

Statistical Programming with SAS/IML® Software

Rick Wicklin

```
/* percentage of births by day of the week */  
declare ScatterPlot plot;  
plot = ScatterPlot.Create(dobj, "Date", "Percentage");  
plot.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);  
colors = BLUE // CYAN // YELLOW // RED;  
run ColorCodeObsByGroups(dobj, "DOW", colors);  
plot.SetMarkerSize(7);
```



features
SAS/IML®
Studio
and the
SAS® Interface
to R

The correct bibliographic citation for this manual is as follows: Wicklin, Rick. 2010. *Statistical Programming with SAS/IML® Software*. Cary, NC: SAS Institute Inc.

Statistical Programming with SAS/IML® Software

Copyright © 2010, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-60764-663-1

ISBN 978-1-60764-770-6 (electronic book)

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.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 2010

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at support.sas.com/publishing or call 1-800-727-3228.

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 registered trademarks or trademarks of their respective companies.

Contents

I Programming in the SAS/IML Language	1
Chapter 1. An Introduction to SAS/IML Software	3
Chapter 2. Getting Started with the SAS/IML Matrix Programming Language	17
Chapter 3. Programming Techniques for Data Analysis	55
Chapter 4. Calling SAS Procedures	89
II Programming in SAS/IML Studio	107
Chapter 5. IMLPlus: Programming in SAS/IML Studio	109
Chapter 6. Understanding IMLPlus Classes	129
Chapter 7. Creating Statistical Graphs	143
Chapter 8. Managing Data in IMLPlus	173
Chapter 9. Drawing on Graphs	187
Chapter 10. Marker Shapes, Colors, and Other Attributes of Data	225
III Applications	249
Chapter 11. Calling Functions in the R Language	251
Chapter 12. Regression Diagnostics	279
Chapter 13. Sampling and Simulation	311
Chapter 14. Bootstrap Methods	349
Chapter 15. Timing Computations and the Performance of Algorithms	371
Chapter 16. Interactive Techniques	385
IV Appendixes	413
Appendix A. Description of Data Sets	415
Appendix B. SAS/IML Operators, Functions, and Statements	419
Appendix C. IMLPlus Classes, Methods, and Statements	427
Appendix D. Modules for Compatibility with SAS/IML 9.22	435
Appendix E. ODS Statements	437
Index	441

Acknowledgments

I would like to thank Robert Rodriguez for suggesting that I write this book and for our discussions regarding content and order of presentation. He and Maura Stokes provided many opportunities for me to hone my writing and programming skills by inviting me to present papers and workshops at conferences.

I thank my colleagues at SAS from whom I have learned many statistical and programming techniques. Among these colleagues, Robert Cohen, Simon Smith, and Randy Tobias stand out as friends who are always willing to share their extensive knowledge.

Thanks to several colleagues who read and commented on early drafts of this book, including Rob Agnelli, Betsy Enstrom, Pushpal Mukhopadhyay, Robert Rodriguez, Randy Tobias, and Donna Watts. I also thank David Pasta, who reviewed the entire book and provided insightful comments, and my editor George McDaniel.

Finally, I would like to thank my wife Nancy Wicklin for her constant support.

Part I

Programming in the SAS/IML Language

Chapter 1

An Introduction to SAS/IML Software

Contents

1.1	Overview of the SAS/IML Language	3
1.2	Comparing the SAS/IML Language and the DATA Step	5
1.3	Overview of SAS/IML Software	6
1.3.1	Overview of the IML Procedure	6
1.3.2	Running a PROC IML Program	7
1.3.3	Overview of SAS/IML Studio	8
1.3.4	Installing and Invoking SAS/IML Studio	10
1.3.5	Running a Program in SAS/IML Studio	11
1.3.6	Using SAS/IML Studio for Exploratory Data Analysis	12
1.4	Who Should Read This Book?	12
1.5	Overview of This Book	13
1.6	Possible Roadmaps through This Book	14
1.7	How to Read the Programs in This Book	14
1.8	Data and Programs Used in This Book	15
1.8.1	Installing the Example Data on a Local SAS Server	16
1.8.2	Installing the Example Data on a Remote SAS Server	16

1.1 Overview of the SAS/IML Language

The acronym IML stands for “interactive matrix language.” The SAS/IML language enables you to read data into vectors and matrices and to manipulate these quantities by using high-level matrix-vector computations. The language enables you to formulate and solve mathematical and statistical problems by using functions and expressions that are similar to those found in textbooks and in research journals. You can write programs that analyze and visualize data or that implement custom algorithms that are not built into any SAS procedure.

The SAS/IML language contains over 300 built-in functions and subroutines. There are also hundreds of functions in Base SAS software that you can call. These functions provide the building blocks for writing statistical analyses. You can write SAS/IML programs by using either of two SAS products: the IML procedure (also called PROC IML) and the SAS/IML Studio application. These two products are discussed in [Section 1.3](#).

4 Chapter 1: An Introduction to SAS/IML Software

As implied by the IML acronym, matrices are a fundamental part of the SAS/IML language. A matrix is a rectangular array of numbers or character strings. In the IML procedure, all variables are matrices. Matrices are used to store many kinds of information. For example, each row in a data matrix represents an observation, and each column represents a variable. In a variance-covariance matrix, the ij th entry represents the sample covariance between the i th and j th variable in a set of data.

As an example of the power and convenience of the SAS/IML language, the following PROC IML statements read certain numeric variables from a data set into a matrix, \mathbf{x} . The program then computes robust estimates of location and scale for each variable. (The location parameter identifies the value of the data's center; the scale parameter tells you about the spread of the data.) Each computation requires only a single statement. The SAS/IML statements are described in Chapter 2, “[Getting Started with the SAS/IML Matrix Programming Language](#).” The data are from a sample data set in the Sashelp library that contains age, height, and weight information for a class of students.

```
/* standardize data by using robust estimates of center and scale */
proc iml;
use Sashelp.Class;                /* open data set for reading */
read all var _NUM_ into x[colname=VarNames]; /* read variables */
close Sashelp.Class;              /* close data set */

/* estimate centers and scales of each variable */
c = median(x);                    /* centers = medians of each column */
s = mad(x);                       /* scales = MAD of each column */
stdX = (x - c) / s;               /* standardize the data */

print c[colname=varNames];        /* print statistics for each column */
print s[colname=varNames];
```

Figure 1.1 Robust Estimates of Location and Scale

c		
Age	Height	Weight
13	62.8	99.5
s		
Age	Height	Weight
1	3.7	14.5

In the PROC IML program, the location of the center of each variable is estimated by calling the MEDIAN function. The scale of each variable is estimated by calling the MAD function, which computes the median absolute deviation (MAD) from the median. (The median is a robust alternative to the mean; the MAD is a robust alternative to the standard deviation.) The data are then standardized (that is, centered and scaled) by subtracting each center from the variable and dividing the result by the scale for that variable. See [Section 2.12](#) for details on how the SAS/IML language interprets quantities such as $(\mathbf{x}-\mathbf{c})/\mathbf{s}$.

The previous program highlights a few features of the SAS/IML language:

- You can read data from a SAS data set into a matrix.
- You can pass matrices to functions.
- Many functions act on the columns of a matrix by default.
- You can perform mathematical operations on matrices and vectors by using a natural syntax.
- You can analyze data and compute statistics without writing loops. Notice in the program that there is no explicit loop over observations, nor is there a loop over the variables.

In general, the SAS/IML language enables you to create compact programs by using a syntax that is natural and convenient for statistical computations. The language is described in detail in Chapter 2, “Getting Started with the SAS/IML Matrix Programming Language.”

1.2 Comparing the SAS/IML Language and the DATA Step

The statistical power of SAS procedures and the data manipulation capabilities of the DATA step are sufficient to serve the analytical needs of many data analysts. However, sometimes you need to implement a proprietary algorithm or an algorithm that has recently been published in a professional journal. Other times, you need to use matrix computations to combine and extend results from procedures. In these situations, you can write a program in the SAS/IML language.

The syntax of the SAS/IML language has much in common with the DATA step: neither language is case-sensitive, variable names can contain up to 32 characters, and statements must end with a semicolon. Furthermore, the syntax for control statements such as the IF-THEN/ELSE statement and the iterative DO statement is the same for both languages. The two languages use the same symbols to test a quantity for equality (=), inequality (^=), and to compare quantities (for example, <=). The SAS/IML language enables you to call the same mathematical functions provided in the DATA step, such as LOG, SQRT, ABS, SIN, COS, CEIL, and FLOOR.

Programming Tip: You can call DATA step functions from your SAS/IML programs.

Conceptually, there are two main differences between the DATA step and a SAS/IML program. First, the DATA step implicitly loops over all observations, whereas a typical SAS/IML program does not. Second, the fundamental unit in the DATA step is an observation, whereas the fundamental unit in the SAS/IML language is a matrix.

In general, SAS/IML software is intended for statistical computing, whereas the DATA step is best for merging, extracting, and transforming data. This is a gross oversimplification, and the author has seen some impressive statistical computations accomplished with the DATA step! However, in many cases, those computations would have been shorter to read and simpler to execute if they had been written in SAS/IML software. The reason is simple: SAS/IML functions can use all of the data to compute statistics, whereas the DATA step functions process one observation at a time. For example, computing the sample standard deviation of a variable in the DATA step is more difficult than computing the same quantity in PROC IML.

1.3 Overview of SAS/IML Software

To run SAS/IML programs, you need to use SAS/IML software. The software consists of two components: the IML procedure and the SAS/IML Studio application.

The IML procedure has been a part of the SAS System since Version 6 in the early 1980s. Traditionally, SAS programmers have used PROC IML to implement computational algorithms that are not built into any SAS procedure.

SAS/IML Studio is newer. It is a programming environment for developing, running, and debugging SAS/IML programs. It includes the ability to call SAS procedures, DATA steps, and macro functions from within SAS/IML programs. It also includes dynamically linked statistical graphics. SAS/IML Studio is useful both for developing programs and also for running programs that exploit interactive features of the software.

SAS/IML Studio has undergone several name changes. The product was initially released as a Web download in 2001 under the name “SAS/IML Workshop.” In 2007, the product was renamed “SAS Stat Studio” and was distributed as part of the SAS/IML product in SAS 9.2. The product was renamed SAS/IML Studio in July 2009 to better emphasize its relationship to the SAS/IML programming language.

SAS/IML software continues to evolve. This book describes features of the IML procedure that are available in SAS 9.2 and mentions new features that are available in the 9.22 release of SAS/IML software. This book describes features of SAS/IML Studio 3.3.

1.3.1 Overview of the IML Procedure

The IML procedure implements the SAS/IML language. PROC IML is an *interactive procedure* in the sense that each statement is executed as it is submitted. You can submit several statements, examine the results, submit more statements, and so on. When you define a matrix, it persists until you quit the procedure, free the memory, or redefine it. This is different from many other SAS procedures, which do not execute any statements until a RUN statement is submitted. The RUN statement in the SAS/IML language is used to execute a module or subroutine, so do not attempt to use the RUN statement as the last statement in a PROC IML call! To exit from PROC IML, use the QUIT statement.

Programming Tip: Do not use a RUN statement at the end of a PROC IML program. In interactive mode, each statement is executed as it is submitted.

You need a license for the SAS/IML product in order to run PROC IML. PROC IML executes on a SAS server.

1.3.2 Running a PROC IML Program

There are several ways to run a SAS/IML program. The traditional way to run PROC IML is to use the SAS Display Manager. You type a program into the Enhanced Editor and choose **Run►Submit** from the main menu. You can also press F3 to run a program.

A second traditional way is to use SAS Enterprise Guide. You type a program into a Program window and click **Run** on the **Program** menu.

In both of these environments, you need to begin your program with the PROC IML statement, as shown in the following program:

```
/* convert temperatures from Celsius to Fahrenheit scale */
proc iml;
Celsius = {-40, 0, 20, 37, 100}; /* some interesting temperatures */
Fahrenheit = 9/5 * Celsius + 32; /* convert to Fahrenheit scale */
print Celsius Fahrenheit;
```

Figure 1.2 Some Temperatures in the Celsius and Fahrenheit Scales

Celsius	Fahrenheit
-40	-40
0	32
20	68
37	98.6
100	212

The PRINT statement displays the value of variables to the current output destination (such as the SAS LISTING destination). You can submit additional statements that use the results of previous assignment statements, as shown in the following statements:

```
Kelvin = Celsius + 273.15; /* convert to Kelvin scale */
print Kelvin;
```

Figure 1.3 Some Temperatures in the Kelvin Scale

Kelvin
233.15
273.15
293.15
310.15
373.15

The variable **Celsius**, defined in the first set of statements, remains available until you quit the procedure or free the variable. When you use the IML procedure in this way, you are essentially using it as a massively powerful calculator.

1.3.3 Overview of SAS/IML Studio

SAS/IML Studio is a programming environment for developing, running, and debugging SAS/IML programs. It is available for no additional fee when you license SAS/IML and SAS/STAT software.

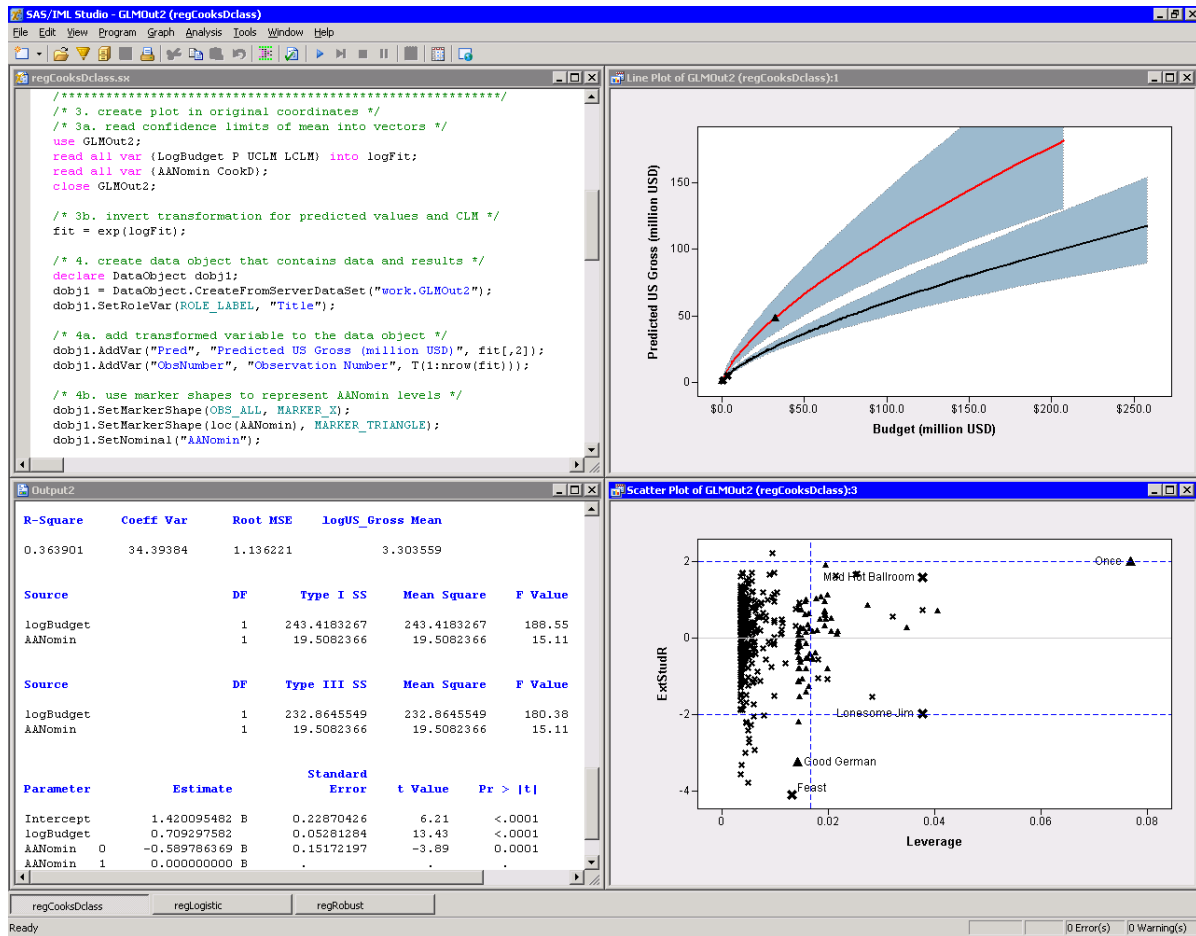
The programming language of SAS/IML Studio is called *IMLPlus*. IMLPlus is an extension of the SAS/IML language that contains additional programming features. IMLPlus combines the flexibility of programming in the SAS/IML language with the power to call SAS procedures and to create and modify dynamically linked statistical graphics. Consequently, IMLPlus contains all of the capabilities of PROC IML, but it can do much more. The IMLPlus language is described in Chapter 5, “[IMLPlus: Programming in SAS/IML Studio](#).”

Programming Tip: An IMLPlus program must be run from SAS/IML Studio. You cannot create IMLPlus graphics or use other IMLPlus features unless you use SAS/IML Studio.

SAS/IML Studio is designed to serve the needs of SAS programmers who need a rich programming environment in which to develop algorithms, to explore data, to investigate relationships between variables, to formulate and compare statistical models, and to detect and understand outliers in the data.

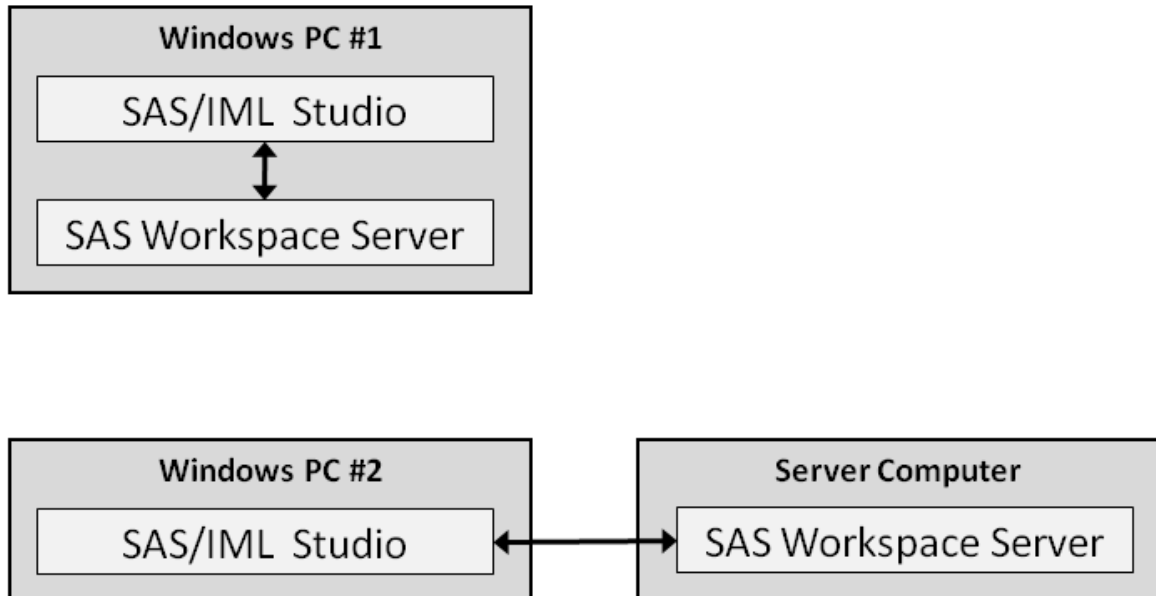
[Figure 1.4](#) shows a SAS/IML Studio workspace with a program window, an output window, and two dynamically linked graph windows. The SAS/IML environment contains many features that assist the programmer in developing programs. For example, the environment is ideal for the programmer who loves to multitask. The environment is multithreaded, which means that you can develop and run several programs simultaneously. Each program environment (called a *workspace*) has its own program window, output window, error log, and windows that display graphs and data tables. While you are working on one program, the windows that belong to other programs are hidden so that you do not get confused about which windows are associated with which analysis. Each workspace also has its own Work library for storing temporary data sets.

Figure 1.4 The SAS/IML Studio Application



The *workspace bar*, shown at the bottom of Figure 1.4, contains a button for each workspace. You can click the workspace bar to navigate between workspaces.

SAS/IML Studio is a client application that runs on a Windows PC. It can connect to one or more SAS Workspace Servers. Figure 1.5 shows two possible architectures. The top image represents the scenario in which SAS Foundation and SAS/IML Studio are both installed on the same Windows PC. In this case, SAS/IML Studio connects to the local version of the SAS Workspace Server when it runs statements in the SAS/IML language or when it calls SAS procedures or DATA steps.

Figure 1.5 Client-Server Architecture for SAS/IML Studio

The bottom image indicates the scenario in which SAS Foundation is installed on a remote computer. In this case, SAS/IML Studio connects to the remote SAS Workspace Server. In fact, *each* workspace in SAS/IML Studio can connect to a *different* server. The remote computer can be running any operating systems that SAS software supports. You need to use the SAS Metadata Server Connection Wizard (found under the **Tools** menu on the SAS/IML Studio main menu) to choose the remote SAS servers to which SAS/IML Studio can connect.

The programs in this book run whether or not the SAS Foundation is installed on the same PC as the SAS/IML Studio application. However, many SAS administrators configure remote SAS servers so that the Sasuser library is read-only, whereas local SAS servers enable users to read and write to the Sasuser library. This affects the installation of the example data sets that are distributed with this book. See [Section 1.8](#).

1.3.4 Installing and Invoking SAS/IML Studio

SAS/IML Studio requires SAS 9.2 and is distributed as part of the SAS/IML product. If SAS/IML Studio was not installed on your PC when the SAS System was installed, you can download and install SAS/IML Studio from support.sas.com/apps/demosdownloads/setupintro.jsp.

After the software is installed, you can invoke SAS/IML Studio by selecting **Start►Programs►SAS►IML Studio 3.3**.

You can create a new program window by clicking **Create a New Program** from the Welcome dialog box, or by selecting **File►New►Workspace** (or CTRL+N). When you create a new program window, you also create a new workspace.

1.3.5 Running a Program in SAS/IML Studio

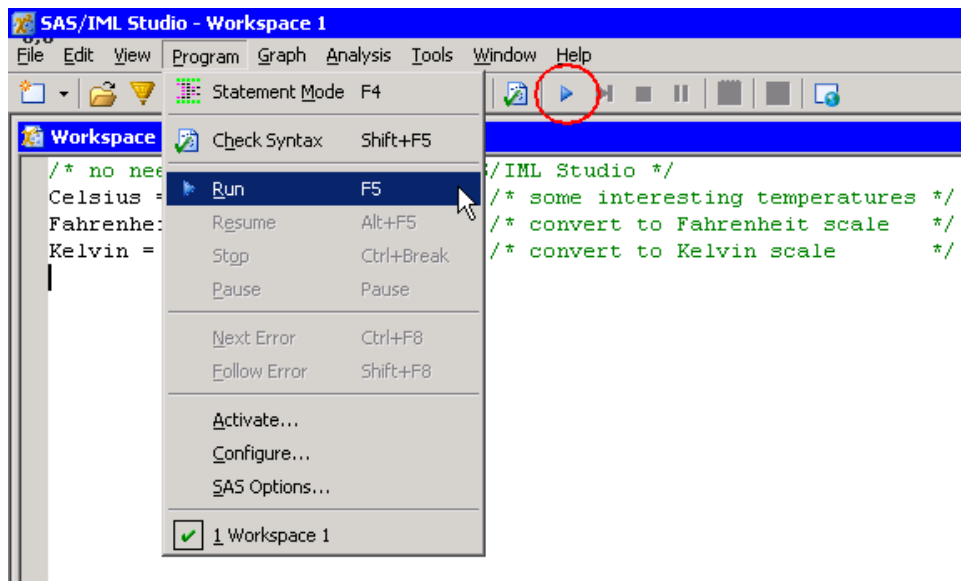
The program in Section 1.3.2 also runs in SAS/IML Studio. The SAS/IML Studio application *assumes* that the program is an IMLPlus program, and therefore you do not need to use the PROC IML statement. In other words, you can run the following program in a SAS/IML Studio program window:

```
/* convert temps: no need for PROC IML statement in SAS/IML Studio */
Celsius = {-40, 0, 20, 37, 100}; /* some interesting temperatures */
Fahrenheit = 9/5 * Celsius + 32; /* convert to Fahrenheit scale */
Kelvin = Celsius + 273.15; /* convert to Kelvin scale */
```

Programming Tip: Programs run from SAS/IML Studio do not require the PROC IML statement.

In SAS/IML Studio, you run a program by choosing **Program►Run** from the main menu, as shown in Figure 1.6. The keyboard shortcut is to press F5. Equivalently, you can click the Run icon (▶) beneath the main menu. The icon is circled in Figure 1.6. To run a portion of a program, you can highlight several statements in the program editor and press F5 or click the Run icon; only the highlighted statements will be executed.

Figure 1.6 Running a Program from the SAS/IML Studio Interface



This book uses the SAS/IML Studio syntax and omits the PROC IML statement. The first two chapters of this book describe basic syntax and features of the SAS/IML language. These statements are valid in both PROC IML and IMLPlus. Later chapters introduce graphs and other features that are not supported by PROC IML.

Programming Tip: Run the programs in this book in SAS/IML Studio. Type the statements into a program window and select **Program►Run** or press F5 to run the program.

1.3.6 Using SAS/IML Studio for Exploratory Data Analysis

The SAS/IML Studio application provides menus and dialog boxes for many point-and-click analyses. By using the graphical user interface (GUI) you can graphically explore data by using dynamically linked statistical graphics. You can use a mouse pointer to select observations in a graph or data table. You can choose menus or click in dialog boxes to change properties of graphs, such as the placement of axis ticks or the title of a graph. You can change properties of a marker such as the shape and color.

You can also use the menu system to do the following:

- compute descriptive statistics and model the distribution of univariate data
- compute smoothers for bivariate scatter plots
- fit various regression models, including linear regression, robust regression, logistic regression, and generalized linear regression
- analyze multivariate data with principal component analysis, exploratory factor analysis, discriminant analysis, and correspondence analysis

All of these graphical and built-in analyses are available to you when you use SAS/IML Studio, and all of these GUI techniques are described in the *SAS/IML Studio User's Guide*.

Some of the exploratory techniques that are described in the SAS/IML Studio documentation are used in this book. They augment and enhance the programming techniques of this book. Often you will use a program to run an analysis and to create graphs, but then you will use interactive techniques to examine the results of the analysis. The regression diagnostics presented in Chapter 12, “[Regression Diagnostics](#),” are canonical examples of using interactive techniques to examine the fit of a statistical model.

If you are not familiar with the dynamically linked graphics in SAS/IML Studio, consider reading Chapters 8–11 of the *SAS/IML Studio User's Guide*.

1.4 Who Should Read This Book?

The goal of this book is to introduce SAS/IML to a wide range of statistical programmers. The examples in this book show how SAS/IML and IMLPlus programs enable you to analyze your data in new and innovative ways. In short, this book intends to show you how writing programs in SAS/IML software enables you to implement analytic techniques that would be difficult or impossible with other SAS software.

The audience for this book is analysts and statistical programmers who use SAS/STAT procedures and the DATA step to explore and model data. It is also intended for programmers who want an introduction to the SAS/IML and IMLPlus languages. The book does not presume prior knowledge of the SAS/IML language, but does presume some familiarity with DATA step concepts such as

missing values, formats, and the length of a character variable. The book also presumes familiarity with basic statistical ideas such as you might encounter by using the UNIVARIATE, CORR, and GLM procedures. For example, this book discusses quantiles, distributions, density estimation, and regression. Only a few sections of the book presume familiarity with basic concepts of linear algebra such as matrix multiplication and solving a system of linear equations.

1.5 Overview of This Book

Depending upon your programming background, your knowledge of statistical programming, and your experience with SAS, you might be able to skip certain chapters of this book. The following list summarizes each chapter of the book:

Chapter 2 A basic introduction to the SAS/IML language. The chapter describes how to define matrices, compare quantities, and call functions and subroutines. It describes basic programming statements such as IF-THEN/ELSE and the iterative DO statement. If you are an experienced SAS/IML programmer, you can skip this chapter.

Chapter 3 An introduction to using the SAS/IML language in data analysis. Even experienced SAS/IML programmers should familiarize themselves with the technique presented in the section “[Analyzing Observations by Categories](#)” on page 68.

Chapter 4 How to call SAS procedures from a SAS/IML program.

Chapter 5 An overview of features in SAS/IML Studio that are not found in PROC IML.

Chapter 6–Chapter 7 These chapters introduce IMLPlus classes and describe how to create graphs.

Chapter 8 This chapter describes how to manage data in IMLPlus.

Chapter 9–Chapter 10 These chapters describe intermediate-level programming techniques for modifying graphs, including drawing on graphs and changing the color and shape of observation markers.

Chapter 11 How to call R functions and packages from a SAS/IML program.

Chapter 12–Chapter 14 Applications of SAS/IML programming to selected modern statistical topics such as regression diagnostics, simulation and sampling, and bootstrap methods.

Chapter 15 How to measure the time required for an algorithm to run on typical data, and also how to investigate how that time changes with the size or characteristics of the data.

Chapter 16 How to write programs that use interactive features of IMLPlus, such as creating dialog boxes and attaching menus to graphs.

The sections titled “Case Study” describe programs that implement some statistical analysis and are longer or more complicated than the simpler examples found elsewhere in the book.

1.6 Possible Roadmaps through This Book

Depending upon your experience and interests, there are several paths through this book. Some readers might identify themselves with one or more of the following personas:

SAS/STAT Programmer: This reader is familiar with the DATA step and with calling SAS/STAT procedures, but has no experience with writing SAS/IML programs. This reader is interested in using SAS/IML software in conjunction with SAS/STAT procedures to fit statistical models.

Novice SAS/IML Programmer: This reader has little or no experience writing SAS/IML programs, but wants to learn the basics of PROC IML and SAS/IML Studio.

Intermediate SAS/IML Programmer: This reader is familiar with the basics of the SAS/IML language and wants to improve his programming skills and learn about the features in SAS/IML Studio.

Advanced SAS/IML Programmer: This reader is proficient in SAS/IML programming and is interested in developing new algorithms and writing modules that implement new computational methods in SAS/IML software.

For these readers, the following table suggests possible paths through this book. A checkmark (✓) indicates that the chapter should be read. Other chapters can be skimmed, since they presumably contain content that is already familiar. Chapters that can be skimmed are indicated in the table by a circled ‘S’ (Ⓢ). Even if you skim a chapter, it is recommended that you read the chapter’s programming tips and techniques.

Table 1.1 Possible Paths through This Book

Persona	Language Chapters			SAS/IML Studio Chapters						Applications Chapters					
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SAS/STAT	✓	✓	✓	Ⓢ	✓	✓	✓	Ⓢ		Ⓢ	✓	✓	✓		
Novice	✓	✓	✓	✓	✓	✓								✓	
Intermediate	Ⓢ	✓	✓	✓	✓	✓	✓					✓		✓	✓
Advanced	Ⓢ	Ⓢ	✓	Ⓢ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

1.7 How to Read the Programs in This Book

In books about statistical programming, there are several approaches for presenting and describing a program. The approach used in this book is to briefly describe what a program is supposed to do,

present the program in its entirety, and then discuss the details of the implementation. Comments in the program call attention to the main steps and to important statements that are explained later in the text. The advantage of this approach is that the reader can see the program in its entirety and read the description of the program after the statements are presented.

For example, the following simple SAS/IML program and subsequent explanation demonstrate how programs in this book are documented and described:

```
/* Present a simple example program with comments.          */
/* The PROC IML statement is not required in SAS/IML Studio. */
x = 3;               /* 1 */               /* NUMBERS indicate steps that */
y = 2;               /* are described in a list    */
z = x + y;           /* 2 */               /* AFTER the program.        */

print z;             /* display result */ /* Other statements are briefly */
                    /* described WITHIN the program. */
```

The program begins with a short comment that explains the purpose of the program. A number that appears in a comment statement means that the program statement is explained in an enumerated list that appears after the program list. For example, the previous program consists of the following main steps:

1. Assign the value 3 to the **x** variable. Notice that the statement ends with a semicolon, as do all SAS statements. The variable **y** is assigned similarly.
2. Define the variable **z** as the sum of the **x** and **y** variables. The result of this assignment is shown in [Figure 1.7](#).

Figure 1.7 The Output from a Simple Program

z 5

For simple or very short programs (such as the one above), the program is often described in paragraph form, rather than with an enumerated list.

1.8 Data and Programs Used in This Book

The data and programs used in this book are available from this book's companion Web site: <http://support.sas.com/publishing/authors/wicklin.html>. Click **Example Code and Data** to obtain the data and programs.

The example data sets used in this book are described in Appendix A, “[Description of Data Sets](#).”

1.8.1 Installing the Example Data on a Local SAS Server

When SAS Foundation and SAS/IML Studio are both installed on the same Windows PC, you can install the example data sets in the Sasuser library. The book's Web site includes instructions on how to download and install the data so that it is accessible from SAS/IML software.

All of the examples in this book are written for this configuration. After you install the data, you can run the examples without modification.

1.8.2 Installing the Example Data on a Remote SAS Server

Many SAS System administrators configure remote SAS servers so that the Sasuser library is read-only. In this case, you cannot install the example data sets in Sasuser.

If your copy of SAS/IML Studio is configured to connect to a SAS Workspace Server that runs on a remote computer, do the following:

1. Ask your site administrator to use the SAS Management Console to set up a library named SPI in which you can store the book's data sets.
2. When you download the data and programs from the book's Web site, install the data sets in the SPI library.
3. When a program in the book refers to the Sasuser library, replace the library name with SPI.

For example, suppose that the book contains the following example that uses the Sasuser library:

```
/* read data installed on local SAS server */
proc iml;
use Sasuser.Vehicles;           /* open data set for reading */
read all var _NUM_ into x;      /* read numerical data      */
close Sasuser.Vehicles;        /* close the data set      */
```

If you want to run the preceding example, you need to use the SPI library instead of the Sasuser library. This means that you will actually run the following program:

```
/* read data installed on remote SAS server */
proc iml;
use SPI.Vehicles;               /* open data set for reading */
read all var _NUM_ into x;      /* read numerical data      */
close SPI.Vehicles;            /* close the data set      */
```

Chapter 2

Getting Started with the SAS/IML Matrix Programming Language

Contents

2.1	Overview of the SAS/IML Language	18
2.2	Creating Matrices	18
2.2.1	Printing a Matrix	19
2.2.2	The Dimensions of a Matrix	20
2.2.3	The Type of a Matrix	21
2.2.4	The Length of a Character Matrix	22
2.3	Using Functions to Create Matrices	24
2.3.1	Constant Matrices	24
2.3.2	Vectors of Sequential Values	25
2.3.3	Pseudorandom Matrices	27
2.4	Transposing a Matrix	28
2.5	Changing the Shape of Matrices	29
2.6	Extracting Data from Matrices	30
2.6.1	Extracting Rows and Columns	31
2.6.2	Matrix Diagonals	33
2.6.3	Printing a Submatrix or Expression	35
2.7	Comparison Operators	36
2.8	Control Statements	38
2.8.1	The IF-THEN/ELSE Statement	38
2.8.2	The Iterative DO Statement	39
2.9	Concatenation Operators	41
2.10	Logical Operators	43
2.11	Operations on Sets	46
2.12	Matrix Operators	47
2.12.1	Elementwise Operators	47
2.12.2	Matrix Computations	49
2.13	Managing the SAS/IML Workspace	51

2.1 Overview of the SAS/IML Language

SAS/IML is a programming language for high-level, matrix-vector computations. Matrices are rectangular arrays that usually contain numbers. A matrix that contains character data is often explicitly called a *character matrix*. In statistical programming, matrices often hold data for analysis. Each row of the matrix is an observation; each column of the matrix is a variable.

If your data are in a matrix, you can carry out many statistical operations by using matrix operations. The SAS/IML language has functions and matrix operations that enable you to manipulate matrices as a unit, regardless of the number of rows or columns in the matrix. For an example, see the section “Case Study: Standardizing the Columns of a Matrix” on page 83.

Operations on numerical matrices are also used to describe a wide variety of statistical techniques, including ordinary least squares (OLS) regression and principal component analysis.

This chapter is an introduction to the SAS/IML syntax. It includes basic information about how to define matrices, compare quantities, and call functions and subroutines. It includes a description of basic programming statements such as IF-THEN/ELSE and the iterative DO statement.

2.2 Creating Matrices

A matrix is an $n \times p$ array of numbers or character strings. The integers n and p are the dimensions of the matrix. The row dimension is n ; the column dimension is p . A vector is a special case of a matrix. An $n \times 1$ matrix is called a *column vector*, whereas a $1 \times n$ matrix is called a *row vector*. A 1×1 matrix is called a *scalar*. In general, this book refers to any SAS/IML variable as a matrix, regardless of its dimensions.

In a SAS/IML program, all variables are matrices, so you do not need to specify the type of a variable. Furthermore, matrices are dynamically reassigned as needed, so you do not need to specify the size or the type (numeric or character) of a matrix. For example, the following statements are all valid:

```
/* create matrices of various types and sizes */
x = 1;                               /* scalar */
x = {1 2 3};                         /* reassign to row vector */
y = {1 2 3, 4 5 6};                 /* 2 x 3 numeric matrix */
y = {"male" "female"};             /* reassign to 1 x 2 character matrix */
```

See the section “Running a PROC IML Program” on page 7 for instructions on how to run SAS/IML programs. The first statement creates **x** as a numerical scalar matrix. The second statement redefines **x** as a numerical row vector; spaces separate entries in different columns. The third statement defines **y** as a 2×3 numerical matrix; commas indicate a new row. The last statement redefines **y** as a 1×2 character matrix.

Programming Tip: When defining a matrix, use a comma to indicate a new row.

2.2.1 Printing a Matrix

You can use the PRINT statement to display the value of one or more matrices. The following statement displays the values of the matrices defined in the previous section:

```
print x, y;
```

Figure 2.1 Numeric and Character Matrices

x		
1	2	3
1	2	3
2	3	4
3	4	5
y		
male	female	
1	2	
2	3	
3	4	

Notice that a comma in the PRINT statement indicates that the second matrix should be displayed on a new row. If you omit the comma, the matrices are displayed side by side.

The PRINT statement has four useful options that affect the way a matrix is displayed:

COLNAME=*matrix*

specifies a character matrix to be used for column headings.

FORMAT=*format*

specifies a valid SAS or user-defined format to use when printing matrix values.

LABEL=*label*

specifies a label for the matrix. If this option is not specified, the name of the matrix is used as a label.

ROWNAME=*matrix*

specifies a character matrix to be used for row headings.

These options are specified by enclosing them in square brackets after the name of the matrix that you want to display, as shown in the following example:

```
/* print marital status of 24 people */
ageGroup = {"<= 45", "> 45"};          /* headings for rows */
status = {"Single" "Married" "Divorced"}; /* headings for columns */
counts = { 5      5      0,          /* data to print */
           2      9      3      };
print counts[colname=status
             rowname=ageGroup
             label="Marital Status by Age Group"];
```

```
pct = counts / 24;                                /* compute proportions */
print pct[format=PERCENT7.1];                     /* print as percentages */
```

Figure 2.2 Matrices Displayed with PRINT Options

Marital Status by Age Group			
	Single	Married	Divorced
<= 45	5	5	0
> 45	2	9	3
pct			
	20.8%	20.8%	0.0%
	8.3%	37.5%	12.5%

You can also use these options by specifying their first letters: **C=**, **F=**, **L=**, and **R=**.

2.2.2 The Dimensions of a Matrix

You can determine the dimensions of a matrix by using the `NROW` and `NCOL` functions, as shown in the following statements:

```
/* dimensions of a matrix */
n_x = nrow(x);
p_x = ncol(x);
print n_x p_x;
```

Figure 2.3 Dimensions of a Matrix

n_x	p_x
1	3
n_u	p_u
0	0

A matrix that contains no elements is called an *empty* matrix. There are several reasons why a matrix can be empty:

- It has not been defined.
- Its memory was freed by using the `FREE` statement.
- It is the result of a query that returned the empty set (such as the intersection of disjoint sets).

The output from the following statements (see [Figure 2.4](#)) shows that each dimension of an empty matrix is zero.

```

/* dimensions of an empty matrix */
n_u = nrow(empty_matrix);
p_u = ncol(empty_matrix);
print n_u p_u;

```

Figure 2.4 Dimensions of an Empty Matrix

n_u	p_u
0	0

2.2.3 The Type of a Matrix

A matrix is either numeric or character or undefined; you cannot create a matrix that contains both numbers and character strings. You can determine whether a matrix is numeric or character by using the TYPE function, as shown in the following statements:

```

/* determine the type of a matrix */
x = {1 2 3};
y = {"male" "female"};
type_x = type(x);
type_y = type(y);
print type_x type_y;

```

Figure 2.5 Types of Matrices

type_x	type_y
N	C

If a matrix is numeric, the TYPE function returns the character 'N'. If a matrix is character, the TYPE function returns the character 'C'. If a matrix is empty, then the TYPE function returns 'U' (for “undefined”) as shown in the following statements:

```

/* handle an empty or undefined matrix */
type_u = type(undefined_matrix);
if type_u = 'U' then
  msg = "The matrix is not defined.";
else
  msg = "The matrix is defined.";
print msg;

```

Figure 2.6 Result of Handling an Undefined Matrix

msg
The matrix is not defined.

Programming Tip: A matrix is either numeric or character or undefined; you cannot create a matrix that contains both numbers and character strings.

2.2.4 The Length of a Character Matrix

SAS/IML character matrices share certain similarities with character variables in the DATA step. In the DATA step, the length of a character variable is determined when the variable is initialized: either explicitly by using the LENGTH statement or implicitly by the length of the first value for the variable. Similarly, every element in a SAS/IML character matrix has the same number of characters: the length of the longest element. This length is determined *when the matrix is initialized*. Strings shorter than the longest element are padded with blanks on the right.

For example, the following statements define a character matrix with length 4:

```
c = {"Low" "Med" "High"};          /* maximum length is 4 characters */
```

The matrix **c** is initialized to have length 4, the length of its longest character string. Shorter strings such as “Low” are padded on the right so that the first element of **c** is stored as Low□ where □ indicates a blank character.

Programming Tip: The length of a character matrix is determined by the length of its longest element at the time it is created.

You can find the length of a character matrix by using the NLENG function, which returns a single number. You can find the number of characters for each element of a character matrix by using the LENGTH function, which returns a number for each element. These functions are shown in the following statements:

```
/* find the length of a character matrix and of each element */  
nlen = nlen(c);          /* length of matrix          */  
len = length(c);         /* number of characters in each element */  
print nlen len;
```

Figure 2.7 Lengths of Elements in a Character Matrix

nlen	len		
4	3	3	4

Notice that the LENGTH function returns a numerical matrix that is the same dimension as its input argument. For example, the ij th element of **len** is the number of characters in the ij th element of **c**. (Strictly speaking, both functions return numbers that represent bytes. Because each English character is one byte long, the numbers also give the number of characters for matrices that contain English strings.)

When you set the value of an element of an existing matrix, the value is truncated if it is too long, as shown in the following statements:

```
/* assign a long string to a matrix with a shorter length */
c[2] = "Medium";          /* value is truncated! */
print c;
```

Figure 2.8 A Truncated Character String

c			
Low	Medi	High	

The output from these statements is shown in Figure 2.8. The matrix **c** was initialized to have length 4, so when you assign a longer string to an element, only the first four characters fit into the matrix element.

You cannot dynamically change the length of a character matrix, but you can copy its contents to a vector that has a longer (or shorter) character length. The following statements use the PUTC function in Base SAS software to copy a vector of values:

```
/* copy character strings to a matrix with a longer length */
c = {"Low" "Med" "High"}; /* maximum length is 4 characters */
d = putc(c, "$6.");        /* copy into vector with length 6 */
d[2] = "Medium";          /* value fits into d without truncation */
nlen = nlength(d);
print nlen, d;
```

Figure 2.9 Result of Changing the Length of a Character Vector

nlen			
6			
d			
Low	Medium	High	

In the previous statements, the PUTC function applies the \$6. format to every element of **c**. The PUTC function returns a matrix with the same dimensions as **c**. This matrix is stored into **d**.

Notice that **c** contains a vector of character strings, but that the PUTC function acted on each element. This is generally true: you can pass a matrix of values to Base SAS functions and expect them to act on each element.

Programming Tip: You can call functions in Base SAS software from SAS/IML programs. In most cases, these functions act on each element of a matrix.

Strings that are smaller than the maximum length of a character matrix are padded with blanks on

the right. You can see this in [Figure 2.9](#) by noticing that there is more space between the words “Low” and “Medium” than between the words “Medium” and “High.” The value stored in the first element of **d** is “Low□□□” where □ indicates a blank character.

The padding of character strings with blanks on the right can cause a problem when you use the CONCAT function or the string concatenation operator (+) to concatenate strings. The solution to this problem is to use the TRIM function in Base SAS software to remove trailing blanks, as shown in the following statements:

```
/* use the '+' operator to concatenate strings */
msg1 = "I like the " + d[1] + " value.";          /* blanks!    */
msg2 = "I like the " + trim(d[1]) + " value.";      /* no blanks */
print msg1, msg2;
```

Figure 2.10 Result of Concatenating Strings and Removing Trailing Blanks

msg1
I like the Low value.
msg2
I like the Low value.

There are times when a character string also has leading blanks. You can use the STRIP function in Base SAS software to remove both leading and trailing blanks. In most situations that involve string concatenation, you will want to remove leading and trailing blanks.

Programming Tip: When concatenating character strings, use the STRIP function to remove leading and trailing blanks from elements of a character matrix.

2.3 Using Functions to Create Matrices

The SAS/IML matrices in the previous section were created by typing in the elements. More typically, matrices are obtained by using SAS/IML functions to generate data or by reading data from a SAS data set. This section describes creating matrices by using SAS/IML functions; creating matrices by reading a SAS data set is covered in Chapter 3, “[Programming Techniques for Data Analysis](#).”

