE function
pairwise distance between points, 612
DIVISION operator
performs elementwise division, 511
DO DATA statement
repeating a loop until, 617
DO function
producing an arithmetic sequence, 615
DO statement
DATA clause, 617
grouping statements as a unit, 616
UNTIL clause, 618
WHILE clause, 619
DO statement, iterative
iteratively executing a DO group, 617
DO UNTIL statement
conditionally executing statements iteratively, 618
DO WHILE statement
conditionally executing statements iteratively, 619
DURATION function
calculating modified duration of noncontingent cash flows, 620

ECHELON function
reducing a matrix to row-echelon normal form, 621
EDIT statement

opening a SAS data set for editing, 621
EIGEN call
computing eigenvalues and eigenvectors, 623
Eigenvalue Decomposition
compared with ODE call, 860
EIGVAL function
computing eigenvalues, 626
EIGVEC function
computing right eigenvectors, 627
ELEMENT function
finding elements that are contained in a set, 628
ELEMENT MAXIMUM operator
selects the larger of two elements, 512
ELEMENT MINIMUM operator
selects the smaller of two elements, 513
END statement
ending a DO loop or DO statement, 629
ENDSUBMIT statement, 236
EXECUTE call
executing statements immediately, 629
EXECUTEFILE call
executing statements in a file, 630
EXP function
calculating the exponential, 632
EXPMATRIX function
exponential of a matrix, 632
ExportDataSetToR subroutine, 247
ExportMatrixToR subroutine, 247
ExportTableToR subroutine, 247

FARMACOV call
generating an ARFIMA(p, d, q) process, 636
FARMAFIT call
estimation of an ARFIMA(p, d, q) model, 638
FARMALIK call
computing the log-likelihood for an ARFIMA(p, d, q) model, 639
FARMASIM call
generating an ARFIMA(p, d, q) process, 641
FDIF call
computing a fractionally differenced process, 643
FEVAL function
evaluate a function indirectly, 643
FFT function
compute the finite Fourier transform, 644
FFTC function
compute the finite Fourier transform of a complex input, 646
FILE statement
opening or pointing to an external file, 647
FIND statement
finding observations, 648
FINISH statement
denoting the end of a module, 649
Forward rates, 650
Fractionally integrated time series analysis
ARFIMA modeling, 310
autocovariance function, 310
example, 307
fractional differencing, 310
generating a fractional time series, 310
log-likelihood function, 310
overview, 307
FREE statement
freeing matrix storage space, 650
FROOT function
univariate root finding, 651
FULL function
converting sparse to dense storage, 652

GAEND call
ending a genetic algorithm optimization, 654
GAGETMEM call
getting current members of the solution population for a genetic algorithm optimization, 654
GAGETVAL call
getting current solution objective function values for a genetic algorithm optimization, 655
GAINIT call
creating an initial solution population for a genetic algorithm optimization, 655
GAREEVAL call
reevaluating the objective function values for a solution population of a genetic algorithm optimization, 656
GAREGEN call
regenerating a solution population by application of selection and genetic operators, 657
GASETCRO call
setting the crossover operator for a genetic algorithm optimization, 657
GASETMUT call
setting the mutation operator for a genetic algorithm optimization, 661
GASETOBJ call
setting the objective function for a genetic algorithm optimization, 663
GASETSEL call
setting the selection parameters for a genetic algorithm optimization, 664
GASETUP function
setting up a genetic algorithm optimization problem, 664
GBXWHSKR call
box-and-whiskers plot, 1134
GENEIG call
generalized eigenvalue problem, 666

Genetic Algorithm Functions, 505
GEOMEAN function
computes geometric means , 668
GINV function
computing generalized inverses, 668
GOTO statement
jumping to a new statement, 670
GPROBCNT call
probability contour plot, 1134
GSCALE call
calculating round numbers for labeling axes, 671
GSORTH call
computing the Gram-Schmidt orthonormalization, 672
GXYPLOT call
create scatter plot, 1134

HADAMARD function, 674
HALF function
computing Cholesky decomposition, 675
HANKEL function
generating a Hankel matrix, 675
HARMEAN function
computes harmonic means , 677
HDIR function
performing a horizontal direct product, 677
HEATMAPCONT call
create a heat map, 678
HEATMAPDISC call
create a heat map, 682
HERMITE function
reducing a matrix to Hermite normal form, 684
HISTOGRAM Call
create a histogram, 685
HOMOGEN function
solving homogeneous linear systems, 687