2.3.1 Constant Matrices

The simplest matrix is a constant matrix. In SAS/IML, the J function creates a constant matrix. The syntax is J(*nrow*, *ncol*, *value*), although you can omit the third argument, which defaults to the

value 1. For example, the following statements create several constant matrices:

```
/* create constant matrices */
c = j(10, 1, 3.14);          /* 10 x 1 column vector      */
r = j( 1, 5);                /* 1 x 5 row vector of 1's   */
m = j(10, 5, 0);             /* 10 x 5 matrix of zeros    */
miss = j(3, 2, .);           /* 3 x 2 matrix of missing values */
```

The first statement creates a column vector with ten rows; each element has the value 3.14. The second statement creates a row vector with five columns; each element is a 1 because the *value* argument is omitted. The third statement creates a 10×5 matrix of zeros. The last statement creates a 3×2 matrix in which every element is a missing value.

Programming Tip: Do not confuse an empty matrix with a matrix that contains missing values or with a zero matrix. An empty matrix has no rows and no columns. A matrix that contains missing values has at least one row and column, as does a matrix that contains zeros.

You can also use the REPEAT function to create a constant matrix or, more generally, a matrix with a repeating pattern of values. The REPEAT function creates a new matrix by repeating a given matrix a specified number of times in each dimension, as shown in the following example:

```
/* create a matrix by repeating values from another matrix */
g = repeat({0 1}, 3, 2);      /* repeat the pattern 3 times down */
print g;                     /* and 2 times across              */
```

Figure 2.11 A Repeated Pattern of Values

g			
0	1	0	1
0	1	0	1
0	1	0	1

The REPEAT function takes three arguments: a matrix of values, the number of times that matrix should be repeated in the vertical dimension, and the number of times that matrix should be repeated in the horizontal dimension. In the example, the vector {0 1} is repeated three times down the rows and twice across the columns.

You can also use the J function and the REPEAT function to create character matrices.

2.3.2 Vectors of Sequential Values

The next simplest matrix to construct is a matrix in which entries follow an arithmetic progression. The DO function (not to be confused with the iterative DO statement) creates a vector with elements that follow an arithmetic series. The syntax is DO(*start*, *stop*, *increment*). Similar to the DO function is the “colon operator,” which can create an arithmetic series with an increment of 1 or -1 .

(The colon operator is also known as the *index creation operator* since it is often used to create an index *start:stop*.) The following statements demonstrate these functions:

```
/* create a vector of sequential values */
i = 1:5;                      /* increment of 1          */
j = 5:1;                      /* increment of -1        */
k = do(1, 10, 2);             /* odd numbers 1, 3, ..., 9 */
print i, j, k;
```

Figure 2.12 Vectors with Sequential Values

i				
1	2	3	4	5
j				
5	4	3	2	1
k				
1	3	5	7	9

The index creation operator (:) has lower precedence than arithmetic operators, as shown by the following statements and by [Figure 2.13](#):

```
/* the index creation operator has low precedence */
n1 = 1;
n2 = 10;
h = n1+2:n2-3;          /* equivalent to 3:7 */
print h;
```

Figure 2.13 Precedence of Index Creation Operator

h				
3	4	5	6	7

Programming Tip: The index creation operator (:) has lower precedence than arithmetic operators. Therefore, $a+b*c:d+e*f$ is equivalent to $(a+b*c):(d+e*f)$.

The DO function requires that its arguments are numerical. But there is one situation in which the index creation operator can be used with character values: the creation of variable names with a common prefix and a numerical suffix. For example, if you have 10 variables and want to name them x_1, x_2, \dots, x_{10} , the index creation operator can create these variable names, as shown in the following statements:

```
/* create variable names with sequential values */
varNames = "x1":"x10";
print varNames;
```


Figure 2.14 Variable Names with Sequential Values

varNames										
x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	

Programming Tip: Use the index creation operator to create a vector of names with a common prefix and a numerical suffix (for example, "x1": "x10").

2.3.3 Pseudorandom Matrices

Probability theory and statistics describe events that contain aspects of randomness. A *random number algorithm* is an algorithm that generates a sequence of numbers whose statistical properties are such that the sequence is indistinguishable from a truly random sequence. Of course, it is technically impossible to *generate* a random sequence, since, by definition, a random process is not deterministic. Therefore, some people prefer use the term *pseudorandom numbers* to describe a sequence of numbers that are generated by a computer and that behave similarly to random variates from a specified distribution.

In SAS software a pseudorandom sequence of numbers is initialized with a *seed value* that determines the sequence. If you use a different seed value, you get a different sequence of numbers.

There are several SAS/IML functions and subroutines that generate pseudorandom variates. The UNIFORM and NORMAL functions generate random variates from the uniform distribution on [0, 1] and from the standard normal distribution, respectively. For example, the following statements generate two variables that are linearly related to each other.

```
/* create pseudorandom vectors */
seed = j(10, 1, 1);          /* set seed (=1) and dimensions */
x = uniform(seed);           /* 10 x 1 pseudorandom uniform vector */
y = 3*x + 2 + normal(seed);  /* linear response plus normal error */
```

The UNIFORM and NORMAL functions are convenient for simple simulations and for quickly generating test data. In the previous statements, the seed value is 1. The size and shape of the **seed** matrix determines the shape of the output from the UNIFORM and NORMAL functions. Since **seed** is a 10×1 vector, the column vector **x** contains 10 pseudorandom numbers in the interval [0, 1]. Similarly, the NORMAL function returns a normally distributed “error vector,” so that the vector **y** is a linear function of **x** plus an error term.

Programming Tip: Use the UNIFORM and NORMAL functions when you need to quickly generate a small random sample for an example or for testing purposes.

The statistical properties of the pseudorandom numbers generated by the UNIFORM and NORMAL functions are not as good as those generated by the newer Mersenne-Twister random number

generator that is implemented in the RANDGEN subroutine. Consequently, you should use the RANDGEN subroutine when you intend to generate millions of pseudorandom numbers.

The RANDGEN subroutine is used extensively in Chapter 13, “Sampling and Simulation.” When you use RANDGEN, you need to allocate a matrix that will hold the random numbers, perhaps by using the J function. For the sake of completeness, the following statements use the RANDGEN subroutine to generate two variables that are linearly related to each other:

```
/* create pseudorandom vectors (better statistical properties) */
call randseed(12345);          /* set seed for RANDGEN          */
x = j(10, 1);                  /* allocate 10 x 1 vector        */
e = x;                         /* allocate 10 x 1 vector        */
call randgen(x, "Uniform");     /* fill x; values from uniform dist */
call randgen(e, "Normal");     /* fill e; values from normal dist */
y = 3*x + 2 + e;               /* linear response plus normal error */
```

Programming Tip: Use the RANDGEN subroutine when the statistical properties of the pseudorandom numbers are important or when you intend to generate millions of variates.

For more information about the numerical properties of SAS routines that generate pseudorandom numbers, see the section “Using Random-Number Functions and CALL Routines” in the *SAS Language Reference: Dictionary*.

2.4 Transposing a Matrix

The vectors created by the DO function and the index creation operators are row vectors. To get a column vector, you can use the T function, which transposes a matrix. The syntax is $T(matrix)$.

The transpose of a row vector is a column vector, and the transpose of a column vector is a row vector. The transpose of a two-dimensional matrix is obtained by flipping the matrix about its main diagonal. Specifically, if a_{ij} is the element in the i th row and j th column of a matrix A , then a_{ji} is the element in the i th row and j th column of the transpose of A . The following statements demonstrate the transpose function:

```
/* transpose a matrix */
s = {1 2 3, 4 5 6, 7 8 9, 10 11 12};          /* 4 x 3 matrix */
transpose = t(s);                             /* 3 x 4 matrix */
print transpose;
```

Figure 2.15 A Transposed Matrix

transpose			
1	4	7	10
2	5	8	11
3	6	9	12

A mathematical notation for the transpose of a matrix A is A' . The SAS/IML language also supports this syntax:

```
sPrime = s` ;           /* alternative notation to transpose a matrix */
```

The transpose operator is described in [Section 2.12](#).

2.5 Changing the Shape of Matrices

Sometimes it is convenient to reshape the data in a matrix. Suppose you have a 1×12 matrix. This same data can fit into many other matrices: for example, a 2×6 matrix, a 3×4 matrix, a 4×3 matrix, and so on. The SHAPE function enables you to specify the number of rows and columns for a new matrix. The values for the new matrix come from an existing matrix, as shown in the following statements:

```
/* reshape a matrix */
x = 1:12;                      /* 1 x 12 matrix */
s = shape(x, 4, 3);            /* reshape data into 4 x 3 matrix */
```

The matrix **s** is identical to the one specified manually in the example in the preceding section.

The data in SAS/IML matrices are stored in row-major order and this ordering of the elements is used in reshaping the data. The SHAPE function does not change the order of the data elements in memory; it merely changes how those data are interpreted as a matrix.

Programming Tip: Data in SAS/IML matrices are stored in row-major order.

The previous example specifies both the number of rows and the number of columns to the SHAPE function. You can also specify only the number of rows or only the number of columns. The dimension that is not specified is determined automatically by dividing the number of elements in the matrix by the number of specified rows or columns. To specify only the number of rows, omit the third argument or use 0 for the number of columns. To specify only the number of columns, specify 0 for the number of rows, as shown in the following statements:

```
/* omit dimension; automatically determined */
s1 = shape(x, 4);              /* 4 rows ==> 3 columns */
s2 = shape(x, 0, 3);           /* 3 columns ==> 4 rows */
```

You can also reshape data into dimensions that are not congruent to the original data by specifying a value to use when the original data “runs out,” as shown in the following statements:

```
/* pad data with a specified value (missing) */
s = shape(1:10, 4, 3, .);      /* 4 x 3, pad with missing value */
print s;
```

Figure 2.16 Result of the SHAPE Function

s		
1	2	3
4	5	6
7	8	9
10	.	.

2.6 Extracting Data from Matrices

This section describes various techniques for extracting or modifying the subsets of a matrix. The subsets include, rows, columns, submatrices, and diagonal elements.

You can specify elements of a matrix with one index or with two. For example, $\mathbf{x}[3]$ is the third element of the matrix \mathbf{x} , specified in row-major order. Similarly, $\mathbf{y}[1,2]$ specifies the element in the first row and the second column of the matrix \mathbf{y} . You can use indices on either side of an assignment statement, as the following statements indicate:

```
/* use indices on either side of assignment statements */
x = {1 2 3, 4 5 6};
y = x[2, 3];          /* value of 2nd row, 3rd column      */
x[2, 3] = 7;          /* changes the value of x[2,3]; y is unchanged */
print x, y;
```

Figure 2.17 Result of Assignment Statements

x		
1	2	3
4	5	7
y		
	6	

If \mathbf{m} is an $n \times p$ matrix, it is an error to refer to $\mathbf{m}[i, j]$ when $i > n$ or $j > p$. It is also an error to use nonpositive indices.

It is an error to refer to a subscript of an undefined matrix. For example, the following statements show that you cannot assign elements of a vector if the vector has not been created:

```
/* ERROR: subscripting a matrix that has not been created */
z[1] = 0;          /* z is undefined */
```

The error message for this mistake is displayed in the SAS log, which is shown in [Figure 2.18](#).

Figure 2.18 Error Message for Undefined Matrix

```

ERROR: (execution) Matrix has not been set to a value.

operation : [ at line 1520 column 2
operands  : z, *LIT1001, *LIT1002

z          0 row          0 col      (type ?, size 0)

*LIT1001    1 row          1 col      (numeric)
           1

*LIT1002    1 row          1 col      (numeric)
           0

statement : ASSIGN at line 1520 column 1

```

Each line of the error message gives information about the error:

- The error is that you are trying to access an element of a matrix that has not been assigned a value.
- The error occurred in the left-bracket (subset) operation.
- The operation involved three operands: the matrix **z** and two unnamed literals.
 - The matrix **z** has zero rows and zero columns. It is an undefined matrix.
 - The first unnamed literal has the value 1.
 - The second has the value 0.
- The error occurred during the assignment statement.

Programming Tip: It is an error to refer to a subscript of an undefined matrix.

2.6.1 Extracting Rows and Columns

You can specify submatrices of a matrix by specifying vectors of indices. For example, the following statements assign the matrix **w** to the values in the odd rows and even columns of the matrix **z**:

```

/* extract submatrix */
z = {1 2 3 4, 5 6 7 8, 9 10 11 12};
rows = {1 3};
cols = {2 4};
w = z[rows, cols];
print w;

```

Figure 2.19 A Submatrix

w	
2	4
10	12

You can specify all rows of a matrix by omitting the row index. Columns are handled similarly. The following statements use the previous definitions of **rows** and **columns** to show this syntax:

```
/* extract only rows or columns */
oddRows = z[rows, ];          /* specified rows; all columns */
evenCols = z[ , cols];        /* all rows; specified columns */
```

Programming Tip: You can specify all rows of a matrix by omitting the row index. Columns are handled similarly.

When you extract a subset of a matrix by specifying a single set of indices, the resulting matrix is always a column vector. At times, this is a surprising result, as shown in the following statements.

```
/* different ways to specify elements of a matrix */
z = {1 3 5 7 9 11 13};          /* row vector */
w = z[{2 4 6}];                 /* column vector with values {3,7,11} */
t = z[ , {2 4 6}];              /* row vector with values {3 7 11} */
print w t;
```

Figure 2.20 Results of Two Indexing Schemes

w	t
3	3
7	7
11	11

Note that **w** is a column vector even though **z** is a row vector! To get a row vector, you must explicitly omit the row index, as for the vector **t** (or use the **SHAPE** function).

Programming Tip: If you extract a subset of a matrix by specifying a single set of indices, the resulting vector is a column vector.

You can also extract all rows *except* those that you enumerate. The **SETDIF** function (described in [Section 2.11](#)) is useful for this. The following statements extract all rows except those contained in the vector **v**:

```

/* extract all rows except those specified in a vector v */
q = {1 2, . 3, 4 5, 6 7, 8 .};
v = {2 5};                               /* specify rows to exclude */
idx = setdif(1:nrow(q), v);               /* start with 1:5, exclude values in v */
r = q[idx, ];                             /* extract submatrix */
print idx, r;

```

Figure 2.21 Rows That Are Not Excluded

idx	
1	3
4	
r	
1	2
4	5
6	7

In the previous statements, the second and fifth rows of the **q** matrix contain missing values. Suppose you want to exclude these rows from a computation. You can define **v** to be the vector that contains the rows to exclude and use the SETDIF function to remove those rows from the vector of all row numbers in **q**. The vector **idx** contains the rows that are retained, as shown in [Figure 2.21](#).

2.6.2 Matrix Diagonals

An important subset of a matrix is the *diagonal* of the matrix. For an $n \times p$ matrix A , the diagonal is the set of elements A_{ii} for $i = 1, \dots, \min(n, p)$. You can extract the diagonal of a matrix by using the VECDIAG function, as shown in the following example:

```

/* extract matrix diagonal */
m = {1  2  3,
     2  5  6,
     3  6 10};
d = vecdiag(m);
print d;

```

Figure 2.22 The Diagonal of a Matrix

d
1
5
10

The VECDIAG function returns a vector of diagonal elements from a matrix, whereas the DIAG function returns a diagonal matrix created from a specified vector of values, as shown in the following example:

```

/* create a diagonal matrix from a vector */
vals = {5, 2, -1};
s = diag(vals);
print s;

```

Figure 2.23 A Diagonal Matrix

s		
5	0	0
0	2	0
0	0	-1

The `I` function returns a square matrix that contains ones on the diagonal. This is called the *identity matrix*. You need to specify the dimension of the matrix, as shown in the following example:

```

/* create an identity matrix */
ident = I(3);                                /* 3 x 3 identity matrix */
print ident;

```

Figure 2.24 An Identity Matrix

ident		
1	0	0
0	1	0
0	0	1

If you want to extract or modify the diagonal values of a general $n \times p$ matrix, you can directly index the elements of the diagonal by using the following statements:

```

/* index the diagonal elements of a matrix */
n = nrow(m);
p = ncol(m);
diagIdx = do(1, n*p, p+1);                /* indices of the diagonal */
print diagIdx;
d2 = m[diagIdx];                          /* extract the diagonal */
print d2;

```

Figure 2.25 Indices and Values of a Matrix Diagonal

diagIdx		
1	5	9
d2		
	1	
	5	
	10	

In the example, the DO function constructs the indices of the diagonal for a general $n \times p$ matrix. These are the correct indices because SAS/IML matrices are stored in row-major order. The indices are stored in the **diagIdx** vector.

You can use the indices in **diagIdx** to extract the diagonal, as shown in the previous example, or to assign to the diagonal elements. For example, a common matrix operation is to subtract a constant from the diagonal of a square matrix, as shown in the following statements:

```
/* compute B = (m - lambda*I) for lambda=1 */
/* First approach: use the formula */
B1 = m - I(n);          /* I(n) gives n x n identity matrix */

/* Second approach: Do not physically create the identity matrix */
B = m;
B[diagIdx] = m[diagIdx] - 1; /* modify the diagonal */
print m B;
```

Figure 2.26 Matrix with Modified Diagonal

m			B		
1	2	3	0	2	3
2	5	6	2	4	6
3	6	10	3	6	9

The example creates the matrix $m - I$ in two different ways. The first way is to use the I function to explicitly create an identity matrix and then to subtract that matrix from m to form the matrix B . This works, but is not as efficient as the alternative approach. The alternative approach initializes the matrix B with the values of m and then subtracts 1 from each diagonal element.

2.6.3 Printing a Submatrix or Expression

There are times when it is convenient to print a submatrix of a matrix, or, in general, a temporary result of some matrix expression. You can print submatrices and expressions by enclosing the term that you want to print in parentheses.

For example, suppose that a data matrix contains hundreds of rows, but you want to print only the first few rows. You can index the rows that you want to print, and enclose the expression in parentheses, as shown in the following statements:

```
x = shape(1:1000, 0, 4);          /* 4 columns, hundreds of rows */
print (x[1:3,]);                 /* print first three rows */
```

Figure 2.27 First Three Rows of a Matrix

	1	2	3	4
	5	6	7	8
	9	10	11	12

The output is shown in [Figure 2.27](#). Notice that the output does not contain any matrix names. This is in contrast to printed output in previous examples in which the name of the matrix is printed in the output. The explanation for this is that when SAS/IML encounters an expression enclosed in parentheses, it creates a temporary matrix to hold the result and the PRINT statement does not output a name for temporary matrices. However, as described in section “[Printing a Matrix](#)” on page 19, you can use the LABEL= and COLNAME= options in the PRINT statement to make the printed output easier to understand, as shown in the following statements:

```
varNames = 'Col1':'Col4';
print (x[1:3,]) [label="My Data" colname=varNames];
```

Figure 2.28 Adding a Label and Column Names to Printed Output

My Data				
Col1	Col2	Col3	Col4	
1	2	3	4	
5	6	7	8	
9	10	11	12	

In the same way, you can print matrix expressions by enclosing the expression in parentheses. For example, the following statements print a linear transformation of a row vector:

```
x = 1:4;
print (3*x + 1);                /* print expression */
```

Figure 2.29 Result of Printing an Expression

4	7	10	13
---	---	----	----

Programming Tip: To print a submatrix, you must enclose the submatrix in parentheses:

```
print (x[1:3]);
```

This is a peculiarity of the PRINT statement. You also need to enclose matrix expressions in parentheses:

```
print (3*x + 1);
```

2.7 Comparison Operators

In the SAS/IML language, the equal sign (=) plays two roles. The equal sign can be an assignment operator or a comparison operator. The other comparison operators are less than (<), less than or equals (<=), greater than (>), greater than or equals (>=), and the not equals operator (^=). Usually you will compare a matrix with a scalar value or with another matrix of the same dimensions, although it is also possible to compare a matrix with a vector. The result of the comparison is a

matrix of zeros and ones. The result matrix has the value one for locations where the comparison is true and the value zero for locations where the comparison is false. This is shown in the following statements:

```
/* comparison operators */
x = {1 2 3, 2 1 1};
s1 = (x=2);
print s1;

z = {1 2 3, 3 2 1};
s2 = (x<z);
print s2;
```

Figure 2.30 Results of Comparison Operators

s1		
0	1	0
1	0	0
s2		
0	0	0
1	1	0

The matrix **s1** has the same dimensions as the matrix **x**. The element **s1[i, j]** has the value 1 when **x[i, j]** equals 2. Similarly, the matrix **s2** has the value 1 in locations where **x[i, j]** is less than **z[i, j]**, and has the value 0 otherwise.

The parentheses in the previous example are not necessary, but might help to remind you that the first equal sign is an assignment, whereas the second is a comparison operator. The SAS/IML language does not support the “multiple assignment” syntax found in some other languages. The expression **x=y=0** does *not* assign the value zero to the matrices **x** and **y**; instead, it compares the matrix **y** with zero, and assigns the result to **x**.

Programming Tip: The comparison operators return a matrix of zeros and ones.

Programming Tip: Unlike the DATA step, the SAS/IML language does not support the mnemonic keywords EQ, NE, GT, LT, GE, or LE as a replacement for the symbols =, ^=, >, <, >=, or <=.

The comparison operators treat a missing value as a value that is less than any valid nonmissing value, as shown in the following example:

```
/* missing values compare as less than any nonmissing value */
m = .;
n = 0;
r = (m<n);
print r;
```

Figure 2.31 Result of Comparing a Missing Value

	r
	1

2.8 Control Statements

This section describes SAS/IML statements that control the flow of a program. This includes the IF-THEN/ELSE statements, the iterative DO statement, the DO/WHILE statement, and the DO/UNTIL statement.

2.8.1 The IF-THEN/ELSE Statement

The comparison operators are most often used in the conditional expression of an IF-THEN/ELSE statement, as shown in the following statements:

```
/* an IF-THEN/ELSE statement */
x = {1 2 3, 2 1 1};
z = {1 2 3, 3 2 1};
if x <= z then
    msg = "all(x <= z)";
else
    msg = "some element of x is greater than the corresponding element of z";
print msg;
```

Figure 2.32 Result of an IF-THEN/ELSE Statement

	msg
	all(x <= z)

The syntax is identical to that used in the DATA step, except that the conditional expression in the SAS/IML language can be a matrix. Recall that the expression $\mathbf{x} \leq \mathbf{z}$ resolves to a matrix of zeros and ones. If every element of $\mathbf{x} \leq \mathbf{z}$ is nonzero (that is, the expression is true for all elements), then the statement following the THEN keyword is executed. Otherwise, the statement following the ELSE keyword is executed. The ELSE statement is optional.

Programming Tip: A matrix expression in an IF-THEN statement is true provided that all elements of the matrix expression are nonzero.

You can use the DO and END statements to group multiple statements that should be executed, as shown in the following statements:

```

/* a DO-END block of statements */
if x <= z then do;
    msg = "all(x <= z)";
    /* more statements ... */
end;
else do;
    msg = "some element of x is greater than the corresponding element of z";
    /* more statements ... */
end;

```

You can use the ALL function to emphasize the condition you are testing:

```

/* the ALL statement */
if all(x <= z) then do;
    msg = "all(x <= z)";
    /* more statements ... */
end;

```

The ALL function returns the value 1 if every element of its argument is nonzero; otherwise, it returns the value 0. If you want to test whether *any* element in a matrix is nonzero, you can use the ANY function, as shown in the following statements:

```

/* the ANY statement */
if any(x < z) then
    msg = "some element of x is less than the corresponding element of z";
print msg;

```

Figure 2.33 Result of the ANY Function

msg
some element of x is less than the corresponding element of z

The ANY function returns 1 if any element of its argument is nonzero; otherwise, it returns 0.

2.8.2 The Iterative DO Statement

The iterative DO statement enables you to repeat a group of statements several times. For example, you can loop over elements in a matrix or loop over variables in a data set. The syntax is identical to that used in the DATA step, as shown in the following statements:

```

/* the iterative DO statement */
x = 1:5;
sum = 0;
do i = 1 to ncol(x);
    sum = sum + x[i];
end;
print sum;
/* inefficient way to sum elements */

```

The statements loop over the columns of **x** and add up each entry. The partial sum at each step in the iteration is accumulated in the variable **sum**. When the loop ends, **sum** contains the sum of the elements in **x**, as shown in [Figure 2.34](#).

Figure 2.34 Result of the Iterative DO Statement

sum
15

In general, you should try to avoid looping over every element in a vector. The SAS/IML language has many functions and statements that enable you to avoid explicit loops. For example, you can rewrite the previous example by using the SUM function to eliminate the loop:

```
x = 1:5;
sum = sum(x) ;                               /* efficient; eliminate DO loop */
```

The SUM function and other functions that act on matrices are described in [Appendix C](#). The section “[Writing Efficient SAS/IML Programs](#)” on page 79 discusses other ways to avoid loops.

The iterative DO statement supports an optional WHILE or UNTIL clause. You can use the WHILE or UNTIL clauses when you want to exit the DO loop after a certain criterion is satisfied, as shown in the following statements:

```
/* an UNTIL clause */
x = 1:5;
sum = 0;
do i = 1 to ncol(x) until(sum > 8);
    sum = sum + x[i];
end;
print sum;
```

Figure 2.35 Result of the UNTIL Clause

sum
10

The WHILE clause is evaluated at the top of the loop, whereas the UNTIL clause is evaluated at the bottom of the loop. Consequently, you can use the WHILE clause when you want to block the execution of the body of the loop. For example, the following statements find the partial sum of a vector until a missing value is encountered:

```
/* a WHILE clause */
x = {1 2 . 4 5};
sum = 0;
do i = 1 to ncol(x) while(x[i]^=.);
    sum = sum + x[i];
end;
print sum;
```

Figure 2.36 Result of the WHILE Clause

sum
3

The statements work correctly because the WHILE clause forces the loop to exit as soon as **i** is incremented to 3. If you try to use an UNTIL clause (such as `until(x[i]=.)`), the body of the loop executes and results in the missing value being added to the **sum** variable.

You can also use the WHILE and UNTIL clauses in conjunction with a noniterative DO statement. In this case, you need to control the iteration yourself, and you need to avoid conditions that might lead to infinite loops. For example, the following statements use a DO-UNTIL statement to approximate the first local maximum of the sine function for $x > 0$:

```
/* a DO-UNTIL statement */
x = 0;
dx = 0.01;
do until(sin(x) > sin(x+dx));
    x = x + dx;
end;
print x;
```

Figure 2.37 Result of the DO-UNTIL Statement

x
1.57

The true value of the local maximum is $\pi/2 \approx 1.57$. The loop adds a small increment to **x** at each iteration. The condition in the UNTIL clause tests whether the sine function is increasing at the current value of **x**. The loop continues until **x** attains a value at which the sine function is no longer increasing.

2.9 Concatenation Operators

A previous section described ways to extract a subset of a matrix. This section describes how you can concatenate matrices together to form a bigger matrix from smaller ones.

The SAS/IML language provides two concatenation operators. The horizontal concatenation operator (**||**) appends new columns to a matrix, or combines two matrices that have the same number of rows. A typical usage is to create a data matrix consisting of columns from several vectors, as shown in the following statements:

```

/* horizontal concatenation */
a = {1,2,3,4,5};                      /* 5 x 1 column vectors */
b = {3,5,4,1,3};
c = {0,1,0,0,1};
x = a || b || c;                      /* 5 x 3 matrix */
print x;

```

Figure 2.38 Result of Horizontal Concatenation

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

The vertical concatenation operator (//) appends new rows or combines matrices that have the same number of columns. A typical usage is that you want to combine several subsets of data, where each subset is contained in its own matrix. For example, the following statements combine a subset of the data (say, for males) with another subset (say, for females):

```

/* vertical concatenation */
males = {1 3 0, 2 5 1, 3 4 0};        /* 3 x 3 matrix */
females = {4 1 0, 5 3 1};             /* 2 x 3 matrix */
x = males // females;                 /* 5 x 3 matrix */

```

Because the concatenation operators allocate a new matrix and copy data into it, you should try to avoid using concatenation operators inside of a loop. For example, suppose you want to construct a vector of even integers {2 4 6 8 10}. The following statements show an inefficient way to construct this vector:

```

/* construct vector of even integers */
free x;                               /* make sure x is empty */
do i = 1 to 5;
  x = x || 2*i;                       /* inefficient! 5 allocations */
end;

```

The program first uses the FREE statement to ensure that **x** is an empty matrix. Inside the DO loop, the program allocates a new matrix for each iteration of the loop. The first iteration concatenates the empty matrix with the scalar that contains 2. At the end of the first iteration, the matrix **x** is 1×1 . At the end of the second iteration, the matrix **x** is 1×2 , and so on.

It is not merely the allocation of memory that is inefficient. The algorithm is also inefficient because of the way it copies elements multiple times. During the five iterations, the value 2 is copied into a new matrix five times. The value 4 is copied four times, and so on. In total, the five values are copied 15 times before the loop ends. If the same algorithm is used to create a vector of length n , the values are copied a total of $n(n-1)/2$ times!

If the purpose of the loop is to compute values for a matrix with a given number of rows and columns, it is usually more efficient to allocate space for the matrix prior to the loop. You can assign the values to the preallocated matrix inside the loop, as shown in the following statements:


```

/* Improved algorithm: allocate the matrix to hold the
   even integers. Assign values into the vector. */
x = j(1, 5, .);                      /* allocate; fill with missing */
do i = 1 to ncol(x);
    x[i] = 2*i;                      /* assign value */
end;

```

This algorithm requires a single allocation of a vector, and no values are copied unnecessarily. Of course, for this simple example, you can avoid the DO loop altogether:

```
x = 2*(1:5);
```

Programming Tip: Whenever possible, avoid using concatenation operators to construct a matrix row-by-row (or column-by-column) inside a loop.

You cannot always avoid concatenation within a DO loop because there are situations in which you do not know in advance how many iterations are required. Nevertheless, even in those situations you can often obtain an upper bound on the number of iterations and preallocate the results matrix. An example is a root-finding algorithm that iterates until convergence. You do not know in advance how many iterations are required, but you can decide to limit the algorithm to no more than 50 iterations before stopping the algorithm. In this case, you can preallocate space for up to 50 results.

The concatenation operators enable you to append one matrix to the end of another. If you need to insert rows or columns into the middle of a matrix, you can use the INSERT function. If you need to remove rows or columns from the middle of a matrix, you can use the REMOVE function. For example, the following statements remove the first, third, and sixth elements of a vector, which has the effect of removing all missing values from the vector:

```

y = {., 1, ., 2, 3, .};
v = remove(y, {1 3 6});

```

See the *SAS/IML User's Guide* for details of the INSERT and REMOVE statements.

2.10 Logical Operators

The IF-THEN/ELSE, DO-UNTIL, and DO-WHILE statements all test whether a condition is true. When the condition involves a matrix, the condition is considered “true” provided that it is true for each element of the matrix.

You can also test whether multiple conditions are true and combine two or more conditions into a single logical expression. The three logical operators are the AND operator (&), the OR operator (!), and the NOT operator (^). The operators perform logical operations on the elements of matrices, as shown in the following statements:

```

/* logical operators */
r = {0 0 1 1};
s = {0 1 0 1};
and = (r & s);
or  = (r | s);
not = ^r;
print and, or, not;

```

Figure 2.39 Results of Logical Operators

	and			
	0	0	0	1
	or			
	0	1	1	1
	not			
	1	1	0	0

As the example indicates, the matrix **and** contains a nonzero value only where the corresponding elements of **r** and **s** are nonzero. Similarly, the matrix **or** contains a nonzero value if either of the corresponding elements of **r** and **s** are both nonzero. Lastly, the matrix **not** is nonzero only where the matrix **r** is zero.

Programming Tip: Unlike the DATA step, the SAS/IML language does not support the mnemonic keywords AND, OR, or NOT as a replacement for the symbols **&**, **|**, and **^**.

You can use the logical operators to combine multiple criteria into a single criterion. For example, the following statements use the logical AND operator to test whether the values of a vector are all in the domain of a function:

```

/* logical combination of criteria */
x = do(-1, 1, 0.5);
if (x>= -1) & (x<=1) then
  y = sqrt(1-x##2); /* the ## operator squares each element of x */
else
  y = "Unable to compute the function.";
print y;

```

Figure 2.40 Result of the Logical AND (&) Operator

y		
0 0.8660254	1 0.8660254	0

There are two conditions in the IF-THEN statement. The first tests whether all of the elements of \mathbf{x} are greater than or equal to -1 . The second tests whether all of the elements of \mathbf{x} are less than or equal to 1 . The logical AND operator ensures that the SQRT function is called only if both of the conditions are satisfied. The parentheses are not necessary, but are included for clarity. For this example, both conditions are satisfied, so the SQRT function is called. The argument to the SQRT function uses the elementwise power operator (##) to square each element of \mathbf{x} . Consequently, the i th element of \mathbf{y} is equal to $\sqrt{1 - x_i^2}$, where x_i is the i th element of \mathbf{x} .

You can often write a logical condition in multiple ways. For example, the following IF-THEN statements are logically equivalent to the previous IF-THEN statement:

```
/* different ways to compute the same logical condition */
if (x< -1) | (x>1) then
  y = "Unable to compute the function.";
else
  y = sqrt(1-x##2);

if ^(x< -1) & ^(x>1) then
  y = sqrt(1-x##2);
else
  y = "Unable to compute the function.";
```

Some programming languages support a feature known as *minimal evaluation* or *short-circuit evaluation*. In these languages, the second argument of a logical AND or OR expression is evaluated only if the first evaluation does not already determine the truth of the expression. For example, in the expression $\mathbf{r} \ \& \ \mathbf{s}$, if \mathbf{r} contains a zero, then the entire expression is false, regardless of the value of \mathbf{s} . Similarly, in the expression $\mathbf{r} \ | \ \mathbf{s}$, if \mathbf{r} contains all nonzero values, then the entire expression is true. The SAS/IML language does *not* support short-circuit evaluation.

Programming Tip: The SAS/IML language does not support short-circuit evaluation.

Why does this matter? Some programmers write logical expressions that assume short-circuiting will occur. For example, in some languages you can write the following logical expression:

```
/* incorrect: second condition evaluated even if first is false */
if (x>0) & (log(x)>1) then do;                /* ERROR if x<=0 */
  /* more statements */
end;
```

This is a valid expression in a language that short-circuits operations because the expression that contains `log(x)` is executed only if \mathbf{x} is positive, as ensured by the first condition. However, in the SAS/IML language, both expressions are evaluated, and then the AND operator combines the results. This means that the LOG function is always called and that the program will stop with an error if some element of \mathbf{x} is nonpositive.

Instead, you need to write the previous statements by using nested IF-THEN statements, as shown in the following example:

```

/* correct: second condition not evaluated if first is false */
if x>0 then
  if log(x)>1 then do;
    /* more statements */
  end;

```

2.11 Operations on Sets

There are three SAS/IML functions that perform operations on sets. The UNION function returns the union of the values in one or more matrices. The XSECT function returns the intersection of the values in two or more matrices. (The intersection is the set of values common to all matrices.) The SETDIF function accepts two arguments, say **A** and **B**, and returns the values in matrix **A** that are not found in matrix **B**.

These set functions are shown in the following statements:

```

/* union, intersection, and difference of sets */
A = 1:4;
B = do(2.5, 4.5, 0.5);
u = union(A, B);           /* union                */
inter = xsect(A, B);       /* intersection        */
dif = setdif(A, B);        /* difference between sets */
print u, inter, dif;

```

As shown in [Figure 2.41](#), each of these functions returns a row vector. The vector **u** contains the union of the values in **A** and **B**, the vector **inter** contains the intersection of the values, and **dif** contains the difference between the sets determined by **A** and **B**.

Figure 2.41 Results of Operations on Sets

u						
1	2	2.5	3	3.5	4	4.5
inter						
		3	4			
dif						
1	2					

In this example, the intersection and the difference between sets were both nonempty, but this is not always the case. The XSECT function returns an empty matrix if there are no elements in the intersection of the matrices. Similarly, the SETDIF function returns an empty matrix if the elements of **A** are a proper subset of the elements of **B**. Consequently, you need to check that the matrix returned by XSECT or SETDIF is nonempty before you print it or use it in further computations.

The following statements show an example in which the intersection is empty:

```
/* check whether intersection and set difference are empty */
A = 1:4;
B = do(5, 8, 0.5);
inter = xsect(A, B);
if ncol(inter)>0 then
    print inter;

dif = setdif(A, B);
if ncol(dif)>0 then
    print dif;
```

Figure 2.42 A Nonempty Set Difference

dif				
1	2	3	4	

The matrix **inter** is created by the XSECT function, but it has zero rows and zero columns. You can therefore use the NCOL function to detect the empty intersection. (You could also check to see if **type(inter)** has the value 'U'.) The program avoids run-time errors by only executing the PRINT statement for nonempty matrices.

Programming Tip: The intersection of sets can be empty, so always check the matrix returned by XSECT and handle the possibility that the matrix is empty. Do the same for set differences found by using the SETDIF function.

2.12 Matrix Operators

The fundamental unit in the SAS/IML language is the matrix. This section describes how to perform arithmetic operations that involve scalars, vectors, and matrices.

2.12.1 Elementwise Operators

The elementwise operators in the SAS/IML language consist of one unary operator and several binary operators. The unary operator is the negation operator (**-**), which is equivalent to multiplication by **-1**. The elementwise binary operators are the addition operator (**+**), the subtraction operator (**-**), the elementwise multiplication operator (**#**), the division operator (**/**), and the power or exponentiation operator (**##**).

In most cases, the SAS/IML language enables you to write concise, high-level expressions that combine scalars, vectors, and matrices. The SAS/IML software determines if it can make sense

of an arithmetic expression in which the matrices have different dimensions. For example, the following statements show how you can write an arithmetic expression that contains a matrix and also a scalar or a vector:

```
/* elementwise matrix operations with scalar or vector */
x = {7 7, 8 9, 7 9, 5 7, 8 8};
grandmean = 7.5;          /* mean of all elements */
y = x - grandmean;        /* subtract 7.5 from each element */
mean = {7 8};             /* mean of each column */
xc = x - mean;            /* subtract (7 8) from each row */
print y xc;
```

The assignment statement for **y** contains an expression that subtracts a scalar value from a 5×2 matrix, **x**. The SAS/IML language assumes that this is shorthand notation for subtracting the scalar 7.5 from every element of **x**. On the assignment statement for **xc**, the program subtracts a 1×2 vector from **x**. Since SAS 9.2, the SAS/IML language has assumed that this is shorthand for subtracting the vector {7 8} from every row of **x**. The resulting matrices are shown in Figure 2.43.

Figure 2.43 Results of Elementwise Operations

y		xc	
-0.5	-0.5	0	-1
0.5	1.5	1	1
-0.5	1.5	0	1
-2.5	-0.5	-2	-1
0.5	0.5	1	0

In general, if **m** is an $n \times p$ matrix, you can perform elementwise operations with a second matrix **v** provided that **v** has one of the following dimensions: 1×1 , $n \times 1$, $1 \times p$, or $n \times p$. The result of the elementwise operation is shown in Table 2.1, which describes the behavior of elementwise operations for the +, -, #, /, and ## operators.

Table 2.1 Behavior of Elementwise Operators (SAS/IML 9.2)

Size of v	Result of m op v
1×1	v applied to each element of m
$n \times 1$	v[i] applied to row m[i,]
$1 \times p$	v[j] applied to column m[, j]
$n \times p$	v[i, j] applied to element m[i, j]

Programming Tip: If you follow the rules presented in Table 2.1, you can perform elementwise operations on rows or columns of a matrix by specifying an appropriate column vector or row vector.

The following statements (which continue from the previous example) give concrete examples for the remaining rules presented in Table 2.1:

```

/* elementwise matrix operations with vector or matrix */
/* r is row vector */
r = {1.225 1};                                /* std dev of each col      */
std_x = xc / r;                                /* divide each column (normalize) */
/* c is column vector */
c = x[,1];                                    /* first column of x        */
y = x - c;                                    /* subtract from each column */
/* m is matrix */
m = {6.5 7.5, 7.9 9.1, 7.5 8.5, 5.6 6.4, 7.5 8.5};
deviations = x - m;                            /* difference between matrices */
print std_x y deviations;

```

The program shows that you can multiply or divide every row (or column) of a matrix by a scalar or a vector in a natural way. For example, the statement that assigns the matrix `std_x` divides each column of `x` by its standard deviation, thus normalizing the scale of each column. The results of this and the other computations are shown in Figure 2.44.

Figure 2.44 More Results of Elementwise Operations

std_x		y	deviations	
0	-1	0	0	0.5
0.8163265	1	0	1	0.1
0	1	0	2	-0.5
-1.632653	-1	0	2	-0.6
0.8163265	0	0	0	0.5

2.12.2 Matrix Computations

The SAS/IML language supports operators for specifying high-level matrix operations such as matrix multiplication. The matrix multiplication operator is `*`. If **A** is an $n \times p$ matrix and **B** is a $p \times m$ matrix, the product **A*****B** is an $n \times m$ matrix. The ij th entry of the product is the sum $\sum_{k=1}^p A_{ik} B_{kj}$. If **B** is a column vector, then set $j = 1$ in the previous formula.

For example, the following statements multiply a matrix and a vector:

```

/* matrix multiplication */
A = {7 7, 8 9, 7 9, 5 7, 8 8};                /* 5 x 2 matrix            */
v = {1, -1};                                  /* 2 x 1 vector            */
y = A*v;                                       /* result is 5 x 1 vector */
print y;

```

Figure 2.45 Result of Matrix Multiplication

y
0
-1
-2
-2
0

As a convenience, you can also use the `*` operator to multiply a matrix by a scalar quantity, such as `3*A`. This performs elementwise multiplication and is equivalent to `3#A`.

Another matrix operator is the transpose operator (```). This operator is typed by using the GRAVE ACCENT key. (The GRAVE ACCENT key is located in the upper left corner on US and UK QWERTY keyboards.) The operator transposes the matrix that follows it. This notation mimics the notation often seen in statistical textbooks that discuss matrix equations.

For example, given an $n \times p$ data matrix X and a vector of n observed responses, y , a goal of ordinary least squares (OLS) regression is to find the linear combination of the columns of X that best approximates y . The so-called *normal equations* provide a computational solution to this problem. The normal equations are written as $(X'X)b = X'y$. Solving this equation means solving for the unknown $p \times 1$ parameter vector b .

The following statements use the matrix transpose operator to compute the $X'X$ matrix and the $X'y$ vector for example data:

```
/* Set up the normal equations (X`X)b = X`y */
x = (1:8)`;                               /* X data: 8 x 1 vector */
y = {5 9 10 15 16 20 22 27}`;             /* corresponding Y data */

/* Step 1: Compute X`X and X`y */
x = j(nrow(x), 1, 1) || x;                /* add intercept column */
xpx = x` * x;                             /* cross-products */
xpy = x` * y;
```

The vector `x` is initially defined as the column vector that results from transposing the row vector `1:8`. The vector `y` is also defined by transposing a row vector. (You can equivalently define `y={5, 9, ..., 27}` by typing commas between each pair of values, but it is easier to type one grave accent than to type seven commas.) The statements next append a vector of ones to the `x` data; this column is used to compute an intercept term in an OLS regression model. The matrix `xpx` and the vector `xpy` are computed by using a natural notation that mimics the mathematics of the problem.

Because the transpose operator can be difficult to see, some programmers prefer to use the `T` function to indicate transposition. However, the SAS/IML language optimizes certain computations such as the computation of `xpx`. For that reason, using the transpose operator can sometimes result in better performance than using the `T` function.

Programming Tip: The SAS/IML language optimizes certain computations that involve the transpose operator (```). Consequently, you should use the transpose operator instead of the `T` function when you are concerned about the speed of matrix computations.

The SAS/IML language also provides functions that enable you to numerically solve the normal equations for the unknown parameter b . Textbooks often write the solution of the normal equations as $b = (X'X)^{-1}X'y$, where A^{-1} indicates the inverse of the matrix A . The SAS/IML language provides the `INV` function for computing the inverse of a function, so many SAS/IML programmers use the following statements to numerically solve the normal equations:


```

/* solve linear system */
/* Solution 1: compute inverse with INV (inefficient) */
xpxi = inv(xpx);           /* form inverse crossproducts */
b = xpxi * xpy;           /* solve for parameter estimates */

```

A better technique is to use the SOLVE function to numerically solve the normal equations:

```

/* Solution 2: compute solution with SOLVE. More efficient */
b = solve(xpx, xpy);       /* solve for parameter estimates */

```

Explicitly solving for the inverse matrix is inefficient if you are only interested in solving a linear equation. Not only does the INV function need to allocate memory, but the INV function (which solves for a *general* solution) is often less accurate than the SOLVE function (which solves for a particular solution). There might occasionally be situations in which you need to compute the inverse matrix, but you should avoid it when you have the option.

Programming Tip: If you need to solve the linear equation $Ab = z$, use the SOLVE function (`b=solve(A, z)`) unless there is a compelling reason to explicitly compute the inverse matrix A^{-1} .

2.13 Managing the SAS/IML Workspace

The SAS/IML language enables you to view and manage the matrices that are kept in memory. At any point in your program, you can see the matrices that are defined, free the memory that is associated with matrices, and store matrices to a SAS catalog.

You can use the SHOW statement to display the names, dimensions, and type of matrices that are defined and are in scope. You can specify a list of matrices or you can use the SHOW NAMES statement to display information about all matrices. Both of these techniques are shown in the following statements:

```

/* display the names, dimensions, and type of matrices */
proc iml;
x = 1:3;           /* define some matrices */
y = j(1e5, 100);   /* large matrix: 10 million elements */
animals = {"Cat" "Dog" "Mouse",
           "Cow" "Pig" "Horse"};
show names;        /* show information about all matrices */
show y animals;    /* show information about specified matrices */

```

Figure 2.46 Information about Defined Matrices

SYMBOL	ROWS	COLS	TYPE	SIZE
animals	2	3	char	5
x	1	3	num	8
y	100000	100	num	8
Number of symbols = 3 (includes those without values)				
y 100000 rows 100 cols (numeric)				
animals 2 rows 3 cols (character, size 5)				

If you no longer need certain matrices, you can use the FREE statement to free the associated memory. For example, the following statements free the memory that is associated with a large matrix:

```
/* delete one or more matrices by listing their names */
free y;
show names;
```

Figure 2.47 Deleting a Matrix

SYMBOL	ROWS	COLS	TYPE	SIZE
animals	2	3	char	5
x	1	3	num	8
Number of symbols = 3 (includes those without values)				

You can free all matrices by placing a slash (/) on the FREE statement, as shown in the following example. Figure 2.48 shows that three matrix names (symbols) have been used in this session, but none of the matrices have values after the FREE statement is executed.

```
/* delete all matrices */
free /;
show names;
```

Figure 2.48 Deleting All Matrices

SYMBOL	ROWS	COLS	TYPE	SIZE
Number of symbols = 3 (includes those without values)				

If you need to shut down your computer or exit the SAS System, but you want to save the state of your program so that you can resume work at a later time, you can save some or all of the matrices that are currently defined. This is especially important if some matrices are the result of a lengthy computation. You can use the STORE statement to save matrices. By default, the STORE statement saves the matrices to a SAS catalog in the Work library. However, the Work library vanishes at the

Figure 2.51 Removing Matrices from Storage

```
Contents of storage library = MYLIB.MYSTORAGE  
  
Matrices:  
  
Modules:
```

As an alternative to using the `_ALL_` keyword, you can specify a list of names on the `STORE` and `LOAD` statements to indicate which matrices you want to store and load. Similarly, you can remove a specified list of matrices from storage.

Chapter 3

Programming Techniques for Data Analysis

Contents

3.1	Overview of Programming Techniques	55
3.2	Reading and Writing Data	56
3.2.1	Creating Matrices from SAS Data Sets	56
3.2.2	Creating SAS Data Sets from Matrices	58
3.3	Frequently Used Techniques in Data Analysis	59
3.3.1	Applying a Variable Transformation	59
3.3.2	Locating Observations That Satisfy a Criterion	60
3.3.3	Assigning Values to Observations That Satisfy a Criterion	65
3.3.4	Handling Missing Values	65
3.3.5	Analyzing Observations by Categories	68
3.4	Defining SAS/IML Modules	72
3.4.1	Function and Subroutine Modules	72
3.4.2	Local Variables	73
3.4.3	Global Symbols	74
3.4.4	Passing Arguments by Reference	74
3.4.5	Evaluation of Arguments	75
3.4.6	Storing Modules	76
3.4.7	The IMLMLIB Library of Modules	78
3.5	Writing Efficient SAS/IML Programs	79
3.5.1	Avoid Loops to Improve Performance	79
3.5.2	Use Subscript Reduction Operators	80
3.5.3	Case Study: Standardizing the Columns of a Matrix	83
3.6	Case Study: Finding the Minimum of a Function	84
3.7	References	88

3.1 Overview of Programming Techniques

This chapter features SAS/IML statements, tips, and techniques in SAS/IML that might not be familiar to a SAS/STAT programmer, but that are useful for writing programs that analyze data.

Many of these techniques are used in subsequent chapters. For example, this chapter describes how to locate observations that satisfy a criterion, how to handle missing values in matrices, how to analyze observations by categories, how to define and call SAS/IML modules, and how to write efficient SAS/IML programs.

3.2 Reading and Writing Data

You can create a matrix by reading data from a SAS data set. You can also create a SAS data set from the contents of a matrix. This section discusses both of these techniques.

3.2.1 Creating Matrices from SAS Data Sets

This section describes how to read data from a SAS libref into SAS/IML vectors and matrices. If you have not already done so, follow the directions in [Section 1.8](#) to install the data used in this book. (The examples in this book read data from the Sasuser library. If you install the data in a different SAS library, use that library instead of Sasuser.)

The Sasuser.Vehicles data contains engine characteristics and fuel efficiency for 1,187 vehicles sold in 2007. You can read data from a SAS data set by using the USE and READ statements. You can read variables into individual vectors by specifying a character matrix that contains the names of the variables that you want to read. The READ statement creates column vectors with those same names, as shown in the following statements:

```
/* read variables from a SAS data set into vectors */
varNames = {"Make" "Model" "Mpg_City" "Mpg_Hwy"};
use Sasuser.Vehicles;          /* open data set for reading */
read all var varNames;         /* create four vectors: Make, ..., Mpg_Hwy */
close Sasuser.Vehicles;        /* close the data set */

idx = 1:3;                     /* print only the first 3 observations */
print (Make[idx])
      (Model[idx])
      (Mpg_City[idx]) [label="Mpg_City"]
      (Mpg_Hwy[idx]) [label="Mpg_Hwy"];
```

Figure 3.1 First Three Observations Read from a SAS Data Set

		Mpg_City	Mpg_Hwy
Aston Martin	V8 Vantage	14	20
Aston Martin	V8 Vantage ASM	15	21
BMW	Z4 3 0I	20	30

The USE statement opens Sasuser.Vehicles for reading. The READ statement reads observations into vectors or into a matrix. In this book, the examples almost always use the ALL option to read

all the observations. See the chapter “Working with SAS Data Sets” in the *SAS/IML User’s Guide* for further details on reading data sets.

The `Sasuser.Vehicles` data has 1,187 observations, but only the first few observations are printed. The section “[Printing a Submatrix or Expression](#)” on page 35 describes the use of parentheses in printing submatrices.

You can also read a set of variables into a matrix (assuming the variables are either all numeric or all character) by using the `INTO` clause on the `READ` statement. The following statements illustrate this approach:

```
/* read variables from a SAS data set into a matrix */
varNames = {"Mpg_City" "Mpg_Hwy" "Engine_Cylinders"};
use Sasuser.Vehicles;
read all var varNames into m; /* create matrix with three columns */
close Sasuser.Vehicles;
print (m[1:3,]) [colname=VarNames];
```

Figure 3.2 First Three Observations Read from a SAS Data Set into a Matrix

Mpg_City	Mpg_Hwy	Engine_Cylinders
14	20	8
15	21	8
20	30	6

The `COLNAME=` option on the `PRINT` statement is used to display the variable names in the output. Notice also that the `READ/INTO` statement does not associate variable names with columns of a matrix. However, you can use the `MATTRIB` statement to permanently associate variable names to columns of a matrix.

You can read only the numeric variable in a data set by specifying the `_NUM_` keyword on the `READ` statement:

```
/* read all numeric variables from a SAS data set into a matrix */
use Sasuser.Vehicles;
read all var _NUM_ into y[colname=NumericNames];
close Sasuser.Vehicles;
print NumericNames;
```

Figure 3.3 The Names of the Numeric Variables Read into a Matrix

NumericNames			
COL1	COL2	COL3	COL4
ROW1 Engine_Liters	Engine_Cylinders	Mpg_City	Mpg_Hwy
NumericNames			
COL5	COL6		
ROW1 Mpg_Hwy_Minus_City	Hybrid		

The matrix **NumericNames** contains the names of the numeric variables that were read; the columns of matrix **y** contain the data for those variables.

3.2.2 Creating SAS Data Sets from Matrices

You can use the CREATE and APPEND statements to write a SAS data set from vectors or matrices. The following statements create a data set called MyData in the Work library:

```
/* create SAS data set from vectors */
call randseed(12345);          /* set seed for RANDGEN          */
x = j(10, 1);                  /* allocate 10 x 1 vector       */
e = x;                          /* allocate 10 x 1 vector       */
call randgen(x, "Uniform");     /* fill x; values from uniform dist */
call randgen(e, "Normal");      /* fill e; values from normal dist */
y = 3*x + 2 + e;                /* linear response plus normal error */

create MyData var {x y};        /* create Work.MyData for writing */
append;                         /* write data in x and y         */
close MyData;                   /* close the data set            */
```

The CREATE statement opens Work.MyData for writing. The variables x and y are created; the type of the variables (numeric or character) is determined by the type of the SAS/IML vectors of the same name. The APPEND statement writes the values of the vectors listed on the VAR clause of the CREATE statement. The CLOSE statement closes the data set. It is a good programming practice to close data sets when you are finished reading or writing the data because it frees up memory and system resources, and also prevents corruption of the data in the event that your program terminates prematurely. (For example, you might need to terminate a program if a computation is taking much longer than you expected.)

Programming Tip: Always close your data sets when you are finished reading or writing the data.

You can write character vectors in the same way. Row vectors are written to data sets as if they were column vectors. If the **x** and **y** vectors do not contain the same number of elements, then the shortened vector is padded with missing values.

If you want to create a data set from a matrix of values, you can use the FROM clause on the CREATE and APPEND statements. If you do not explicitly specify names for the data set variables, the default names are COL1, COL2, and so forth. You can explicitly specify names for the data set variables by using the COLNAME= option to the FROM clause, as shown in the following statements:

```
/* create SAS data set from a matrix */
call randseed(12345);          /* set seed for RANDGEN          */
x = j(10, 3);                  /* allocate 10 x 3 matrix       */
call randgen(x, "Normal");      /* fill x; values from normal dist */

create MyData2 from x[colname={"Random1" "Random2" "Random3"}];
append from x;
close MyData2;
```


Most of the examples in this book use the Work libref for temporary storage. You can create a permanent data set by using the LIBNAME statement to create a libref, as shown in the following statements:

```
/* create a libref and create a permanent SAS data set */
libname MyLib 'your data directory';
create MyLib.MyData var {x y};
append;
close MyLib.MyData;
```

Programming Tip: In SAS/IML Studio, the SAS Workspace Server might be a different computer than the PC that runs the SAS/IML Studio application. In this case, remember that the LIBNAME statement creates a libref that refers to a directory on the SAS server, not on the PC client.

3.3 Frequently Used Techniques in Data Analysis

The SAS/IML language can be very useful in data analysis. This section describes SAS/IML statements and techniques that you can use to analyze your data. In particular, this section describes how to do the following:

- transform variables
- locate observations that satisfy a criterion
- handle missing values
- analyze observations for each level of a categorical variable

3.3.1 Applying a Variable Transformation

It is common to transform data during exploratory data analysis and statistical modeling. For example, you can apply a linear transformation to scale or center a variable. If a variable is heavily skewed, it is common to apply a logarithmic transformation in an attempt to work with a more symmetric distribution.

Logarithmic transformations are common when dealing with variables that describe how much something costs. For example, in the Movies data set, the Budget and US_Gross variables are prime candidates for a log transformation. Both natural logarithms and base-10 logarithms are frequently used in practice.

Transforming variables in SAS/IML software is easy since the language supports common mathematical functions and algebraic expressions. In addition to logarithmic transformations, frequently used transformations in data analysis include square root transformations, inverse transformations, power transformations, and exponential transformations. All of these are easily programmed in

SAS/IML software, often with a single statement. For example, the following statements apply a natural log transform to the Budget variable in the Movies data set:

```
/* apply log transform to data */
use Sasuser.Movies;                /* open data set for reading */
read all var {"Budget"};           /* read data */
close Sasuser.Movies;              /* close data set */

logBudget = log(Budget);            /* apply log transform */
```

A log transformation is well-defined for these data because the budget of a movie is always positive. If some data vector, say y , is not positive, it is common to add a constant to all values in order to produce a vector that contains positive values. You can show that $y + c > 0$, provided that $c > -\min(y)$. The following statements exploit that fact to compute a \log_{10} transformation for nonpositive data:

```
/* apply log10 transform to data with negative values */
y = {10 0 -29 -20 273 70};
c = 1 - min(y);                /* (y + c) > 0 whenever c > -min(y) */
log10Y = log10(y + c);          /* apply log10 transformation */
print log10Y;
```

Figure 3.4 A \log_{10} Transformation

log10Y					
1.60206	1.4771213	0	1	2.4814426	2

The section “[Variable Transformations](#)” on page 181 describes an alternative approach for applying a log transformation to a variable that contains nonpositive values.

For other analyses (especially multivariate analyses), you might want to standardize the data to have zero mean and unit standard deviation. Standardizing a variable is described in section “[Case Study: Standardizing the Columns of a Matrix](#)” on page 83.

3.3.2 Locating Observations That Satisfy a Criterion

A frequent task in data analysis is to locate observations that satisfy some criterion. For example, you might want to locate all of the males in your data, or all of the events that happened prior to some target date. The LOC function is the key to efficiently finding these observations.

Many experienced SAS programmers are not familiar with the LOC function. That is unfortunate, because it is a powerful function. Perhaps the reason for its obscurity is that it does not have an analogue in the DATA step. Or perhaps the SAS/IML documentation is partially to blame, since the documentation states that the LOC function “finds nonzero elements of a matrix.” The casual reader can be forgiven for overlooking the power of such a seemingly bland function.

The power of the LOC function is best understood by looking first at the SAS/IML comparison operators (less than, equal to, greater than, and so forth) as described in the section “[Comparison Operators](#)” on page 36. The SAS/IML comparison operators create a matrix of zeros and ones,

where 0 indicates that the comparison is false and where 1 indicates that the comparison is true. For example, the following are valid SAS/IML statements:

```
/* compare data; the comparison operator returns zeros and ones */
x = 1:5;
t = (x > 2);          /* t = {0 0 1 1 1} */
```

The matrix **t** contains nonzero values in exactly the locations at which **x** satisfies the given condition. Consequently, if you want to know the indices for which **x** is greater than two, you can use the LOC function, as shown in the following statements:

```
/* find indices that correspond to nonzero values */
idx = loc(t);          /* idx = {3 4 5}; */
print t, idx;
```

Figure 3.5 Indices That Satisfy a Condition

t				
0	0	1	1	1
idx				
	3	4	5	

In practice, you do not need to create the named vector **t**. SAS/IML creates a temporary matrix when it needs to compute an intermediate result. Therefore, an experienced SAS/IML programmer writes the following statements:

```
/* find indices that satisfy a condition by using the LOC function */
x = 1:5;
idx = loc(x > 2);      /* idx = {3 4 5}; */
```

Now the utility of the LOC function becomes clear: you can use the LOC function to find observations that satisfy arbitrarily complex criteria! For example, if you are trying to find vehicles that are fuel efficient, you can write the following statements:

```
/* find vehicles that get at least 33 mpg in the city */
varNames = {"Make" "Model" "Mpg_City" "Engine_Cylinders"};
use Sasuser.Vehicles;          /* read data */
read all var varNames;
close Sasuser.Vehicles;

idx = loc(Mpg_City >= 33);
print (Make[idx])
      (Model[idx])
      (Mpg_City[idx]) [label="Mpg_City"];
```

Figure 3.6 Vehicles with Mpg_City ≥ 33

		Mpg_City
Toyota	Yaris	34
Toyota	Yaris	34
Honda	Civic Hybrid	49
Nissan	Altima Hybrid	42
Toyota	Camry Hybrid	40
Toyota	Prius Hybrid	60
Honda	Fit	33
Ford	Escape Hybrid FWD	36

Continuing the previous program, if you want to find vehicles that have six cylinders and that are fuel efficient, you use the AND (&) operator in conjunction with the LOC function, as shown in the following statements:

```
/* find vehicles that have six cylinders and get > 30 mpg in the city */
idx = loc(Engine_Cylinders=6 & Mpg_City>30);
numSatisfy = ncol(idx);
print numSatisfy "vehicles satisfy the criteria.";
print (Make[idx]) (Model[idx]) (Mpg_City[idx]) [label="Mpg_City"];
```

Figure 3.7 Vehicles That Satisfy Multiple Criteria

numSatisfy		
		4 vehicles satisfy the criteria.
		Mpg_City
Lexus	RX 400h 2WD	32
Toyota	Highlander Hybrid 2WD	32
Lexus	RX 400h 4WD	31
Toyota	Highlander Hybrid 4WD	31

Notice that you can use the NCOL function to count observations that satisfy the criteria.

Programming Tip: Use the LOC function to identify observations that satisfy some criteria. Use the NCOL function to count the observations that satisfy the criteria.

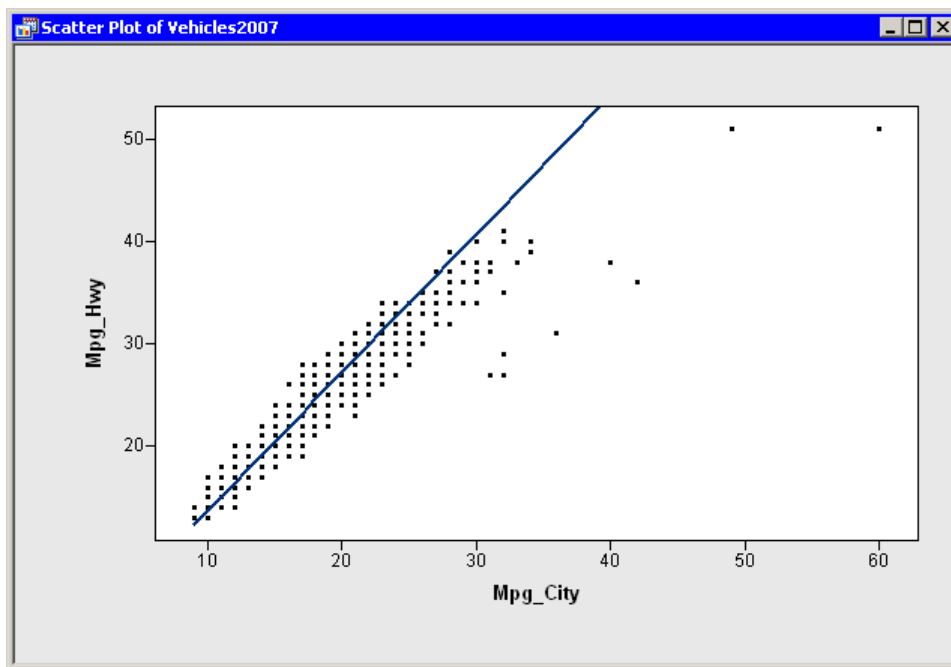
You can use the LOC function to do the following:

- find extreme values in univariate data
- find observations or variables that contain missing values
- identify observations that are more than a certain distance from the mean or median

- detect outliers in a regression model by finding observations whose absolute residuals are large
- detect observations that have a high influence on the regression model by finding observations with a large Cook's D , leverage, or PRESS value

As an example, the following statements demonstrate how to use the LOC function to detect outliers in a regression model. [Figure 3.8](#) shows a scatter plot of the Mpg_Hwy versus Mpg_City, along with a robust regression curve. (Chapter 4, “[Calling SAS Procedures](#),” shows how you can call a SAS procedure such as ROBUSTREG to compute the predicted and residual values for a model. Chapter 7, “[Creating Statistical Graphs](#),” shows how to create the graph in SAS/IML Studio.)

Figure 3.8 Scatter Plot with Robust Regression Line



For now, assume that the parameters for the regression line are estimated by some robust regression algorithm. The estimates are approximately given by $b = (0.24, 1.35)$. The following statements use the LOC statement to extract and print information about the vehicles that have large residuals:

```
/* find vehicles with large residuals for a linear model */
varNames = {"Make" "Model" "Mpg_City" "Mpg_Hwy" "Hybrid"};
use Sasuser.Vehicles;                /* read data */
read all var varNames;
close Sasuser.Vehicles;

Pred_Mpg_Hwy = 0.24 + 1.35*Mpg_City; /* create predicted values */
residual = Mpg_Hwy - Pred_Mpg_Hwy;   /* create residual values */
idx = loc(abs(residual) > 10);
print (Make[idx])
      (Model[idx])
      (residual[idx]) [label="Residual"]
      (Hybrid[idx]) [label="Hybrid?"];
```

By now, the first few statements in the program are familiar: define the variables of interest and use the USE and READ statements to create vectors that contain the values for the given variables. The subsequent statement creates predicted values based on some regression model. The predicted values are used to form the residuals. The LOC function finds observations (shown in Figure 3.9) whose observed value for Mpg_Hwy is more than 10 MPG different than the predicted value.

Figure 3.9 Vehicles with Large Residuals

		Residual	Hybrid?
Honda	Civic Hybrid	-15.39	1
Nissan	Altima Hybrid	-20.94	1
Toyota	Camry Hybrid	-16.24	1
Toyota	Prius Hybrid	-30.24	1
Ford	Escape Hybrid FWD	-17.84	1
Lexus	RX 400h 2WD	-16.44	1
Toyota	Highlander Hybrid 2WD	-16.44	1
Ford	Escape Hybrid 4WD	-14.44	1
Lexus	RX 400h 4WD	-15.09	1
Mercury	Mariner Hybrid 4WD	-14.44	1
Toyota	Highlander Hybrid 4WD	-15.09	1

Notice that the value of the Hybrid indicator variable is 1 for all of these observations. This indicates that the hybrid vehicles do not satisfy the same linear relationship as the other vehicles. A revised model might incorporate the Hybrid variable as a classification variable in the model.

There is a potential problem with each of the previous examples: what happens if there are no observations that satisfy the specified criterion? In that case, the LOC function returns an empty matrix that has zero rows and zero columns. It is invalid to use an empty matrix as an index. In other words, it is an error to form an expression such as `residual[idx]` when `idx` is empty.

You can determine whether a matrix is empty by using one of three functions: TYPE, NROW, or NCOL. (These functions are described in Section 2.2.) Which function you use is a matter of personal preference. This book uses the NCOL function. Consequently, the following statements show how to handle the case in which there are no sufficiently large residuals:

```
/* handle the situation when no observations satisfy a criterion */
Pred_Mpg_City = 0.24 + 1.35*Mpg_Hwy;
residual = Mpg_City - Pred_Mpg_City;
idx = loc(abs(residual) > 10);
if ncol(idx) > 0 then
    print (Make[idx]) (Model[idx]) (residual[idx]) [label="Residual"]
        (Hybrid[idx]) [label="Hybrid?"];
else
    print "No observations satisfy the criterion.";
```

Programming Tip: Never use the result of the LOC function without knowing that the result is a nonempty matrix. You can use the TYPE, NROW, or NCOL functions to check whether a matrix is empty.

3.3.3 Assigning Values to Observations That Satisfy a Criterion

The previous section indicates that hybrid-electric vehicles are different from traditional vehicles in terms of the relationship between MPG_Hwy and MPG_City. Suppose you compute two linear models—one for hybrid vehicles and the other for traditional vehicles—and you want to assign a vector of predicted values for every observation in the Vehicles data set. You can use the LOC function in conjunction with an IF-THEN statement to conditionally assign the predicted values, as shown in the following statements:

```
/* assign predicted values based on whether vehicle is hybrid-electric */
varNames = {"Mpg_City" "Mpg_Hwy" "Hybrid"};
use Sasuser.Vehicles;      /* read data */
read all var varNames;
close Sasuser.Vehicles;

/* first approach: use the LOC function */
Pred = j(nrow(Hybrid), 1); /* allocate vector for predicted values */
idx = loc(Hybrid=1);      /* indices that satisfy a criterion */
if ncol(idx)>0 then        /* assign an expression to those indices */
    Pred[idx] = 6.91 + 0.7*MPG_City[idx];
idx = loc(Hybrid^=1);      /* the other indices */
if ncol(idx)>0 then        /* assign a different expression */
    Pred[idx] = 0.09 + 1.36*MPG_City[idx];
```

The predicted values for the hybrid-electric vehicles are based on one model; the predicted values for traditional vehicles are based on another model. The LOC function finds the indices for which each model applies. (A less efficient alternative to the LOC function would be to loop over all observations and to use an IF-THEN/ELSE statement inside the loop to check whether each vehicle is hybrid-electric. You should avoid looping over observations whenever possible.)

There is, in fact, another efficient way to conditionally assign values to a vector. The CHOOSE function is a function that takes three arguments. The first argument is a logical expression. The second argument contains values that are assigned to all observations that satisfy the logical expression; the third argument contains values that are assigned to all observations that do not satisfy the logical expression. The following statement is equivalent to several statements in the previous program:

```
/* second approach: use the CHOOSE function */
Pred = choose(Hybrid=1,      /* if the i_th vehicle is hybrid */
              6.91+0.7*MPG_City, /* assign this value, */
              0.09+1.36*MPG_City); /* otherwise, assign this value */
```

Programming Tip: Use the CHOOSE function to conditionally assign values to elements, based on some criterion.

3.3.4 Handling Missing Values

It is common to have missing values in data. For example, in the Sasuser.Movies data, there are 25 missing values for the World_Gross variable. Similarly, there are missing values for the Distributor variable for those movies that are “independent films.” Missing values can occur for a variety of

reasons such as nonresponse in a survey or because no measurement was taken. Missing values can occur for mathematical or statistical reasons. For example, the logarithm of a nonpositive value or the skewness of constant data are both often represented by a missing value.

There are two common techniques for dealing with missing values during data analysis: excluding observations with any missing variable values from the analysis or imputing values to replace the missing value. Most SAS procedures (for example, UNIVARIATE, REG, and PRINCOMP) exclude the entire observation if the observation has a missing value in any relevant variable. The MI and MIANALYZE procedures in SAS/STAT use multiple imputation to handle missing values. This book does not discuss multiple imputation methods.

Many SAS/IML built-in subroutines and matrix operators handle missing values by omitting them from a computation. For example, you can compute the sum of the elements in a matrix by using the SUM function or by using the summation subscript reduction operator (+), which is described in “Use Subscript Reduction Operators” on page 80. If the matrix contains missing values, those missing values are excluded from the arithmetic operations. Similarly, the Median module (which is distributed with SAS/IML as part of the IMLMLIB module library) excludes missing values when computing the median of each column of a matrix.

However, some SAS/IML subroutines and operators cannot check for missing values without becoming inordinately inefficient. For example, matrix and vector multiplications stop with an error if there are missing values in either factor. Other SAS/IML statistical and mathematical functions similarly report errors if they are called with arguments that contain missing values. Consequently, if you want to write robust, reusable programs, you should write programs that explicitly handle missing values in your data.

For example, suppose you wanted to compute the mean of the worldwide revenue for the movies in the Movies data set. The following statements look correct upon a first glance:

```
/* incorrect computation for data that contain missing values */
use Sasuser.Movies;                /* read data */
read all var {"World_Gross"};
close Sasuser.Movies;

/* mean is wrong because of missing values */
wrong_mean = sum(World_Gross) / nrow(World_Gross);
print wrong_mean;
```

Figure 3.10 An Incorrect Computation

wrong_mean
116.76681

The value for the mean (116.77) is wrong. Missing values are the reason. The SUM function skips missing values as it adds up the elements in the **World_Gross** vector, but the NROW function counts *all* of the rows, even those with missing values.

There are two ways to get the correct answer: use functions that account for the missing values, or delete the missing values from the data. There are several functions that account for missing values, as shown in the following statements:


```

/* use functions that account for missing values */
mean1 = World_Gross[:];
mean2 = mean(World_Gross);
mean3 = sum(World_Gross)/countn(World_Gross);
print mean1 mean2 mean3;

```

Figure 3.11 Result of Calling Functions That Handle Missing Values

mean1	mean2	mean3
125.50684	125.50684	125.50684

The subscript reduction operator (:) is described in [Section 3.5](#). The MEAN function is a function that computes the mean of each column in a matrix. The COUNTN function returns the number of nonmissing values in each column of a matrix.

The alternative way to handle missing values is to delete elements in a vector (or rows in a matrix) that are missing. This is an easy application of using the LOC function. The following statements exclude observations with missing values:

```

/* exclude observations with missing values */
nonMissing = loc(World_Gross ^= .);
if ncol(nonMissing)=0 then mean = .;
else do;
    World_Gross = World_Gross[nonMissing,];
    mean4 = sum(World_Gross) / nrow(World_Gross);
end;
print mean4;

```

Figure 3.12 Result of Deleting Missing Values

mean4
125.50684

This time the value for the mean (125.51) is correct because only the nonmissing values are used in the calculation. [Section 3.4](#) extends this example to create a module that excludes all rows of a matrix that contain a missing value.

You can use this same technique to exclude observations that satisfy any criterion—including a criterion that has nothing to do with missing values. To exclude observations that satisfy a criterion, first locate all observations that do *not* satisfy the criterion, and then extract those observations from the vector. In this way, you can exclude all of the R-rated movies from the movies data, all of the films that had a large budget, and so forth.

Programming Tip: To exclude observations that satisfy a certain condition, use the LOC function to find and extract the observations that do *not* satisfy the condition.

The observant reader who is familiar with DATA step programming might ask “Can’t you exclude those observations when you read the data by using a WHERE clause?” Yes, you can. You can specify the WHERE clause on the READ statement, as the following statement indicates:

```
read all var {"World_Gross"} where (World_Gross ^= .);
```

However, this does not eliminate the need for the technique of this section. For example, suppose that you want to read several variables (say X and Y) in a single READ statement. Do you want to exclude observations from some X variable because a value is missing in the Y variable? The answer is often “yes” for a multivariate analysis, but is often “no” for univariate analyses. One way to handle both situations is to read in all of the data but to exclude missing values in a way that makes sense for a particular analysis.

A second consideration is that much of SAS/IML programming is done inside of a module (modules are introduced in the section “[Defining SAS/IML Modules](#)”). A module typically receives a matrix as an argument, and has no control over whether or not it contains missing values. Consequently, the module has to handle missing values.

A third consideration is that a matrix can be created from many sources: SAS data sets on a server, data residing in a DataObject, data from a Microsoft Excel worksheet, an R data frame, or values resulting from a SAS/IML computation. (These situations are discussed in Chapter 6, “[Understanding IMLPlus Classes](#).”) In general, an IMLPlus programmer cannot control whether matrices contain missing values, so it is best to write programs that handle them.

Programming Tip: Write programs that handle missing values in the data.

3.3.5 Analyzing Observations by Categories

The LOC function is often used in concert with the UNIQUE function when you are analyzing categorical variables. The UNIQUE function returns a vector that contains a sorted list of unique values. The LOC and UNIQUE functions both return their results as row vectors. Consequently, you can use the NCOL function to count how many unique values there are, and also use the NCOL function to count how many observations are associated with each unique value.

For example, the simplest usage of UNIQUE and LOC is to carry out a one-way frequency analysis of a variable. The following statements count the number of different categories in the MPAARating variable in the movies data set:

```
/* count the number of observations in each category */
use Sasuser.Movies;
read all var {"MPAARating"};          /* 1 */
close Sasuser.Movies;

categories = unique(MPAARating);       /* 2 */
count = j(ncol(categories), 1, 0);     /* 3 */
do i = 1 to ncol(categories);
    idx = loc(MPAARating = categories[i]); /* 4 */
    count[i] = ncol(idx);               /* 5 */
end;
print count[rowname=categories];
```

Figure 3.13 A One-Way Frequency Analysis

	count
G	13
NR	3
PG	66
PG-13	149
R	128

The previous program contains five main steps:

1. Read the MPAARating variable. This creates a vector of the same name.
2. Use the UNIQUE function to get a row vector that contains unique categories of the MPAARating variable. The type (numeric or character) of the **categories** vector is the same as the type of the MPAARating variable.
3. Create a constant vector, **count**, to hold the results. The size of the **count** vector is the number of unique categories in MPAARating. The **count** vector is initialized to the zero vector, but **count[i]** is overwritten by the actual number of observations belonging to the *i*th category.
4. Inside a loop over all categories, use the LOC function to find the observations belonging to the *i*th category. Notice that there is at least one observation in each category because the loop is over the categories found by the UNIQUE function. Therefore, you do not need to check whether **idx** is empty.
5. The number of observations in the *i*th category are counted by using the NCOL function. For this computation, the actual observations are not needed, only the number of observations.

This algorithm is an *extremely* useful programming technique. You can also use this technique to loop over levels of a categorical variable and to conduct an analysis for the observations in each level.

Programming Technique: If **c** is a categorical variable, then the following statements analyze all observations in each category of **c**:

```
uC = unique(c);
do i = 1 to nrow(uC);
    idx = loc(c = uC[i]);
    /* idx contains the observations in the i_th category */
    /* Analyze those observations here. */
end;
```

Programming Tip: If you are computing a quantity in a loop, use the J function prior to the loop to allocate a vector to hold the results.

You can use the UNIQUE-LOC technique twice (by using two nested loops) in order to analyze observations that belong two categories. For example the following statements create a two-way frequency table for the Year and MPAARating variables of the Movies data:

```
/* count the number of observations in each pair of categories */
use Sasuser.Movies;                               /* read data */
read all var {"Year" "MPAARating"};
close Sasuser.Movies;

/* create two-way frequency table */
uYear = unique(Year);
uMPAARating = unique(MPAARating);
Table = j(ncol(uYear), ncol(uMPAARating), 0);      /* 1 */
do j = 1 to ncol(uMPAARating);                     /* 2 */
  jdx = loc(MPAARating = uMPAARating[j]);          /* 3 */
  t = Year[jdx];                                    /* 4 */
  do i = 1 to ncol(uYear);
    idx = loc(t=uYear[i]);                          /* 5 */
    Table[i,j] = ncol(idx);
  end;
end;
YearLabel = char(uYear,4);                          /* 6 */
print Table[rowname=YearLabel colname=uMPAARating];
```

Figure 3.14 A Two-Way Frequency Analysis of MPAA Rating by Year

	Table				
	G	NR	PG	PG-13	R
2005	6	0	26	63	42
2006	5	3	23	57	48
2007	2	0	17	29	38

The example follows the same general principles as the one-variable example:

1. Create a matrix to hold the results before entering a loop.
2. Loop over categories, not over observations.
3. Use the LOC function to find observations in the j th category of the MPAARating variable.

In addition, the example adds new ideas:

4. Before entering a second loop, restrict your attention to only the observations that satisfy the first condition (that is, movies with a particular MPAA rating). Notice that you do not have to check whether **jdx** is an empty matrix because the value **uMPAARating[j]** is one of the values returned by the UNIQUE function.
5. Find movies of a given rating that also belong to the i th year. The element **Table[i, j]** contains the count of the number of movies in the joint level of Year and MPAARating. This

count might be zero. Therefore, it is important to check that `idx` is nonempty if you use it as an index into the data.

6. The `ROWNAME=` and `COLNAME=` options to the `PRINT` statement require character vectors. You can convert a numerical matrix to a character matrix by using the `CHAR` function in SAS/IML. (The Base `PUTN` function is also useful if you need to apply a format to the numerical values.)

The purpose of this example is not to form a two-way table, since the `FREQ` procedure already does this quite well. Rather, the purpose is to show a framework in which you can compute statistics on observations that belong to a joint level of two categorical variables. The matrix `idx` contains the observation numbers in the j th category of one variable and the i th category of the other variable.

This same technique enables you to loop over joint levels of two categorical variables and to conduct an analysis for the observations in each joint level.

As a closing comment, notice that the `LOC` function finds the observations in each category without sorting the data. Would this process be easier if you sort the data? Sometimes. If the goal of the analysis is to conduct BY group processing, then calling the `SORT` procedure followed by a `DATA` step or SAS/STAT procedure might be simpler and more efficient than programming the analysis in PROC IML. For example, if the statements that produce [Figure 3.13](#) were changed to compute, say, the quartiles of `US_Gross` for each `MPAARating` category, you could compute the same quantities by using the following SAS procedures:

```
/* sort the data by MPAARating */
proc sort data=Sasuser.Movies out=movies;
    by MPAARating;
run;

/* compute quartiles for each BY group */
proc means data=movies noprint;
    by MPAARating;
    var US_Gross;
    output out=OutStat Q1=Gross_Q1 median=Gross_Q2 Q3=Gross_Q3;
run;

proc print data=OutStat noobs;
    var MPAARating Gross_Q1 Gross_Q2 Gross_Q3;
run;
```

Figure 3.15 Using Base SAS for BY Group Analysis

MPAARating	Gross_Q1	Gross_Q2	Gross_Q3
G	\$24.4	\$56.1	\$83.0
NR	\$2.0	\$3.1	\$18.8
PG	\$17.0	\$48.5	\$82.2
PG-13	\$16.3	\$42.1	\$80.3
R	\$8.2	\$22.6	\$49.8

However, in general, the SAS/IML technique has value for several reasons:

1. You might be writing the analysis in a SAS/IML module (see [Section 3.4](#)), or the data may not exist in any SAS data set. In these instances, it might be more efficient to compute a desired analysis completely in the SAS/IML language than to write the data to a data set, call a SAS procedure, and then read the results into SAS/IML matrices.
2. The goal of the technique might not be to compute a statistic. In an IMLPlus program, you might want to color observations according to their category, or exclude observations from an analysis if some criterion is true.
3. You can use the technique to compute statistics that are not available in any SAS procedure. For example, you can use the technique to implement a proprietary algorithm or to compute a new statistic that recently appeared in a journal article.

3.4 Defining SAS/IML Modules

A powerful programming feature in the SAS/IML language is the ability to define your own module. A module is a user-defined function or subroutine that can be called from programs. A module can return a value or not. A module that returns a value is called a function module. You can use a function module as part of an expression. A module that does not return a value is called a subroutine module. You can call a subroutine module by using the CALL or RUN statement.

A module enables you to reuse and maintain related SAS/IML statements in a convenient way. You can pass matrices from your program into the module to provide input data and to control the way the module behaves.

3.4.1 Function and Subroutine Modules

You define a module by using the START and FINISH statements. The START statement defines the name of the module and the arguments.

There are two kinds of modules: functions and subroutines. You can use the RETURN statement to return a value from a function module. A subroutine module usually modifies one or more matrices that are passed into the module.

For example, the following statements define a function module that returns the Pearson correlation coefficient between two column vectors:

```
/* module to compute Pearson correlation between two column vectors */
start CorrCoef(x, y);                               /* assume no missing values: */
  xStd = standard(x);                               /* standardize data          */
  yStd = standard(y);
  df = nrow(x) - 1;                                 /* degrees of freedom        */
  return ( xStd` * yStd / df );
finish;
```

```

u = {1,2,3,4,5};
v = {2,3,1,4,4};
r = CorrCoef(u,v);
print r;

```

Figure 3.16 Result from a User-Defined Function Module

<p style="text-align: center;">r</p> <p style="text-align: center;">0.6063391</p>
--

The module (named CorrCoef) is defined to accept two matrix arguments. Inside the module, these vectors are named **x** and **y**. The **x** and **y** vectors are standardized by calling the Standard module, which is included in the IMLMLIB module library. (The IMLMLIB library is described in “[The IMLMLIB Library of Modules](#)” on page 78.) The Standard module applies a linear transformation to the data so that it has zero mean and unit standard deviation: $x \rightarrow (x - \bar{x})/s_x$ where \bar{x} is the sample mean and s_x is the sample standard deviation. The CorrCoef module returns the inner product of the standardized vectors, divided by the degrees of freedom. (Recall that the transpose operator (`) transposes a matrix.)

In spite of the recommendations in the section “[Handling Missing Values](#)” on page 65, the CorrCoef module does not handle missing values in the **x** and **y** vectors.

Programming Tip: Use modules to encapsulate self-contained computational pieces, especially if you want to reuse them.

3.4.2 Local Variables

A module usually maintains its own set of variables. This means that names used for variables inside a module do not conflict with variables of the same name that exist outside the module. Consequently, the matrix **df** defined inside the CorrCoef module does not overwrite or conflict with any other variable of the same name that might exist in some other module or in the main program.

Programming Tip: Variables defined in a module are local to that module. Local variables do not conflict with variables in the main program that have the same name.

If you use the SHOW NAMES statement inside a module, it displays the names, dimensions, and types of all matrices that are local to the module.

3.4.3 Global Symbols

You can force a variable to be a global variable by using the optional GLOBAL clause on the START statement for the module. An example of a module that uses a GLOBAL clause is shown in the following statements:

```
/* use the GLOBAL clause in a module definition */
start HasValue(x) global(g_Value);
    idx = loc(x=g_Value);          /* find value, if it exists */
    return ( ncol(idx)>0 );         /* return 1 if value exists */
finish;

g_Value = 1;                      /* global value to search for */
v = {4,2,1,3,8};                  /* data to search */
z = HasValue(v);
```

A more realistic example of using the GLOBAL clause is described in [Section 11.8.3](#).

Programming Tip: If you use a global variable in a module, it is a good idea to prefix the variable name by `g_` (or some other mnemonic prefix) to help you remember that the variable is global.

In PROC IML, a module that has no arguments implicitly uses variables in the global scope of the program. This is discussed in [Section 5.7.2](#).

3.4.4 Passing Arguments by Reference

SAS/IML passes matrices to modules “by reference,” which means that if you change one of the arguments inside the module, the corresponding matrix also changes outside the module. This is demonstrated in the following example:

```
/* matrices are passed by reference to modules */
start ReverseRows(x);              /* define subroutine module */
    r = x[nrow(x):1, ];            /* reverse rows of argument */
    x = r;                         /* reassign the argument matrix */
finish;

u = {1,2,3,4,5};                  /* original values */
run ReverseRows(u);                /* values of u are changed */
print u;
```

Figure 3.17 Result of Passing a Matrix by Reference

u
5
4
3
2
1

The module `ReverseRows` reverses the position of the rows of a matrix. Inside the module, the argument to the module is called `x`. The module reverses the rows of `x` and overwrites `x` with new values. The module alters the matrix `u`.

Programming Tip: In a module, any changes made to the arguments also change the matrices passed into the module.

3.4.5 Evaluation of Arguments

The SAS/IML language resolves all matrix expressions prior to calling a subroutine. For example, consider the following statements:

```
/* pass temporary matrices to a function or subroutine */
w = {1 2, 2 3, 3 1, 4 4, 5 4};
r = CorrCoef(w[,1], w[,2]);
```

The statements call the `CorrCoef` module defined in [Section 3.4.1](#). The arguments to the module are matrix expressions that each extract a column from the `w` matrix. When the SAS/IML language processes the `CorrCoef` function call, the following occurs:

1. The expression `w[,1]` is evaluated. The results are placed in a temporary matrix called something like `_TEM0001`.
2. The expression `w[,2]` is evaluated. The results are placed in a temporary matrix called something like `_TEM0002`.
3. The `_TEM0001` and `_TEM0002` vectors are passed into the `CorrCoef` function as arguments to the module. The module computes the correlation coefficient for the centered data.
4. When the `CorrCoef` module returns, the return value of the module is copied into the `r` matrix. The two temporary vectors, `_TEM0001` and `_TEM0002`, are no longer needed, so they are deleted.

In general, SAS/IML allocates temporary matrices when necessary to hold intermediate expressions. These temporary matrices are deleted when they are no longer needed. In most cases, you do not need to worry about temporary matrices: SAS/IML software handles the creation and deletion without any intervention from you. However, if you use an expression as the argument to a module, SAS/IML creates a temporary matrix to hold the expression. If the module modifies that argument, then it is the temporary matrix that is modified. This is usually not what you intend.

For example, the following statements call the `ReverseRows` module defined in [Section 3.4.4](#):

```
/* be careful when you pass a temporary matrix to a subroutine */
w = {1 2, 2 3, 3 1, 4 4, 5 4};
run ReverseRows(w[,1]);      /* w[,1] creates a temporary matrix */
print w;                     /* w is unchanged */
```

Figure 3.18 An Unmodified Matrix

w	
1	2
2	3
3	1
4	4
5	4

In this example, the first column of the **w** matrix is passed as the input to the ReverseRows module. You might initially guess that the elements in the first column of **w** will be reversed by the module call. But this is not what happens! Instead, the following occurs:

1. The SAS/IML language copies the first column of **w** into a temporary vector (say, **_TEM0003**) and passes that vector into the ReverseRows module.
2. The module reverses the rows of **_TEM0003**.
3. When the subroutine returns, the temporary vector **_TEM0003** is deleted.

Because a copy of the column was passed in, the matrix **w** is unchanged.

Programming Tip: Use matrix expressions only for *input* parameters to a module.

3.4.6 Storing Modules

An important advantage of modules is that you can write and debug a module and then store it so that you can call it in future programs. You store a module by using the STORE statement. In SAS/IML you need to use the LOAD statement to load a stored module before you can call it. (In IMLPlus, modules are stored into Windows directories. SAS/IML Studio automatically searches certain directories in order to find the definition for an unresolved module. See the section “[IMLPlus Modules](#)” on page 114.)

As a final example of writing a module, the following statements define and store a module that returns the rows in a matrix that contain only nonmissing values. You can use this module to delete all rows that contain a missing value in any variable.

```
/* create a module to return the nonmissing rows of a matrix */
start LocNonMissingRows(x);
  c = cmiss(x);          /* matrix of 0 and 1's          */
  r = c[,+];             /* number of missing in row    */
  nonMissingIdx = loc(r=0); /* rows that do not contain missing */
  return ( nonMissingIdx );
finish;
store module=LocNonMissingRows;
```

```

z = {1 2, 3 ., 5 6, 7 8, . 10, 11 12};
nonMissing = LocNonMissingRows(z);
if ncol(nonMissing)>0 then
    z = z[nonMissing, ];
print z;

```

Figure 3.19 Nonmissing Rows of a Matrix

z	
1	2
5	6
7	8
11	12

The `LocNonMissingRows` module calls the `CMISS` function in Base SAS software to create a matrix, `c`, of zeros and ones. The matrix returned by the `CMISS` function has the value 1 at locations for which the corresponding element of `x` is missing, and has the value 0 at all other locations. The column vector `r` is the sum across each row of all of the elements of `c`. The vector `r` is computed by using the summation subscript reduction operator (+), which is described in [Section 3.5.2](#). The module uses the `LOC` function to find all rows that contain no missing values. These row numbers are returned by the function. In the example, the module returns the vector `{1 3 4 6}`. The example then extracts these rows from the matrix `z`, which results in a matrix that does not contain any missing values.

If the argument to the module has a missing value in every row, then the module returns an empty matrix. Consequently, a careful programmer will check that the matrix `nonMissing` is nonempty prior to using it. If the module is sent a matrix, `z`, that does not contain missing values, the module returns a vector identical to `1:nrow(z)`.

In PROC IML, the `STORE` statement stores the `LocNonMissingRows` module in the default “storage library,” which is a SAS catalog that is used to store modules and matrices. The default storage library is `Work.IMLStore`. Notice that this catalog is in the `Work` library and recall that the `Work` library vanishes when you exit the SAS System. Consequently, if you want a stored module to persist between SAS sessions, you need to save the module to a permanent library.