I function
creating an identity matrix, 688
ICCEPSTRUM function
computes the inverse complex cepstrum, 688
IF-THEN/ELSE statement
conditionally executing statements, 690
IFFT function
computing the inverse finite Fourier transform, 692
IFFTC function
compute the inverse finite Fourier transform of a complex input, 693
IMLMLIB Module Library
modules reference, 1025, 1134, 1135
overview, 1129, 1131, 1132
ImportDataSetFromR subroutine, 248
ImportMatrixFromR subroutine, 248
ImportTableFromR function, 248
INDEX CREATION operator
creates an index vector, 514
INDEX statement
indexing a variable in a SAS data set, 697
INFILE statement
opening a file for input, 699
INPUT statement
inputting data, 699
INSERT function
inserting one matrix inside another, 701
INT function
truncating a value, 702
INV function
computing a matrix inverse, 703
Inverses
Moore-Penrose inverse, 568, 962, 971–973
INVUPDT function
updating a matrix inverse, 704
IPF call
performing an iterative proportional fit, 706
ISEMPTY function
determine whether a matrix is empty, 718
ISSKIPPED function
determine whether a module argument is skipped, 718
Iterative Algorithm, 719
ITSOLVER call
solving a sparse linear system by using iterative methods, 719

J function
creating a matrix of identical values, 722

Kalman filter subroutines
covariance filtering and prediction, 291
diffuse covariance filtering and prediction, 291
diffuse fixed-interval smoothing, 291
examples, 292
fixed-interval smoothing, 291
one-step forecast for SSM, 730
one-step predictions, 724, 727
overview, 290
smoothed estimate, 727
smoothed state vectors, 733
syntax, 724
Kernel PCA
nonlinear kernel principal components, 736
score a kernel PCA model, 740
KRONECKER product
takes the direct product of two matrices, 510
KURTOSIS function
compute sample kurtosis, 744

LABEL

quadratic form maximization, 797
LAG function
computing lagged values, 745
LAMBERTW function
computes either branch of the Lambert W function, 746
LCP call
solving the linear complementarity problem, 751
Least absolute value regression, 747–750
LEAVE statement
exits a DO loop, 754
LENGTH function
finding the lengths of character matrix elements, 755
Linear Algebra Functions, 499
Linear least squares
full-rank example, 903, 904
QR decomposition, 946
rank-deficient solutions, 962, 965, 966, 968–970
LINK statement
jumping to another statement, 755
LIST CREATION operator
create a list, 516
LIST ITEM
list item, 516
LIST statement
displaying observations of a data set, 756
LIST SUBLIST
list sublist, 517
LISTADDITEM call
adds a new item to end of a list, 757
LISTCREATE function
creates a new list, 757
LISTDELETEITEM call
deletes an item from a list, 758
LISTDELETENAME call
removes the name of an item, 759
LISTGETALLNAMES function
gets names for all named items, 759
LISTGETITEM function
gets the value of an item, 760
LISTGETNAME function
gets the names used in a list, 761
LISTGETSUBITEM function
gets the value of an item in a nested sublist, 762
LISTINDEX function
gets the numeric positions of items, 763
LISTINSERTITEM call
inserts an item at a specified position, 764
LISTLEN function
gets the number of items in a list, 764
Lists, 503
LISTSETITEM call
sets the value of an existing list item, 765
LISTSETNAME call
sets the name of an item, 765
LISTSETSUBITEM call
sets the value of an item in a nested sublist, 766
LMS call
performing robust regression, 767
LOAD statement
loading modules and matrices, 776
LOC function
finding nonzero elements of a matrix, 776
LOG function
taking the natural logarithm, 777
LOGABSDET function
compute the log of the absolute value of the determinant, 778
LOGICAL operator
perform elementwise logical comparisons, 517
LP call
solving the linear programming problem, 778
LPSOLVE call
solving the linear programming problem, 779
LTS call
performs robust regression, 781
LUPDT call, 788

MAD function
univariate median absolute deviation, 789
MAGIC Function
return magic square, 791
MAHALANOBIS function
compute Mahalanobis distance, 792
MARG call
evaluating marginal totals, 793
Matrix decomposition
Cholesky decomposition, 962–964
complete orthogonal decomposition, 532, 566
downdating and updating, 945, 946, 961
QR decomposition, 872–874, 876, 899, 901–904
matrix exponential, 632
Matrix Inquiry Functions, 492
Matrix Reshaping Functions, 493
Matrix Sorting And By-Group Processing Functions, 493
MATTRIB statement
associating printing attributes with matrices, 795
MAX function
finding the maximum value of matrix, 797
MCD call, 800
MEAN function
computing sample means, 804
Median computation, 806
MILPSOLVE call
solving the mixed integer linear programming problem, 806

MIN function
finding the smallest element of a matrix, 810
Minimum Residual Algorithm, 465, 469
MOD function
computing the modulo (remainder), 810
MODULEI call, 811
MODULEIC function
calling an external function, 812
MODULEIN function
calling an external function, 812
Modules, 502
MODULESTACK function
return the calling sequence for a module), 813
MULTIPLICATION operator, elementwise
performs elementwise multiplication, 518
MULTIPLICATION operator, matrix
performs matrix multiplication, 520
Multivariate sampling, 913, 928–931
MVE call, 814