You can set the storage location for a module in PROC IML by using the `RESET STORAGE` statement. For example, the following statements store the `LocNonMissingRows` module in a catalog in the `Sasuser` library:

```

reset storage=Sasuser.MyModules;          /* set location for storage */
store module=LocNonMissingRows;

```

By using the `RESET STORAGE` statement prior to any `STORE` statement, you can ensure that you are storing modules to a permanent storage location. Similarly, you can use the `RESET STORAGE` statement prior to a `LOAD` statement to load a module definition from a permanent storage location.

Programming Tip: In PROC IML, you can store modules to a permanent storage location by using the `RESET STORAGE` statement.

In SAS/IML Studio, the STORE and LOAD statements work differently. For details, see the section “[IMLPlus Modules](#)” on page 114.

In SAS/IML 9.22, you can use the COUNTMISS function to write the LocNonMissingRows module more efficiently:

```
/* SAS/IML 9.22: a module to return the nonmissing rows of a matrix */
start LocNonMissingRows(x);
  r = countmiss(x, "row");      /* number of missing in row      */
  nonMissingIdx = loc(r=0);     /* rows that do not contain missing */
  return ( nonMissingIdx );
finish;
```

This module is more efficient because it uses less memory. The first version uses the CMISS function to create a matrix that is the same size as the input matrix **x**. The second version of the function does not create that (potentially large) matrix.

3.4.7 The IMLMLIB Library of Modules

SAS/IML software is distributed with a set of predefined modules called the IMLMLIB modules, because the modules are stored in a SAS catalog called IMLMLIB. You can call these modules without having to LOAD them. [Table 3.1](#) describes some of the modules that are useful in data analysis.

Table 3.1 Frequently Used Modules in IMLMLIB

Statement	Description
ColVec	Reshapes a matrix into a column vector.
Corr	Computes a correlation matrix from a data matrix. This module was replaced by the more general CORR function in SAS/IML 9.22.
Median	Computes the median value for each column of a matrix.
Quartile	Computes the minimum, lower quartile, median, upper quartile, and maximum values for each column of a matrix.
RandNormal, ...	The RandNormal module generates a random sample from a multivariate normal distribution. There are also modules for generating a random sample from other multivariate distributions.
RowVec	Reshapes a matrix into a row vector.
Standard	Standardizes each column of a matrix.

3.5 Writing Efficient SAS/IML Programs

What does it mean to write efficient SAS/IML programs? The simplest rule to follow is “whenever possible, avoid writing loops.” SAS/IML provides many high-level operators and built-in subroutines for working with matrices and vectors, rather than working with each element of an array.

The previous section introduced the LOC and UNIQUE functions. A first step to writing efficient programs is to use those functions whenever you can.

3.5.1 Avoid Loops to Improve Performance

Take another look at the algorithm on page 68 that counts the number of observations that belong to each category. Notice that the algorithm does not loop over the observations. There is a loop over the categories, but no loop over observations. This is in sharp contrast to the following inefficient statements, which compute the same quantities as the statements that produce Figure 3.13.

```
/* inefficient algorithm which loops over observations */
use Sasuser.Movies;
read all var {"MPAARating"};          /* 1 */
close Sasuser.Movies;

/* inefficient: do not imitate */
categories = MPAARating[1];          /* 2 */
count = 1;
do obs = 2 to nrow(MPAARating);      /* 3 */
  i = loc(categories = MPAARating[obs]); /* 4 */
  if ncol(i) = 0 then do;             /* 5 */
    categories = categories // MPAARating[obs];
    count = count // 1;
  end;
  else count[i] = count[i] + 1;        /* 6 */
end;
print count[rowname=categories];
```

Figure 3.20 Result of Inefficient Statements

count	
PG-13	149
PG	66
R	128
G	13
NR	3

The main steps of this program are as follows:

1. Read the `MPAARating` variable.
2. Initialize the matrix `categories` with the first category. When the program encounters a category that has not previously been observed, the `categories` vector will grow.
3. Loop over all observations.
4. Inside the loop, compare the current observation with the categories that have been seen previously. (Some beginning programmers replace this statement with a second loop over the number of categories, leading to even worse performance!)
5. If `i` is empty, then `MPAARating[obs]` is a new category. Use vertical concatenation to add a new row to the vector that keeps track of categories and to the vector that keeps track of the count for each category.
6. If `i` is not empty, then increment `count[i]`, the number of observations encountered for this category.

While the two algorithms compute the same quantity, the statements that produce [Figure 3.13](#) are more efficient for the following reasons:

- The efficient algorithm loops over the number of unique categories. This is typically a small number. The inefficient algorithm loops over the number of observations, which is typically a much larger number.
- The efficient algorithm determines the size of all vectors before beginning the loop and preallocates any storage it needs. The inefficient algorithm uses a vertical concatenation operator (`//`) to dynamically enlarge the result vectors every time a new category is encountered. This inefficient technique is discussed in the section “[Concatenation Operators](#)” on page 41.

Programming Tip: If a program contains a loop over all observations, you can often make the program more efficient by eliminating the loop and using the `LOC` function instead.

Not only is the program that produces [Figure 3.13](#) more efficient than the program that produces [Figure 3.20](#), but also the results of the former program are computed in a standard form. The categories of `MPAARating` are produced in alphabetical order, and therefore the results are independent of the way that the `Movies` data set is sorted. In contrast, the second algorithm computes the results in the order that the categories appear in the data set.

3.5.2 Use Subscript Reduction Operators

One way to avoid writing unnecessary loops is to take full advantage of the subscript reduction operators for matrices. These operators enable you to perform common statistical operations (such as sums, means, and sums of squares) on either the rows or the columns of a matrix. For example, the following statements compute the sum and mean of columns and of rows for a matrix; the results are shown in [Figure 3.21](#):

```

/* compute sum and mean of each column */
x = {1 2 3,
      4 5 6,
      7 8 9,
      4 3 .};
colSums = x[+, ];
colMeans = x[:, ];
rowSums = x[ ,+];
rowMeans = x[ ,:];
print colSums, colMeans, rowSums rowMeans;

```

Figure 3.21 Sums and Means of Rows and Columns

colSums		
16	18	18
colMeans		
4	4.5	6
rowSums	rowMeans	
6	2	
15	5	
24	8	
7	3.5	

The expression `x[+,]` uses the '+' subscript operator to “reduce” the matrix by summing the elements of each row for all columns. (Recall that not specifying a column in the second subscript is equivalent to specifying all columns.) The expression `x[:,]` uses the ':' subscript operator to compute the mean for each column. The row sums and means are computed similarly. Note that the subscript reduction operators correctly handle the missing value in the third column.

A common use of subscript reduction operators is to compute the marginal frequencies in a two-way frequency table. For example, the following statements compute the row total, column total, and grand total for a frequency table of two variables in the Movies data:

```

/* compute column sums, row sums, and grand sum for a matrix */
uYear = 2005:2007;
uMPAARating = {"G"      "NR"      "PG"      "PG-13"    "R"};
Table = {
           6      0      26      63      42,
           5      3      23      57      48,
           2      0      17      29      38};
colSums = Table[+, ];
rowSums = Table[ ,+];
Total = Table[+];
YearLabel = char(uYear, 7);
print Table[rowname=YearLabel colname=uMPAARating] rowSums,
      colSums[rowname="colSums" label=""] Total[label=""];

```

Figure 3.22 Marginal Frequencies for a Matrix

Table	G	NR	PG	PG-13	R	rowSums
2005	6	0	26	63	42	137
2006	5	3	23	57	48	136
2007	2	0	17	29	38	86
colSums	13	3	66	149	128	359

The section “[Analyzing Observations by Categories](#)” on page 68 describes how to compute the cell counts for this table from the Movies data set. The new feature in this example is the use of the '+' subscript operator to compute three different marginal sums.

The following table summarizes the subscript reduction operators for matrices and an equivalent way to perform the operation that uses function calls.

Table 3.2 Subscript Reduction Operators for Matrices

Operator	Action	Equivalent Function
+	Addition	sum(x)
#	Multiplication	prod(x) /* SAS/IML 9.22 */
><	Minimum	min(x)
<>	Maximum	max(x)
>:<	Index of minimum	loc(x=min(x))[1]
<:>	Index of maximum	loc(x=max(x))[1]
:	Mean	mean(x) /* SAS/IML 9.22 */
##	Sum of squares	ssq(x)

Beginning in SAS/IML 9.22, the SAS/IML language supports a MEAN function and a PROD function. The MEAN function computes the mean of each column in a matrix, so it is equivalent to $\mathbf{x}[:,]$. The PROD function returns the product of all nonmissing elements of its arguments.

3.5.3 Case Study: Standardizing the Columns of a Matrix

The CorrCoef module in the section “[Defining SAS/IML Modules](#)” on page 72 calls the standard module in the IMLMLIB module library. This section describes how you can write statements that reproduce the results of that module.

The following statements standardize each column in a numeric matrix to have mean zero and unit standard deviation:

```
/* standardize data: assume no missing values and no constant column */
x = {7 7, 8 9, 7 9, 5 7, 8 8};
xc = x - x[1,:];          /* 1. center the data */
ss = xc[##,];             /* 2. sum of squares for columns of xc */
n = nrow(x);              /* 3. assume no missing values */
std = sqrt(ss/(n-1));      /* 4. sample standard deviation */
std_x = xc / std;         /* 5. divide each column (standardize) */
print std_x;
```

The output from the program is shown in [Figure 3.23](#). Each statement is explained in the following list:

1. The row vector `x[1,:]` contains the means of each column of `x`. Consequently, the matrix `xc` contains the centered data: the mean of each column is subtracted from the data in that column.
2. The row vector `ss` contains the sum of squares for each centered column.
3. Assuming that there are no missing values, the scalar quantity `n` contains the number of nonmissing values in each column.
4. The row vector `std` contains the standard deviation of each column. Each element of `std` is nonzero, provided that no column of `x` is constant (that is, contains the same value for all observations).
5. The matrix `std_x` contains the standardized data that results from dividing the matrix `xc` by the row vector `std`.

Figure 3.23 Standardized Columns of a Matrix

std_x	
0	-1
0.8164966	1
0	1
-1.632993	-1
0.8164966	0

You can relax the assumption that the data do not contain missing values: count the number of nonmissing observations in each column and compute the standard deviation for each column by using the number of nonmissing values for that column. This is left as an exercise for the reader.

Beginning in SAS/IML 9.22, the SAS/IML language supports a MEAN function that computes the mean of each column and a VAR function that computes the variance of each column. With the addition of these functions, the following statements standardize the data:

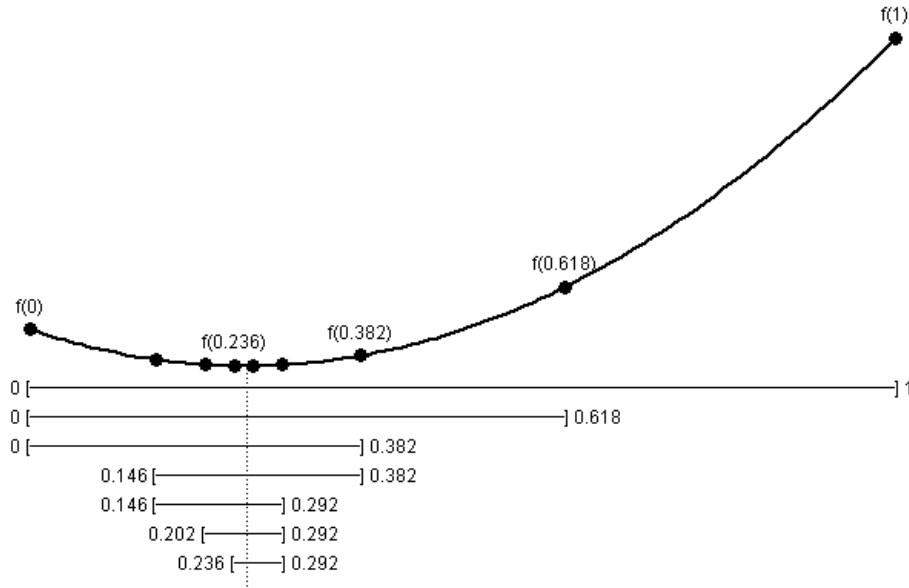
```
/* standardize data: SAS/IML 9.22 and beyond */
xc = x - mean(x);          /* center the data          */
std = sqrt(var(xc));       /* sample standard deviation */
std_x = xc / std;         /* divide each column (standardize) */
```

The MEAN and VAR functions both handle missing values. Consequently, the previous statements correctly standardize data that contain missing values.

3.6 Case Study: Finding the Minimum of a Function

This section shows how you can write a SAS/IML module to minimize a continuous function of one variable on a closed interval. To make the problem easier, assume that the function is *unimodal*, meaning that it has a single minimum on the interval.

There are many techniques for finding the minimum of a continuous function on a closed interval. The technique presented in this section is known as the *golden section search* (Thisted 1988). The basic idea is to bracket the minimum, meaning that you must find three points $a < x_1 < b$ such that the value $f(x_1)$ is less than the function evaluated at either endpoint. In the golden section search, the initial interval $[a, x_1]$ is smaller than the interval $[x_1, b]$. Therefore, a new point $x_2 \in [x_1, b]$ is chosen and the function is evaluated at x_2 . If $f(x_1) < f(x_2)$, then $a < x_1 < x_2$ is a new smaller interval that brackets the minimum. Otherwise, $x_1 < x_2 < b$ is a new smaller interval that brackets the minimum. In either case, the process repeats with the new smaller interval. This process is schematically indicated in [Figure 3.24](#).

Figure 3.24 A Schematic Illustration of the Golden Section Search

In the figure, $a = 0$, $b = 1$, $x_1 = 0.382$, $x_2 = 0.618$, and the function is minimized at the value indicated by the vertical dotted line. Because $f(x_1) < f(x_2)$, the algorithm chooses $[a, x_2]$ as the new bracketing interval. A new point is chosen in the larger of the intervals $[a, x_1]$ or $[x_1, x_2]$. For the example in the figure, the new point is $x_3 = 0.236$. Because $f(x_3) < f(x_1)$, the algorithm chooses $[a, x_1]$ as the new bracketing interval, and so forth.

The golden section algorithm guarantees that the length of the bracketing interval decreases at a constant rate, and that rate is related to the “golden ratio” that gives the algorithm its name. The following statements define a SAS/IML module that implements the golden section search in order to find the value of x that minimizes the function defined in the module Func:

```

/* define module that implements the golden section search */
start GoldenSection(a0,b0,eps);
/* Find the value that minimizes the continuous unimodal function
 * f (defined by the module "Func") on the interval [a,b] by using
 * a golden section search. The algorithm stops when b-a < eps.
 *
 * Example:
 * start Func(x);
 *   return ( x#(x-0.5) );
 * finish;
 * x0 = GoldenSection(0,1,1e-4);
 */

w = (sqrt(5)-1)/2;                               /* "golden ratio" - 1 */
a = a0; b = b0;
x1 = w*a + (1-w)*b;
x2 = (1-w)*a + w*b;
fx1 = Func(x1);
fx2 = Func(x2);

```

```

do while (b-a>eps);
  if fx1 < fx2 then do;                /* choose new right endpoint */
    b = x2;                          /* update x1 and x2          */
    x2 = x1;
    fx2 = fx1;
    x1 = w*a + (1-w)*b;
    fx1 = Func(x1);
  end;
  else do;                            /* choose new left endpoint */
    a = x1;                          /* update x1 and x2          */
    x1 = x2;
    fx1 = fx2;
    x2 = (1-w)*a + w*b;
    fx2 = Func(x2);
  end;
end;
return ( choose(fx1<=fx2, a, b) );
finish;

```

To test the module, the following statements define a module that returns values of the function $f(x) = x(x - 0.5)$. This function is minimized at $x = 0.25$. The iteration history is not printed, but is shown in Figure 3.24 for this function. (Incidentally, Figure 3.24 was created in SAS/IML Studio by using drawing commands that are discussed in Chapter 7, “Creating Statistical Graphs.”) The output is shown in Figure 3.25.

```

/* Minimize unimodal function by using GoldenSection module */
/* define unimodal function on [0,1]. Minimum at x=0.25.    */
start Func(x);
  return ( x*(x-0.5) );
finish;

/* search for minimum on an interval */
x0 = GoldenSection(0, 1, 1e-4);
y = Func(x0);
QuadSoln = x0 || y;
print QuadSoln[colname={"XMin", "f(XMin)"}];

```

Figure 3.25 Minimum Found by Golden Section Search

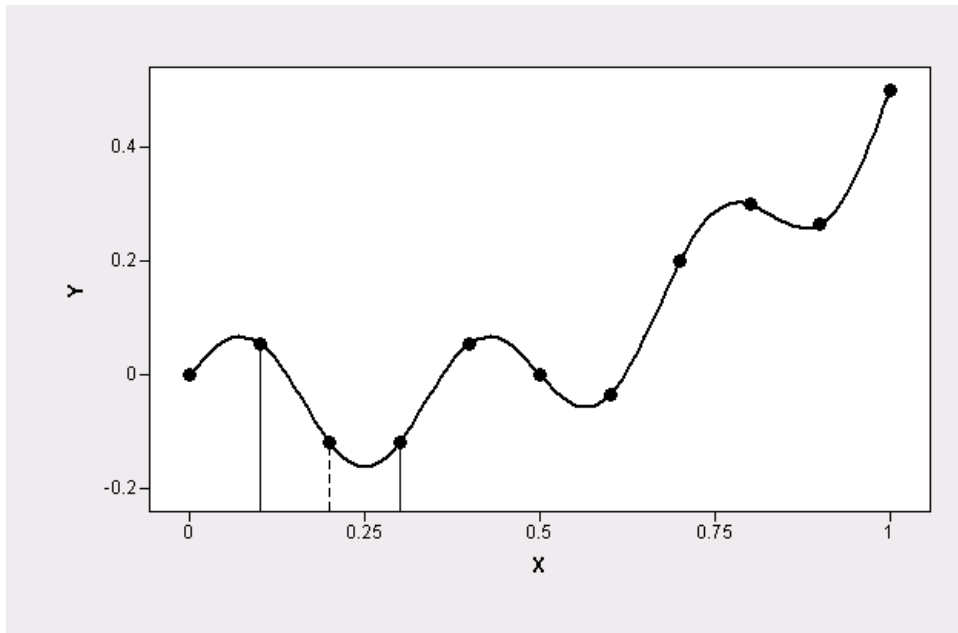
QuadSoln	
XMin	f(XMin)
0.2500228	-0.0625

Although the golden section search assumes that the function is unimodal, you can often use it in conjunction with other algorithms to refine intervals where a local minimum occurs. You might be interested in a function that has many local minima, but if you can find an interval that brackets a given minimum, you can use the golden section search to home in on the minimum value in that interval.

For example, Figure 3.26 shows the graph of the function $f(x) = x(x - 0.5) + 0.1 \sin(6\pi x)$ on the interval $[0, 1]$. The global minimum occurs at $x = 0.25$. If you evaluate the function on a

coarse grid, as shown in Figure 3.26, you can assume that the true minimum of the function occurs near the smallest value of the function on the grid. The evaluation of f on a grid is a “presearch,” which enables you to find a bracketing interval for the minimum. In Figure 3.26 the function f is evaluated at $0, 0.1, \dots, 1$. The smallest value of f on that set of points occurs at 0.2 , so you can use the bracketing interval $[0.1, 0.3]$ as an initial interval for the golden section search.

Figure 3.26 Locating an Interval That Brackets a Minimum



The following statements use this technique to find the minimum of the function f on the interval $[0, 1]$:

```
/* evaluate function on a grid, followed by golden section search */
/* This function has three local minima on [0,1]. Global min at 0.25. */
start Func(x);
    pi = constant('PI');                                /* 3.14159... */
    return ( x#(x-0.5) + 0.1 *sin(6*pi*x) );
finish;

/* use "presearch" evaluation on coarse grid on [0,1] */
x = do(0, 1, 0.1);
y = Func(x);
yMinIdx = y[>:<];
aIdx = max(yMinIdx-1, 1);
bIdx = min(yMinIdx+1, ncol(x));
a = x[aIdx];
b = x[bIdx];
print a b;

x0 = GoldenSection(a, b, 1e-4); /* 4. finds minimum on [a,b] */
y = Func(x0);
Soln = x0 || y;
print Soln[colname={"XMin", "f(XMin)"}];
```

Figure 3.27 Minimum Found by Presearch Followed by Golden Section Search

a	b
0.1	0.3
Soln	
XMin	f(XMin)
0.249974	-0.1625

The following list explains a few of the steps in the example:

1. Notice that the function module is written so that its argument, \mathbf{x} , can be a vector. The quadratic term in the function is implemented by using the elementwise multiplication operator, `#`. This enables you to call `Func` on every point in the grid with a single call to the module.
2. The example uses the “index of minimum” operator (`>:<`) to find the (first) index of \mathbf{y} with the smallest value. As shown in [Table 3.2](#), the expression `y[>:<]` is equivalent to the longer expression `y[loc(y=min(y))[1]]`.
3. Handle the case where the minimum value of the function occurs at an endpoint of the interval. In the example, the minimum value of the function on the grid occurs for `yMinIdx` equal to 3. The bracketing interval is therefore chosen to have endpoints `x[2]` and `x[4]`. But if `yMinIdx` were equal to 1, the endpoints of the bracketing interval would need to be `x[1]` and `x[2]`. The case where the minimum occurs at `x[11]` is handled similarly.
4. The minimum is found on the bracketing interval $[a, b]$. Note in [Figure 3.26](#) that the function is actually unimodal on this interval.

The example requires that the module is named `Func`, but in `IMLPlus` you can use the `ALIAS` statement to get rid of this restriction. For details, see “[Creating an Alias for a Module](#)” on page 117.

3.7 References

Thisted, R. A. (1988), *Elements of Statistical Computing: Numerical Computation*, London: Chapman & Hall.

Chapter 4

Calling SAS Procedures

Contents

4.1	Overview of Calling SAS Procedures	89
4.2	Calling a SAS Procedure from IMLPlus	90
4.3	Transferring Data between Matrices and Procedures	91
4.4	Passing Parameters to SAS Procedures	93
4.5	Case Study: Computing a Kernel Density Estimate	94
4.6	Creating Names for Output Variables	96
4.7	Creating Macro Variables from Matrices	99
4.8	Handling Errors When Calling a Procedure	101
4.9	Calling SAS Functions That Require Lists of Values	103

4.1 Overview of Calling SAS Procedures

The SAS/IML language is very powerful and enables you to compute many statistical quantities. However, the SAS System has hundreds of procedures that compute thousands of statistics. If you want to compute a statistic that is already produced by a SAS procedure, why not call that procedure directly?

For example, you might want to compute the sample skewness of univariate data, which is a statistic produced by the UNIVARIATE procedure. Or you might want to fit a linear model as implemented by the GLM procedure, or fit a kernel density estimate as computed by the KDE procedure. In these cases, it makes sense to take advantage of the power of SAS procedures and their output, rather than duplicate the computation with SAS/IML statements.

This chapter describes how to call a SAS procedure from an IMLPlus program. The SUBMIT and ENDSUBMIT statements are the only IMLPlus statements in this chapter. All other statements are standard SAS/IML statements. Beginning in SAS/IML 9.22, the SUBMIT and ENDSUBMIT statements are also available in PROC IML, although not all features of the IMLPlus implementation are available in PROC IML.

4.2 Calling a SAS Procedure from IMLPlus

When you run an IMLPlus program in SAS/IML Studio, the IMLPlus interpreter knows how to make sense of two kinds of statements: SAS/IML statements and IMLPlus statements. IMLPlus statements are extensions to the SAS/IML language that run in SAS/IML Studio. This section introduces two IMLPlus statements: the `SUBMIT` statement and the `ENDSUBMIT` statement.

SAS/IML Studio executes some IMLPlus statements (such as graphics commands) on the client PC, and sends the other statements to the SAS Workspace Server for execution. Statements sent to the SAS Workspace Server are assumed to be “pure” SAS/IML statements. If you want to execute a global statement (for example, an `ODS`, `OPTIONS`, or macro statement) or run a SAS procedure or `DATA` step, you need to precede the SAS statements with a `SUBMIT` statement and follow them with an `ENDSUBMIT` statement. The statements from the `SUBMIT` statement to the `ENDSUBMIT` statement are collectively referred to as a *SUBMIT block*.

Programming Tip: In IMLPlus, you must enclose a global statement (for example, an `ODS`, `OPTIONS`, or macro statement) in a `SUBMIT` block.

The following example shows how to call the `UNIVARIATE` procedure on data in a SAS libref from within an IMLPlus program. The program prints descriptive statistics such as the skewness and kurtosis of the `Budget` variable in the `Movies` data set:

```
/* call UNIVARIATE procedure from an IMLPlus program */
x = 1:10;                /* some matrices are defined before the call */
submit;                  /* IMLPlus statement: call a SAS procedure */
proc univariate data=Sasuser.Movies;
  var Budget;
  ods select Moments;
run;
endsubmit;               /* return from the SAS procedure */
y = sum(x);              /* the matrices are still defined! */
```

Figure 4.1 Output from the `UNIVARIATE` Procedure Called from an IMLPlus Program

The UNIVARIATE Procedure			
Variable: Budget (Budget (million USD))			
Moments			
N	359	Sum Weights	359
Mean	44.8273677	Sum Observations	16093.025
Std Deviation	41.9658513	Variance	1761.13267
Skewness	1.76633581	Kurtosis	3.49160391
Uncorrected SS	1351893.45	Corrected SS	630485.497
Coeff Variation	93.6165861	Std Error Mean	2.21487292

In the program, the matrix \mathbf{x} is defined prior to calling the UNIVARIATE procedure. After the UNIVARIATE procedure exits, the matrix is still available for further computations. This behavior is in contrast to the standard behavior in SAS programs where calling a new procedure (here, UNIVARIATE) causes the previous procedure (here, IML) to exit.

The SUBMIT block enables you to call any SAS procedure, DATA step, or macro as if it were a built-in SAS/IML subroutine. Any variables in your IMLPlus program that were in scope prior to calling the SUBMIT block are still in scope after calling the SUBMIT block.

Programming Tip: Use the SUBMIT and ENDSUBMIT statements to call a SAS procedure or DATA step from an IMLPlus program.

The simple example in this section merely prints some statistics about the `Budget` variable; there is no interaction between the `IMLPlus` portion of the program and the call to the `UNIVARIATE` procedure. The next sections show how to transfer data between SAS/IML matrices and SAS procedures, and how to pass parameters from SAS/IML software into SAS procedures.

4.3 Transferring Data between Matrices and Procedures

```

/* read results of procedure and compare with model */
use PE;
read all var {"Parameter" "Estimate"};
close PE;
print Parameter beta Estimate;

```

Figure 4.2 Parameters and Estimates for Linear Model

Parameter	beta	Estimate
Intercept	2	1.9576601
x	3	2.9926217

The program uses SAS/IML statements to generate 100 pairs of (x, y) values that are related by the model $y = 2 + 3x + e$, where $e \sim N(0, 1)$. The program writes those data to the MyData data set and calls the GLM procedure to estimate the coefficients of the model. The ODS OUTPUT statement creates the PE data set from the ParameterEstimates table that is produced by PROC GLM. The PE data set contains the least squares estimates for the model. These estimates are read into the **Estimate** vector. The program concludes by printing the parameters and the least squares estimates for the linear model.

Programming Tip: Use the CREATE and APPEND statements to transfer data from SAS/IML to SAS procedures. Use output data sets or ODS OUTPUT statements to transfer data from SAS procedures to SAS/IML matrices.

In addition to the many SAS/IML and Base SAS functions, you can call SAS procedures, DATA steps, and macro functions from within your IMLPlus program. This feature means that you have access to the thousands of statistics that are available from SAS procedures. In short, IMLPlus enables you to access a wide range of powerful techniques for advanced statistical analysis.

Programming Technique: Use a SAS procedure to compute statistics that are needed by your program. Many IMLPlus programs consist of one or more of the following steps:

1. Use SAS/IML to compute some quantity
2. Write a SAS data set
3. Call one or more SAS procedures to analyze the data
4. Read the results of the procedures into SAS/IML matrices

4.4 Passing Parameters to SAS Procedures

The previous section uses a SAS data set to communicate information between SAS/IML vectors and a SAS procedure. However, there are occasions in which it is convenient to pass information directly from a SAS/IML vector to a SAS procedure. For example, you might want to communicate the name of a data set or the name of a variable to a SAS procedure.

You can pass the contents of a SAS/IML vector as a parameter for a SUBMIT block by listing the name of the vector in the SUBMIT statement. You refer to the contents of the vector by preceding the vector name with an ampersand (&), as shown in the following program:

```
/* pass the contents of character matrices to a procedure */
VarName = "Budget";
DSName  = "Sasuser.Movies";

submit VarName DSName;                /* IMLPlus statement */
proc univariate data=&DSName;
    var &VarName;
    ods select Moments;
run;
endsubmit;
```

Figure 4.3 Result of Passing Character Parameters to the UNIVARIATE Procedure

The UNIVARIATE Procedure			
Variable: Budget (Budget (million USD))			
Moments			
N	359	Sum Weights	359
Mean	44.8273677	Sum Observations	16093.025
Std Deviation	41.9658513	Variance	1761.13267
Skewness	1.76633581	Kurtosis	3.49160391
Uncorrected SS	1351893.45	Corrected SS	630485.497
Coeff Variation	93.6165861	Std Error Mean	2.21487292

In this program, the **VarName** and **DSName** vectors are listed in the SUBMIT statement. Inside the SUBMIT block, the contents of these vectors are substituted for the tokens **&VarName** and **&DSName**. Consequently, the SAS System receives the following statements:

```
proc univariate data=Sasuser.Movies;
    var Budget;
    ods select Moments;
run;
```

Although the use of the ampersand to refer to the contents of a variable is reminiscent of string substitution in the SAS macro language, this example does not create any macro variables. The substitution is made by IMLPlus before the SUBMIT block is sent to the SAS System.

Programming Technique: You can pass the names of variables and data sets and the values of parameters to SAS procedures. To pass the contents of a SAS/IML matrix, **m**, list the matrix in the SUBMIT statement and reference the matrix inside the SUBMIT block by using an ampersand (&m).

You can list an $n \times p$ matrix as a parameter in the SUBMIT statement. The elements of the matrix are substituted in row-major order. The substitution is valid both for character matrices and for numerical matrices, as shown in the following example:

```
/* pass the contents of a numeric matrix to a procedure */
percentiles = {10 20 30,
               40 50 60,
               70 80 90};

submit percentiles;
proc univariate data=Sasuser.Movies noprint;
  var Budget;
  output out=out pctlpre=p pctlpts=&percentiles;
run;
proc print data=out noobs; run;
endsubmit;
```

Figure 4.4 Result of Passing Numerical Parameters to the UNIVARIATE Procedure

p10	p20	p30	p40	p50	p60	p70	p80	p90
7	14	20	25	30	40	53	70	100

For the sake of this example, the program creates 3×3 numerical matrix **percentiles**. The contents of this matrix are substituted (in row-major order) for the **&percentiles** token inside the SUBMIT block. Therefore, the OUTPUT statement of the UNIVARIATE procedure contains the same PCTLPTS= options as if you had typed the following:

```
pctlpts=10 20 30 40 50 60 70 80 90
```

The result of this example is that the UNIVARIATE procedure creates an output data set named out that contains the deciles of the Budget variable of the Movies data set. The output data set is displayed by the PRINT procedure, as shown in Figure 4.4.

4.5 Case Study: Computing a Kernel Density Estimate

By using the techniques in this chapter, you can call SAS procedures from within your IMLPlus programs and can pass parameters from SAS/IML matrices to the procedures. This section uses these techniques to call the UNIVARIATE procedure to compute a kernel density estimate (KDE) of a numerical variable in a data set.

The KERNEL option in the HISTOGRAM statement of the UNIVARIATE procedure supports three ways to specify the bandwidth for a KDE. You can use the *C=value* option to explicitly specify the bandwidth. You can use the *C=MISE* option to specify that UNIVARIATE should compute a bandwidth that minimizes the approximate mean integrated square error. You can use the *C=SJPI* option to specify a bandwidth according to the Sheather-Jones plug-in method.

The following program defines the ComputeKDE module and calls the module to compute a kernel density estimate for the Engine_Liters variable in the Vehicles data set. You can use the *Bandwidth* argument to specify a bandwidth for the KDE.

```

/* define module to compute a kernel density estimate */
/* input arguments:  DSName      name of SAS data set
*                   VarName      name of variable
*                   Bandwidth    numerical bandwidth, "MISE", or "SJPI"
* output arguments:  x           evenly spaced x values
*                   f           corresponding density value
*/
start ComputeKDE(x, f, DSName, VarName, Bandwidth);    /* 1 */
  submit DSName VarName Bandwidth;                    /* 2 */
  proc univariate data=&DSName;                        /* 3 */
    var &VarName;
    histogram / noplot kernel(C=&Bandwidth) outkernel=KerOut;
  run;
  endsubmit;

  use KerOut;
  read all var {_value_} into x;                      /* 4 */
  read all var {_density_} into f;
  close KerOut;
finish;

DSName = "Sasuser.Vehicles";
VarName = "Engine_Liters";
Bandwidth = "MISE";
run ComputeKDE(x, f, DSName, VarName, Bandwidth);    /* 5 */

```

The module ComputeKDE requires three input arguments: the name of a SAS data set, the name of a numeric variable, and a parameter that specifies a standardized bandwidth for the kernel function. The bandwidth parameter can be a numeric value or a character string with the value “MISE” or “SJPI.” (In fact, the module is written so that if you pass a vector of bandwidths, then the module computes multiple kernel density estimates!)

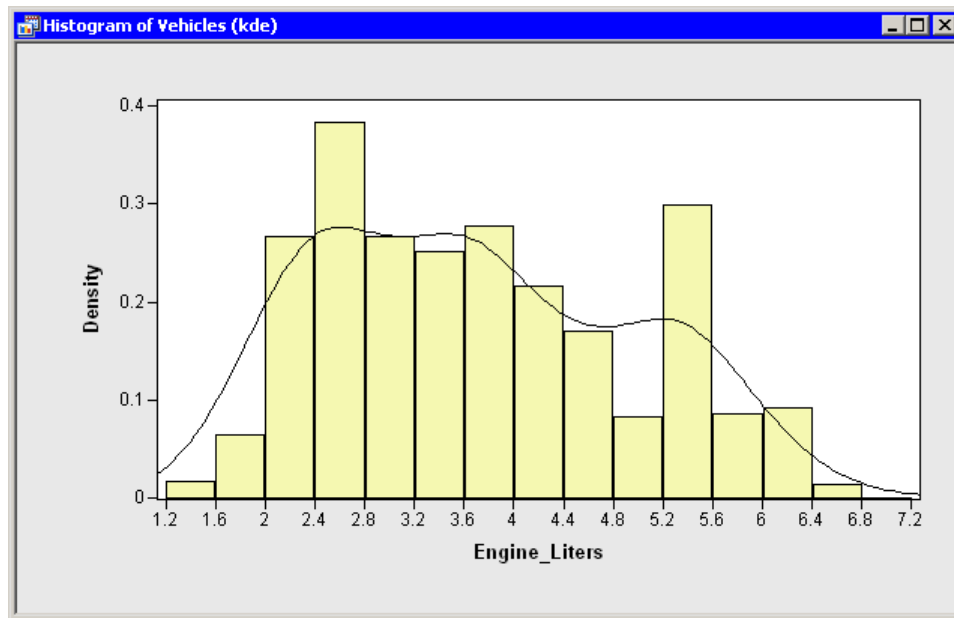
The module also has two arguments that are used to return the kernel density estimate. By convention, output arguments to SAS/IML subroutines are listed prior to the input arguments. When the module returns, the first argument contains evenly spaced *x* values, and the second argument contains the corresponding density values.

The program consists of the following main steps:

1. The program begins by defining the ComputeKDE module and naming the arguments to the module.
2. The input arguments are listed in the SUBMIT statement.

3. The UNIVARIATE procedure is called. The values of the input parameters are substituted into the procedure statements. The HISTOGRAM statement computes the kernel density estimate for the given bandwidth(s) and writes the results to the KerOut data set.
4. The KerOut data set contains two variables named `_value_` and `_density_`. The READ statements read those variables into the `x` and `f` vectors, respectively, that were provided as output arguments.
5. With the module defined, the remainder of the program specifies the input arguments and calls the module.

Figure 4.5 Histogram with Kernel Density Estimate



The histogram shown in Figure 4.5 shows the results of the module overlaid on a histogram. Chapter 7, “Creating Statistical Graphs,” discusses how to create graphs in IMLPlus. Only a few additional IMLPlus statements are required to create the graph.

4.6 Creating Names for Output Variables

One use for parameter substitution in the SUBMIT statement is to pass in the names of variables for the OUTPUT statement of a SAS procedure.

For example, suppose you want the GLM procedure to output the predicted values and the lower and upper confidence limits for mean predictions. The `P=` option to the OUTPUT statement specifies a variable that will contain the predicted values. The `LCLM=` and `UCLM=` options specify the confidence limits. A useful convention is to name the corresponding output variable `P_Y`, where *Y*

is the name of the response variable. In general, you can name an output variable by appending the name of the response variable to the option that specifies the output statistic.

You can concatenate string values in the SAS/IML language by using the concatenation operator (+) or by using the CONCAT function. The following statements demonstrate this approach by using SAS/IML to form the names of output variables that follow this convention:

```

/* form the names of output variables */
yVarName = "Mpg_Hwy";
Options = {"P", "LCLM", "UCLM"};           /* 1 */
blank32 = subpad("", 1, 32);               /* 2 */
outputVarNames = j(nrow(Options), 1, blank32); /* 3 */
do i = 1 to nrow(outputVarNames);          /* 4 */
    outputVarNames[i] = strip(Options[i]) + "_" + yVarName;
end;
outputStats = Options + "=" + outputVarNames; /* 5 */
print outputStats;

submit outputStats;
proc glm data=Sasuser.Vehicles;
    model Mpg_Hwy = Engine_Liters | Engine_Liters; /* 6 */
    output out=GLMOut &outputStats;              /* 7 */
quit;
endsubmit;

use GLMOut;
read all var outputVarNames into m;           /* 8 */
close GLMOut;

print (m[1:5,]) [colname=outputVarNames];

```

The main steps of the program are as follows:

1. Specify the name of the response variable and the options to the OUTPUT statement (P=, LCLM=, and UCLM=).
2. SAS variable names cannot exceed 32 characters. Create a string with 32 characters (called **blank32**) by using the SUBPAD function. This matrix is used in the subsequent statement to initialize a character vector.
3. Initialize the **outputVarNames** vector to have a row for each option. The character vector has space to store up to 32 characters in each element.
4. For each option, create the name for the corresponding variable: concatenate the option onto the name of the response variable, separating these strings with an underscore. You need a loop for this step because you need to strip trailing blanks from the prefix before you concatenate the prefix and the variable name.
5. Form the option-value pairs by concatenating the options with an equal sign and with the names of the output variables. [Figure 4.6](#) shows the vector of option-value pairs.

Figure 4.6 Constructing Option/Name Pairs for the OUTPUT Statement

```

                                outputStats

P      =P_Mpg_Hwy
LCLM=LCLM_Mpg_Hwy
UCLM=UCLM_Mpg_Hwy

```

6. Use the “bar operator” (|) to specify quadratic terms on the MODEL statement in PROC GLM. The bar operator is described in the section “Specification of Effects” in the documentation of the GLM procedure in the *SAS/STAT User’s Guide*.
7. Pass the vector of option-value pairs to the procedure by listing the **outputStats** matrix in the SUBMIT statement. Reference the option-pairs on the OUTPUT statement by using an ampersand.
8. Read the results of the analysis into a SAS/IML matrix. [Figure 4.7](#) displays the first 5 rows of the results.

Figure 4.7 Reading Output Variables

	P_Mpg_Hwy	LCLM_Mpg_Hwy	UCLM_Mpg_Hwy
ROW1	21.599269	21.349346	21.849192
ROW2	21.599269	21.349346	21.849192
ROW3	26.543056	26.324998	26.761114
ROW4	26.543056	26.324998	26.761114
ROW5	26.543056	26.324998	26.761114

It is instructive to consider what happens at Step 4 if the name of the response variable has length 32. In this case, the string that results from concatenating an option and the response variable has a length greater than 32. However, when this too-long string is assigned into an element of the **outputVarNames** vector, the too-long string is truncated to 32 characters. Consequently, the names of the output variables are always valid SAS variable names.

Programming Tip: To ensure that the names of variables do not exceed 32 characters, initialize the vector of names with 32 blanks. You can create a string of 32 blanks by calling the SUBPAD function: `subpad("", 1, 32)`.

Programming Tip: When concatenating character strings, use the STRIP function to remove trailing and leading blanks.

4.7 Creating Macro Variables from Matrices

This section describes how to create macro variables from SAS/IML matrices. This is an advanced topic that can be omitted during an initial reading.

Section 4.4 describes how to pass values from a matrix into a SUBMIT block. The values of the matrix are substituted into the SUBMIT block before the code is sent to the SAS System for processing. The section mentioned that although the syntax for this string substitution is the same as is used by the SAS macro language, no macro variable is actually created.

However, it is possible to create a macro variable that contains the contents of a 1×1 character matrix, and this section describes how to create such a macro variable. The technique uses pure SAS/IML statements; it does not require IMLPlus.

The macro variable is created by using the SYMPUT subroutine in the Base SAS language. The following program shows an example that uses the SYMPUT subroutine:

```
/* create a macro variable from a scalar character matrix */
c = "My String";
call symput("MyMacroVar", c);
```

The program creates a macro variable that contains the contents of the scalar matrix, **c**. If **c** is not a scalar matrix, then this approach does not work correctly.

Programming Tip: You can use the SYMPUT subroutine to store the value of a scalar SAS/IML character variable into a macro variable.

If you want to ensure that the macro variable was correctly assigned, you can print the value to the SAS log with the %PUT statement. In the SAS/IML language, you can use the %PUT statement directly from a SAS/IML program, as shown in the following statements:

```
proc iml;
%put &MyMacroVar;          /* PROC IML statement */
quit;
```

In SAS/IML Studio the %PUT statement must be inside a SUBMIT block, as shown in the following statements:

```
submit;
%put &MyMacroVar;
endsubmit;
```

IMLPlus has a shorthand notation for a SUBMIT block that contains a single statement. The previous SUBMIT block can also be specified with the following statement:

```
@%put &MyMacroVar;          /* IMLPlus statement */
```

Programming Tip: In IMLPlus, you can send a single statement to the SAS System by preceding the statement with the @ symbol. This is useful for submitting global statements such as ODS, OPTIONS, or macro statements.

It is possible, although unwieldy, to create a macro variable that contains the contents of a nonscalar matrix. You first need to concatenate the elements of the matrix into a long string, as shown in the following program:

```
/* convert a character matrix to a single string */
c = {"a", "character", "matrix"};
s = "";
do i = 1 to nrow(c);
    s = strip(concat(s, " ", c[i]));
end;
print s;
```

Figure 4.8 Concatenated Matrix Elements

<p>s</p> <p>a character matrix</p>

The program uses the STRIP function to remove trailing blanks from each element of **c**, and concatenates the strings together with the CONCAT function. At the end of the program the variable **s** is a scalar matrix that contains each of the strings in the original **c** matrix. Consequently, it can be placed into a macro variable as shown in the previous section. Notice, however, that this requires much more effort than to use substitution in the SUBMIT statement as shown in the following program:

```
/* concatenate strings by using parameter substitution */
c = {"a", "character", "matrix"};
submit c;
%put &c;
endsubmit;
```

In this example, (which does not create a macro variable), the SUBMIT block is equivalent to the following statement:

```
%put a character matrix;
```

In a similar manner, you can use the CHAR function or the PUTN function to convert numeric values into character values and then store the converted values in a macro with the SYMPUT subroutine. However, it is simpler to use substitution in the SUBMIT statement.

Programming Tip: Pass the values of SAS/IML matrices into SAS procedures by listing the matrices in the SUBMIT statement and referring to them within the SUBMIT block. Rarely do you need to create a macro variable in IMLPlus programs.

4.8 Handling Errors When Calling a Procedure

If you write an IMLPlus program that calls a SAS procedure, you might need to know whether the procedure stopped with an error. For example, the data could be degenerate in some way such as having too many missing values or having collinear variables. Or you could incorrectly specify an option. You can use the `OK=` option in the `SUBMIT` statement to determine whether a procedure generated a SAS error and to handle the error if one occurs.

The next program is a variation of the example in the section “[Passing Parameters to SAS Procedures](#)” on page 93. In that program, an array of percentile values was passed into a `SUBMIT` block and substituted into a call to the `UNIVARIATE` procedure. Recall that a valid percentile (as specified in the `PCTLPTS=` option in the `OUTPUT` statement of the `UNIVARIATE` procedure) is a number in the range $[0, 100]$. If you specify a number outside that range in the `PCTLPTS=` option, the `UNIVARIATE` procedure generates an error.

The next program intentionally substitutes a bad value into the `PCTLPTS=` option:

```
/* use OK= option to check for errors in procedures */
percentiles = -1;                /* intentionally cause an error! */
submit percentiles / ok=mOK;
proc univariate data=Sasuser.Movies noprint;
    var Budget;
    output out=out pctlpre=p pctlpts=&percentiles;
run;
endsubmit;

if ^mOK then do;
    errCode = symgetn("syserr");
    msg = symget("syserrortext");
    print errCode msg;
end;
else do;
    /* process results */
end;
```

Figure 4.9 Error Codes and Messages from Procedures

errCode	msg
2001	A percentile value is not within the required range of 0 to 100

In the example, the value of the `percentiles` matrix is -1 , which is an invalid percentile. The value is substituted into the call to the `UNIVARIATE` procedure, which stops with an error. Without the `OK=` option, a `SUBMIT` block that generates an error will stop the execution of the IMLPlus program. By using the `OK=` option, the IMLPlus program catches the error and sets the value of the `mOK` variable to 0. The program resumes execution after the `ENDSUBMIT` statement.

Typically, a procedure that generates an error writes an error message to the SAS log, and also

sets two read-only automatic macro variables. The macro variable `SYSERR` is set to zero if the procedure completes without error; otherwise, it is set to a nonzero value. The macro variable `SYSERRORTEXT` contains the last error message written to the SAS log.