NAME function
listing the names of arguments, 819
NCOL function
finding the number of columns of a matrix, 820
NLENG function
finding the size of an element, 822
Nonlinear optimization subroutines
advanced examples, 349
conjugate gradient optimization, 825
control parameters vector, 346, 347
double-dogleg optimization, 826, 828
feasible point computation, 832
finite difference approximations, 828–831
finite-difference approximations, 331, 332
global vs. local optima, 324
hybrid quasi-Newton optimization, 833, 835
Kuhn-Tucker conditions, 325
least squares methods, 833, 835, 836
Levenberg-Marquardt optimization, 835, 836
Nelder-Mead simplex optimization, 836–838, 840
Newton-Raphson optimization, 840–842
Newton-Raphson ridge optimization, 843, 845
objective function and derivatives, 326–331
options vector, 335–339
parameter constraints, 333–335
printing optimization history, 348, 349
quadratic optimization, 850, 851, 853
quasi-Newton optimization, 845, 846, 848–850
return codes, 326
termination criteria, 339, 341, 343–345
trust-region optimization, 853, 854
NORM function
finding a vector or matrix norm, 854
NROW function
finding the number of rows of a matrix, 855
NUM function
producing a numeric representation of a character matrix, 856
Numerical Analysis Functions, 498
Numerical integration, 904–907, 909
adaptive Romberg method, 905
of differential equations, 856, 857, 859–862
specifying subintervals, 904
two-dimensional integration, 907

ODSGRAPH call, 862
OPSCAL Function, 865
Optimization Subroutines, 500
ORPOL function
generating orthogonal polynomials, 866
Orthogonal factorization, 962–964
Orthogonalization
by ORTVEC call, 872–874, 876

PACKAGE HELP statement
display help files, 877
PACKAGE INFO statement
display information, 878
PACKAGE INSTALL statement
install one or more packages, 878
PACKAGE LIBNAME statement
create a libref, 879
PACKAGE LIST statement
list packages, 880
PACKAGE LOAD statement
create a libref, 881
PACKAGE statement
install and use packages, 876
PACKAGE UNINSTALL statement
uninstall packages, 882
Packages, 504
PALETTE function
return color palette, 883
PARENTNAME function
return name of argument, 882
PAUSE statement
interrupting module execution, 888
percentiles, 898
POLYROOT function
finding zeros of a real polynomial, 888
POWER operator, elementwise
raises each element to a power, 521
POWER operator, matrix
raises a matrix to a power, 521
PRINT statement
printing matrix values, 889
Printing matrices, 1138
PROD function

multiplying all elements, 891
PRODUCT function
multiplying matrices of polynomials, 891
PURGE statement
removing observations marked for deletion, 892
PUSH call, 894
PUT statement
writing data to an external file, 895
PV function
calculating present value, 896

QNTL call
computing sample quantiles, 898
Quadratic form maximization, 798, 799
quantiles, 898
QUARTILE function
quartile computation, 910
quartiles, 910
QUEUE call
queuing SAS statements, 910
QUIT statement
exiting from PROC IML, 911

R language, 243
RANCOMB function
generate random combinations, 912
RANDFUN function
generating random numbers, 914
RANDGEN call
generating random numbers, 915
Random multivariate sampling, 913, 928–931
Random Number Generation, 495
RANDSEED call
generating random numbers, 935
RANGE function
finding the range of values, 936
RANK function
ranking elements of a matrix, 937
RANKTIE function
ranking elements of a matrix, 939
RANPERK function
generate random permutations, 933
RANPERM function
generate random permutations, 934
RATES function
converting interest rates, 941
RATIO function
dividing matrix polynomials, 942
RCEPSTRUM function
computes the real cepstrum, 943
READ statement
reading observations from a data set, 948
Reduction Functions, 492
Regression, 1136
best subsets, 799
least absolute value, 747–750
response surface, 1135
REMOVE function
discarding elements from a matrix, 950
REMOVE statement
removing matrices from storage, 950
RENAME call
renaming SAS data sets, 951
REPEAT function
creating a new matrix of repeated values, 951
REPLACE statement
replacing values, 952
RESET statement
setting processing options, 953
Reshaping matrices, 960
Response surface regression, 1135
RESUME statement
resuming execution, 955
RETURN statement
returning to caller, 955
ROOT function
performing the Cholesky decomposition of a matrix, 956
ROW function, 958
ROWCAT function
concatenating rows without blank compression, 958
ROWCATC function
concatenating rows with blank compression, 959
RUN statement
executing statements in a module, 961

SAMPLE statement
sampling from a finite set, 974
SAVE statement
saving data, 975
Scalar Functions, 492
SCATTER call
create scatter plot, 976
Sequential tests, 979–982, 984–986, 988–990
group sequential methods, 986
minimizing average sample number (ASN), 988–990
randomized clinical trials, 988–990
scaling, 981
shifting, 982
SERIES call
create series plot, 991
Set Functions, 501
SETDIF function
comparing elements of two matrices, 993
SETIN statement
making a data set current for input, 994