In the example program, the IF-THEN statement following the SUBMIT block handles possible errors by checking the value of the `mOK` variable. If the value is zero, the program uses the `SYMGETN` and `SYMGET` functions to get the values of the `SYSERR` and `SYSERRORTEXT` macro variables and prints those values. You can learn more about SAS error processing in *SAS Language Reference: Concepts*.

Programming Tip: SAS procedures can terminate because of errors. Use the `OK=` option in the SUBMIT statement if you want your IMLPlus program to run to completion, regardless of potential errors encountered in a SUBMIT block.

Be aware that interactive procedures such as `DATASETS` and `REG` behave differently from non-interactive procedures with regard to generating errors. An interactive procedure is designed to execute when it reaches a RUN statement, and to stay running even if an error occurs. Consequently, if you end an interactive procedure with a RUN statement, the value of the `OK=` matrix will always be nonzero, even if an error occurred. To use the `OK=` option correctly, use the QUIT statement to instruct an interactive procedure to run and exit.

Programming Tip: Terminate interactive procedures with the QUIT statement when you use the `OK=` option in the SUBMIT statement.

SAS procedures can also generate warnings. The `OK=` option does not help you to detect and handle warnings because the `OK=` option returns nonzero (success) even if the SUBMIT block generates a warning. However, there is a trick that sometimes enables your IMLPlus program to detect and handle warnings. Many (although not all!) SAS procedures set the value of the `SYSERR` macro variable to the value 4 when they write a warning message to the SAS log. You can use the following program statements when it is important to handle warnings in a SUBMIT block:

```
/* handle warnings in a SUBMIT block */
errCode = symgetn("syseerr");
if errCode=4 then do;
    msg = symget("syswarningtext");
    /* handle warning */
end;
else do;
    /* process results */
end;
```

The read-only macro variable `SYSWARNINGTEXT` contains the last warning message written to the SAS log. If a SUBMIT block generates a single warning, then you can obtain the warning message through the `SYSWARNINGTEXT` macro variable. Unfortunately, if the SUBMIT block generates multiple warnings, only the last warning is obtainable through the `SYSWARNINGTEXT` macro variable.

Notice that you cannot directly access the contents of the SYSERR macro variable in IMLPlus by using an ampersand. In SAS/IML you can write the following:

```
errCode = &syserr;    /* valid in PROC IML, but not in IMLPlus */
```

However, in IMLPlus you must use the SYMGETN (or SYMGET) functions to read macro variables.

Programming Tip: In IMLPlus, you cannot use an ampersand (&) to reference a macro variable. Instead, you must use the SYMGETN (or SYMGET) functions to read macro variables.

4.9 Calling SAS Functions That Require Lists of Values

In addition to calling SAS procedures from a SAS/IML program, the SAS/IML language enables you to call Base SAS functions and subroutines. In most cases, you do not need to do anything special in order to use these functions from a SAS/IML program. For example, the LOG, CEIL, PUTN, and PDF functions are all part of Base SAS, but you can easily call them from SAS/IML programs. In fact, you can pass in a vector of values to most Base SAS functions. For example, `log(x)` returns the logarithms of **x**, whether **x** is a scalar value or a matrix.

However, some Base SAS functions require that you explicitly list arguments. For example, if you want to compute the harmonic mean of, say, 10 numbers, the HARMEAN function in Base SAS is available, but the documented syntax requires a comma-separated list of 10 arguments, as shown in the following example:

```
/* find the harmonic mean of 10 numbers */
/* First way: list numbers in call to Base SAS HARMEAN function */
submit;
data a;
    h = harmean(1,2,5,3,5,4,3,2,1,2);
run;
proc print;
run;
endsubmit;
```

Figure 4.10 Computing a Harmonic Mean in Base SAS

Obs	h
1	2.076125

This syntax is not convenient for a SAS/IML programmer who has a vector that contains the 10 values. Therefore, SAS/IML provides a vectorized override to the HARMEAN function:

```

/* Second way: Use SAS/IML version of the HARMEAN function */
x = {1,2,5,3,5,4,3,2,1,2};
h = harmean(x);
print h;

```

Figure 4.11 Computing a Harmonic Mean in SAS/IML Software

h
2.0761246

Unfortunately, not every Base SAS function has a vectorized version available in SAS/IML. Consequently, you might find that you need to call a Base SAS function whose arguments must be specified as a list of values. This section presents a general technique for writing a module that “wraps” a Base SAS function so that it becomes “vectorized” and can be called from PROC IML.

The IRR function is an example of a Base SAS function for which SAS/IML software does not provide a vectorized override. The IRR function is a financial function that “returns the internal rate of return over a specified base period of time for the set of cash payments c_0, c_1, \dots, c_n ,” according to the *SAS Language Reference: Dictionary*. You can call the IRR function from SAS/IML by explicitly forming a list, as shown in the following statement:

```

/* call Base SAS function directly to find the answer */
rate1 = IRR(1,-400,100,200,300);
print rate1;

```

Figure 4.12 Result Calling a Base SAS Function

rate1
19.43771

However, in SAS/IML the parameters are typically in a vector, so it would be more useful to be able to call a vector version of IRR such as shown in the following (incorrect) attempt:

```

c = {-400, 100, 200, 300};
rate2 = IRR(1,c);          /* WRONG! Function is expecting a list! */

```

Although the previous statement is not valid syntax, you can write a module that uses the PUTN function to convert the numeric vector of parameters into a comma-separated list, and then calls Base SAS function by using the EXECUTE subroutine in the PROC IML language. This idea is developed in the following example:

```

/* define a module in PROC IML that calls the IRR function in Base SAS.
 * This technique works for any Base SAS function that
 * is expecting a list of arguments instead of a vector.
 */
start MyIRR(freq, c);
/* convert numeric values in a matrix into a single string */
  args = strip(putn(freq,"BEST12.")); /* string of the freq value */
  cc = putn(c, "BEST12."); /* character vector of c */
  do i = 1 to nrow(cc)*ncol(cc); /* concatenate all values */
    args = concat(args, ",", strip(cc[i]));
  end;
  cmd = "rate = IRR(" + args + ");" ; /* function call as string */
  print cmd; /* optional: print command */

  call execute(cmd); /* execute the string */
  return ( rate ); /* return result */
finish;

freq = 1;
c = {-400, 100, 200, 300};
/* call Base SAS function with vector arg */
rate = MyIRR(freq, c);
print rate;

```

The module converts the numerical arguments into a comma-separated list. First, the matrix **args** is created as a character matrix that contains the first argument, converted to text. The **cc** matrix is a character matrix that contains the vector of cash payments, also converted to text. The DO loop concatenates all of these values into a big comma-separated list. The **cmd** matrix is then created and contains the SAS/IML statement that calls the IRR function with a list of arguments. The EXECUTE subroutine executes that command, and the value returned by IRR (in **rate**) is returned as the return value of the MyIRR module.

Figure 4.13 Result of a Module That Calls a Base SAS Function

rate
19.43771

As of SAS/IML Studio 3.3, the EXECUTE subroutine is not available in IMLPlus. However, you can write a similar module that uses the SUBMIT statement to call the DATA step to execute the code in the **cmd** matrix. You can then read the **rate** value from the SAS data set created by the DATA step. This is left as an exercise for the reader.

You can generalize the MyIRR module and define a new module (say, MyBaseFcn) that enables you to call *any* Base SAS function that expects a list of numerical arguments. All you need to do is to define the module so that it takes the name of a Base SAS function as a first argument, and a vector of parameters as a second argument. Then you can produce the output in [Figure 4.13](#) with the following statements:

```
/* call Base SAS function with vector arg */  
rate = MyBaseFcn("IRR", freq/c);
```

Writing the MyBaseFcn module is left as an exercise.

Part II

Programming in SAS/IML Studio

Chapter 5

IMLPlus: Programming in SAS/IML Studio

Contents

5.1	Overview of the IMLPlus Language	109
5.2	Calling SAS Procedures	111
5.2.1	Passing Parameters to a SAS Procedure	111
5.2.2	Checking the Return Code from a SAS Procedure	111
5.3	Calling R Functions	112
5.4	IMLPlus Graphs	112
5.5	Managing Data in Memory	112
5.6	Using Expressions When Reading or Writing Data	113
5.7	IMLPlus Modules	114
5.7.1	Storing and Loading IMLPlus Modules	114
5.7.2	Local Variables in Modules	117
5.7.3	Creating an Alias for a Module	117
5.8	The IMLPlus Module Library	119
5.9	Features for Debugging Programs	121
5.9.1	Jumping to the Location of an Error	121
5.9.2	Jumping to Errors in Modules	123
5.9.3	Using the Auxiliary Input Window as a Debugging Aid	124
5.9.4	Using the PAUSE Statement as a Debugging Aid	125
5.10	Querying for User Input	126
5.11	Differences between IMLPlus and the IML Procedure	126

5.1 Overview of the IMLPlus Language

The programming language of SAS/IML Studio is called *IMLPlus*. IMLPlus is an extension of SAS/IML that contains additional programming features. IMLPlus combines the flexibility of programming in the SAS/IML language with the power to call SAS procedures and to create and modify dynamically linked statistical graphics. Consequently, IMLPlus contains all of the capabilities of PROC IML that are described in earlier chapters, but it can also do much, much more.

This chapter describes statements and features of the IMLPlus language that are not supported in the IML procedure in SAS 9.2. You can use these features from the SAS/IML Studio environment, but not from PROC IML. Consequently, this chapter also answers the question, “What are the advantages to using SAS/IML Studio instead of PROC IML?”

Many of these features are described elsewhere in the book, but are collected and summarized here for easy reference. These IMLPlus statements and features enable you to use SAS/IML Studio to do the following:

- Call SAS procedures from within IMLPlus and pass parameters to a SAS procedure from SAS/IML matrices
- Call R functions from within IMLPlus
- Create dynamically linked statistical graphs
- Query, manipulate, and modify data that are stored in memory
- Read and write data by using expressions for data set names
- Define and use modules that are automatically loaded at run time, have local variables, and can be defined anywhere in the program
- Dynamically define an alias for a module name
- Access a rich library of modules, including dialog boxes that prompt for user input
- Jump directly to the location of a programming error
- Debug your program or interact with your data by using the PAUSE statement

Some IMLPlus features have proved to be very popular and so might be implemented in future releases of the IML procedure. For example, the 9.22 and 9.3 releases of SAS/IML software include a SUBMIT block for submitting SAS statements and for calling R functions from PROC IML.

When you run an IMLPlus program, you are implicitly “in IML,” meaning that you do not need to use the PROC IML statement at the top of your program, nor do you need to end your program with the QUIT statement.

Programming Tip: IMLPlus programs do not require the PROC IML or QUIT statements.

Although not required, the PROC IML and the QUIT statements are supported in IMLPlus. The PROC IML statement frees all matrices, clears user module definitions, closes any open data sets and files, and resets all SAS/IML options. The QUIT statement stops the execution of the program.

5.2 Calling SAS Procedures

A powerful feature in SAS/IML Studio is the ability to call SAS procedures from within an IMLPlus program. This enables your IMLPlus programs to compute and use any statistic available from any SAS procedure, any DATA step, and any macro program. You can call SAS procedures and DATA steps by writing SAS statements between a SUBMIT and an ENDSUBMIT statement. The SAS statements in between are called a *SUBMIT block*. The statements in the SUBMIT block are sent to the SAS System for execution.

This feature is described in Chapter 4, “Calling SAS Procedures.”

You should also use the SUBMIT statement for global SAS statements such as the LIBNAME statement, the FILENAME statement, and the TITLE statement, the OPTIONS statement, and so on. In IMLPlus, global statements that are executed in a SUBMIT block also persist outside the SUBMIT block. For example, if you define a libref by using a LIBNAME statement inside a SUBMIT block, you can also use that libref in IMLPlus statements that are executed after the ENDSUBMIT statement.

Programming Tip: Use the SUBMIT and ENDSUBMIT statements to send statements to SAS from within an IMLPlus program.

5.2.1 Passing Parameters to a SAS Procedure

It is possible to pass parameters to a SAS procedure from SAS/IML matrices. In this way, you can specify names of variables, data sets, and options to the procedure. To pass the contents of a matrix to a SAS procedure, list the name of the matrix on the SUBMIT statement. Inside the SUBMIT block, refer to the contents of the matrix by preceding the name of the matrix with an ampersand (&). IMLPlus substitutes the values of the matrix (converted to a row vector, with spaces between consecutive elements) before sending the SUBMIT block to the SAS System for processing.

This feature is described in Chapter 4, “Calling SAS Procedures.”

5.2.2 Checking the Return Code from a SAS Procedure

If there is an error in a SUBMIT block, your IMLPlus program will stop unless you handle the error. You can handle an error by using the OK= option on the SUBMIT statement. The OK= option enables you to specify the name of a SAS/IML matrix that contains a return code from the procedure. The matrix contains the value 0 if there was an error in the SUBMIT block; otherwise, the matrix contains a nonzero value.

This feature is described in Chapter 4, “Calling SAS Procedures.”

You can also determine the cause of an error in many situations by examining certain SAS macro variables that are created by the procedures. This feature is described in [Section 4.8](#).

5.3 Calling R Functions

You can call built-in or user-written functions in the R statistical programming language from within an IMLPlus program. You can load user-defined packages and call functions in those packages. You can exchange data between R and various SAS data formats. This feature is described in Chapter 11, “Calling Functions in the R Language.”

5.4 IMLPlus Graphs

The most powerful computational feature in SAS/IML Studio is the ability to call SAS procedures from within an IMLPlus program, but SAS/IML Studio also provides rich functionality for creating and modifying statistical graphs.

The IMLPlus graphs have several characteristics that appeal to data analysts:

1. The graphs are easy to create. You can create a graph with a single programming statement, or by using the SAS/IML Studio GUI through the **Graph** menu. SAS/IML Studio automatically handles the details of laying out the graph, including axes, labels, and margins. Graphs are introduced in Chapter 7, “Creating Statistical Graphs.”
 2. The graphs are dynamically linked to each other. Observations that are selected in one graph are shown as highlighted in all other graphs that view the same data. This topic is introduced in Section 6.6.
 3. You can modify the default graph attributes, such as the placement of ticks and margins. This feature is described in Chapter 10, “Marker Shapes, Colors, and Other Attributes of Data.”
 4. You can draw markers, lines, polygons, text, and other objects on a graph. This enables you to overlay the results of a statistical computation onto a scatter plot, histogram, or other graph. Collectively, the methods that draw on a plot are referred to as *Draw methods* because the names of the methods all begin with the “Draw” prefix. This feature is described in Chapter 9, “Drawing on Graphs.”
-

5.5 Managing Data in Memory

The dynamically linked graphics in SAS/IML Studio are possible because SAS/IML Studio maintains an in-memory version of the data. You can programmatically query, manipulate, and modify these data. The data are managed by a class known as the `DataObject` class. You can use methods in the `DataObject` class to get or set properties of variables and observations. For details on the `DataObject` class, see Chapter 8, “Managing Data in IMLPlus.”

5.6 Using Expressions When Reading or Writing Data

In the IML procedure, the names of data sets on the CREATE and USE statements have to be specified at the time that the statement is written. For example, the following statement is valid:

```
use Sashelp.Class;                                /* valid statement */
```

In the preceding statement, the data set name is resolved at the time that the statement is parsed. In PROC IML, it is not valid to use a character matrix to define a data set name at run time. The following statements are *not* valid:

```
DataName = "Sashelp.Class";  
use DataName;                                /* invalid statement */
```

In the second set of statements, the USE statement tries to find the Work.DataName data set instead of recognizing that **DataName** is a character matrix that contains the string "Sashelp.Class." In the IML procedure, you can work around this by using the SAS Macro Language Facility.

In IMLPlus, the traditional syntax is supported, but you can also refer to the contents of a character matrix by placing parentheses around the matrix, as shown in the following statement:

```
use (DataName);                                /* valid IMLPlus statement */
```

Consequently, the names of data sets can be resolved at run time, rather than at parse time.

The statements in the following table are enhanced in IMLPlus to accept matrix expressions.

Table 5.1 Data Set Statements That Support Expressions in IMLPlus

CLOSE	CREATE	EDIT	SETIN
SETOUT	SORT	USE	

Furthermore, in IMLPlus you can use a matrix expression (enclosed in parentheses) for the WHERE clause of any SAS/IML statement that has a WHERE clause. For example, the following statements are also valid in IMLPlus:

```
whereName = "Mpg_City";  
DataName = "Sasuser.Vehicles";  
  
use (DataName) where((whereName)>=33);  /* valid IMLPlus statement */  
read all var {"Make" "Model"};  
close (DataName);
```

5.7 IMLPlus Modules

IMLPlus modules have the same basic syntax as modules in PROC IML (see [Section 3.4](#)). In particular, IMLPlus modules are defined with the START and FINISH statements. However, IMLPlus modules differ from modules in PROC IML in several important ways:

- Not only do IMLPlus modules accept SAS/IML matrices as arguments, but also they accept Java objects such as plots and data objects. This feature is described in Chapter 6, “[Understanding IMLPlus Classes](#).”
- Although PROC IML modules must be defined prior to being called, you can define an IMLPlus module anywhere in your program. For example, you can put all of your modules at the end of your program. This is possible because the IMLPlus parser parses the entire program prior to executing the program.
- In PROC IML you must load a module by using the LOAD statement prior to calling the module. This is not required in IMLPlus. IMLPlus modules are stored in directories on your Windows PC. If you add these directories to your module search path, then the modules are automatically loaded whenever they are called from a program.
- You can dynamically define an *alias* for a module name. An alias is an alternative name for a module. It is similar to a function pointer in the C programming language.

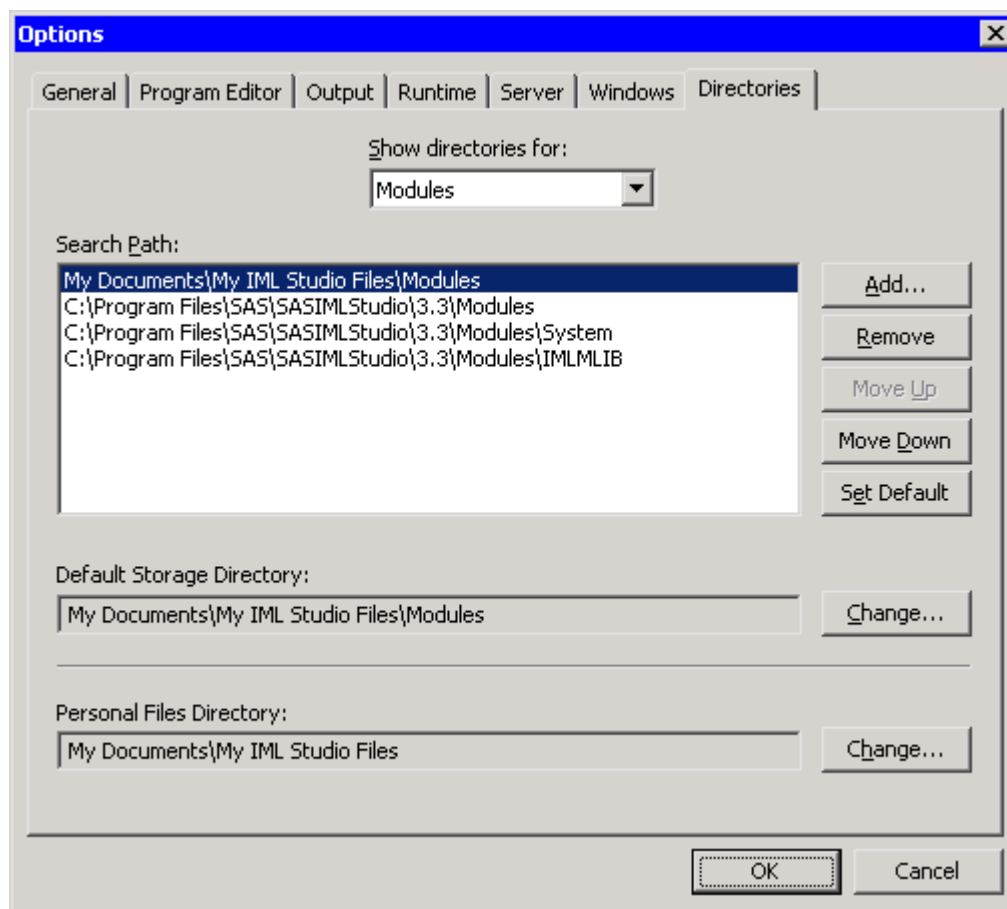
These features are described in the following sections.

5.7.1 Storing and Loading IMLPlus Modules

A module is a group of statements that can be called from programs. In IMLPlus, the module can be defined anywhere in your program: at the top of the program, at the bottom, or anywhere in the middle.

If you write a module that you want to call from multiple programs, then you need to store the module. In SAS/IML Studio, modules are stored in a directory of the Windows PC that is running SAS/IML Studio. This is different from PROC IML, which stores modules in a catalog on a SAS Workspace Server. To store a module, you can use the STORE statement. But in what directory is the module stored? The module is stored in the *default module storage directory* (DMSD) on your Windows PC. To find the DMSD, select **Tools►Options** from the SAS/IML Studio main menu. On the Options dialog box, select the **Directories** tab, as shown in [Figure 5.1](#).

Figure 5.1 The Module Search Path



The **Directories** tab shows the directories that are automatically searched when IMLPlus tries to find a module. They are searched in the order shown in the **Search Path** list.

When you use the STORE statement, IMLPlus writes two files to the directory specified in the **Default Storage Directory** field. The root name for the files is the same as the name of the module. The source code that defines the module is written to a file with the extension `.sxs`. The executable form of the module is stored in a file with the extension `.sxx`.

What is potentially confusing is that IMLPlus stores the source code in the DMSD *even if you are editing that source code in a different file!* This could result in having two versions of the module source code.

For example, suppose you are writing a module called `MyModule` that is defined in the file `C:\MyProgram.sx`. (The `sx` extension is used for SAS/IML Studio programs.) When you run the statement `store module=MyModule`, SAS/IML Studio creates a `MyModule.sxs` file and a `MyModule.sxx` file in your DMSD. If there is a run-time error in the module, SAS/IML Studio displays the version of the source code that is on the module search path, not the source code in `C:\MyProgram.sx`. This might cause you to edit the copy in the DMSD instead of fixing the error in `C:\MyProgram.sx`.

5.7.1.1 Conventions for Saving Modules

One way to avoid the previous problem is to carefully adhere to the following conventions when you write a stored module. Assume the name of your module is `MyModule`.

1. Accept the current DMSD as shown in [Figure 5.1](#), or set a new DMSD. The DMSD will contain the source and executable files for the module.
2. Create a new program window in SAS/IML Studio (**File►New►Workspace** or CTRL+N). Type or paste in only the module definition and the STORE statement, as shown in the following statements:

```
start MyModule( /* ... arguments ... */ );
  /* include all statements in the body of the module */
  x = 1;
  y = 2;
  /* and so forth */
finish;
store module=MyModule;
```

3. Save the program to a file called *MyModule.sxs* in your current DMSD. Note that this file has the *same name* as the module.
4. Run the program (**Program►Run** or F5) to define and store the module.

You can now call the module from another program. By creating a file called *MyModule.sxs* in the DMSD, you prevent IMLPlus from creating a copy of the module source code. The file *MyModule.sxs* contains the only copy of the module source; if you need to update the module, edit that file.

Programming Technique: Use the technique in this section to avoid having multiple copies of the source code for a stored module.

5.7.1.2 Loading IMLPlus Modules

Unlike PROC IML, IMLPlus does not require that you use the LOAD statement prior to running a user-defined module. Provided that the module is on your module search path, the module does not need to be explicitly loaded. You can refer to the section “Summary of Module Storage Statements” in the online Help for other ways in which the LOAD, STORE, FREE, and REMOVE statements are different in IMLPlus than in PROC IML.

Programming Tip: SAS/IML Studio automatically finds and loads modules that are stored in directories on the search path.

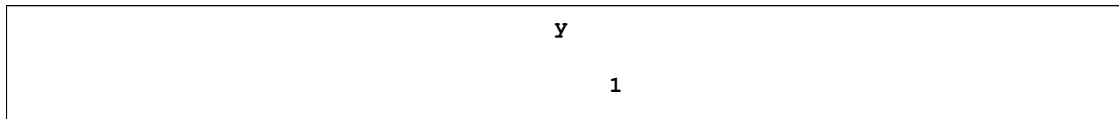
5.7.2 Local Variables in Modules

In PROC IML, a module that has no arguments implicitly uses variables in the global scope of the program. Inside the module, you can access variables from the global scope. Also, any new variables that you define in the module are accessible outside the module. For example, the following statements define a module in PROC IML that takes no arguments:

```
/* define PROC IML module with no arguments */
proc iml;
x=1;
start MyMod;                /* module with no arguments      */
    y=x;                    /* x and y are both global variables */
finish;

run MyMod;                  /* assigns y to the value of x      */
print y;                   /* y is available outside the module */
```

Figure 5.2 A Variable That Is Global in Scope



Notice that the matrix **x** is defined in the main program. When the MyMod module is run, the module uses the matrix **x** to initialize the matrix **y**. Even though **y** is created within the module, the matrix **y** is also available outside the module.

In IMLPlus, you can use parentheses on the START statement to declare that you want the module to have local variables. The following statements define a module with local variables:

```
/* define IMLPlus module with no arguments */
x=1;                                /* x is defined in the main program */
start MyFunc();                    /* no arguments; local variables    */
    y=2;                            /* y is a local variable             */
    return ( y );                  /* return the value of y            */
finish;

t = MyFunc();                      /* copies values into t              */
```

In this program, the MyFunc module takes no arguments, but the module contains local variables because parentheses are used on the START statement. Inside the module, the matrix **x** is not available unless you explicitly use a GLOBAL clause on the START statement. The local matrix **y** is defined in the module; you cannot access **y** from the main program.

5.7.3 Creating an Alias for a Module

An *alias* is the name of a module that can be set at run time. This means, for example, that you can pass the name of a module (that is, a character string) as an argument to another module, and the second module can call the first module.

As a simple example, this section describes how you might write a module to estimate the location parameter for univariate data. The sample mean is one estimator you can use, but a robust estimator such as the median is sometimes a better choice. You can define a module named `Location` that takes a string as an argument. The string determines whether to call a module that computes the mean or whether to call a module that computes the median.

The `ALIAS` statement enables you to do this. It enables you to specify the *name* of a module, and then call the module with that name. The following statements implement an IMLPlus function that uses the `ALIAS` statement:

```
/* write an IMLPlus module that estimates a location parameter */
start MyMean(x);
    return ( x[,] );
finish;

/* The Median module already exists in IMLMLIB.
 * The syntax is Median(x).
 */

/* module that calls either the MyMean or the Median module */
start Location(StatName, x);
    alias *MyFunc(x) StatName;      /* MyFunc is an ALIAS                */
    y = MyFunc(x);                  /* call function that is passed in */
    return ( y );                   /* value is returned                */
finish;

x = {1 2 1, 1 4 3, 1 5 3, 1 5 18};
mean = Location("MyMean", x);
median = Location("Median", x);
print mean, median;
```

Figure 5.3 The `ALIAS` Statement in IMLPlus

mean		
1	4	6.25
median		
1	4.5	3

The `Location` module uses the `ALIAS` statement to associate the name `MyFunc` to the module name that is contained in the `StatName` matrix. When the statement `MyFunc(x)` is executed, it can call the `MyMean` or `Median` module—or any other function module that takes one argument! Consequently, if you create a `Trim10` module at some future date that computes the 10% trimmed mean, you can immediately make the following call without making any changes to the `Location` module:

```
trimMean = Location("Trim10", x);
```

Notice that an alternative way to write the `Location` module would be to use an `IF-THEN/ELSE` statement to look at the name of the `StatName` argument and conditionally call the appropriate module. However, that approach is not extensible: if you create a `Trim10` module at some future date, you will also need to modify the definition of the `Location` module.

Notice that MyFunc is not a module, since it is not defined by using START and FINISH statements. Instead, it is an alias to a module. Some programming languages (such as C) have the concept of a *function pointer*. If you are familiar with function pointers, an alias is a similar concept.

IMLPlus needs to know whether an alias is to a function or a subroutine, and needs to know how many arguments are in the function. Some programming languages call this the *prototype* of a function. If you create an alias to a function (that is, to a module that returns a value), then prefix the name of the alias with an asterisk:

```
alias *MyFunc(x) StatName;      /* the module returns a value */
```

If, on the other hand, you create an alias to a subroutine (that is, to a module that does not return a value), then do not use an asterisk:

```
alias MySubroutine(x) StatName; /* the module is a subroutine */
```

5.8 The IMLPlus Module Library

The IMLMLIB library of modules is distributed with SAS/IML software. This is a collection of documented modules that you can call when writing SAS/IML programs. You can call these modules from PROC IML and from SAS/IML Studio.

In addition, SAS/IML Studio 3.3 is distributed with more than 70 additional IMLPlus modules. These modules are only available from IMLPlus programs. The modules are contained in the **Modules** subdirectory of the SAS/IML Studio *installation directory* (for example, **C:\Program Files\SAS\SASIMLStudio\3.3\Modules**), which is automatically part of the module search path. (See [Figure 5.1](#).) The modules are documented in the SAS/IML Studio online Help, which you can display by selecting **Help►Help Topics** from the main menu, and then by selecting the chapter titled “IMLPlus Module Reference.” Most of the modules are documented in the sections titled “General Purpose,” “Graphics,” and “User Interface.”

The next tables describe modules that are frequently used in statistical programs. Each module is documented fully in the online Help, and also contains an example that you can run to examine how the module works.

Table 5.2 General Purpose: Frequently Used IMLPlus Modules

IMLPlus Module	Description
CopyServerDataToDataObject	Copies variables from a SAS data set in a SAS libref into a data object (see Section 8.8.3)
GetPersonalFilesDirectory	Returns the Windows directory in which you should store data sets, modules, and programs related to SAS/IML Studio (see Section 12.10)

A *color ramp* is a sequence of colors that smoothly varies from one color to another. A color ramp is often used to visualize values of a continuous variable. Colors and coloring observations are

described in Chapter 10, “[Marker Shapes, Colors, and Other Attributes of Data](#).” The following modules are useful for creating graphs that use color to visualize values of a variable.

Table 5.3 Graphics: Frequently Used IMLPlus Modules

IMLPlus Module	Description
BlendColors	Uses linear interpolation to create a color ramp from a vector of colors
ColorCodeObs	Sets the color of observation markers by mapping values of a single variable onto a color ramp
ColorCodeObsByGroups	Sets the color of observation markers by mapping values of one or more nominal variables to a set of colors
DrawInset	Displays an inset on a graph (see Section 9.2)
DrawLegend	Displays a legend on a graph (see Section 9.2)
DrawContinuousLegend	Displays a color ramp on a graph
DrawPolygonsByGroups	Displays a collection of polygons (such as a map) on a graph
IntToRGB	Converts a hexadecimal representation of colors into RGB triples (see Section 10.2.2)
RGBToInt	Converts RGB triples of colors into a hexadecimal representation (see Section 10.2.2)

The “User Interface” modules display dialog boxes of various types. SAS/IML Studio 3.3 has a variety of dialog boxes that you can use to solicit input from a person running a program. For example, you can display a simple dialog box such as a message box or a list box. A frequently used module is DoDialogGetListItems, which displays a dialog box with a list of items and prompts the user to select one or more items. These modules are described in the section “[Querying for User Input](#)” on page 126.

Table 5.4 User Interface: Frequently Used IMLPlus Modules

IMLPlus Module	Description
DoDialogGetDouble	Displays a dialog box that prompts the user for a number
DoDialogGetListItem	Displays a dialog box that prompts the user to select an item from a list
DoDialogGetListItems	Displays a dialog box that prompts the user to select one or more items from a list (see Section 16.5)
DoDialogGetString	Displays a dialog box that prompts the user for a character string
DoDialogModifyDouble	Displays a dialog box that prompts the user to accept or modify a default numeric value (see Section 16.5)
DoErrorMessageBoxOK	Displays an error dialog box that contains an OK button
DoMessageBoxOK	Displays a message dialog box that contains an OK button (see Section 16.5)
DoMessageBoxYesNo	Displays a message dialog box that contains a Yes button and a No button (see Section 16.5)

5.9 Features for Debugging Programs

Debugging a SAS/IML program in the SAS Enhanced Editor can be challenging. SAS/IML Studio has three features that make debugging programs easier. The first feature is that SAS/IML Studio has a program editor that colors keywords in the IMLPlus language. The second feature is that you can jump directly to the location of a programming error in the program window. The third feature is that you can use the PAUSE statement in conjunction with the Auxiliary Input window to debug your program.

5.9.1 Jumping to the Location of an Error

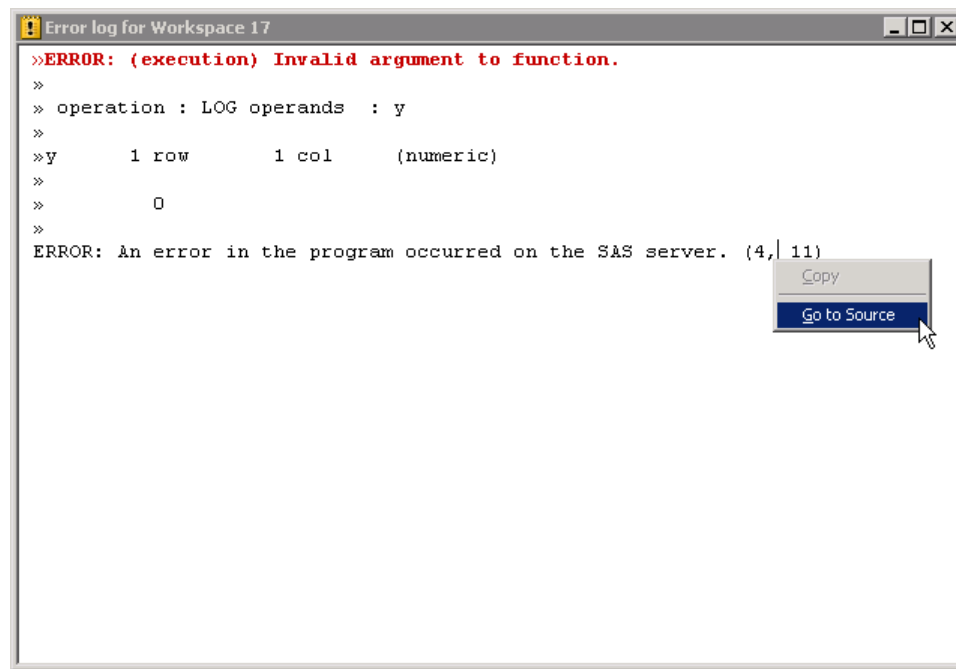
There are two kinds of errors that cause a program to report an error: the parse-time error and the run-time error. Common parse-time errors include mistyping a statement, forgetting a semicolon, or failing to close a set of parentheses. In all these cases, the syntax of the program is incorrect. In SAS/IML Studio, you can select **Program►Check Syntax** to check your program for parse-time errors.

On the other hand, a run-time error does not occur until the program is actually run. Common run-time errors include adding matrices that are different sizes, taking the logarithm of a negative value, and using the matrix index operator to specify indices that do not exist.

If you run a program and SAS/IML Studio reports a parse-time or run-time error, then the Error Log window appears. (The Error Log window appears automatically provided that the **Auto activate** property is selected in the **Window** tab of the **Tools►Options** dialog box. If the **Auto activate** property is not selected, you can view the Error Window by pressing the F8 key.) The error message in the Error Log window typically ends with a pair of numbers that indicate the position (line number and column number) of the error in the program window, as shown in [Figure 5.4](#). To jump directly to the location of the error in the program window, do the following:

1. Position the mouse cursor on the error message.
2. Right-click the error message. A pop-up menu is displayed.
3. Select the **Go to Source** menu item.

The program window becomes active and SAS/IML Studio positions the cursor at the location of the error. You can also go to an error location by pressing CTRL+G.

Figure 5.4 Jumping to an Error Location

The following statements contain a run-time error:

```
/* define loop that contains a run-time error */
do x = 5 to 1 by -1;
  y = log(x);
  z = log(y);
end;
```

When you run the program, the following message appears in the Error Log:

```
>>ERROR: (execution) Invalid argument to function.
>>
>> operation : LOG operands : y
>>
>>y      1 row      1 col      (numeric)
>>
>>      0
>>
ERROR: An error in the program occurred on the SAS server. (4, 11)
```

The first eight lines in the error message begin with the >> character. Lines beginning with the >> character are output by the SAS server (in this case, from SAS/IML software). The remaining lines are output by the IMLPlus language interpreter. The lines from the SAS server indicate the following:

1. The error occurs in the argument to a function.
2. The function being evaluated is the LOG function.

3. The argument is the matrix **y**, which has one row, one column, is numeric, and has the value 0.

The portion of the message from the IMLPlus interpreter indicates that the error occurred on (or near) line 4 and column 11. If you right-click on the final line of the error message and choose **Go to Source** from the pop-up menu, then the cursor is placed on the indicated line and column. From this information, you should be able to determine that the program is unable to evaluate the LOG function at 0. (Recall that the logarithm function is not defined for nonpositive numbers.)

Parse-time errors are handled similarly.

5.9.2 Jumping to Errors in Modules

If an error occurs inside a module (or inside many nested modules), you can press CTRL+G to go to the statement in the main program that calls the outermost module in which the error occurred. You can then *descend into the module* by pressing SHIFT+F8. This means that SAS/IML Studio will display the source code for the module and position the cursor on the line in the module at which the error occurred. You can continue to follow errors into as many modules as necessary.

The following example is essentially the same as the previous one, except this time the run-time error occurs in a module:

```
/* define and call module that contains a run-time error */
start MyMod(k);
  do x = 5 to k by -1;
    y = log(x);
    z = log(y);
  end;
finish;

run MyMod(1);
```

When you run the program, the following message appears in the Error Log:

```
>>ERROR: (execution) Invalid argument to function.
>>
>> operation : LOG operands : y
>>
>>y      1 row      1 col      (numeric)
>>
>>      0
>>
ERROR: An error occurred while executing module "MyMod". (9, 1)
An error in the program occurred on the SAS server. (+4, 14)
```

The error message from the SAS server (the first eight lines) is the same as for the previous example. The message from the IMLPlus interpreter (the last two lines) is different. The first line says that the error occurred in the module MyMod when it was called from line 9 in the program. The second line indicates that, relative to the first line of the module, the error occurred on line 4, column 14. The START statement is considered the first line of a module. You can right-click on either line to

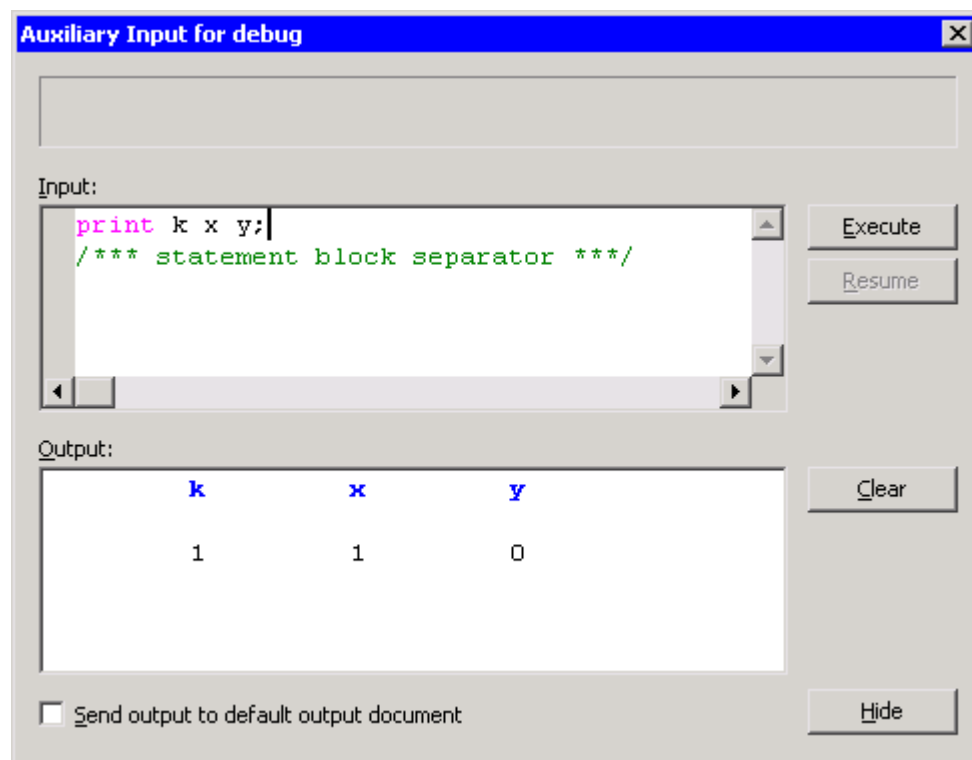
go to the corresponding error location. Alternatively, you can press CTRL+G to go to the first error location, and SHIFT+F8 to descend into the module.

5.9.3 Using the Auxiliary Input Window as a Debugging Aid

The Auxiliary Input window is a pop-up window in which you can execute program statements when your program is paused. A program is implicitly paused when it encounters a run-time error. You can also explicitly pause a program by using the PAUSE statement (see the next section). A common use of the Auxiliary Input window is to print the value of variables in order to better understand why a program has experienced a run-time error.

In many situations, you can select **View►Auxiliary Input** (or press F7) to display the Auxiliary Input window, as shown in Figure 5.5. You can type any valid IMLPlus statement into the **Input** area. Press the **Execute** button to run only those statements. The statements are executed at the scope of the innermost module that contains the error.

Figure 5.5 The Auxiliary Input Window



For example, after the previous example stops with an error, you can press F7 to display the Auxiliary Input window, and then type the following statement in the **Input** area:

```
print k x y;
```

When you press **Execute**, the output from the PRINT statement is shown in the Auxiliary Input window. Notice that the variables **k**, **x**, and **y** are defined only in the module. This indicates that the Auxiliary Input window is executing statements within the scope of the MyMod module.

If the source code for the module that contains the error is not in the same program window as the statement that calls the module, then SAS/IML Studio opens a new workspace that contains the module source code. This often happens when the error occurs within a saved module.

5.9.4 Using the PAUSE Statement as a Debugging Aid

The error log can help you determine *where* an error occurred, but the PAUSE statement can help you determine *why* a program is not giving correct answers. The PAUSE statement can be used to debug a program that gives wrong answers, but that does not stop with an error. You can use the PAUSE statement in the main program and also inside modules.

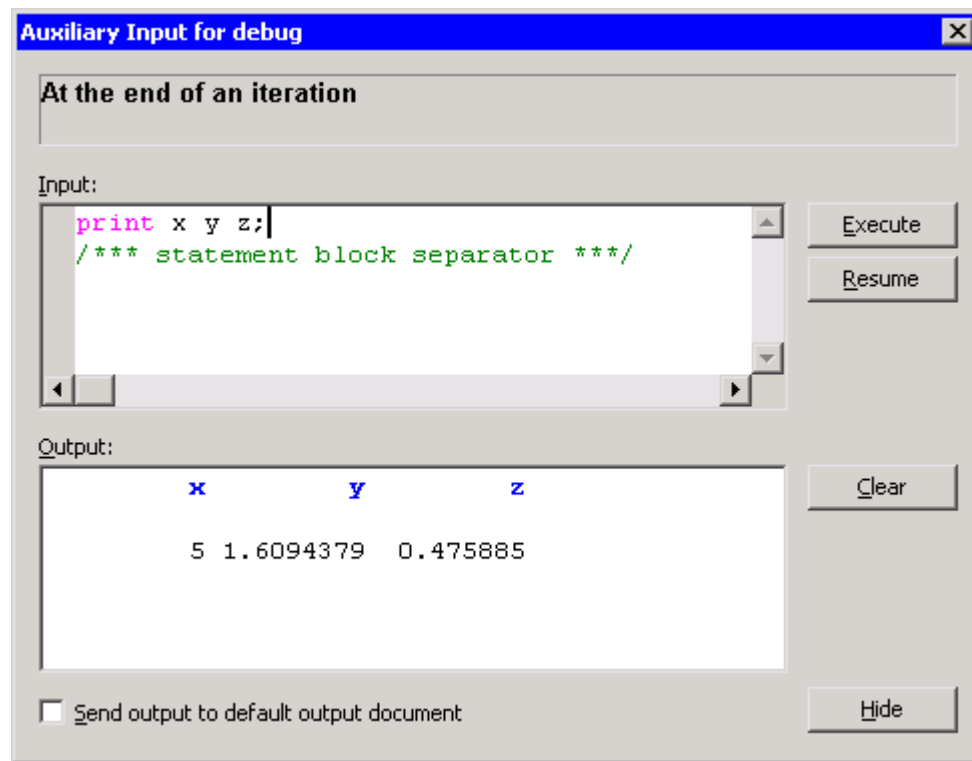
For example, suppose you want to monitor the values of variables in a loop. The following statements use a PAUSE statement at the end of a loop to pause the program and display the Auxiliary Input window:

```
/* pause program at the end of each iteration */
do x = 5 to 1 by -1;
  y = log(x);
  z = log(y);
  pause "At the end of an iteration";
end;
```

The Auxiliary Input window displays the string specified on the PAUSE statement. You can type any valid IMLPlus statements into the **Input** area and click **Execute** to execute the statements. [Figure 5.6](#) shows the Auxiliary Input window that appears at the end of the first iteration. A PRINT statement was typed into the **Input** area; the values of the **x**, **y**, and **z** variables at the end of the first iteration are shown in the **Output** area. When you click **Resume**, the program continues from the statement following the PAUSE statement.

If the text on the PAUSE statement begins with the string “NoDialog:”, then the Auxiliary Input window is not displayed, but the program still pauses its execution. The remainder of the message is displayed in the SAS/IML Studio status bar (located in the lower left corner of the SAS/IML Studio application). This enables the person running the program to interact with the data before resuming the program, as described in Chapter 16, “[Interactive Techniques](#).”