SETOUT statement
making a data set current for output, 995
SHAPE function
reshaping and repeating values, 996
SHAPECOL function
reshaping and repeating values, 997
SHOW statement
printing system information, 998
SIGN REVERSE operator
reverses the signs of elements, 523
SKEWNESS function
compute sample skewness, 1000
SOLVE function
solving a system of linear equations, 1000
SOLVELIN call
solving a sparse symmetric linear system by direct decomposition, 1001
SORT call
sorting a matrix, 1002
SORT statement
sorting a SAS data set, 1003
SORTNDX call
creating a sorted index for a matrix, 1004
SOUND call
producing a tone, 1005
SPARSE function
converting dense to sparse storage, 1006
Sparse Matrix Algorithms, 465, 719
preconditioners, 465, 1001
SPECTROGRAM call
visualize a signal in time-frequency space, 1007
Splines, 1013, 1022
integration of splines, 1020
SPOT function
calculating spot rates, 1022
SQRSYM function
converting to a square matrix, 1023
SQRT function
calculating the square root, 1024
SQRVECH function
converting to a square matrix, 1024
SSQ function
calculating the sum of squares, 1025
Standardizing numeric data, 1025
START statement
defining a module, 1026
Statistical Functions, 495
Statistical Graphics, 504
STD function
computing sample standard deviation, 1028
STOP statement
stopping execution of statements, 1028
STORAGE function
listing names of matrices and modules, 1028
STORE statement
storing matrices and modules, 1029
SUBMIT statement, 236
parameter substitution, 237, 250
R statements, 245
submit R statements, 1031
submit SAS statements, 1031
SUBSCRIPTS
select submatrices, 523
subscript reduction operators, 525
SUBSTR function
taking substrings of matrix elements, 1033
Substring replacement, 960
SUBTRACTION operator
subtracts corresponding matrix elements, 526
SUM function
summing all elements, 1034
SUMMARY statement
computing summary statistics, 1034
SVD call
computing the singular value decomposition, 1037
SWEEP function
sweeping a matrix, 1038
SYMSQR function
converting to a symmetric matrix, 1040

T function
transposing a matrix, 1041
TABLEADDVAR call
adds columns from a matrix to a table, 1041
TABLECREATE function
creates a table from a matrix, 1042
TABLECREATEFROMDATASET function
creates a table from a SAS data set, 1044
TABLEGETVARDATA function
creates a matrix from columns of a table, 1044
TABLEGETVARFORMAT function
returns the formats of the specified columns, 1045
TABLEGETVARINDEX function
returns the column indices for specified names, 1045
TABLEGETVARINFORMAT function
returns the informats of the specified columns, 1046
TABLEGETVARLABEL function
returns the labels of the specified columns, 1047
TABLEGETVARNAME function
returns the names of the specified columns, 1047
TABLEGETVARTYPE function
returns the types of the specified columns, 1048
TABLEISEXISTINGVAR function
indicates whether the specified column names exist, 1048

TABLEISVARNUMERIC function
indicates whether the specified columns are numeric, 1049
TABLEPRINT call
displays a table, 1050
TABLERENAMEVAR call
changes the names of columns, 1053
Tables, 503
TABLESETVARFORMAT call
sets the formats of the specified columns, 1053
TABLESETVARINFORMAT call
sets the informats of the specified columns, 1054
TABLESETVARLABEL call
sets the labels of the specified columns, 1054
TABLESORT call
sorts a table, 1055
TABLEWRITETODATASET call
creates a SAS data set from a table, 1056
TABULATE call
counting the number of elements in each category, 1057
Termination Statements, 504
TFHILBERT function
compute the Hilbert transform, 1057
TFPWV function
compute the pseudo-Wigner-Ville distribution, 1058
TFSTFT function
compute the short-time Fourier transform, 1061
TFWINDOW function
compute a window function for a short-time Fourier transform, 1063
Time series analysis and control
deprecated, 310
Time Series Functions, 496
TOEPLITZ function
generating a Toeplitz matrix, 1067
TPSPLINE call
computing thin-plate smoothing splines, 1069
TPSPLNEV call
evaluating thin-plate smoothing splines, 1071
TRACE function
summing diagonal elements, 1073
TRANSPOSE operator
transposes a matrix, 527
Triangular linear systems, 1074
TYPE function
determining matrix types, 1077

UNION function
performing unions of sets, 1077
UNIQUE function
sorting and removing duplicates, 1078
UNIQUEBY function
processing BY groups in a matrix, 1078
USE statement
opening SAS data sets, 1080