Figure 5.6 The Auxiliary Input Window as a Debugging Aid



5.10 Querying for User Input

SAS/IML Studio distributes modules that create and display several simple dialog boxes. All of these modules begin with the prefix “DoDialog” or “DoMessageBox.” You can use these dialog boxes to query for user input or to display information to the user. These dialog boxes are *modal*, meaning that the program halts execution until the dialog box is dismissed. These modules are described in Chapter 16, “Interactive Techniques.”

5.11 Differences between IMLPlus and the IML Procedure

This chapter primarily focuses on ways that the IMLPlus programming language extends the SAS/IML language. Most computational modules and programs that run in the IML procedure also run in IMLPlus without modification. However, there are a small number of SAS/IML statements and functions that are not supported in IMLPlus. The differences are described in the online Help in the chapter “The IMLPlus Language.”

Of these differences, the most important PROC IML statement that is not available in IMLPlus is the EXECUTE subroutine.

Another major difference is that IMLPlus programs do not intrinsically support SAS System global statements. For example, the following statements are not supported in IMLPlus:

- Global SAS statements such as LIBNAME, FILENAME, and TITLE.
- The OPTIONS statement.
- The ODS statement.
- Macro language statements. This includes using %let to define macro variables and using %macro to define macro functions, in addition to making calls to macro functions. Use the SYMGET and SYMPUT functions in Base SAS software to create macro variables from SAS/IML matrices, and vice versa.

You can still use these statements provided that you use them in a SUBMIT block. You can also submit these global statements by using the @ symbol, as described in Chapter 3, “[Programming Techniques for Data Analysis](#).”

Programming Tip: In IMLPlus, you can submit a single SAS statement (for example, a global statement such as a LIBNAME, OPTIONS, or ODS statement) by preceding the statement with the @ symbol. For example, the following statement is valid in IMLPlus:

```
@ods trace on;
```

IMLPlus does not support the PROC IML low-level drawing routines such as GSTART, GOPEN, GPOLY, GPOINT, and GSHOW. Instead, IMLPlus provides high-level statistical graphs such as scatter plots, histograms, and box plots. IMLPlus also provides methods for drawing markers, lines, polygons, text, and other graphical objects on a plot. [Section 9.10](#) provides a comparison between drawing in IMLPlus and drawing in PROC IML.

Chapter 6

Understanding IMLPlus Classes

Contents

6.1	Overview of Understanding IMLPlus Classes	129
6.2	Object-Oriented Terminology	130
6.3	The DataObject Class	131
6.4	Base and Derived Classes	134
6.5	Creating a Graph	135
6.6	Creating Dynamically Linked Graphs	136
6.7	The Plot Class: A Base Class for Graphs	138
6.8	The Data Table Class	138
6.9	The DataView Class: A Base Class for Graphs and Data Tables	139
6.10	Passing Objects to IMLPlus Modules	139
6.11	Using a Base Class in a Module	141

6.1 Overview of Understanding IMLPlus Classes

This chapter describes how to programmatically create, manipulate, and modify the graphs and data tables that are available in SAS/IML Studio. Because the “Plus” portion of the IMLPlus language is object-oriented, the chapter begins by introducing object-oriented terms such as class, object, method, signature, base class, and derived class. The remainder of the chapter describes several important IMLPlus classes such as the DataObject class and the Plot class, and shows how you can pass objects of these classes to IMLPlus modules.

You can run the programs in this chapter only from the SAS/IML Studio environment. PROC IML does *not* support classes.

Programming Tip: You must use SAS/IML Studio to run IMLPlus programs.

6.2 Object-Oriented Terminology

The IMLPlus programming language borrows ideas from object-oriented programming, particularly Java. An *object* is a variable that refers to a class. The class is a “template” for the object: it specifies the data and the functions (called *methods*) that query, retrieve, and manipulate the data. In IMLPlus, a variable is implicitly assumed to be a SAS/IML matrix unless you use the **declare** keyword to specify that the variable is an object.

To call class methods in IMLPlus, you use a “dot notation” syntax in which the method name is appended to the name of the object, such as *ObjectName.MethodName()*. Class names, object names, and method names are case-sensitive, which is the Java convention. The **declare** keyword is also case-sensitive.

Programming Tip: In IMLPlus, the names of classes, objects, and methods are case-sensitive. You use the **declare** keyword to specify that a variable is an object. The **declare** keyword must be in lowercase characters.

IMLPlus includes many classes that are useful in data analysis such as the classes that define statistical graphs: ScatterPlot, Histogram, BarChart, and so on. In addition, you can call any other Java classes from IMLPlus.

Programming Tip: IMLPlus programs support Java classes and objects. You can instantiate objects, call their methods, and even pass them to IMLPlus modules.

In IMLPlus, the names of classes are written in mixed case and begin with a capital letter. Often the name of a method is formed by concatenating a verb and a direct object. For example, SetAxisLabel, DrawLine, and GetVarData are all names of IMLPlus methods.

Programming Tip: In IMLPlus, the names of classes are written in mixed case and begin with a capital letter.

The calling syntax for a method (that is, the number and order of arguments) is called a *signature*. Java methods can have multiple signatures. Often a short signature is identical to a longer signature, but with default values for certain parameters.

IMLPlus classes obey most of the same conventions as Java classes. The SAS/IML Studio online Help chapter “The IMLPlus Language” contains a section “Java-oriented Language Extensions” that documents the similarities and differences between IMLPlus and Java. You can display the online Help by selecting **Help►Help Topics** from the main menu.

6.3 The DataObject Class

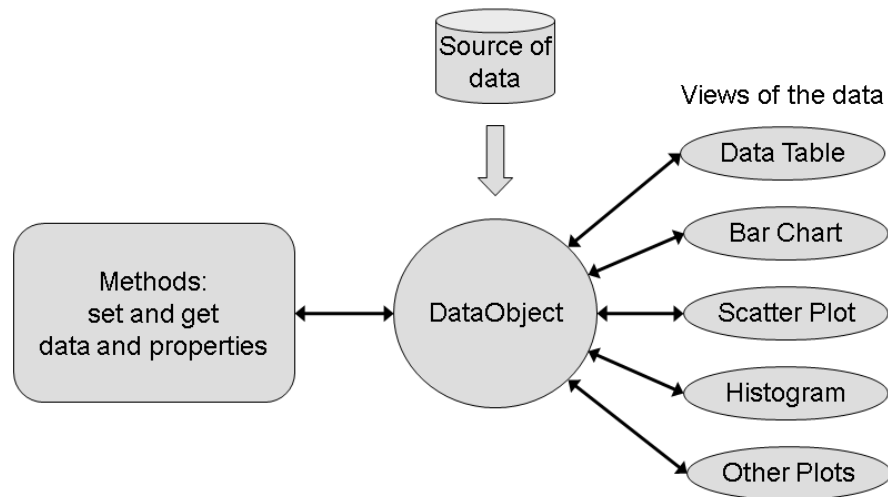
The most important class in the IMLPlus language is the DataObject class. The DataObject class manages an in-memory copy of data. An object of the DataObject class is typically created (or *instantiated*) from a SAS data set, but it can also be created from a SAS/IML matrix, an R data frame, an Excel spreadsheet, or other sources. When you use the SAS/IML Studio GUI to open a data set, SAS/IML Studio reads the entire data set into memory. This enables dynamically linked graphs and tables: selecting observations in one graph is quickly reflected in other views of the same data because the data are in memory.

The DataObject class contains methods that query, retrieve, and manipulate the data. The class also manages graphical information about observations such as the shape and color of markers, the selected state of observations, and whether observations are displayed in plots or are excluded.

Programming Tip: You can instantiate an object of the DataObject class from any of several formats: a SAS data set, a SAS/IML matrix, an R data frame, an Excel spreadsheet, and so on. The DataObject class provides a uniform interface to the data, regardless of the data source.

The first chapter of the *SAS/IML Studio for SAS/STAT Users* documentation devotes several pages and diagrams to explaining the DataObject class and its role in coordinating graphical and tabular views of the data. It is well worth reading. [Figure 6.1](#) shows a schematic description of the three primary roles of the DataObject class:

- to read data into memory from various sources
- to be a uniform interface to data by providing methods that set and get data and properties of observations and variables
- to coordinate the display of data in dynamically linked graphs and data tables

Figure 6.1 Schematic Description of the Roles of the DataObject Class

This book refers to an “object of the DataObject class” as a *data object*. The following example creates a data object from the Movies data set and calls two methods in the DataObject class:

```

/* create data object and call methods */
declare DataObject dobj;                               /* 1 */
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies"); /* 2 */

dobj.GetVarNames(varNames);                             /* 3 */
print varNames;

dobj.GetVarData("Budget", b);                           /* 4 */
budgetSummary = quartile(b);                             /* 5 */
print budgetSummary[rowname={"Min" "Q1" "Median" "Q3" "Max"}];

```

The following list describes each step of the program:

1. Use the **declare** keyword to specify that the variable **dobj** is an object of the DataObject class. The variable **dobj** is not yet created.
2. Instantiate the variable **dobj** from the SAS data set **Sasuser.Movies**. (You can also use other methods to create a data object from other sources of data, as described in Chapter 8, “[Managing Data in IMLPlus](#).”) You can now call all methods in the DataObject class on the object, **dobj**.
3. Call the **GetVarNames** method in the DataObject class by using the *ObjectName.MethodName()* syntax. The argument to this method is the name of a SAS/IML vector, **varNames**. The method creates the **varNames** vector and fills it with the names of the variables in the data object. These names are printed and are shown in [Figure 6.2](#).
4. Call the **GetVarData** method in the DataObject class. The method retrieves the data from the **Budget** variable and copies it into a SAS/IML vector called **b**.

5. Call the QUARTILE module (which is part of the IMLMLIB module library) to compute the five-number summary of the data. The minimum, first quartile, second quartile (median), third quartile, and maximum values of the data are stored in the vector **budgetSummary**, which is shown in [Figure 6.2](#).

Figure 6.2 Result of Calling Methods in the DataObject Class

varNames	
Title	
MPAARating	
ReleaseDate	
Budget	
US_Gross	
World_Gross	
Sex	
Violence	
Profanity	
NumAANomin	
NumAAWon	
AA Nomin	
AA Won	
Distributor	
Year	
budgetSummary	
Min	0.15
Q1	16.5
Median	30
Q3	60
Max	258

Programming Technique: If **dobj** is an object of the DataObject class, use the **GetVarData** method to create a SAS/IML vector that contains the data for a variable:

```
dobj.GetVarData("VarName", v);
```

You can also use the **GetVarData** method to create a matrix that contains the data for several variables.

Chapter 8, “[Managing Data in IMLPlus](#),” gives further examples of methods in the DataObject class. Frequently used methods in the DataObject class are listed in [Appendix C](#).

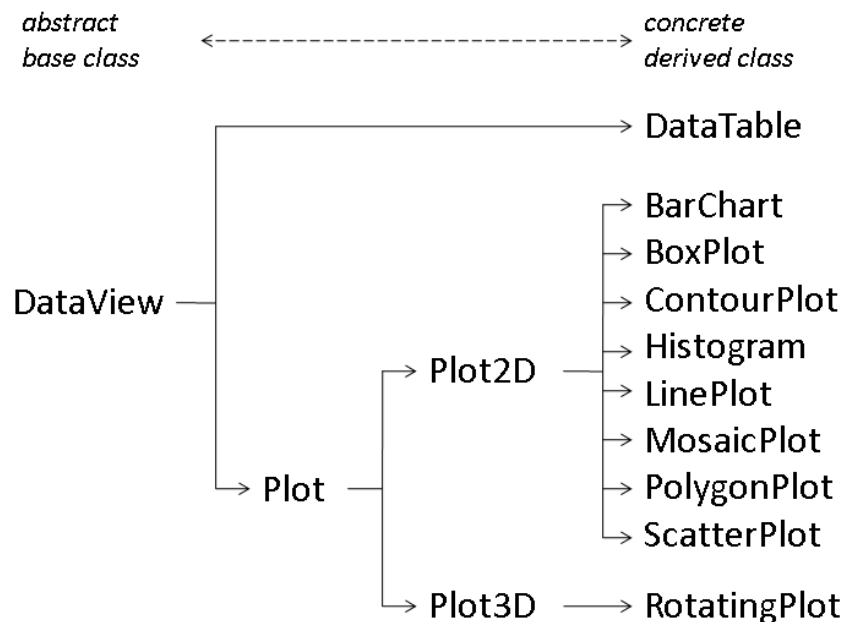
You can find a complete list of all IMLPlus classes and methods in the online Help. To view the documentation for IMLPlus classes, select **Help►Help Topics** from the SAS/IML Studio main menu, and then select the chapter titled “IMLPlus Class Reference.”

6.4 Base and Derived Classes

Some IMLPlus classes are derived from other classes, which means that the derived class inherits the methods and properties of the so-called base class. For example, the Histogram class is derived from the Plot2D class, which in turn is derived from the Plot class, which is derived from the DataView class. Methods available in a base class are (usually) available to a derived class.

The class hierarchy of IMLPlus data views is shown in [Figure 6.3](#). The classes listed in the far right column are *concrete classes*, meaning that they can be instantiated. The other classes are *abstract classes* that cannot be directly instantiated. For example, you cannot instantiate an object for the DataView class, but you can call methods in the DataView class for data tables, bar charts, scatter plots, and any other objects of a class derived from the DataView class.

Figure 6.3 DataView Class Hierarchy



[Figure 6.3](#) is useful for finding documentation for class methods in the SAS/IML Studio online Help. All IMLPlus methods (and an example of their use) are documented in the online Help, in the chapter “IMLPlus Class Reference.” To see which methods are available for a given type of graph (for example, a bar chart), you need to look not only in the documentation for the class itself (**BarChart**) but also in the documentation for all base classes (**Plot2D**, **Plot**, and **DataView**). Most methods for graphs are located in the **Plot** class, which contains nearly 100 methods. In contrast, the **BarChart** class has only six methods that are not defined in a base class.

Programming Tip: Most of the methods you can call from a graph or data table are documented in base classes such as the Plot and DataView classes.

Notice that the class hierarchy presented in [Figure 6.3](#) does not contain the DataObject class. The DataObject class is not part of the DataView class hierarchy. In fact, the hierarchy for the DataObject class is remarkably simple: the DataObject class does not have any base nor derived classes.

6.5 Creating a Graph

Suppose you want to create a graph such as a bar chart. In technical terms, you want to instantiate an object of the BarChart class. To do this, you need to call the BarChart.Create method. There are several signatures for the Create method, but the simplest way to create a bar chart is to create it from a variable in a data object.

The following program creates a data object from the Movies data set. It then instantiates a bar chart and calls methods defined in several base classes. Each base class for the BarChart class is shown in [Figure 6.3](#).

```
/* call base class methods for object of BarChart class */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

/* create a bar chart from a variable in a data object */
declare BarChart bar;
bar = BarChart.Create(dobj, "MPAARating");

/* call various methods that modify attributes of the bar chart */
bar.ShowPercentage(); /* BarChart method */
bar.SetAxisViewRange(YAXIS, 0, 50); /* Plot2D method */
bar.ShowReferenceLines(); /* Plot method */
bar.SetWindowPosition(50, 50, 50, 50); /* DataView method */
```

The bar chart is shown in [Figure 6.4](#). Of the four methods in the program, only one is defined in the BarChart class. The other three methods are defined in one of the three base classes for the BarChart class (see [Figure 6.3](#)).

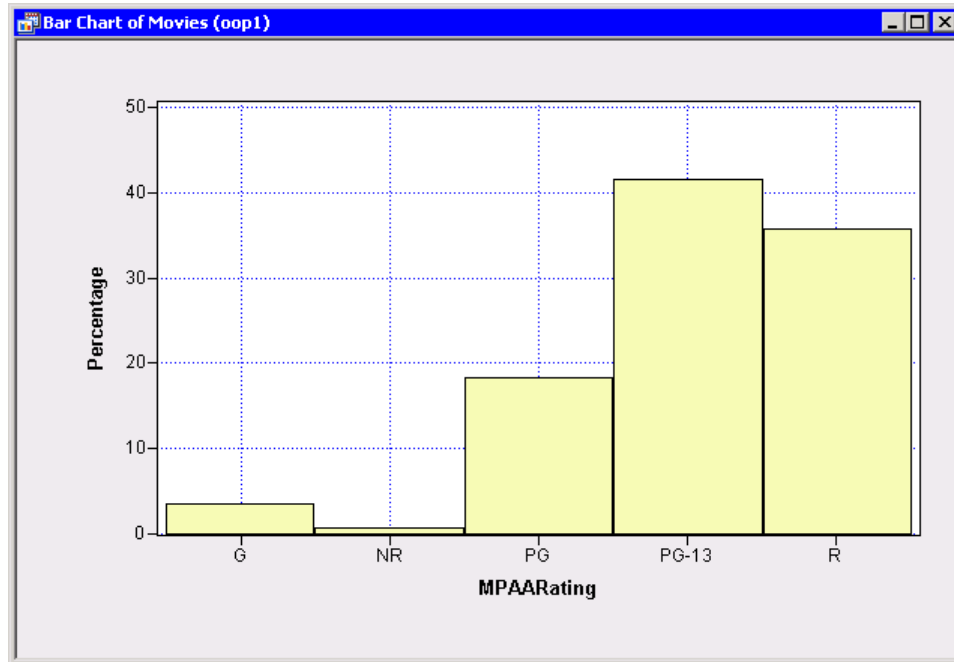
BarChart class method The ShowPercentage method in the BarChart class specifies whether the Y axis of the bar chart displays frequencies or percents.

Plot2D class method The SetAxisViewRange method in the Plot2D base class enables you to explicitly set the range of an axis in a two-dimensional graph. All graphs except the rotating plot can call this method.

Plot class method The ShowReferenceLines method in the Plot class (which adds reference lines to a graph) can be called by all graphs, but not by a data table.

DataView class method Any graph or data table can call the `SetWindowPosition` method in the `DataView` class because the `DataView` class is a base class for all classes that display data in graphical or tabular forms. The `SetWindowPosition` method sets the location and size of a data view within the SAS/IML Studio application.

Figure 6.4 Result of Calling Methods in Base Classes



As mentioned previously, the `DataObject` class is not part of the `DataView` class hierarchy. However, every data view has an associated data object. In the previous program, the bar chart was explicitly created from a data object, but even when a graph is created directly from vectors of data (as described in Chapter 7, “Creating Statistical Graphs”), there is an underlying data object. You can obtain the data object from a graph or data table by calling the `GetDataObject` method in the `DataView` class.

Programming Tip: You can get the data object that is associated with a graph or data table by calling the `GetDataObject` method in the `DataView` class.

6.6 Creating Dynamically Linked Graphs

The program in the previous section created a bar chart of the data in the `MPAARating` variable in a data object created from the `Movies` data set. It is simple to extend that program to create a second graph (say, a scatter plot) that is dynamically linked to the bar chart: merely create the second graph *from the same data object*, as shown in the following statements:

```

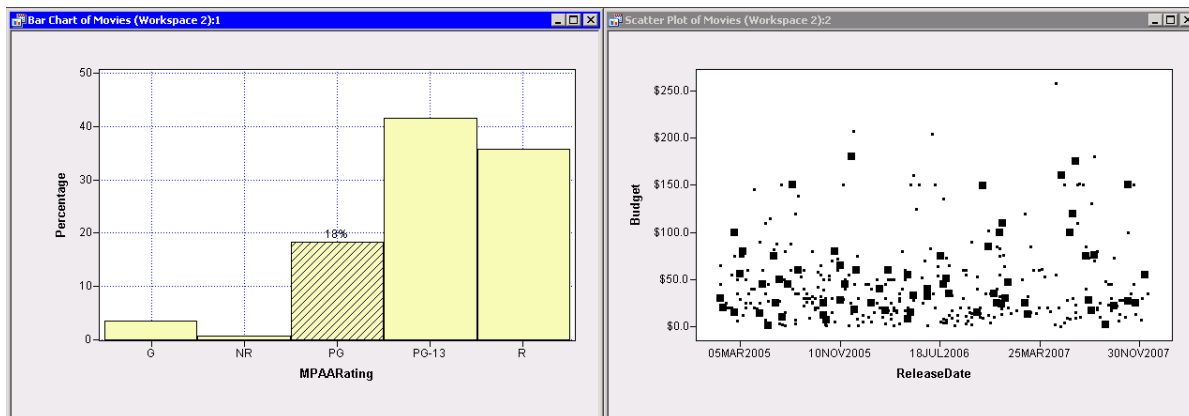
/* create a second graph. The two graphs are dynamically linked. */
declare ScatterPlot plot;
plot = ScatterPlot.Create(dobj, "ReleaseDate", "Budget");

```

In the program, the `dobj` variable is the same data object that was created from the Movies data and that was used to create the `bar` object. Because the scatter plot and the bar chart are created from a common data object, they are automatically linked to each other and to any other graphical or tabular view of the same data object.

The resulting graphs are shown in Figure 6.5. The figure shows the graphs after clicking on the PG category in the bar chart. Note that the 66 movies that are rated PG are selected. These observations are displayed as highlighted—both in the bar chart and also in the scatter plot. From the scatter plot you can observe a few facts about the PG-rated movies in these data: six PG-rated movies had budgets near or above 150 million dollars; there were no big-budget PG-rated movies in the spring or summer of 2006; and there were only two PG-rated movies released in the spring of 2007.

Figure 6.5 Linked Graphs



You can also select observations in the scatter plot and see the MPAA ratings for those observations. You can use dynamically linked graphs such as these to investigate outliers or to explore relationships between movie ratings, budgets, and release dates. By creating graphs from a common data object, you enable exploratory analyses that are impossible with graphs that are not linked together.

Programming Tip: Create graphs from a common data object. All graphs that share the same data object are dynamically linked to each other.

6.7 The Plot Class: A Base Class for Graphs

The Plot class is a base class for all plots. You cannot directly instantiate an object of the Plot class, but any graph can call the nearly 100 methods in this class.

The most frequently used methods in this class are related to modifying plot attributes or to drawing on a plot, as discussed in Chapter 9, “Drawing on Graphs.” Appendix C lists frequently used methods in the Plot class.

6.8 The Data Table Class

If you want to display a tabular view of data, you can create an object of the DataTable class. A data table is dynamically linked to any other graph or table that shares the same data object. For example, the following statements create a data table that is dynamically linked to the two graphs that are shown in Figure 6.5:

```
/* create tabular view of data */  
declare DataTable dt;  
dt = DataTable.Create(dobj);
```

The data table might appear behind the program window because SAS/IML Studio tries to keep the program window in the foreground of a workspace. This can result in the program window obscuring tables or graphs. It can be confusing (and annoying) to create a data table that is obscured, so the following techniques are often useful:

- nudge the programming window away from the upper left corner of the workspace, or
- use the ActivateWindow method in the DataView class to force the data table to display on top of other windows.

The following statement displays the data table on top of all other windows:

```
dt.ActivateWindow();           /* display on top; make active */
```

The data table shows selected observations as highlighted. For example, Figure 6.6 shows a data table in which all PG-rated movies are selected.

Figure 6.6 A Data Table That Shows Selected Observations

	19	Title	MPAARating	ReleaseDate	Budget	US_Gross	World_Gross
[66]			Norm	Int	Int	Int	Int
1	■ x ²	Coach Carter	PG-13	14JAN2005	\$45.0	\$67.3	\$76.7
2	■ x ²	Elektra	PG-13	14JAN2005	\$65.0	\$24.4	\$56.4
3	■ x ²	Racing Stripes	PG	14JAN2005	\$30.0	\$49.8	\$93.8
4	■ x ²	Assault on Precinct 13	R	19JAN2005	\$30.0	\$20.0	\$36.0
5	■ x ²	Are We There Yet?	PG	21JAN2005	\$20.0	\$82.7	\$97.9
6	■ x ²	Alone in the Dark	R	28JAN2005	\$20.0	\$5.2	\$8.2
7	■ x ²	Hide and Seek	R	28JAN2005	\$25.0	\$51.1	\$123.1
8	■ x ²	Boogeyman	PG-13	04FEB2005	\$20.0	\$46.8	\$67.2
9	■ x ²	Hitch	PG-13	11FEB2005	\$55.0	\$177.8	\$366.8
10	■ x ²	Pooh's Heffalump Movie	G	11FEB2005	\$20.0	\$18.1	\$52.9
11	■ x ²	Because of Winn-Dixie	PG	18FEB2005	\$15.0	\$32.6	\$33.6
12	■ x ²	Constantine	R	18FEB2005	\$75.0	\$76.0	\$230.0
13	■ x ²	Son of the Mask	PG	18FEB2005	\$100.0	\$17.0	\$59.9
14	■ x ²	Cursed	PG-13	25FEB2005	\$35.0	\$19.3	\$25.1
15	■ x ²	Diary of a Mad Black Woman	PG-13	25FEB2005	\$5.5	\$50.4	\$50.4
16	■ x ²	Man of the House	PG-13	25FEB2005	\$50.0	\$19.7	\$22.1
17	■ x ²	Be Cool	PG-13	04MAR2005	\$75.0	\$55.8	\$94.8
18	■ x ²	Jacket	R	04MAR2005	\$28.5	\$6.3	\$15.5
19	■ x ²	Pacifier	PG	04MAR2005	\$56.0	\$113.0	\$198.0
20	■ x ²	Hostage	R	11MAR2005	\$75.0	\$34.6	\$77.6
21	■ x ²	Robots	PG	11MAR2005	\$80.0	\$126.2	\$260.7

6.9 The DataView Class: A Base Class for Graphs and Data Tables

The DataView class is the base class for the DataTable class and the Plot class. You cannot instantiate a DataView object, but you can call methods in this class from any data table and any graph.

The most frequently used methods in this class are related to positioning or showing a window. Less common methods include the methods related to *action menu*, which are discussed in Chapter 16, “Interactive Techniques.” Appendix C lists frequently used methods for the DataView class.

6.10 Passing Objects to IMLPlus Modules

In the IML procedure, each argument to a module is a SAS/IML matrix. In IMLPlus, you can also pass any Java object to a module, as shown in the following module definition:

```
/* pass Java object to a module */
start PrintNames(DataObject dobj);
  dobj.GetVarNames(varNames);
  print varNames;
finish;
```

```
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
run PrintNames(dobj);
```

The PrintNames module displays the names of all variables in a data object. For example, if you run the module by passing in a data object instantiated from the Sasuser.Movies data, then the module prints the names shown in Figure 6.2.

Recall from the section “Defining SAS/IML Modules” on page 72 that matrices are passed by reference to SAS/IML modules and that changes made to a matrix inside a module also change the matrix in the calling environment. IMLPlus modules behave similarly when you pass them a Java object or array: you can create or modify the object in an IMLPlus module. (However, primitive Java data types such as `int`, `double`, and `boolean` are passed by value.) For example, the following module creates a new data object and returns the data object to the main environment:

```
/* create data object inside module; use it outside of module */
start CreateNormalData(DataObject d, nObs, nVar);
    x = j(nObs, nVar);                                /* 1 */
    call randgen(x, "normal");                         /* 2 */
    d = DataObject.Create("Matrix", x);                 /* 3 */
finish;

/* begin main program */
declare DataObject dobj;
run CreateNormalData(dobj, 100, 5);                    /* call module */
declare DataTable ndt;
ndt = DataTable.Create(dobj);                          /* create the data table */
ndt.ActivateWindow();                                  /* display on top */
```

The module takes three arguments. The first argument is a data object with the local name `d`. The second and third arguments are scalar values that indicate the number of observations and the number of variables to create.

Programming Tip: You can pass IMLPlus objects as arguments to a module. When you define the module, specify the name of the class in addition to the name of the argument on the START statement.

The module consists of the following steps:

1. Allocate a matrix that has `nObs` rows and `nVar` columns.
2. Fill the matrix with values generated from the standard normal distribution.
3. Create a new data object from the matrix data and assign it to the variable `d`. (The variable names are automatically assigned to be A1, A2, A3, and so on.) The data object that was passed into the module is replaced by the new data object. The new data object becomes available to the calling environment upon the module’s return.

Figure 6.7 shows a data table linked to the data object that was instantiated inside the module. Notice that the data object is named `dobj` in the main program. The `dobj` object was “the null

object” (that is, undefined) prior to calling the module, but when the module returns, the data object contains the data created inside the module.

Figure 6.7 Data from Data Object Created within Module

	5	A1	A2	A3	A4	A5
		Int	Int	Int	Int	Int
1	x^2	-0.518064	0.0680106	1.07636	-0.402842	-1.32616
2	x^2	-1.64442	-0.632617	0.320593	1.06598	-0.912332
3	x^2	-0.391428	-0.291688	0.885911	0.386763	-0.635892
4	x^2	0.598769	0.460208	-0.0450018	-0.0500899	-1.56282
5	x^2	-0.623606	-1.1065	1.6097	-0.0632889	0.73627
6	x^2	-1.31169	-0.777936	1.08441	1.08172	0.754304
7	x^2	0.0671497	-0.116315	2.2547	0.139838	-0.755494
8	x^2	-0.486328	0.588107	1.94486	0.857253	-1.82671
9	x^2	0.171682	-0.300966	-1.1788	-0.336611	-0.247146
10	x^2	0.38221	-0.0435254	-0.200008	-0.950313	-0.591924
11	x^2	0.0817885	-1.1389	-1.04928	0.777281	-0.165358
12	x^2	-0.236545	-1.29936	0.390157	-1.34831	0.520695
13	x^2	-0.712655	0.0458695	0.566029	-0.313448	1.20332
14	x^2	-1.10731	0.80316	0.340709	-1.29629	-1.18954
15	x^2	1.54769	0.910296	-0.496414	-0.61667	-0.0927346
16	x^2	-0.717609	-0.648569	-1.10577	1.51265	0.295093
17	x^2	1.26348	0.734982	0.867899	0.318915	-1.19846
18	x^2	-0.0735598	-1.09613	-1.31104	0.412888	-0.203104
19	x^2	-0.220291	-0.155763	1.81443	-0.262905	1.63971
20	x^2	0.18378	0.334892	0.485095	0.599964	-0.356811
21	x^2	-1.43824	-0.938205	-1.01184	-2.12734	0.837938
22	x^2	2.33933	0.412057	0.827836	0.391277	-0.866279

Programming Tip: You can create a matrix or object within an IMLPlus module and return that object to the calling routine.

6.11 Using a Base Class in a Module

The signature of a module determines what kind of objects can be passed to it. If you want a module to accept several different kinds of graphs, it is often useful to specify that the module argument is an object of the Plot class. Because the Plot class is a base class for all graphs, you can call the module on any graph provided that the module calls only methods in the Plot class. For example, the following statements define a module with a Plot parameter and then call the module twice, once with a scatter plot and once with a bar chart.

```
/* define module whose argument is a base-class object */
start ConfigPlot(Plot p);
  p.ShowReferenceLines();
  p.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
  p.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);
finish;
```

```

/* begin main program */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

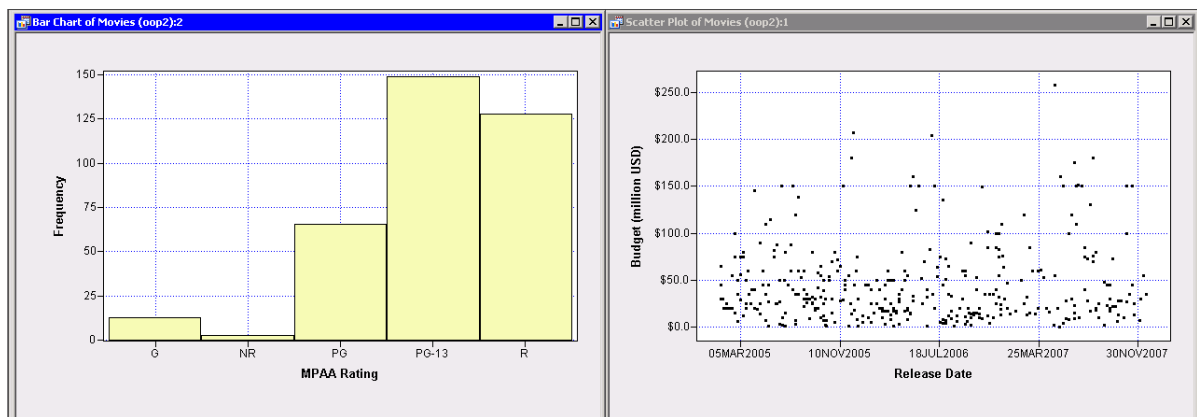
declare ScatterPlot scat;
scat = ScatterPlot.Create(dobj, "ReleaseDate", "Budget");
run ConfigPlot(scat);

declare BarChart bar;
bar = BarChart.Create(dobj, "MPAARating");
run ConfigPlot(bar);

```

The ConfigPlot module adds vertical and horizontal reference lines to the graph and changes the axis labels to show variable labels instead of variable names. The resulting graphs are shown in Figure 6.8.

Figure 6.8 Graphs Configured by a Module That Uses a Base Class



Programming Tip: If a module accepts a graph as a parameter, and if the module only calls methods in a base class such as Plot, define the module parameter to be a Plot object rather than a concrete class such as a scatter plot or histogram.

Chapter 7

Creating Statistical Graphs

Contents

7.1	Overview of Creating Statistical Graphs	144
7.2	The Source of Data for a Graph	144
7.3	Bar Charts	145
7.3.1	Creating a Bar Chart from a Vector	145
7.3.2	Creating a Bar Chart from a Data Object	146
7.3.3	Modifying the Appearance of a Graph	146
7.3.4	Frequently Used Bar Chart Methods	147
7.4	Histograms	149
7.4.1	Creating a Histogram from a Vector	149
7.4.2	Creating a Histogram from a Data Object	150
7.4.3	Frequently Used Histogram Methods	150
7.5	Scatter Plots	153
7.5.1	Creating a Scatter Plot from Vectors	154
7.5.2	Creating a Scatter Plot from a Data Object	154
7.6	Line Plots	155
7.6.1	Creating a Line Plot for a Single Variable	155
7.6.2	Creating a Line Plot for Several Variables	156
7.6.3	Creating a Line Plot with a Classification Variable	158
7.6.4	Frequently Used Line Plot Methods	161
7.7	Box Plots	161
7.7.1	Creating a Box Plot	162
7.7.2	Creating a Grouped Box Plot	164
7.7.3	Frequently Used Box Plot Methods	165
7.8	Summary of Graph Types	167
7.9	Displaying the Data Used to Create a Graph	168
7.10	Changing the Format of a Graph Axis	169
7.11	Summary of Creating Graphs	172
7.12	References	172

7.1 Overview of Creating Statistical Graphs

This chapter introduces some of the statistical graphs available in IMLPlus. It describes how to create these graphs from SAS/IML vectors and from a data object. The chapter also introduces the concept of calling methods in order to modify simple attributes of graphs.

The chapter uses some object-oriented terminology that is described in Chapter 6, “[Understanding IMLPlus Classes](#).” However, you do not need to master object-oriented programming in order to follow the examples in this chapter. By imitating the syntax in the examples, you can create and use simple graphs in your own IMLPlus programs. The following list summarizes important terms and concepts:

- A *class* is a “template” that defines a graph, including what data are required to create it and what functions can be used to modify the graph. A bar chart is defined by the BarChart class, a histogram is defined by the Histogram class, and so on.
- Graphs are created and modified by functions called *methods*. Each class has its own methods, although some methods are common to several classes.
- An *object* is a programming variable that refers to a class. In IMLPlus, you can use the **declare** keyword to specify that a variable in your program is an object of some class. For example, the following statement declares that **bar** is an object of the BarChart class:

```
declare BarChart bar;
```

- Each graph is attached to an in-memory copy of the data. This in-memory copy is known as a “data object” since it is an object of the DataObject class. Graphs that are attached to the same data object are dynamically linked to each other. The DataObject class is discussed in Chapter 8, “[Managing Data in IMLPlus](#).”

The examples in this chapter do not run in PROC IML, so be sure you are using SAS/IML Studio.

7.2 The Source of Data for a Graph

In general, there are two ways to create a graph in IMLPlus: from SAS/IML vectors and from an object of the DataObject class. The DataObject class is introduced in [Chapter 6](#) and is described more fully in [Chapter 8](#).

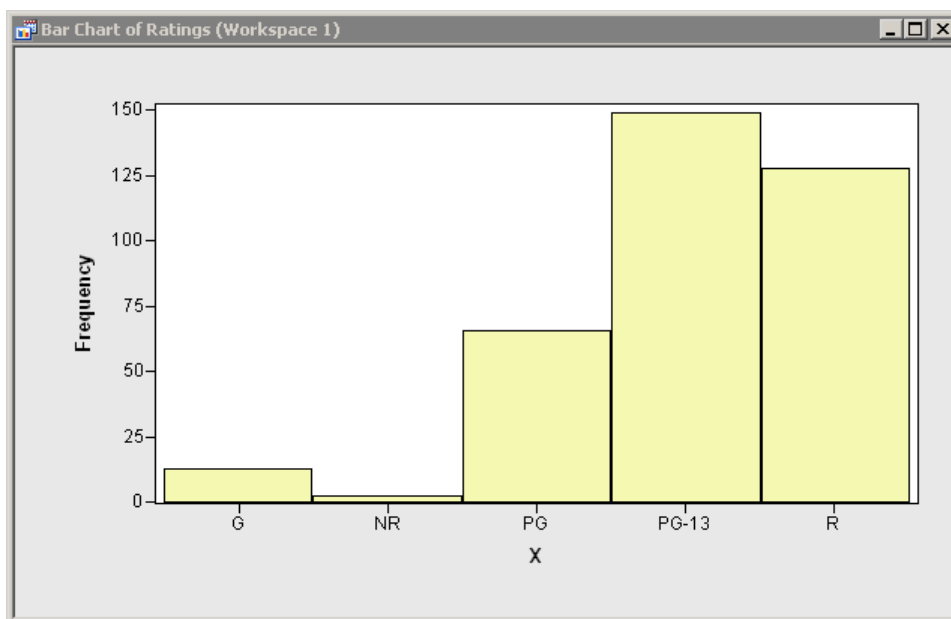
If you already have the data in SAS/IML vectors and you just want a quick and convenient way to visualize the data, you can create graphs directly from the vectors. However, if you want to create several graphs that are dynamically linked to each other, you should create the graphs from a common data object.

Programming Tip: You can create IMLPlus graphs from SAS/IML vectors to quickly visualize the contents of the vectors.

7.3 Bar Charts

The section “Analyzing Observations by Categories” on page 68 shows how to read a categorical variable from a SAS data set and how to use the LOC and UNIQUE functions to count and display a table that shows the frequency of each category in the variable. A bar chart is a graphical representation of the same information. For example, [Figure 7.1](#) displays the number of movies in the Movies data set for each rating category. This graphically displays the tabular data shown in [Figure 3.13](#).

Figure 7.1 Bar Chart of Movie Ratings



7.3.1 Creating a Bar Chart from a Vector

You can create [Figure 7.1](#) from a vector that contains the Motion Picture Association of America (MPAA) rating for each movie. The following program shows one way to create the bar chart:

```
/* create a bar chart from a vector of data */
use Sasuser.Movies;
read all var {"MPAARating"};          /* 1 */
close Sasuser.Movies;

declare BarChart bar;                 /* 2 */
bar = BarChart.Create("Ratings", MPAARating); /* 3 */
```

The program contains three main steps:

1. Create the vector **MPAARating** from the data in the MPAARating variable.
2. Use the **declare** keyword to specify that **bar** is an object of the BarChart class.
3. Create the BarChart object from the data in the **MPAARating** vector. The first argument to this method is a string that names the associated data object; the second argument is the vector that contains the data.

The Create method displays the graph, as shown in [Figure 7.1](#). The bar chart shows that most movies made in the US between 2005–2007 were rated PG-13 or R. Only a small number of movies were rated G. There were also a smaller number of “not rated” (NR) movies that were never submitted to the MPAA for rating.

7.3.2 Creating a Bar Chart from a Data Object

If the data are in a data object, you can create a similar bar chart.

Create a data object from the Movies data by using the following statements:

```
/* create data object from SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
```

The following statements create a bar chart:

```
/* create a bar chart from a data object */
declare BarChart bar2;
bar2 = BarChart.Create(dobj, "MPAARating");
```

The program creates the BarChart object, **bar2**, from the data in the MPAARating variable. The first argument to this method is a data object; the second argument is the name of the variable that contains the data.

The bar chart created in this way is linked to the data object and to all other graph and data tables that share the same data object. It looks like [Figure 7.1](#), except that the horizontal axis is labeled by the name of the MPAARating variable.

7.3.3 Modifying the Appearance of a Graph

You can call BarChart class methods to modify the appearance of the bar chart. Each method causes the graph to update.

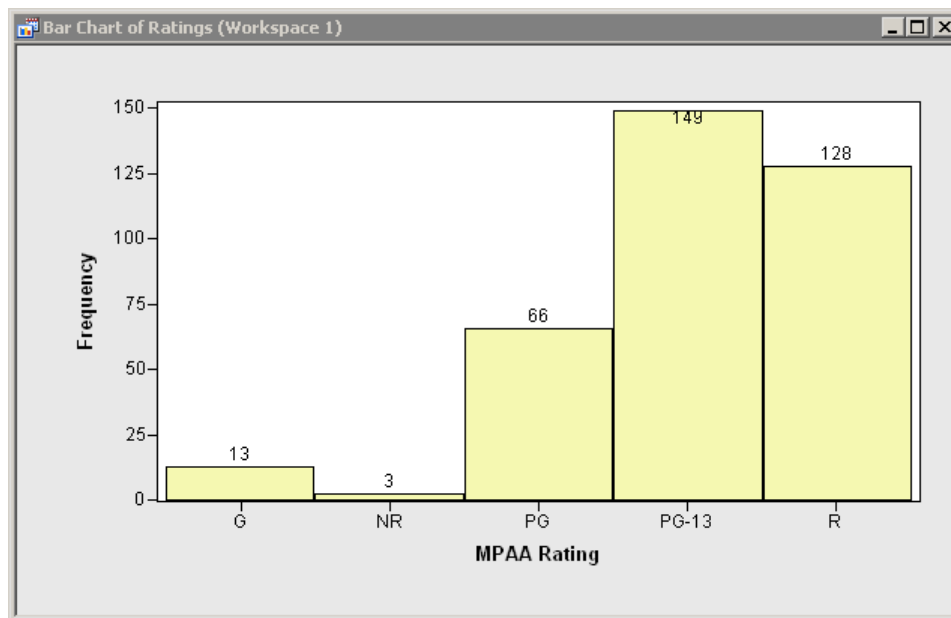
Programming Tip: A graph is created and displayed when the Create method is executed. You can call additional methods to modify attributes of the graph.

For example, suppose you want to modify the appearance of the graph in [Figure 7.1](#). This graph corresponds to the `bar` object. You might want to change the label for the horizontal axis (which currently reads “X”) and add labels to the bar chart that indicate the number of observations in each MPAA category. You need to call methods in the `BarChart` class. The `bar` object is a variable in the IMLPlus program, and therefore you can call methods on the `bar` object that change the appearance of the bar chart. You can call methods accessible to the `BarChart` class by using the `ObjectName.MethodName()` syntax, as shown in the following statements:

```
bar.SetAxisLabel(XAXIS, "MPAA Rating"); /* change the axis label */
bar.ShowBarLabels(); /* show counts for each bar */
```

The first statement sets the label for the horizontal axis to be “MPAA Rating” instead. The second statement causes the bar chart to display the counts for each category. Both statements are optional, but result in a more interpretable graph, as shown in [Figure 7.2](#). The bar chart updates itself after each method call.

Figure 7.2 Modified Bar Chart of Movie Ratings



7.3.4 Frequently Used Bar Chart Methods

Each graph type has methods that control the appearance of the graph. [Table 7.1](#) summarizes frequently used methods in the `BarChart` class.

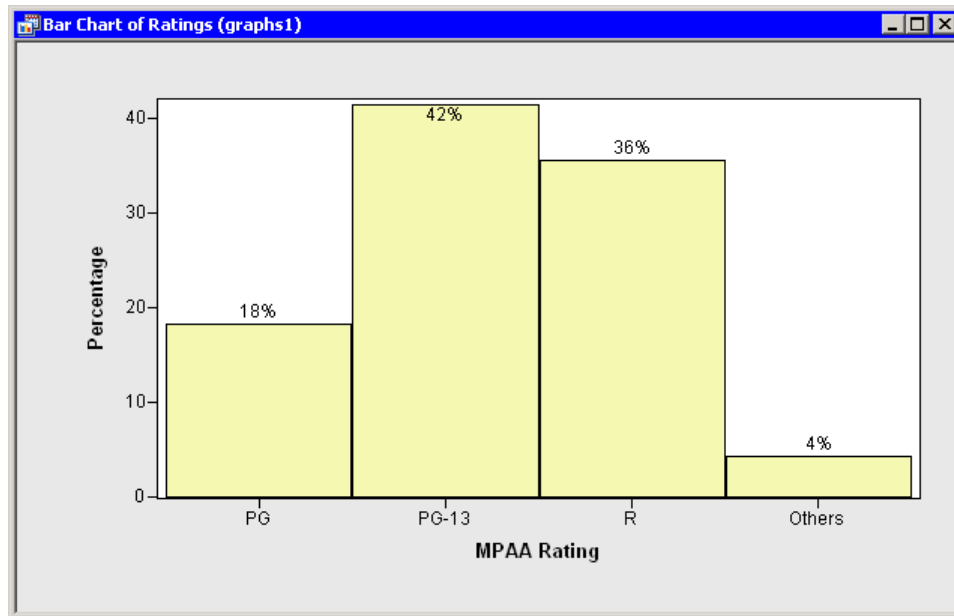
Table 7.1 Frequently Used Methods in the BarChart Class

Method	Description
BarChart.Create	Creates a bar chart
SetOtherThreshold	Specifies a cutoff percentage for determining which categories are combined into a generic category called “Others”
ShowBarLabels	Shows or hides labels of the count or percentage for each category
ShowPercentage	Specifies whether the graph’s vertical axis displays counts or percentages

The complete set of BarChart methods are documented in the SAS/IML Studio online Help. To view the online Help, select **Help►Help Topics** from the SAS/IML Studio main menu, and then select the chapter titled “IMLPlus Class Reference.”