VALSET call
perform indirect assignments, 1081
VALUE function
retrieving values, 1081
VAR Function
computing a sample variance, 1082
VARMACOV Call
computing cross-covariance matrices, 1083
VARMALIK Call
computing log-likelihood function, 1085
VARMASIM Call
generating VARMA(p,q) time series, 1086
VEC operator, 998
VECDIAG function
creating vector from diagonal, 1088
VECH function, 1089
Vector time series analysis
cross-covariance matrix, 306
example, 303, 305
generating a multivariate normal, 307
generating a multivariate time series, 307
log-likelihood function, 306
overview, 303
roots of VARMA characteristic function, 307
VNORMAL Call
generating multivariate normal random series, 1089
VTSROOT Call
calculating characteristic roots, 1091

Wavelet Analysis Functions, 505
WAVFT call
computing fast wavelet transform, 1092
WAVGET call
extracting wavelet information, 1095
WAVIFT call
computing inverse fast wavelet transform, 1097
WAVPRINT call
printing wavelet information, 1099
WAVTHRSH call
thresholding wavelet detail coefficients, 1100
WIDETOLONG call
transforms data from wide to long form, 1100

XMULT function
performing extended-precision matrix multiplication, 1102
XSECT function
intersecting sets, 1103

YIELD function

calculating yield-to-maturity of a cash-flow stream, 1103

Syntax Index

ABORT statement, 528
ABS function, 528
ADDITION operator, 506
ALL function, 529
ALLCOMB function, 529
ALLPERM function, 531
ANY function, 532
APPCORT call, 532
APPEND statement, 534
APPLY function, 537
ARMACOV call, 539
ARMALIK call, 540
ARMASIM function, 542

BAR call, 543
Basic time series subroutines
    ARMACOV subroutine, 289
    ARMALIK subroutine, 289
    ARMASIM function, 289
    example, 288
    overview, 288
BIN function, 547
BLANKSTR function, 549
BLOCK function, 549
BOX call, 550
BRANKS function, 553
BSPLINE function, 554
BTRAN function, 556
BYTE function, 557

CALL statement, 558
CCEPSTRUM call, 558
CHANGE call, 560
CHAR function, 561
CHOOSE function, 562
CLOSE statement, 563
CLOSEFILE statement, 564
COL function, 565
COLVEC function, 566
COMPARISON operator, 507
COMPORT call, 566
CONCAT function, 569
CONCATENATION operator, horizontal, 508
CONCATENATION operator, vertical, 509
CONTENTS function, 570
CONTINUE statement, 571
CONVEXIT function, 571
CORR function, 572
CORR2COV function, 574
COUNTMISS function, 574
COUNTN function, 575
COUNTUNIQUE function, 576
COV function, 577
COV2CORR function, 578
COVLAG function, 578
CREATE statement, 579
CSHAPE function, 582
CUPROD function, 584
CUSUM function, 584
CV function, 585
CVEXHULL function, 585

DATASETS function, 586
DELETE call, 586
DELETE statement, 587
DESIGN function, 588
DESIGNF function, 589
DET function, 589
DFCONV function, 590
DFDESIGN call, 591
DFFILT function, 593
DFFREQZ function, 596
DFFREQZZPK function, 597
DFMEDFILT function, 598
DFORDER call, 600
DFSOSFILT function, 605
DFSOSFREQZ function, 607
DFSOSFREQZZPK function, 608
DIAG function, 610
DIF function, 611
DIMENSION function, 612
DIRECT PRODUCT operator, 510
DISTANCE function, 612
DIVISION operator, 511
DO DATA statement, 617
DO function, 615
DO statement, 616
DO statement, iterative, 617
DO UNTIL statement, 618
DO WHILE statement, 619
DURATION function, 620

ECHELON function, 621
EDIT statement, 621
EIGEN call, 623
EIGVAL function, 626
EIGVEC function, 627
ELEMENT function, 628

ELEMENT MAXIMUM operator, 512
ELEMENT MINIMUM operator, 513
END statement, 629
ENDSUBMIT statement, 629
EXECUTE call, 629
EXECUTEFILE call, 630
EXP function, 632
EXPANDGRID Function, 633
EXPMATRIX function, 632
EXPORTDATASETTOR call, 634
EXPORTMATRIXTOR call, 635
EXPORTTABLETOR call, 635

FARMACOV call, 636
FARMAFIT call, 638
FARMALIK call, 639
FARMASIM call, 641
FDIF call, 643
FEVAL function, 643
FFT function, 644
FFTC function, 646
FILE statement, 647
FIND statement, 648
FINISH statement, 649
FORWARD function, 650
Fractional time series subroutines
    syntax, 310
Fractionally integrated time series subroutines
    example, 307
    FARMACOV subroutine, 310
    FARMAFIT subroutine, 310
    FARMALIK subroutine, 310
    FARMASIM subroutine, 310
    FDIF subroutine, 310
    overview, 307
FREE statement, 650
FROOT function, 651
FULL function, 652

GAEND call, 654
GAGETMEM call, 654
GAGETVAL call, 655
GAINIT call, 655
GAREEVAL call, 656
GAREGEN call, 657
GASETCRO call, 657
GASETMUT call, 661
GASETOBJ call, 663
GASETSEL call, 664
GASETUP function, 664
GBXWHSKR call, 1134
GENEIG call, 666
GEOMEAN function, 668
GINV function, 668
GOTO statement, 670
GPROBCNT call, 1134
GSCALE call, 671
GSORTH call, 672
GXYPLOT call, 1134

HADAMARD function, 674
HALF function, 675
HANKEL function, 675
HARMEAN function, 677
HDIR function, 677
HEATMAPCONT call, 678
HEATMAPDISC call, 682
HERMITE function, 684
HISTOGRAM Call, 685
HOMOGEN function, 687

I function, 688
ICCEPSTRUM function, 688
IF-THEN/ELSE statement, 690
IFFT function, 692
IFFTC function, 693
IMLMLIB Module Library
    modules reference, 1025, 1134, 1135
    overview, 1129, 1131
IMPORTDATASETFROMR call, 694
IMPORTMATRIXFROMR call, 696
IMPORTTABLEFROMR function, 697
INDEX CREATION operator, 514
INDEX statement, 697
INFILE statement, 699
INPUT statement, 699
INSERT function, 701
INT function, 702
INV function, 703
INVUPDT function, 704
IPF call, 706
ISEMPTY function, 718
ISSKIPPED function, 718
ITSOLVER call, 719

J function, 722
JROOT function, 723

KALCVF call, 293, 300, 301, 724, 727
KALCVS call, 727
KALDFF call, 301, 730
KALDFS call, 733
Kalman filter subroutines
    examples, 292
    KALCVF subroutine, 291
    KALCVS subroutine, 291
    KALDFF subroutine, 291
    KALDFS subroutine, 291
    overview, 290

syntax, 291, 724
KPCASCORE function, 740
KPCATRAIN call, 736
KRONECKER product, 510
KURTOSIS function, 744

LAG function, 745
LAMBERTW function, 746
LAV call, 747–750
LCP call, 751
LEAVE statement, 754
LENGTH function, 755
LINK statement, 755
LIST CREATION operator, 516
LIST ITEM, 516
LIST statement, 756
LIST SUBLIST, 517
LISTADDITEM call, 757
LISTCREATE function, 757
LISTDELETEITEM call, 758
LISTDELETENAME call, 759
LISTGETALLNAMES function, 759
LISTGETITEM function, 760
LISTGETNAME function, 761
LISTGETSUBITEM function, 762
LISTINDEX function, 763
LISTINSERTITEM call, 764
LISTLEN function, 764
LISTSETITEM call, 765
LISTSETNAME call, 765
LISTSETSUBITEM call, 766
LMS call, 767
LOAD statement, 776
LOC function, 776
LOG function, 777
LOGABSDET function, 778
LOGICAL operator, 517
LP call, 778
LPSOLVE call, 779
LTS call, 781
LUPDT call, 788

MAD function, 789
MAGIC Function, 791
MAHALANOBIS function, 792
MARG call, 793
MATTRIB statement, 795
MAX function, 797
MAXQFORM call, 797–799
MCD call, 800
MEAN function, 804
MEDIAN function, 806
MILPSOLVE call, 806
MIN function, 810
MOD function, 810
MODULEI call, 811
MODULEIC function, 812
MODULEIN function, 812
MODULESTACK function, 813
MULTIPLICATION operator, elementwise, 518
MULTIPLICATION operator, matrix, 520
MVE call, 814

NAME function, 819
NCOL function, 820
NDX2SUB function, 820
NLENG function, 822
Nonlinear optimization subroutines
advanced examples, 349
details, 324
introductory examples, 315
NLPCG Call, 353
NLPCG call, 825
NLPDD Call, 357, 361, 378, 826, 828
NLPDD call, 826
NLPFDD Call, 368, 829, 831
NLPFDD call, 828, 830
NLPFEA call, 832
NLPHQN Call, 833, 835
NLPHQN call, 833
NLPLM Call, 369, 836
NLPLM call, 835
NLPNMS Call, 838, 840
NLPNMS call, 836, 837
NLPNRA Call, 841, 842
NLPNRA call, 840
NLPNRR Call, 355, 845
NLPNRR call, 843
NLPQN Call, 362, 364, 382, 846, 848–850
NLPQN call, 845
NLPQUA Call, 851, 853
NLPQUA call, 850
NLPTR Call, 351, 367, 854
NLPTR call, 853
overview, 313
syntax, 822
NORM function, 854
NROW function, 855
NUM function, 856

ODE call, 856, 857, 859–862
ODSGRAPH call, 862
OK= option
SUBMIT statement, 1031
OPSCAL function, 865
ORPOL function, 866
ORTVEC call, 872–874, 876