The result of the ShowBarLabels method is shown in Figure 7.2. The other methods are used in the following statements to create Figure 7.3.

```
bar.ShowPercentage(true);      /* display percents          */
bar.SetOtherThreshold(5.0);    /* merge bars that have small counts */
```

Figure 7.3 Result of Calling Bar Chart Methods

The ShowPercentage method specifies the units of the vertical axis: frequency or percentage. The SetOtherThreshold method is useful when there are many categories that contain a relatively small number of observations. The argument to the method is a percentage in the range [0, 100]. For example, 5.0 means 5% of observations. Any categories that contain fewer than 5% of the total observations are combined into an “Others” category, as shown in Figure 7.3. This enables you to more easily explore and analyze data in the relatively large categories.

The BarChart class can also access methods provided by any of its base classes. For example,


```

declare Histogram hist;                                /* 2 */
hist = Histogram.Create("Budget", Budget);             /* 3 */
hist.SetAxisLabel(XAXIS, "Budget (million $)"); /* change axis label */

```

The program contains three main steps:

1. Create the vector **Budget** from the data in the Budget variable.
2. Use the **declare** keyword to specify that **hist** is an object of the Histogram class.
3. Create the Histogram object from the data in the **Budget** vector. The first argument to this method is a string that names the associated data object; the second argument is the vector that contains the data.

The resulting histogram is shown in [Figure 7.4](#). The histogram shows that about two-thirds of movies in the years 2005–2007 had budgets less than 50 million dollars. The distribution of budgets has a long tail; several movies had budgets more than 150 million dollars, and one movie had a budget in excess of 250 million dollars.

7.4.2 Creating a Histogram from a Data Object

If the data are in a data object, you can create a similar histogram.

Create a data object from the Movies data by using the following statements:

```

/* create data object from SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

```

The following statements create a histogram:

```

/* create a histogram from a data object */
declare Histogram hist2;
hist2 = Histogram.Create(dobj, "Budget");

```

The program creates the Histogram object, **hist2**, from the data in the Budget variable. The first argument to this method is a data object; the second argument is the name of the variable that contains the data.

The histogram created in this way is linked to the data object and to all other graph and data tables that share the same data object. It looks like [Figure 7.4](#), except that the horizontal axis is labeled by the name of the Budget variable.

7.4.3 Frequently Used Histogram Methods

Each graph type has methods that control the appearance of the graph. [Table 7.2](#) summarizes frequently used methods in the Histogram class.

Table 7.2 Frequently Used Methods in the Histogram Class

Method	Description
Histogram.Create	Creates a histogram
ReBin	Specifies the offset and width for the histogram bins
ShowBarLabels	Shows or hides labels of the count, percentage, or density for each category
ShowDensity	Specifies whether the graph's vertical axis displays counts or density
ShowPercentage	Specifies whether the graph's vertical axis displays counts or percentages

To view the complete documentation for graph methods, select **Help►Help Topics** from the SAS/IML Studio main menu, and then select the chapter titled “IMLPlus Class Reference.”

The ShowBarLabels and ShowPercentage methods in the Histogram class behave identically to their counterparts in the BarChart class and are discussed in [Section 7.3.4](#). The ShowDensity method scales the vertical axis so that the sum of the histogram bars is unity. This scale is useful when you want to overlay a parametric or kernel density estimate on the histogram, as described in the section “[Case Study: Plotting a Density Estimate](#)” on page 214.

The *SAS/IML Studio for SAS/STAT Users* documentation has a chapter on “Adjusting Axes and Ticks” that describes how to use the ReBin method. The ReBin method is useful when you want to specify a new bin width for the histogram. For example, you might want bin widths for ages of adults to be an integral multiple of five. Alternatively, you might want to choose a bin width that is optimal for exhibiting features of the data density.

The following statements modify the histogram shown in [Figure 7.4](#), which corresponds to the `hist` object. The program compares the default histogram bin widths (computed according to an algorithm by Terrell and Scott (1985)) with robust bin widths suggested by Freedman and Diaconis, as presented in Scott (1992, p. 75). The robust bin widths are computed as $2.603 \text{ IQR } n^{-1/3}$, where IQR is the interquartile range of the data and n is the number of nonmissing observations in the data.

The following statements continue the program in [Section 7.4.1](#):

```

/* calculate new histogram bins */
/* get current anchor and bin width: Terrell and Scott (1985) */
x0 = hist.GetAxisTickAnchor(XAXIS);          /* 4 */
h0 = hist.GetAxisTickUnit(XAXIS);

/* Freedman-Diaconis robust rule (1981) */
nNonmissing = sum(Budget^=.);                /* 5 */
q = quartile(Budget);                        /* 6 */
IQR = q[4] - q[2];
h = 2.603 * IQR * nNonmissing##(-1/3);       /* 7 */
print x0 h0 h;

h = round(h, 1);                            /* round to integer */
hist.ReBin(x0, h);                          /* 8 */

```

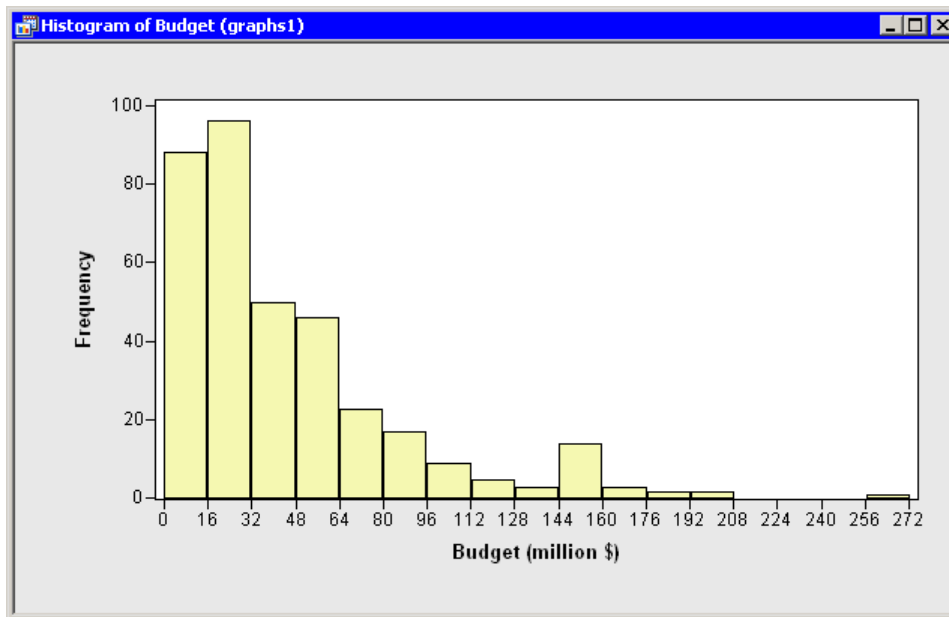
Figure 7.5 Tick Anchor, Default Bin Width, and New Bin Width

<code>x0</code>	<code>h0</code>	<code>h</code>
0	25	15.931816

The previous statements implement five new steps:

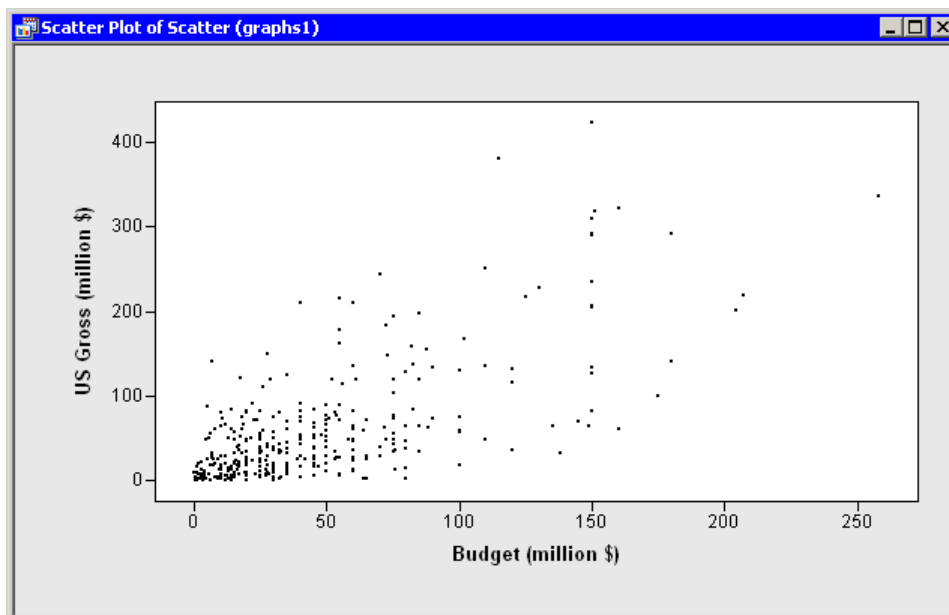
4. Get the current histogram anchor (`x0`) and bin width (`h0`). The `GetAxisTickAnchor` and `GetAxisTickUnit` methods are provided by the `Plot` class, which is a base class of the `Histogram` class.
5. Compute the number of nonmissing observations (`nNonMissing`). Although the `Budget` variable does not contain missing values, it is a good idea to write programs that handle missing values, as discussed in the section “[Handling Missing Values](#)” on page 65.
6. Call the `QUARTILE` module (which is part of the `IMLMLIB` module library) to compute the five-number summary of the data. The vector `q` contains the minimum, first quartile, second quartile (median), third quartile, and maximum values of the data. The interquartile range, `IQR`, is computed as `q[4]-q[2]`.
7. Compute the Freedman-Diaconis bin width. The bin width is approximately 15.9 and is stored in the matrix `h`.
8. Call the `ReBin` method with the existing anchor and the new bin width. For this example, the bin width is rounded to the nearest integer so that the axis tick labels look nicer.

The revised histogram is shown in [Figure 7.6](#). The histogram looks similar to [Figure 7.4](#), but the new Freedman-Diaconis bandwidth is substantially smaller than the default bin width, as shown in [Figure 7.5](#). With the smaller bin width, the distribution appears to be bimodal, with one peak near 20 million dollars and another smaller peak near 150 million dollars.

Figure 7.6 Rebinned Histogram of Movie Budgets

7.5 Scatter Plots

A scatter plot displays the values of two variables plotted against each other. In many cases it enables you to explore bivariate relationships in your data. For example, [Figure 7.7](#) plots the value of the Budget variable versus the value of the US_Gross variable for movies in the Movies data set.

Figure 7.7 Scatter Plot of Gross Revenue versus Movie Budgets

7.5.1 Creating a Scatter Plot from Vectors

You can create a scatter plot from data in two vectors in much the same way as you create a histogram. The data in a scatter plot is usually continuous numeric data. However, the IMLPlus scatter plot also supports categorical data (including character data) for either axis.

The following program shows one way to create a scatter plot of the Budget variable versus the US_Gross variable:

```
/* create a scatter plot from vectors of data */
use Sasuser.Movies;
read all var {"Budget" "US_Gross"};
close Sasuser.Movies;

declare ScatterPlot p;
p = ScatterPlot.Create("Scatter", Budget, US_Gross);
p.SetAxisLabel(XAXIS, "Budget (million $)");
p.SetAxisLabel(YAXIS, "US Gross (million $)");
```

The program contains the same steps as the program in [Section 7.4](#), except that this program creates an object of the ScatterPlot class. The resulting scatter plot is shown in [Figure 7.7](#).

The scatter plot shows that these two variables are correlated: movies produced with relatively modest budgets typically bring in modest revenues, whereas big-budget movies often bring in larger revenues. However, the graph also shows movies that generated revenue that is disproportionate to their budgets. Some movies with small budgets generated a relatively large gross revenue. Other movies were disappointments: their revenue was only a fraction of the cost of producing the movie.

7.5.2 Creating a Scatter Plot from a Data Object

If the data are in a data object, you can create a similar scatter plot.

Create a data object from the Movies data by using the following statements:

```
/* create data object from SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
```

The following statements create a scatter plot:

```
/* create a scatter plot from a data object */
declare ScatterPlot p2;
p2 = ScatterPlot.Create(dobj, "Budget", "US_Gross");
```

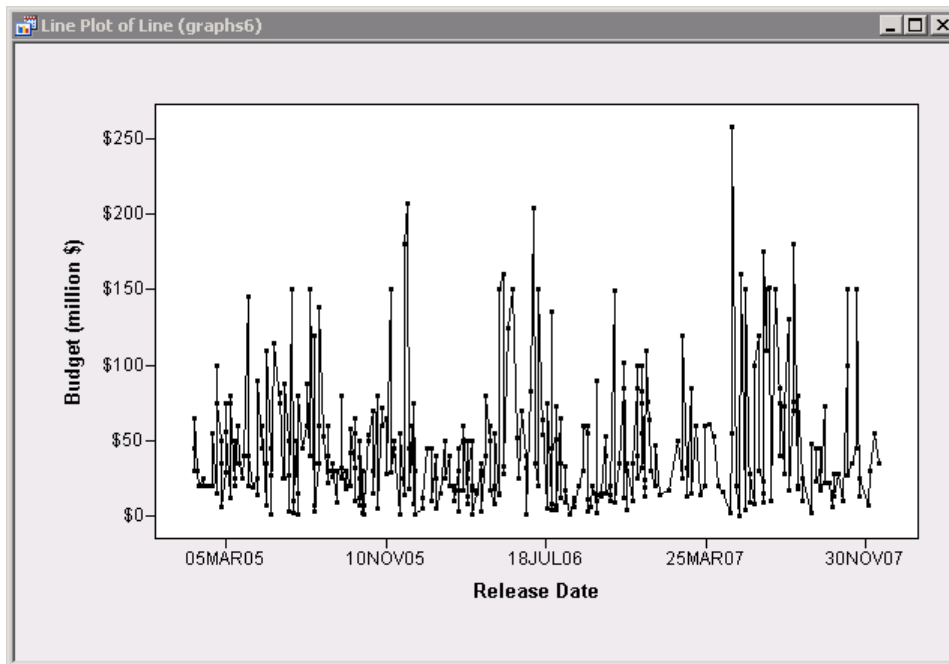
The program creates the ScatterPlot object, `p2`, from the data in the Budget and US_Gross variables. The first argument to this method is a data object; the second and third arguments are the name of the variables that contains the data.

The scatter plot created in this way is linked to the data object and to all other graph and data tables that share the same data object.

7.6 Line Plots

A line plot displays the values of one or more variables plotted against another variable, which is often a time variable. A line plot is useful for investigating how quantities change over time. For example, [Figure 7.8](#) plots the value of the Budget variable versus the value of the ReleaseDate variable for movies in the Movies data set.

Figure 7.8 Line Plot of Movie Budgets versus Date of Release



You can create a line plot from data in two (or more) vectors in the same way as you create a scatter plot. Often, each variable in a line plot is continuous. However, the IMLPlus line plot also supports categorical data (including character data) for either axis.

7.6.1 Creating a Line Plot for a Single Variable

The simplest line plot shows a single variable plotted against time.

7.6.1.1 Creating a Line Plot from Vectors

The following program shows one way to create a line plot of the Budget variable versus the ReleaseDate variable:

```

/* create a line plot from vectors of data */
use Sasuser.Movies;
read all var {"ReleaseDate" "US_Gross"};
close Sasuser.Movies;

declare LinePlot line;
line = LinePlot.Create("Line", ReleaseDate, US_Gross);
line.SetAxisLabel(XAXIS, "Release Date");
line.SetAxisLabel(YAXIS, "Budget (million $)");

```

The program contains the same steps as the program in [Section 7.5](#). (Not shown are statements that place a DATE7. format on the horizontal axis; these statements are described in [Section 7.10](#).) The resulting line plot is shown in [Figure 7.8](#).

The line plot shows that movies with large budgets are often released in the May–July and November–December time periods.

7.6.1.2 Creating a Line Plot from a Data Object

If the data are in a data object, you can create a similar line plot.

Create a data object from the Movies data by using the following statements:

```

/* create data object from SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

```

The following statements create a line plot:

```

/* create a line plot from a data object */
declare LinePlot line2;
line2 = LinePlot.Create(dobj, "ReleaseDate", "US_Gross");

```

The program creates the LinePlot object, `line2`, from the data in the ReleaseDate and US_Gross variables. The first argument to this method is a data object; the second and third arguments are the names of the variables that contain the data.

The line plot created in this way is linked to the data object and to all other graph and data tables that share the same data object.

7.6.2 Creating a Line Plot for Several Variables

Although [Figure 7.8](#) shows a plot of a single Y variable versus time, you can use a line plot to display several Y variables simultaneously.

7.6.2.1 Creating a Line Plot from Vectors

The following statements create a graph that displays three functions:

```

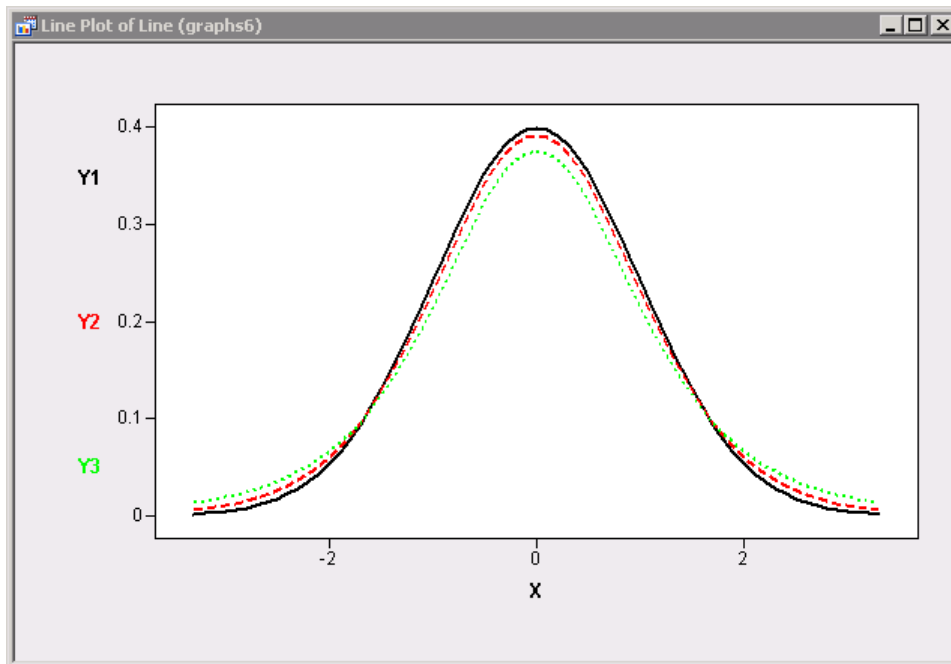
/* create a line plot of several Y variables */
x = t(do(-3.3, 3.3, 0.1));    /* evenly spaced points (t=transpose) */
normal = pdf("normal", x);    /* evaluate normal density at x      */
t4 = pdf("t", x, 4);          /* evaluate t distrib with 4 d.o.f   */
t12 = pdf("t", x, 12);        /* evaluate t distrib with 12 d.o.f  */

declare LinePlot lp;
lp = LinePlot.Create("Line", x, normal || t12 || t4);
lp.ShowObs(false);           /* do not show observation markers   */
lp.SetLineWidth(2);          /* set all line widths to 2          */
lp.SetLineStyle(2, DASHED);   /* set style of second line          */
lp.SetLineStyle(3, DOTTED);   /* set style of third line           */

```

The graph is shown in [Figure 7.9](#). The program uses the PDF function to evaluate three different probability density functions on a set of evenly spaced values: the standardized normal distribution, a Student's t distribution with four degrees of freedom, and a t distribution with 12 degrees of freedom. The line plot enables you to compare these three distributions. The markers are not displayed so that the three curves are easier to see. The line plot has several methods that enable you to control the color, style, and width of each line. In this program, the `SetLineWidth` method sets the width of all lines, and the `SetLineStyle` method sets the line styles for the second and third lines.

Figure 7.9 Line Plot of Probability Density Functions



7.6.2.2 Creating a Line Plot from a Data Object

The data for the previous line plot do not exist in a SAS data set. Nevertheless, you can create a data object from the data in the vectors and then create a line plot from the data object.

Assume that the `x`, `normal`, `t4`, and `t12` vectors contain the data as defined in the previous section. You can create a data object for these data by using the following statements:

```
/* create data object from vectors */
varNames = {"x" "pdf_normal" "pdf_t12" "pdf_t4"};
declare DataObject dobjVec;
dobjVec = DataObject.Create("Data", varNames, x || normal || t12 || t4 );
```

Notice that the data object is created from a matrix by using the `Create` method. The first argument is an identifier for the data object. This string is used when the data object is written to a data set. It is also used in the title bar of a graph to identify the data object that underlies the graph. The second argument is a list of variable names. The third argument is a matrix of data; each column corresponds to a variable.

The following statements create a line plot:

```
/* create a line plot of several variables from a data object */
declare LinePlot lp2;
lp2 = LinePlot.Create(dobjVec, "x", {"pdf_normal" "pdf_t12" "pdf_t4"});
```

The program creates the `LinePlot` object, `lp2`, from variables in the data object. Each PDF variable is plotted against the `x` variable. The first argument to the `Create` method is a data object; the second and third arguments are the names of the variables that contain the data. You can continue the program by calling the `ShowObs`, `SetLineWidth`, and `SetLineStyle` methods that are used in the previous section.

The line plot created in this way is linked to the data object and to all other graph and data tables that share the same data object.

7.6.3 Creating a Line Plot with a Classification Variable

The `LinePlot` class also enables you to create a line plot by specifying a single Y variable, one or more classification variables, and a time variable. The joint levels of the classification variables determine which observations are joined in the line plot. This is useful for comparing two or more groups. Line plots of this type are created by using the `CreateWithGroup` method of the `LinePlot` class.

7.6.3.1 Creating a Line Plot from Vectors

Recall that the `Sex`, `Violence`, and `Profanity` variables in the `Movies` data set represent a measurement of the amount of sexual content, violence, and profane language (respectively) for each movie, as judged by the raters at the kids-in-mind.com Web site. It is interesting to investigate how these quantities relate to the MPAA rating of a movie.

The following statements create a line plot that enables you to compare the values of a quantity as a function of time for movies rated G, PG, PG-13, and R. The quantity is the total amount of sexual content, violence, and profane language for each movie.

```

/* create a line plot by specifying a classification (group) variable */
use Sasuser.Movies where (MPAARating^="NR");
read all var {"ReleaseDate" "MPAARating" "Sex" "Violence" "Profanity"};
close Sasuser.Movies;

Total = Sex + Violence + Profanity;    /* score for "adult situations" */

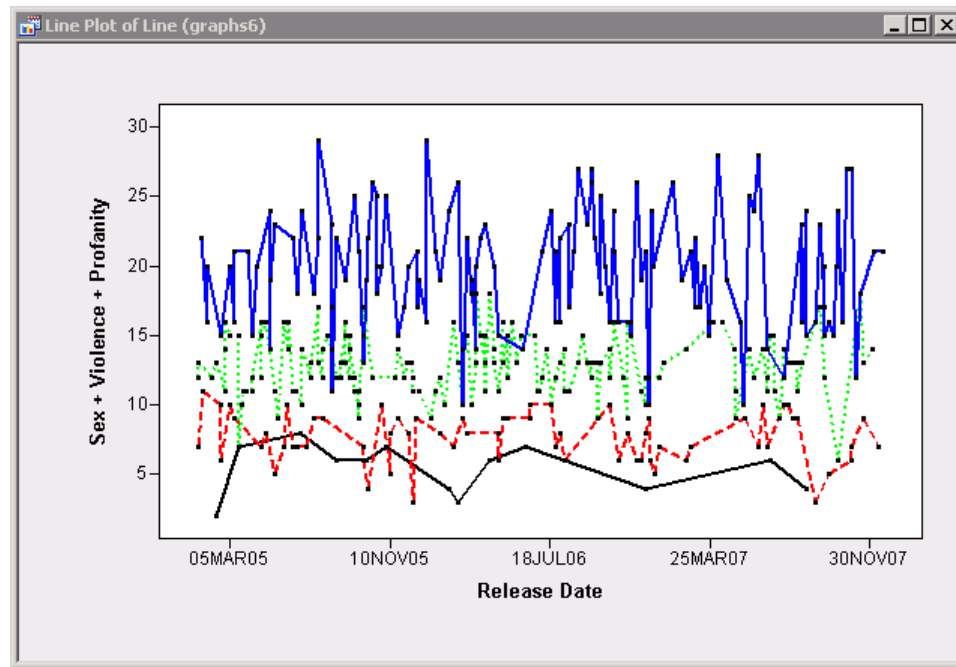
declare LinePlot lpg;
lpg = LinePlot.CreateWithGroup("Line",
                                ReleaseDate, /* x coordinate          */
                                Total,        /* y quantity to plot         */
                                MPAARating    /* classification variable    */
                                );
lpg.SetAxisLabel(XAXIS, "Release Date");
lpg.SetAxisLabel(YAXIS, "Sex + Violence + Profanity");
lpg.SetLineWidth(2);
lpg.SetLineStyle(2, DASHED);
lpg.SetLineStyle(3, DOTTED);

```

The graph is shown in [Figure 7.10](#). The **Total** vector contains the sum of the **Sex**, **Violence**, and **Profanity** vectors. This is a measure of the level of “adult situations” in each movie. As shown in [Figure 7.10](#), the R-rated movies have the highest average adult situation score, with a mean near 20. Movies rated PG-13 have a mean score near 13, whereas PG-rated movies have a mean score near 7.5. The G-rated movies have a mean score of about 5. The four time series appear to be stationary in time. That is, there does not seem to be any trends in the time period spanned by the data.

The graph also indicates that the MPAA rating board and the raters at the kids-in-mind.com Web site appear to be using consistent standards in evaluating movies. Consequently, the **Total** vector might serve as a useful discriminant function for predicting a movie’s MPAA rating based on the measures of sexual content, violence, and profanity as determined by the Web site. For example, you could use the Web site ratings to predict MPAA ratings for the three movies in the data that were not rated by the MPAA.

Lastly, the graph shows that there was a two-month period (March–April, 2007) during which no PG- or G-rated movies were released.

Figure 7.10 Line Plot of Adult Content for Each MPAA Rating

7.6.3.2 Creating a Line Plot from a Data Object

If the data are in a data object, you can create a similar line plot. There are two ways to create a data object that contains the Total variable, which is the sum of other variables. The simplest approach (used in this section) is to use the DATA step to create a new data set that contains the Total variable. (The alternative approach, used in the section “[Variable Transformations](#)” on page 181, is to use the GetVarData and AddVar methods in the DataObject class.)

Recall from [Chapter 4](#) that you can use the SUBMIT statement to execute SAS statements. The following statements call the DATA step and create a data object from the resulting data set:

```
/* use the DATA step to transform a variable */
submit;
data NewMovies;
    set Sasuser.Movies;
    Total = Sex + Violence + Profanity; /* score for "adult situations" */
run;
endsubmit;

declare DataObject dobjNew;
dobjNew = DataObject.CreateFromServerDataSet("Work.NewMovies");
```

The following statements create a line plot:

```

/* create a line plot with a group variable from a data object */
declare LinePlot lpg2;
lpg2 = LinePlot.CreateWithGroup(dobjNew,
                                "ReleaseDate", /* x coordinate      */
                                "Total",        /* y quantity to plot    */
                                "MPAARating"    /* classification variable */
                                );

```

The program creates the LinePlot object, **lpg2**, from variables in the data object. The first argument to the CreateWithGroup method is the data object. The second and third arguments are the names of the variables that contain the data. The third argument names the variables whose levels define the groups used in the graph.

The line plot created in this way is linked to the data object and to all other graph and data tables that share the same data object.

7.6.4 Frequently Used Line Plot Methods

Table 7.3 summarizes frequently used methods in the LinePlot class.

Table 7.3 Frequently Used Methods in the LinePlot Class

Method	Description
LinePlot.Create	Creates a line plot with one or more Y variables
LinePlot.CreateWithGroup	Creates a line plot in which each line is determined by categories of a grouping variable
AddVar	Adds a new Y variable to an existing line plot
ConnectPoints	Specifies whether to connect observations for each line
SetLineAttributes	Specifies the color, style, and width of a line
SetLineColor	Specifies the color of a line
SetLineStyle	Specifies the style of a line
SetLineWidth	Specifies the width of a line
SetLineMarkerShape	Specifies the marker shape for a line
ShowPoints	Specifies whether to show the markers for a line

To view the complete documentation for graph methods, select **Help►Help Topics** from the SAS/IML Studio main menu, and then select the chapter titled “IMLPlus Class Reference.”

7.7 Box Plots

A box plot is a schematic representation of the distribution of a variable. The left graph in [Figure 7.11](#) shows a box plot for the US_Gross variable restricted to the G-rated movies in the Movies

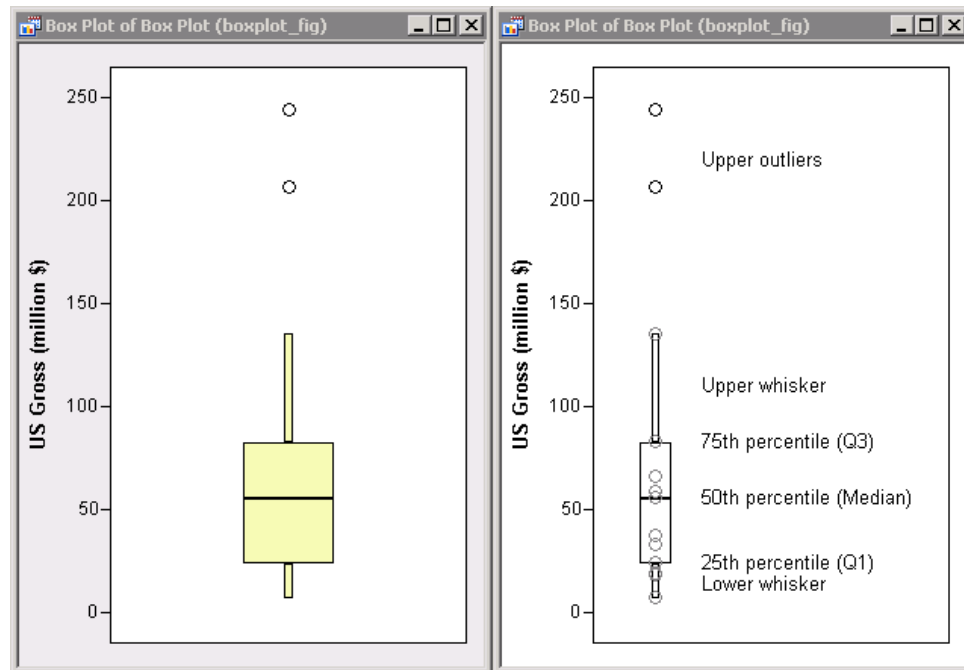
data set. (The left graph also shows the default appearance of a box plot in SAS/IML Studio.) The right graph shows a box plot for the same data, but labels important features of the box plot.

A box plot enables you to find the *five-number summary* of the data: the minimum value, the 25th percentile, the 50th percentile, the 75th percentile, and the maximum value. (The 25th, 50th, and 75th percentiles are also called the first quartile (Q1), the median, and the third quartile (Q3), respectively.) For example, Figure 7.11 indicates the five-number summary for the US gross revenues for G-rated movies. The minimum value is about 8 million dollars. The quartiles of the data are approximately 25, 55, and 85 million dollars. The maximum value is about 245 million dollars.

A box plot has a wide central box that contains 50% of the data. The upper and lower edges of the box are positioned at the 25th and 75th percentiles of the data. A line segment inside the box indicates the median value of the data. The height of the box indicates the interquartile range (IQR), which is a robust estimate of the scale of the data.

Above and below the main box are two thinner boxes that are called *whiskers*. The lengths of the whiskers are determined by the IQR. Specifically, the upper (respectively, lower) whisker extends to an observation whose distance from the 75th (respectively, 25th) percentile does not exceed 1.5 IQR. If there are observations whose distance to the main box exceeds 1.5 IQR, these observations are plotted individually and are called univariate outliers.

Figure 7.11 Main Features of a Box Plot



7.7.1 Creating a Box Plot

You can create a box plot that schematically shows the distribution of a single variable. Figure 7.12 shows a box plot for the US_Gross variable for all movies in the Movies data set.

7.7.1.1 Creating a Box Plot from a Vector

The following program shows one way to create the box plot shown in [Figure 7.12](#):

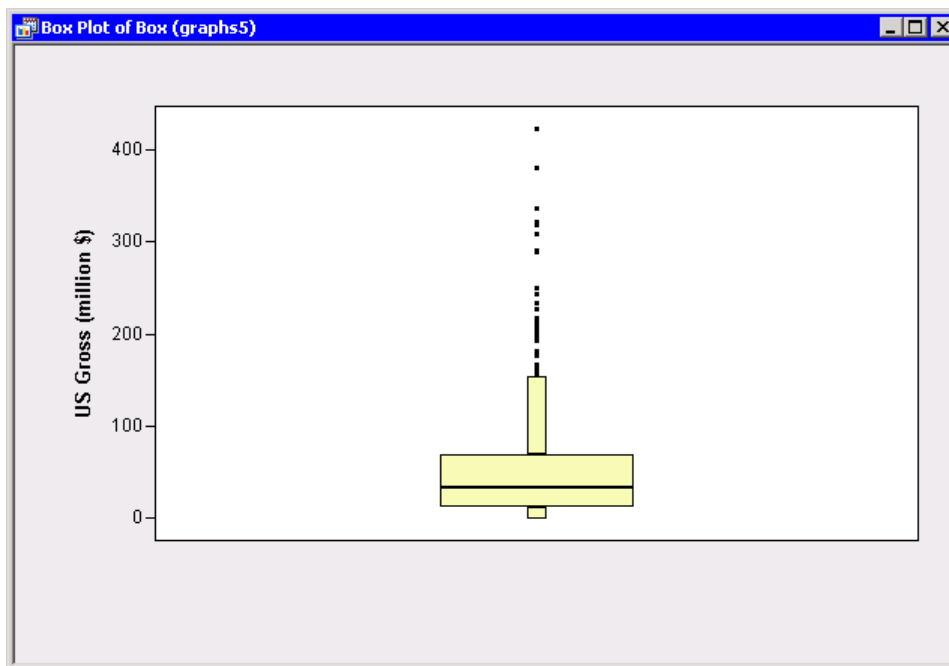
```
/* create a box plot from a vector of data */
use Sasuser.Movies;
read all var {"US_Gross"};
close Sasuser.Movies;

declare BoxPlot box;
box = BoxPlot.Create("Box", US_Gross);
box.SetAxisLabel(YAXIS, "US Gross (million $)");
```

The program contains three main steps:

1. Create the vector **US_Gross** from the data in the **US_Gross** variable.
2. Use the **declare** keyword to specify that **box** is an object of the **BoxPlot** class.
3. Create the **BoxPlot** object from the data in the **US_Gross** vector. The first argument to this method is a string that names the associated data object; the second argument is the vector that contains the data.

Figure 7.12 A Box Plot of Gross Revenue



7.7.1.2 Creating a Box Plot from a Data Object

If the data are in a data object, you can create a similar box plot.

Create a data object from the **Movies** data by using the following statements:

```
/* create data object from SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
```

The following statements create a box plot of the US gross revenues of all movies:

```
/* create a box plot from a data object */
declare BoxPlot box2;
box2 = BoxPlot.Create(dobj, "US_Gross");
```

The program creates the BoxPlot object, `box2`, from the data in the `US_Gross` variable. The first argument to this method is a data object; the second argument is the name of the variable that contains the data.

The box plot created in this way is linked to the data object and to all other graph and data tables that share the same data object.

7.7.2 Creating a Grouped Box Plot

If there is a grouping variable in the data, you can create a box plot for the data in each group.

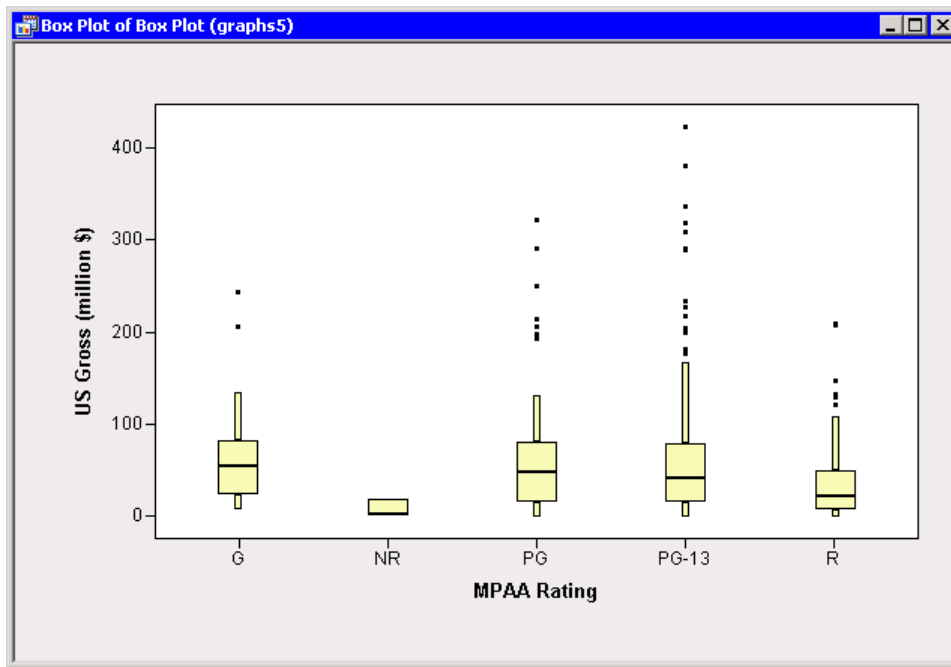
7.7.2.1 Creating a Grouped Box Plot from Vectors

The following statements create a box plot for the `US_Gross` variable, grouped by each MPAA category in the `Movies` data set:

```
/* create a box plot from vectors of data */
use Sasuser.Movies;
read all var {"MPAARating" "US_Gross"};
close Sasuser.Movies;

declare BoxPlot bg;
bg = BoxPlot.Create("Box Plot", MPAARating, US_Gross);
bg.SetAxisLabel(XAXIS, "MPAA Rating");
bg.SetAxisLabel(YAXIS, "US Gross (million $)");
```

The box plot is shown in [Figure 7.13](#). It shows features of the distribution of the `US_Gross` variable for each ratings category. You can see that the movies rated G, PG, and PG-13 have similar median US gross revenues. Furthermore, the first and third quartiles of the data are similar for those movies. However, the PG and PG-13 movies seem to have more extreme values than the G-rated movies. The R-rated movies generate comparatively less revenue, as seen in the box plot by the smaller median and Q3 value. There are only a few NR (not rated) movies, but these movies did not generate large revenues when compared with the other rating categories.

Figure 7.13 A Box Plot of Gross Revenue versus MPAA Rating

7.7.2.2 Creating a Grouped Box Plot from a Data Object

If the data are in a data object, you can create a similar box plot.

Create a data object from the Movies data by using the following statements:

```
/* create data object from SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
```

The following statements create a box plot:

```
/* create a box plot with a group variable from a data object */
declare BoxPlot bg2;
bg2 = BoxPlot.Create(dobj, "MPAARating", "US_Gross");
```

The program creates the BoxPlot object, **bg2**, from the data in the MPAARating and US_Gross variables. The first argument to this method is a data object; the second and third arguments are the names of variables that contain the data.

The box plot created in this way is linked to the data object and to all other graph and data tables that share the same data object.

7.7.3 Frequently Used Box Plot Methods

The BoxPlot class has methods that control the appearance of box plots. [Table 7.4](#) summarizes the frequently used methods in the BoxPlot class.

Table 7.4 Frequently Used Methods in the BoxPlot Class

Method	Description
BoxPlot.Create	Creates a bar chart
SetWhiskerLength	Specifies the length of the box plot whiskers as a multiple of the interquartile range
ShowMeanMarker	Specifies whether to overlay the mean and standard deviation on the box plot
ShowNotches	Specifies whether to display a notched box plot. This variation of the box plot indicates that medians of two box plots are significantly different at approximately the 0.05 significance level if the corresponding notches do not overlap.

The complete set of BoxPlot methods are documented in the SAS/IML Studio online Help.

You can use the SetWhiskerLength method to control which observations are plotted as outliers. The default multiplier is 1.5. If you set the whisker length to zero, then all observations in the first and fourth quartiles are plotted. In contrast, if you set the whisker length to 3, then only extreme outliers are explicitly plotted.

A box plot shows the median and IQR for the data, but sometimes it is useful to compare these statistics to their nonrobust counterparts, the mean and the standard deviation. You can use the ShowMeanMarker method to overlay a line segment that represents the mean of the data, and an ellipse (or diamond) that indicates the standard deviation.

The box plot displays the sample median. You can use a notched box plot to compare the median values of two groups: the medians are different (at approximately a 0.05 significance level) if their notched regions do not overlap.

The following statements modify [Figure 7.13](#) to illustrate the methods in this section. The result is shown in [Figure 7.14](#).

```

bg.SetWhiskerLength(3);
bg.ShowMeanMarker();
bg.ShowNotches();

```


Table 7.5 IMLPlus Graphs

Graph	Comments
BarChart	Described in the section “ Bar Charts ” on page 145.
BoxPlot	Described in the section “ Box Plots ” on page 161.
ContourPlot	Useful when you want to visualize a fitted surface or a regression model with two explanatory variables. This graph uses a simple contouring algorithm that is not suitable for noisy or highly correlated data.
Histogram	Described in the section “ Histograms ” on page 149.
LinePlot	Described in the section “ Line Plots ” on page 155.
MosaicPlot	Useful in understanding the relationships between two or three categorical variables. If you analyze categorical data, learn how to interpret this graph.
PolygonPlot	Useful in creating interactive maps.
RotatingPlot	A three-dimensional scatter plot. It also has the capability to plot a fitted surface or a regression model with two explanatory variables.
ScatterPlot	Described in the section “ Scatter Plots ” on page 153.

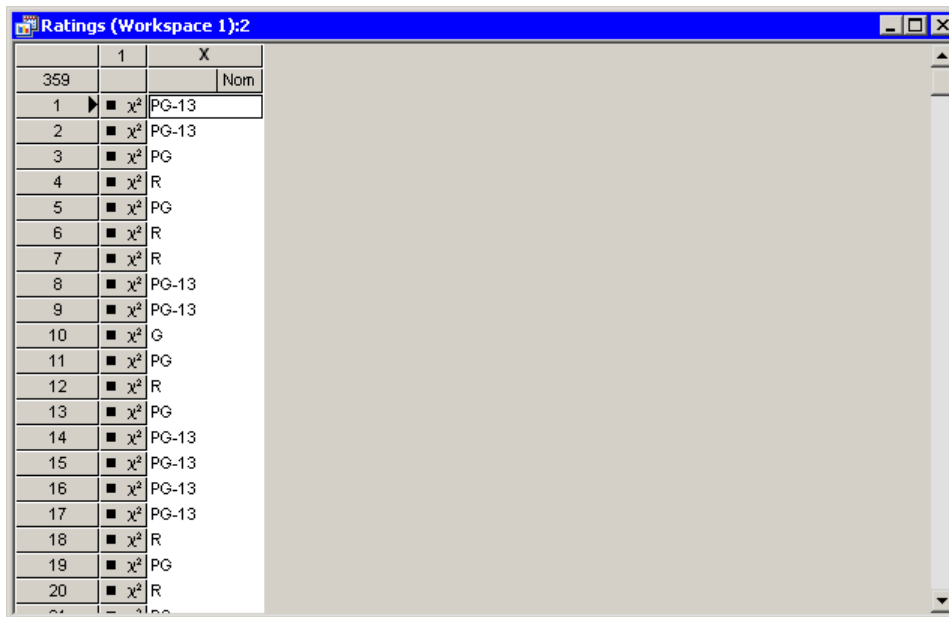
7.9 Displaying the Data Used to Create a Graph

Each section of this chapter contains an example of how to create a graph from vectors. What is not initially apparent is that creating a graph from vectors also creates an object of the `DataObject` class that contains the contents of the vector.

In SAS/IML Studio you can display a tabular view of the data that underlies a graph by pressing F9 when the graph is the active window. You can click in a graph window to make it the active window. The title bar of the active window usually has a different color than the title bars of inactive windows.

Programming Tip: You can view the data that are used to create a graph by pressing F9 when the graph is the active window.

For example, [Figure 7.2](#) shows a bar chart that is created from a vector. If you press F9 in the bar chart, SAS/IML Studio displays the data table in [Figure 7.15](#). Notice that the data are the contents of the vector that is used to create the graph. Other variables that are in the `Movies` data set are not present.

Figure 7.15 Data Table for the Bar Chart of Movie Ratings


	1	X
359		Nom
1	X ²	PG-13
2	X ²	PG-13
3	X ²	PG
4	X ²	R
5	X ²	PG
6	X ²	R
7	X ²	R
8	X ²	PG-13
9	X ²	PG-13
10	X ²	G
11	X ²	PG
12	X ²	R
13	X ²	PG
14	X ²	PG-13
15	X ²	PG-13
16	X ²	PG-13
17	X ²	PG-13
18	X ²	R
19	X ²	PG
20	X ²	R

In contrast, [Section 7.3.2](#) shows how to create a data object directly from the `Movies` data set and then create the bar chart from the data object. If you press F9 in the bar chart that is created in this way, then you see a tabular view of the entire `Movies` data set. Furthermore, the second approach preserves variable properties such as formats and labels.