PACKAGE HELP statement, 877

PACKAGE INFO statement, 878
PACKAGE INSTALL statement, 878
PACKAGE LIBNAME statement, 879
PACKAGE LIST statement, 880
PACKAGE LOAD statement, 881
PACKAGE statement, 876
PACKAGE UNINSTALL statement, 882
PALETTE function, 883
PARENTNAME function, 882
PAUSE statement, 888
POLYROOT function, 888
POWER operator, elementwise, 521
POWER operator, matrix, 521
PRINT statement, 889
PROC IML Statement, 10, 473
PROD function, 891
PRODUCT function, 891
PURGE statement, 892
PUSH call, 894
PUT statement, 895
PV function, 896

QNTL call, 898
QR call, 899, 901–904
QUAD call, 904–907, 909
QUADREG call, 1135
QUARTILE function, 910
QUEUE call, 910
QUIT statement, 911

R option
    SUBMIT statement, 1031
RANCOMB function, 912
RANDDIRICHLET function, 913
RANDFUN function, 914
RANDGEN call, 915
RANDMULTINOMIAL function, 928
RANDMVT function, 929
RANDNORMAL function, 930
RANDSEED call, 935
RANDWISHART function, 931
RANGE function, 936
RANK function, 937
RANKTIE function, 939
RANPERK function, 933
RANPERM function, 934
RATES function, 941
RATIO function, 942
RCEPSTRUM function, 943
RDODT call, 945, 946
READ statement, 948
REGRESS call, 1136
REMOVE function, 950
REMOVE statement, 950
RENAME call, 951
REPEAT function, 951
REPLACE statement, 952
RESET statement, 953
RESUME statement, 955
RETURN statement, 955
ROOT function, 956
ROW function, 958
ROWCAT function, 958
ROWCATC function, 959
ROWVEC function, 960
RSUBSTR function, 960
RUN statement, 961
RUPDT call, 945, 946, 961
RZLIND call, 962–966, 968–973

SAMPLE statement, 974
SAVE statement, 975
SCATTER call, 976
SEQ call, 979–982, 984–986, 988–990
SEQSCALE call, 979–982, 984–986, 988–990
SEQSHIFT call, 979–982, 984–986, 988–990
SERIES call, 991
SETDIF function, 993
SETIN statement, 994
SETOUT statement, 995
SHAPE function, 996
SHAPECOL function, 997
SHOW statement, 998
SIGN REVERSE operator, 523
SKEWNESS function, 1000
SOLVE function, 1000
SOLVELIN call, 1001
SORT call, 1002
SORT statement, 1003
SORTNDX call, 1004
SOUND call, 1005
SPARSE function, 1006
SPECTROGRAM call, 1007
SPLINE call, 1013
SPLINEC call, 1013
SPLINEV function, 1022
SPOT function, 1022
SQRSYM function, 1023
SQRT function, 1024
SQRVECH function, 1024
SSQ function, 1025
STANDARD function, 1025
START statement, 1026
STD function, 1028
STOP statement, 1028
STORAGE function, 1028
STORE statement, 1029
SUB2NDX call, 1030

SUBMIT statement, 1031
    OK= option, 1031
    R option, 1031
SUBSCRIPTS, 523
SUBSTR function, 1033
SUBTRACTION operator, 526
SUM function, 1034
SUMMARY statement, 1034
SVD call, 1037
SWEEP function, 1038
SYMSIZE= option, 10, 473
    PROC IML statement, 10
SYMSQR function, 1040

T function, 1041
TABLEADDVAR call, 1041
TABLECREATE function, 1042
TABLECREATEFROMDATASET function, 1044
TABLEGETVARDATA function, 1044
TABLEGETVARFORMAT function, 1045
TABLEGETVARINDEX function, 1045
TABLEGETVARINFORMAT function, 1046
TABLEGETVARLABEL function, 1047
TABLEGETVARNAME function, 1047
TABLEGETVARTYPE function, 1048
TABLEISEXISTINGVAR function, 1048
TABLEISVARNUMERIC function, 1049
TABLEPRINT call, 1050
TABLERENAMEVAR call, 1053
TABLESETVARFORMAT call, 1053
TABLESETVARINFORMAT call, 1054
TABLESETVARLABEL call, 1054
TABLESORT call, 1055
TABLEWRITETODATASET call, 1056
TABPRT call, 1138
TABULATE call, 1057
TFHILBERT function, 1057
TFPWV function, 1058
TFSTFT function, 1061
TFWINDOW function, 1063
TIMSAC subroutines
    deprecated, 310
    TSBAYSEA subroutine, 1075
    TSDECOMP subroutine, 1075
    TSMLOCAR subroutine, 1075
    TSMLOMAR subroutine, 1075
    TSMULMAR subroutine, 1076
    TSPEARS subroutine, 1076
    TSPRED subroutine, 1076
    TSROOT subroutine, 1076
    TSTVCAR subroutine, 1076
    TSUNIMAR subroutine, 1077
TOEPLITZ function, 1067
TPSPLINE call, 1069
TPSPLNEV call, 1071
TRACE function, 1073
TRANSPOSE operator, 527
TRISOLV function, 1074
TYPE function, 1077

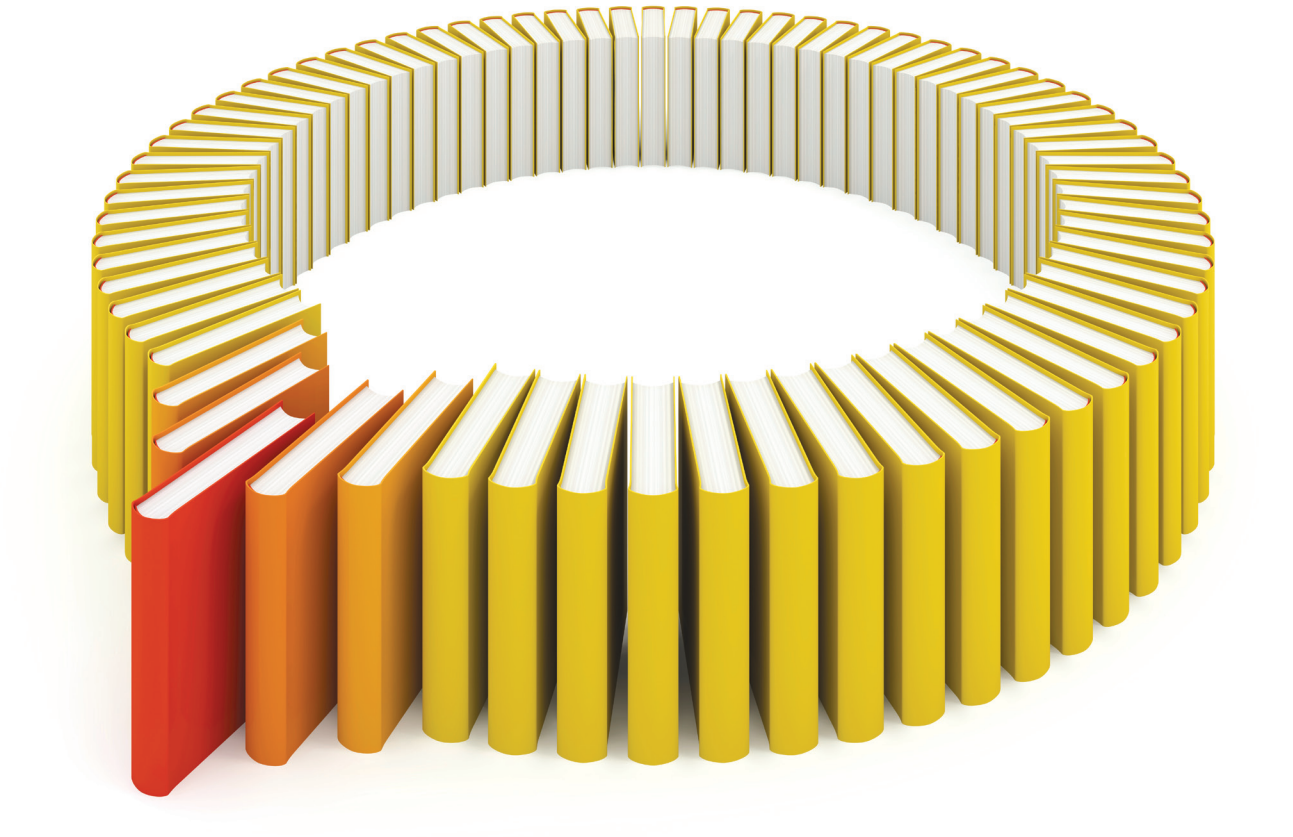
UNION function, 1077
UNIQUE function, 1078
UNIQUEBY function, 1078
USE statement, 1080

VALSET call, 1081
VALUE function, 1081
VAR Function, 1082
VARMACOV Call, 1083
VARMALIK Call, 1085
VARMASIM Call, 1086
VECDIAG function, 1088
VECH function, 1089
Vector time series subroutines
    example, 303, 305
    overview, 303
    syntax, 306
    VARMACOV subroutine, 306
    VARMALIK subroutine, 306
    VARMASIM subroutine, 307
    VNORMAL subroutine, 307
    VTSROOT subroutine, 307
VNORMAL Call, 1089
VTSROOT Call, 1091

WAVFT call, 1092
WAVGET call, 1095
WAVIFT call, 1097
WAVPRINT call, 1099
WAVTHRSH call, 1100
WIDETOLONG call, 1100
WORKSIZE= option, 10, 473
    PROC IML statement, 10

XMULT function, 1102
XSECT function, 1103

YIELD function, 1103



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.

sas
THE POWER TO KNOW®

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613