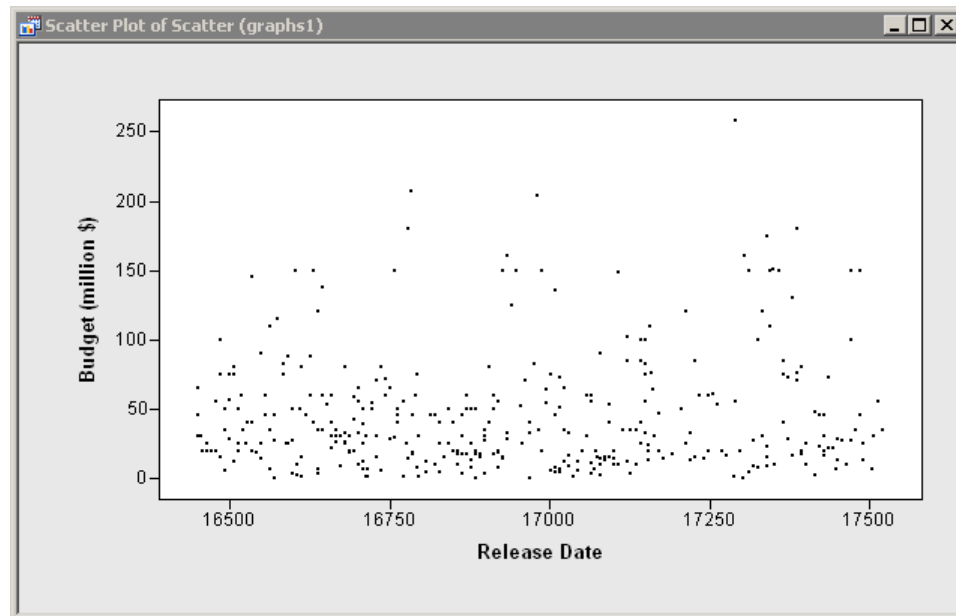
7.10 Changing the Format of a Graph Axis

Suppose that a variable in a SAS data set has a SAS format (for example, the `DATE7.` format). How can you get the scatter plot to display the formatted values shown in [Figure 7.17](#)? There are two ways. The preferred way is to create the scatter plot from an object of the `DataObject` class; the scatter plot automatically uses the format that is associated with a variable in a data object.

However, if you use the `READ` statement to create a SAS/IML vector from a variable with a SAS format, the vector contains the raw data, not the formatted values. Consequently, a graph that you create from that vector does not display any formatted values. For example, the following program creates [Figure 7.16](#):

```
/* create a scatter plot from vectors of data */
use Sasuser.Movies;
read all var {"ReleaseDate" "Budget"};
close Sasuser.Movies;

declare ScatterPlot p;
p = ScatterPlot.Create("Scatter", ReleaseDate, Budget);
p.SetAxisLabel(XAXIS, "Release Date");
p.SetAxisLabel(YAXIS, "Budget (million $)");
```

Figure 7.16 Scatter Plot of Movie Budgets versus Date of Release

Notice that each tick mark on the horizontal axis is a numerical value. These are representative of the data in the **ReleaseDate** vector. When a **DATEw.** format is applied to these data, they are displayed as dates. [Figure 7.16](#) would be more understandable if the ticks on the horizontal axis displayed dates. The following statements use the **DATE7.** format to print the date values that are displayed on the horizontal axis of the scatter plot:

```
x = do(16500, 17500, 250);      /* row vector of sequential values */
print x, x[format=DATE7.];
```

Figure 7.17 Result of Applying the **DATE7.** Format

x				
16500	16750	17000	17250	17500
x				
05MAR05	10NOV05	18JUL06	25MAR07	30NOV07

Recall from [Section 7.9](#) that *every* graph has a data object that is associated with it. There is a method that enables you to get the object of the **DataObject** class that is associated with a graph or data table. The method is named **GetDataObject**, and it is implemented in the **DataView** class. (Recall from [Section 6.9](#) that all graph classes and the **DataTable** class are derived from the **DataView** class.) You can then use the **SetVarFormat** method of the **DataObject** class to set a format for the horizontal variable, which is named **X**. You can set a format for the **Y** variable in the same way, as shown in the following statements:

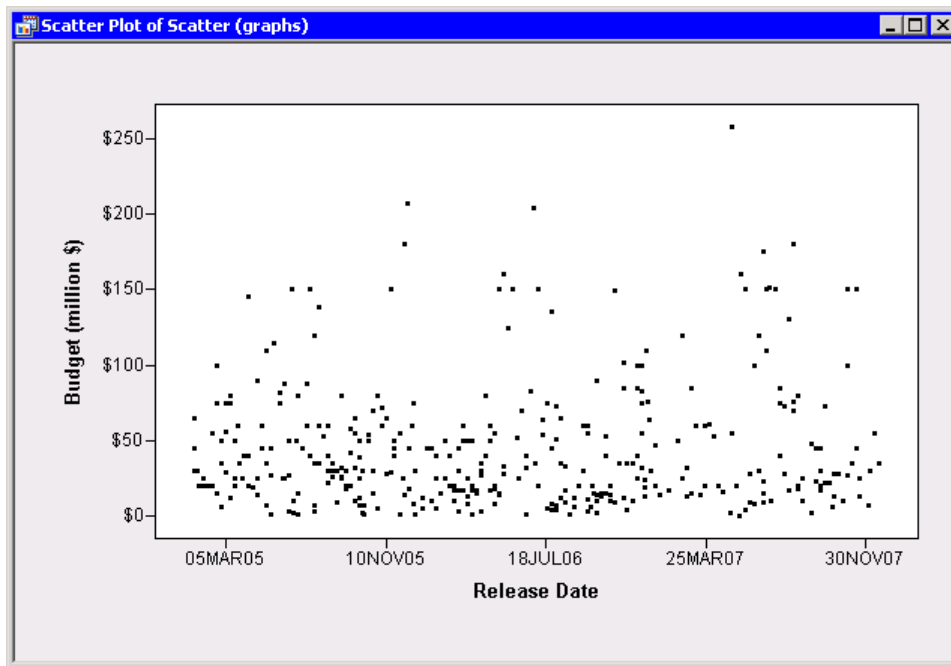

```

/* set formats for variables in a data object */
declare DataObject dobj;
dobj = p.GetDataObject();
dobj.SetVarFormat("X", "DATE7.");
dobj.SetVarFormat("Y", "DOLLAR4.");

```

Because graphs and data tables automatically respond to changes made to the associated data object, the scatter plot updates its axes as shown in [Figure 7.18](#). The scatter plot now more clearly shows the release dates for movies released during 2005–2007.

Figure 7.18 Scatter Plot with Formatted Tick Labels



You can use the same technique to change other attributes of the graph. For example, you can change a marker's shape or color by calling a method in the associated data object. These techniques are described in Chapter 10, "Marker Shapes, Colors, and Other Attributes of Data."

Programming Tip: You can call the `GetDataObject` method (in the `DataView` class) to get the object of the `DataObject` class that is associated with a graph or data table. Graphs and data tables automatically respond to changes in the associated data object.

7.11 Summary of Creating Graphs

You can create graphs in two ways: from SAS/IML vectors and from an object of the `DataObject` class. It is the `DataObject` class that enables you to create dynamically linked graphs. Graphs that are created from the same data object are automatically linked to each other.

Creating a graph from SAS/IML vectors is quick and often convenient, but it has the following drawbacks:

1. The graphs that are created from vectors initially display “X” for the label of the horizontal axis. You can use the `SetAxisLabel` method to explicitly set a label, but this is somewhat inconvenient.
2. If a variable contains a format, that format is lost when the variable is read into a SAS/IML vector. For example, the graphs in this chapter that use the `ReleaseDate` variable must explicitly set a `DATEw.` format if the dates are to be displayed correctly.
3. Any variable not specified when creating the graph is lost. For example, you cannot label observations by the names of movies in order to better identify interesting observations.
4. Two graphs created from the same data are completely independent. When you create graphs from vectors, you cannot select PG-rated movies in a bar chart and see those same movies highlighted in a second graph, as shown in [Figure 6.5](#).

Graphs created from a data object do not suffer from these drawbacks. `IMLPlus` graphics created from a common data object are dynamically linked to each other and automatically display variable names and formats. Most of the graphs in this book are created from a data object.

7.12 References

- Scott, D. W. (1992), *Multivariate Density Estimation: Theory, Practice, and Visualization*, New York: John Wiley & Sons.
- Terrell, G. R. and Scott, D. W. (1985), “Oversmoothed Nonparametric Density Estimates,” *Journal of the American Statistical Association*, 80, 209–214.

Chapter 8

Managing Data in IMLPlus

Contents

8.1	Overview of Managing Data in IMLPlus	173
8.2	Creating a Data Object	174
8.3	Creating a Data Object from a SAS Data Set	174
8.4	Creating Linked Graphs from a Data Object	175
8.5	Creating a Data Object from a Matrix	177
8.6	Creating a SAS Data Set from a Data Object	177
8.7	Creating a Matrix from a Data Object	179
8.8	Adding New Variables to a Data Object	180
8.8.1	Variable Transformations	181
8.8.2	Adding Variables for Predicted and Residual Values	182
8.8.3	A Module to Add Variables from a SAS Data Set	184
8.9	Review: The Purpose of the DataObject Class	185

8.1 Overview of Managing Data in IMLPlus

As introduced in [Section 6.3](#), the most important class in the IMLPlus language is the DataObject class. The DataObject class manages an in-memory copy of data. The class also manages graphical information about observations such as the shape and color of markers, the selected state of observations, and whether observations are displayed in plots or are excluded.

As shown in Chapter 7, “[Creating Statistical Graphs](#),” it is the DataObject class that enables you to create dynamically linked graphs. Graphs that are created from the same data object are automatically linked to each other.

This chapter describes how to create a data object from a source of data and how to create a SAS data set or a SAS/IML matrix from a data object. The chapter also describes how to modify a data object by adding new variables such as predicted values, residual values, and transformed variables.

8.2 Creating a Data Object

The `DataObject` class has several “Create” methods that instantiate a data object. A data object can be instantiated from any of several sources of data: from a SAS/IML matrix, from a Microsoft Excel worksheet, from an R data frame, from a SAS server data set, or from a SAS data set stored on the client PC. A *server data set* is one that is in a SAS library such as `Work`, `Sashelp`, or in a libref that you defined by using the `LIBNAME` statement. A *client data set* is one that can be accessed by the operating system of the computer running SAS/IML Studio. For example, this could be a data set on a hard drive or USB drive of the local PC, or any data set that is accessible through a mounted, networked drive.

The following table lists methods that create a data object from various data sources:

Table 8.1 Creating a Data Object

Method	Source
Create	SAS/IML matrix
CreateFromExcelFile	Microsoft Excel workbook
CreateFromFile	SAS data set on local PC or networked drive
CreateFromR	Data frame or matrix in R
CreateFromServerDataSet	SAS data set in libref

8.3 Creating a Data Object from a SAS Data Set

Most SAS programmers store data in a SAS data set. The `CreateFromFile` method instantiates a data object from a SAS data set on the client PC (the file has a *sas7bdat* extension). The `CreateFromServerDataSet` method instantiates a data object from a SAS data set on a SAS server, which is typically located in `Work`, `Sasuser`, or a user-defined libref such as `MyLib`.

Creating a data object from a SAS data set is easy: you need to declare the name of the data object and then instantiate the object, as shown in the following statements:

```
/* create a data object from a SAS data set */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
```

The data object is invisible, so the statements do not produce any windows. If you want a tabular view of the data, you can create an instance of the `DataTable` class, as described in [Section 6.8](#).

8.4 Creating Linked Graphs from a Data Object

The `DataObject` class provides a uniform interface to setting properties of the data, for retrieving data, and for creating dynamically linked graphs. By using the `DataObject` class, you can read data from various sources and then manipulate the data without regard to the details of how the data are stored.

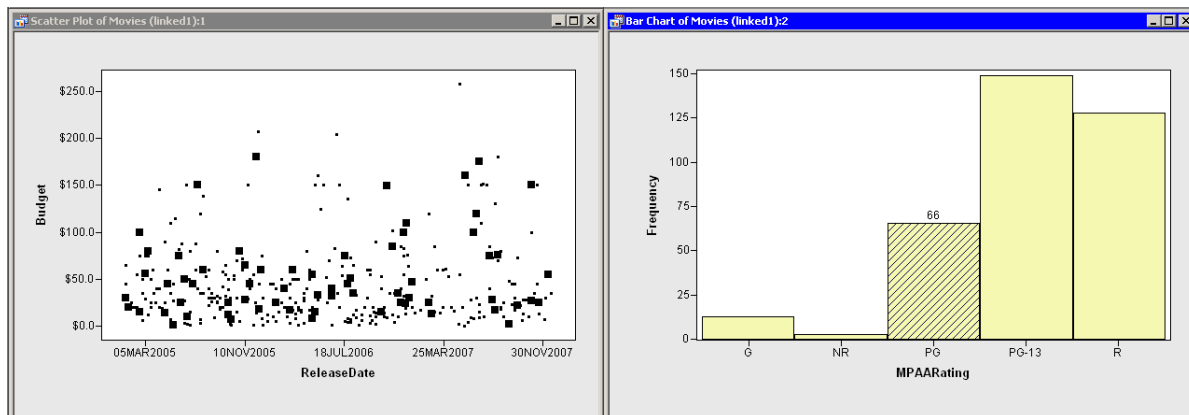
The following statements (which continue the program in the previous section) create a scatter plot and a bar chart. The first argument to each `Create` method is a data object. Subsequent arguments name variables in that data object. Because the scatter plot and the bar chart are created from a common data object, they are automatically linked to each other and to any other graphical or tabular view of the same data object.

```
/* create plots from a common data object */
declare ScatterPlot plot;
plot = ScatterPlot.Create(dobj, "ReleaseDate", "Budget");

declare BarChart bar;
bar = BarChart.Create(dobj, "MPAARating");
```

The resulting graphs are shown in [Figure 8.1](#). The figure shows the graphs after clicking on the PG category in the bar chart. Note that the 66 movies that are rated PG are selected. These observations are displayed as highlighted—both in the bar chart and also in the scatter plot.

Figure 8.1 Linked Graphs



Programming Tip: Create graphs from a common data object. All graphs that share the same data object are dynamically linked to each other.

Sometimes it is useful to label observations in a scatter plot by using values of a particular variable. You can tell the data object which variable is the “label variable” by using the `SetRoleVar` method in the `DataObject` class, as shown in the following statement:

```
dobj.SetRoleVar(ROLE_LABEL, "Title");
```

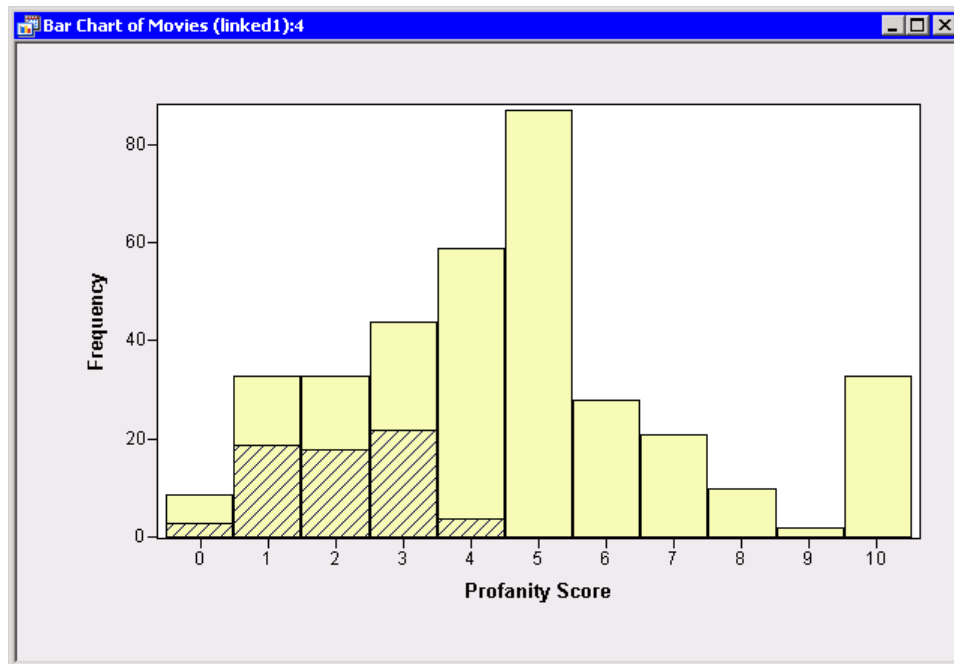
When you click on an observation marker in a scatter plot, the value of the “label variable” is displayed in the graph. For this example, the scatter plot displays the title of the movie. For example, click on the movie with the largest budget to discover that the movie is *Spider-Man 3*.

Figure 8.2 shows another example of a linked graph. It is created by the following statements:

```
declare BarChart bar1;
bar1 = BarChart.Create(dobj, "Profanity");
```

Recall that the Profanity variable contains a number 0–10 that represents the level of profane language in a movie as reported by the kids-in-mind.com Web site. The bar chart shows both the distribution of the Profanity variable and the distribution of the selected PG-rated movies (shown with cross-hatching). The bar chart makes it easy to compare the conditional distribution (that is, the distribution of Profanity given that the movie is rated PG) with the distribution of all movies. Figure 8.2 shows that PG-rated movies tend to have relatively low levels of profanity: the mean for the PG-rated movies appears to be close to 2, whereas the general mean for all of the movies appears to be closer to 4.5.

Figure 8.2 Conditional Distribution (Cross-Hatched) of Profanity for PG-Rated Movies



After a data object is instantiated, there is no connection between the data object and the source of the data (in this case, `Sasuser.Movies`). You can delete or modify data in the data object without affecting the data source. Similarly, the source of the data can be deleted or modified without altering the in-memory data object.

Programming Tip: After you create a data object, the in-memory data are independent of the source data.

8.5 Creating a Data Object from a Matrix

When the data you want to graph are in a SAS/IML matrix, you can create a data object directly from the matrix data. The `Create` method instantiates a data object from a SAS/IML matrix. For example, the following statements demonstrate how to create a data object from a matrix that contains three random variables:

```
/* create a data object from data in a matrix */
x = j(100, 3);                      /* 100 observations, 3 variables */
call randgen(x, "Normal");           /* 1 */
varNames = 'x1':'x3';                /* 2 */
declare DataObject dobj;
dobj = DataObject.Create("Normal Data", varNames, x); /* 3 */
```

The program contains the following steps:

1. The matrix **x**, which contains 100 observations and three columns, is filled with pseudorandom numbers from the standard normal distribution.
2. The character vector **varNames** contains names to assign to the columns of **x**. The notation **'x1':'x3'** uses the index operator to generate a vector of names with a common prefix.
3. A data object is created from **x**. The columns are named according to the values of **varNames**.

The previous program does not create any output or graphs. However, you can see the contents of a data object by creating a data table from the data object:

```
DataTable.Create(dobj);
```

The data table might appear behind the program window. If so, move your programming window to reveal the data table.

8.6 Creating a SAS Data Set from a Data Object

You might want to save the contents of a data object, especially if you have added variables that you intend to use in future analyses. The following table summarizes the methods in the `DataObject` class that are frequently used to save the contents of a data object to a SAS data set.

Table 8.2 Creating a Data Set from a Data Object

Method	Destination
<code>WriteToFile</code>	SAS data set on local PC or networked drive
<code>WriteToServerDataSet</code>	SAS data set in libref
<code>WriteVarsToServerDataSet</code>	SAS data set in libref

The `WriteVarsToServerDataSet` method is useful for writing a subset of variables to a SAS data set. This is especially useful as a prelude to calling a SAS procedure. As explained in [Section 8.2](#), a data object can be created from many sources of data, so if you want to call a SAS procedure, you need to make sure that the relevant variables are in a SAS data set on the SAS server. You could write the entire data object to a libref such as `Work`, but it is more efficient to write only the variables that are actually needed for the analysis.

For example, suppose you want to write a module named `Skewness` that calls the `MEANS` procedure to compute the skewness statistic for each specified variable in a data object. The following statements implement and call the `Skewness` module:

```
/* define a module that computes the skewness of variables in a
 * data object
 */
start Skewness(DataObject dobj, VarNames);           /* 1 */
    dobj.WriteVarsToServerDataSet(VarNames,
                                   "Work", "Temp", true); /* 2 */
    submit VarNames;
        proc means data=Temp noprint;               /* 3 */
            var &VarNames;
            output out=Skew skewness= ;              /* 4 */
        run;
    endsubmit;

    use Skew;
    read all var VarNames into x;                    /* 5 */
    close Skew;

    call delete("Work", "Temp");                      /* 6 */
    call delete("Work", "Skew");

    return ( x );
finish;

/* begin the main program */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Vehicles");

vars = {"MPG_Hwy" "MPG_City" "Engine_Liters"};
s = Skewness(dobj, vars);                             /* call the module */
print s[colname=vars label="Skewness"];
```

The `Skewness` module consists of the following main steps:

1. The `Skewness` module is defined to take two arguments: a data object and a vector of variable names.
2. The `WriteVarToServerDataSet` method writes a SAS data set named `Work.Temp` that contains the variables in the data object that are specified in the `VarNames` vector. The last argument to the method (**true**) specifies that observations that are excluded from analysis are not copied to the data set. (See the section “[Attributes of Observations](#)” on page 246 for further details.)

3. The MEANS procedure is called from within a SUBMIT block. The procedure reads the data from `Work.Temp` and analyzes the variables specified in the `VarNames` vector.
4. The OUTPUT statement outputs the skewness statistic for each variable. The program does not specify a value for the SKEWNESS= option, therefore the output variables have the same names as the input variables.
5. The USE and READ statements read the output from the MEANS procedure into a row vector named `x`.
6. The DELETE subroutine deletes the temporary data sets created during the computation.

The results of calling the Skewness module are shown in [Figure 8.7](#).

Figure 8.3 The Skewness of Three Variables

Skewness		
MPG_Hwy	MPG_City	Engine_Liters
0.3004436	1.4733749	0.3029371

In this example, the `Sasuser.Vehicles` data are already in a SAS data set in a libref, so it was not necessary to use the `WriteVarsToServerDataSet` method. However, the Skewness module is written so that it works for *any* data in a data object, regardless of the data source. Furthermore, recall that a data object is independent of the data from which it was instantiated. You can use `DataObject` class methods to delete observations or to exclude them from the analysis (see [Section 10.5.2](#)). Therefore, it is a good programming practice to write variables in a data object to a SAS data set prior to calling a SAS procedure.

Programming Tip: Use the `WriteVarsToServerDataSet` method in the `DataObject` class to write variables in a data object to a SAS data set prior to calling a SAS procedure.

8.7 Creating a Matrix from a Data Object

You might want to use SAS/IML operations and functions to analyze data that are in a data object. You can get data from one or more variables in a data object by using the `GetVarData` method in the `DataObject` class.

The `GetVarData` method can be called in two ways: you can get all observations for a set of variables, or you can specify the observation numbers that you want to get. Getting all of the observations is shown in the following statements, which get all values for the Hybrid variable in the `Vehicles` data set:

```

/* extract values from a variable into a vector */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Vehicles");
dobj.GetVarData("Hybrid", h);

```

In the previous statements, the `h` vector contains the data in the Hybrid variable. In particular, the vector contains a one for each observation that is for a hybrid-electric vehicle, and a zero for other observations. You can use the LOC function to find conventional vehicles, and then extract only those observations for each of several variables:

```

/* extract certain observations into a matrix */
idx = loc(h=0); /* find conventional vehicles */
VarNames = {"MPG_Hwy" "MPG_City" "Engine_Liters"};
/* extract data for specified vars and specified obs into matrix x */
dobj.GetVarData(VarNames, idx, x);
corr = corr(x); /* correlation between vars */
print corr[r=VarNames c=VarNames];

```

The previous statements fill the `x` matrix with data from the specified variables and for the specified observations. Then a correlation matrix is computed by using the CORR function. The result is shown in Figure 8.4. Notice that the `x` matrix contains observations only for the traditional (not hybrid-electric) vehicles.

Figure 8.4 Correlation Matrix for Conventional Vehicles

corr			
	MPG_Hwy	MPG_City	Engine_Liters
MPG_Hwy	1	0.9458319	-0.815963
MPG_City	0.9458319	1	-0.847236
Engine_Liters	-0.815963	-0.847236	1

Programming Tip: Use the GetVarData method in the DataObject class to copy data from a data object into a SAS/IML matrix.

8.8 Adding New Variables to a Data Object

As mentioned previously, after a data object is instantiated you can delete or modify data in the data object without affecting the data source. For example, you can add or delete variables. Deleting a variable is accomplished with the DeleteVar in the DataObject class. You can add new variables with the AddVar, AddVars, and AddAnalysisVar methods.

In practice, adding variables occurs more frequently than deleting variables. There are several reasons for needing to add new variables to a DataObject. This chapter discusses two reasons: transformations of variables, and adding predicted and residual values from a regression model.

Programming Technique: If `dobj` is an object of the `DataObject` class, use the `AddVar` method to add a variable to the data object:

```
dobj.AddVar("VarName", "Variable Label", v);
```

8.8.1 Variable Transformations

As mentioned in [Section 3.3.1](#), it is common to transform data during exploratory data analysis and statistical modeling. If a variable is heavily skewed, it is common to apply a logarithmic transformation in an attempt to work with a more symmetric distribution.

The program in this section creates a data object from the `Movies` data set. Methods in the `DataObject` class are used to copy the `Budget` variable into a SAS/IML vector. SAS/IML functions are used to transform the data, and then a `DataObject` method is used to create a new variable in the data object that contains the transformed data. The program is as follows:

```
/* transform data, add new variable, and create histogram */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.GetVarData("Budget", b);                               /* 1 */

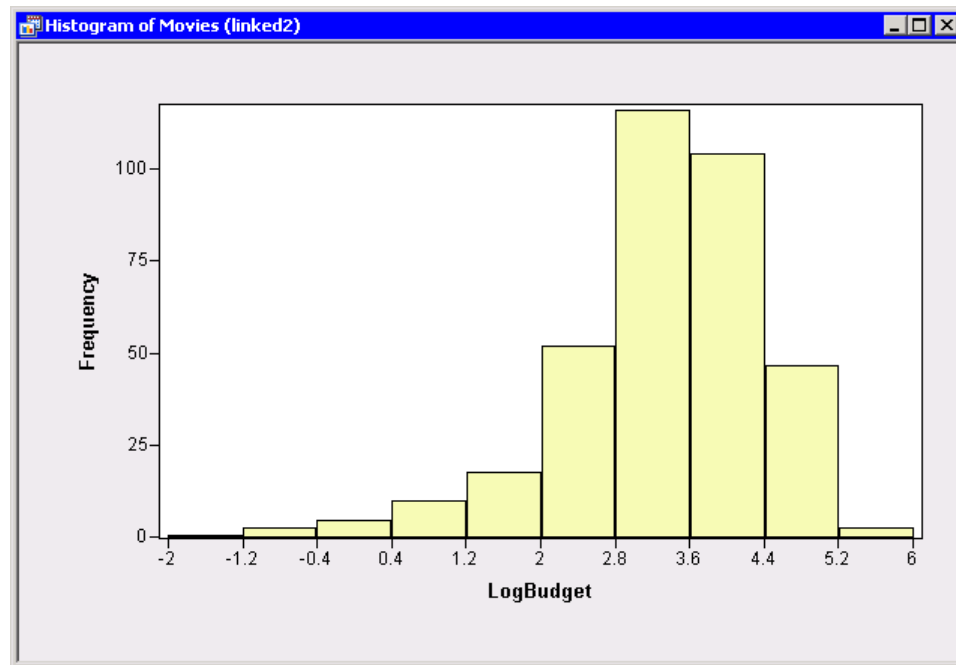
/* apply log transform */
log_b = log(b);                                             /* 2 */
dobj.AddVar("LogBudget", "log(Budget)", log_b);             /* 3 */
Histogram.Create(dobj, "LogBudget");                       /* 4 */
```

As usual, the object of the `DataObject` class is called `dobj`. It is instantiated from the `Movies` data. The program contains the following steps:

1. The `GetVarData` method retrieves the data in the `Budget` variable and puts the data into a vector called `b`.
2. The `LOG` function applies a logarithmic transform to the data and stores the transformed data in the `log_b` vector.
3. The `AddVar` method adds the transformed data to the data object. The first argument is the name of the new variable, in this case `LogBudget`. The second argument is the label for the new variable. The third argument is the vector that contains the data for the new variable.
4. The data object is invisible, so it is reassuring to create a data table or graph to make sure that the program worked. In this case, the program creates a histogram of the new variable.

The result of the program is shown in [Figure 8.5](#). The logarithmic transformation has created a new variable whose distribution is more symmetric than the distribution of the original data.

Figure 8.5 Transformed Data



You can also call the AddVar method with only two arguments, as shown in the following statement:

```
dobj.AddVar("LogBudget", log_b);          /* alternative signature */
```

In this case the label for the new variable is omitted. Consequently the name of the variable is also used as the label for the new variable.

Notice that the Budget variable contains only positive values, so it is safe to take the log of each observation. If you are not certain whether the data contains nonpositive values, you must take greater care in writing the SAS/IML statements that transform the data. A common convention is to assign a missing value to the logarithm of a nonpositive value, as shown in the following statements:

Programming Technique: The following statements apply a log transformation to data in the matrix **x**. Nonpositive elements of **x** are transformed to a missing value.

```
/* assign y=log(x), but assign missing values for any x[i]<=0 */
y = j(nrow(x),ncol(x),.); /* initialize result with missing values */
idx = loc(x>0);           /* find positive values of x */
if ncol(idx)>0 then        /* transform only those values... */
    y[idx] = log(x[idx]); /* leaving missing values in y for x<=0 */
```

8.8.2 Adding Variables for Predicted and Residual Values

In [Section 4.3](#), the GLM procedure is used to model a response variable. You can modify the program in that section to model the Mpg_Hwy variable by a quadratic function of the size of a

vehicle's engine, as represented by the `Engine_Liters` variable. The GLM procedure can write an output data set that contains variables such as the predicted and residual values for the data.

This section describes how you can add variables that contain predicted and residual values to a data object. As shown in the example in the previous section, you first read the variables into vectors, and then add the vectors to the data object by calling the `AddVar` method.

For example, the following program calls the GLM procedure to compute a quadratic regression model. The procedure creates an output data set that contains predicted and residual values for a linear model. The program begins by creating a data object from the `Vehicles` data, creating a scatter plot of two variables, and calling the GLM procedure:

```
/* call a SAS procedure to create predicted and residual values */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Vehicles");
dobj.SetRoleVar(ROLE_LABEL, "Model");      /* label for selected obs */

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Engine_Liters", "Mpg_Hwy");

submit;
proc glm data=Sasuser.Vehicles;
    model Mpg_Hwy = Engine_Liters | Engine_Liters;
    output out=GLMOut P=Pred R=Resid;
quit;
endsubmit;
```

The `Pred` and `Resid` variables are in the `GLMOut` data set. After you read them into SAS/IML vectors, you can call the `AddVar` method twice, once for each variable, or you call the `AddVars` method to add both variables at once, as shown in the following statements:

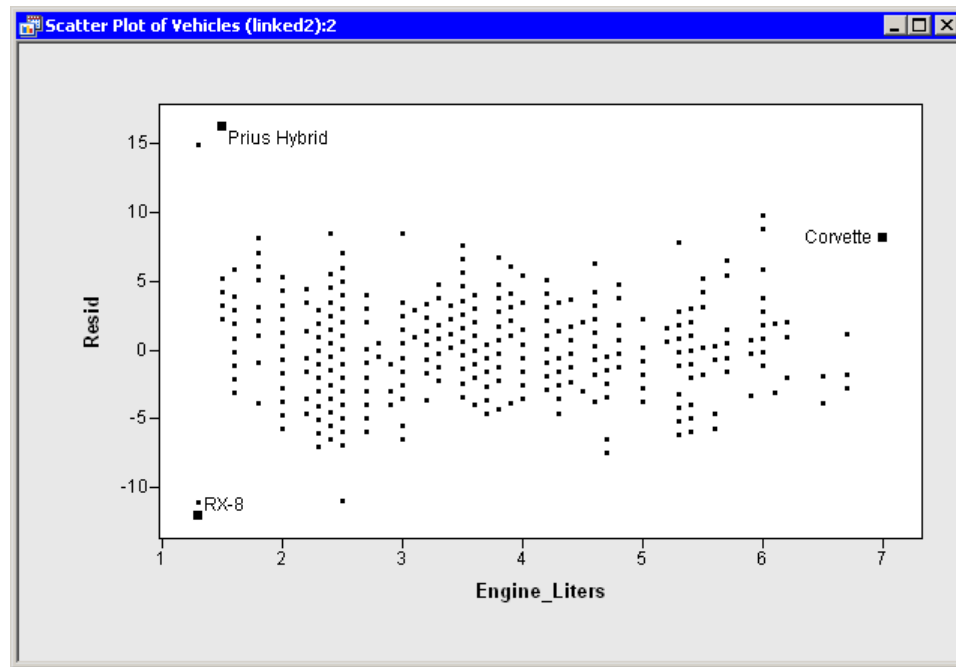
```
/* add predicted and residual values to a data object */
use GLMOut;
read all var {"Pred" "Resid"};
close GLMOut;

dobj.AddVars( {"Pred" "Resid"},
              {"Predicted Values" "Residual Values"},
              pred || resid );
ScatterPlot.Create(dobj, "Engine_Liters", "Resid");
```

The arguments to the `AddVars` method are similar to those for the `AddVar` method: the first argument names the new variables, the second (optional) argument specifies the variables' labels, and the third provides the data.

The program creates a scatter plot, as shown in [Figure 8.6](#). The scatter plot does not reveal any pattern to the residuals that might indicate an incorrectly specified model. Several outliers in the residual plot are selected. The selected observations indicate that the Prius Hybrid and the Corvette get better gas mileage than would be expected from the model, whereas the RX-8 gets substantially less mileage than would be expected.

Figure 8.6 A Residual Plot



8.8.3 A Module to Add Variables from a SAS Data Set

The analysis in the previous section calls a SAS procedure (GLM) and writes the results of the procedure to an output data set. These results are then read into vectors and added to an IMLPlus data object. This sequence of operations occurs so frequently that SAS/IML Studio is distributed with a module that reads variables from a SAS data set in a libref and adds those variables to a data object. The module is named `CopyServerDataToDataObject`; it is documented in the online Help chapter "IMLPlus Module Reference."

The module is implemented so that it not only adds the variables but also preserves the formats of variables. For example, the following module call is an alternative way to add the `Pred` and `Resid` variables from the `GLMOut` data set:

```
/* add predicted and residual values to a data object */
VarNamesInDataSet    = {"Pred" "Resid"};
VarNamesInDataObject = {"Pred" "Resid"}; /* same for this example */
Labels = {"Predicted Values" "Residual Values"};
ok = CopyServerDataToDataObject("work", "GLMOut", dobj,
    VarNamesInDataSet, VarNamesInDataObject, Labels,
    1 /* replace variable if it already exists */ );
```

The module returns 1 if it succeeds and returns 0 if it fails. Notice that you can specify the names of the variables in the data object independently from the names of the variables in the SAS data set.

Programming Tip: Use the `CopyServerDataToDataObject` module to add variables in a SAS data set to a data object.

The last argument to the module specifies what to do if one of the variables you are adding has the same name as an existing variable. The various options are documented in the online Help.

If you specify an empty matrix for the penultimate argument, then the module uses the labels (if any) in the SAS data set as labels for the corresponding new variables in the data object. How can you specify an empty matrix? By using a matrix name that has not been assigned a value in the program. A useful convention is to reserve the matrix `_NULL_` for the empty matrix, and never assign `_NULL_` a value. With that convention, the simplest way to add variables from a SAS data set into a data object is shown in the following statements:

```
ok = CopyServerDataToDataObject("work", "GLMOut", dobj,
    VarNamesInDataSet, VarNamesInDataObject, _NULL_, 1 );
```

Programming Tip: Adopt the convention never to assign a value to the SAS/IML matrix `_NULL_`. You can then confidently use `_NULL_` as an empty matrix.

If you need the data in a SAS/IML vector, you can retrieve the data from a data object by using the `GetVarData` method in the `DataObject` class. This is described in “[Creating a Matrix from a Data Object](#)” on page 179.

8.9 Review: The Purpose of the DataObject Class

A data object serves the following purposes:

- to read data into memory from various sources
- to be a uniform programming interface by providing methods that set and get data and properties of observations and variables
- to coordinate the display of data in dynamically linked graphs and data tables

These roles of the `DataObject` class are shown schematically in [Figure 6.1](#).

You can use methods in the data object to manage the properties of observations and variables. For example, each observation is represented by a marker in scatter plots. The data object ensures that the properties of each observation marker (for example, color and shape) are the same in all graphical or tabular views of the data. The same is true for properties of variables: properties such as formats, labels, and roles are maintained by the data object. Chapter 10, “[Marker Shapes, Colors, and Other Attributes of Data](#),” describes how to manage these and other properties of observations and variables.

When you are exploring data, an important property of an observation is whether or not the observation is in a selected state. Each observation in a data object can be in a selected state or in an unselected state. The data object makes sure that observations that are selected in one graph are highlighted in all other graphs. For example, in [Figure 8.1](#), the PG-rated movies are selected in the

bar chart. When you click on a bar, the scatter plot updates and highlights the markers for those same movies. Similarly, when you select a group of observations in the scatter plot, the bar chart displays the distribution of ratings for the selected observations.

Chapter 9

Drawing on Graphs

Contents

9.1	Drawing on a Graph	187
9.1.1	Example: Overlaying a Regression Curve on a Scatter Plot	188
9.1.2	Graph Coordinate Systems and Drawing Regions	191
9.1.3	Drawing in the Foreground and Background	197
9.1.4	Case Study: Adding a Prediction Band to a Scatter Plot	198
9.1.5	Practical Differences between the Coordinate Systems	202
9.2	Drawing Legends and Insets	203
9.2.1	Drawing a Legend	204
9.2.2	Drawing an Inset	206
9.3	Adjusting Graph Margins	208
9.4	A Module to Add Lines to a Graph	210
9.5	Case Study: A Module to Draw a Rug Plot on a Graph	212
9.6	Case Study: Plotting a Density Estimate	214
9.7	Case Study: Plotting a Loess Curve	216
9.8	Changing Tick Positions for a Date Axis	220
9.9	Case Study: Drawing Arbitrary Figures and Diagrams	222
9.10	A Comparison between Drawing in IMLPlus and PROC IML	224

9.1 Drawing on a Graph

The chapter describes how to overlay curves in the coordinate system of the data and also how to draw objects in different regions of a graph.

The statistical graphics in SAS/IML Studio display data stored in a data object. When you use a statistical analysis to model that data, it is often convenient to add a line, curve, or some other feature to the graph that helps you to visualize, interpret, and evaluate the model. For example, you might want to add a curve that shows predicted values for a scatter plot of two variables. Or, you might want to add a confidence ellipse to a scatter plot of bivariate normal data. Or, you might want to add a density estimate to a histogram, as shown in [Figure 4.5](#).

Drawing lines, curves, or shapes on an IMLPlus graph is accomplished with the IMLPlus *drawing subsystem*. This is a collection of methods implemented in the Plot class, which is the base class for the IMLPlus graphics classes. The methods, which all begin with the prefix “Draw,” are

documented in the online Help, in the chapter titled “IMLPlus Class Reference.” [Appendix C](#) summarizes methods that are used in this chapter.

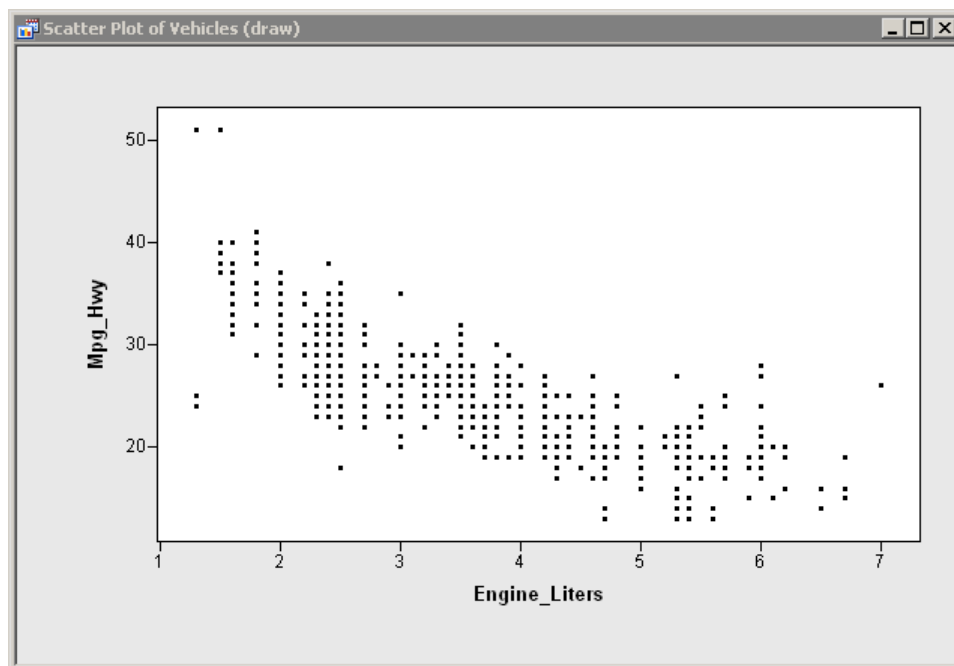
9.1.1 Example: Overlaying a Regression Curve on a Scatter Plot

Suppose that in analyzing the Vehicles data set, you suspect that the fuel efficiency of a vehicle on the highway is related to the power of the engine, as measured by the engine displacement. The following statements create a scatter plot of the Mpg_Hwy variable versus the Engine_Liters variable:

```
/* create a scatter plot */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Vehicles");

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Engine_Liters", "Mpg_Hwy");
```

Figure 9.1 Scatter Plot of Vehicle Characteristics



The scatter plot is shown in [Figure 9.1](#). Based on this graph, you decide to model the relationship between these variables with a quadratic polynomial. You continue the program with the following statements that call PROC GLM and produce the output that is shown in [Figure 9.2](#):

```
/* call SAS procedure to model relationship between variables */
submit;
ods exclude ModelANOVA;
proc glm data=Sasuser.Vehicles;
    model Mpg_Hwy = Engine_Liters | Engine_Liters;
    output out=GLMOut P=Pred R=Resid;
quit;
```

```
endsubmit;
```

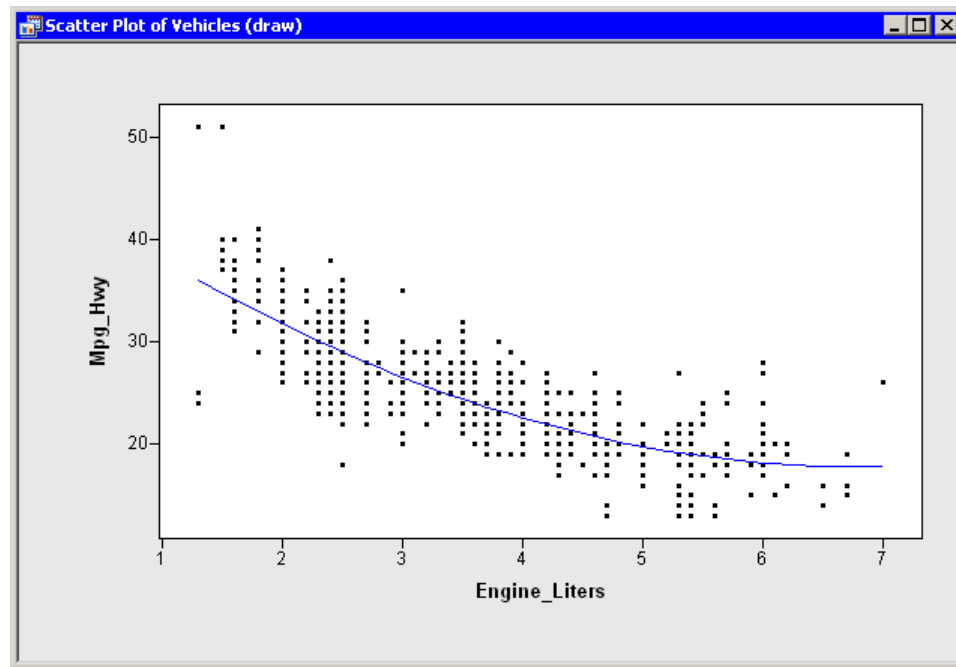
Figure 9.2 Output from PROC GLM for a Quadratic Regression Model

The GLM Procedure					
Number of Observations Read		1187			
Number of Observations Used		1187			
The GLM Procedure					
Dependent Variable: Mpg_Hwy		MPG Highway			
Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	24059.98860	12029.99430	1307.87	<.0001
Error	1184	10890.63313	9.19817		
Corrected Total	1186	34950.62174			
R-Square		Coeff Var	Root MSE	Mpg_Hwy Mean	
0.688399		12.32409	3.032848	24.60910	
Parameter	Estimate	Standard Error	t Value	Pr > t	
Intercept	45.69140443	0.81971214	55.74	<.0001	
Engine_Liters	-8.18269184	0.45126695	-18.13	<.0001	
Engine_Li*Engine_Lit	0.59996970	0.05748606	10.44	<.0001	

The GLM OUTPUT statement creates an output data set, GLMOut. The data set includes the Pred variable, which contains predicted values for the model. The following statements use the IMLPlus drawing subsystem to overlay these predicted values on the scatter plot shown in Figure 9.1. The scatter plot with the predicted values curve is shown in Figure 9.3.

```
/* read results and overlay curve on graph of data */
use GLMOut;
read all var {"Engine_Liters" "Pred"};          /* 1 */
close GLMOut;

/* need to sort data by X variable before plotting */
m = Engine_Liters || Pred;                      /* 2 */
call sort(m, 1);                               /* 3 */
p.DrawUseDataCoordinates();                     /* 4 */
p.DrawSetPenColor(BLUE);                       /* 5 */
p.DrawLine(m[,1], m[,2]);                      /* 6 */
```

Figure 9.3 Predicted Values Overlaid on a Scatter Plot

The previous statements consist of six steps:

1. Read the explanatory variable into the **Engine_Liters** vector and the predicted values into the **Pred** vector.
2. Horizontally concatenate the **Engine_Liters** and **Pred** vectors to form an $n \times 2$ matrix, **m**.
3. Call the SAS/IML SORT subroutine to sort the rows of **m** by the values of the first column.
4. Set the coordinate system for drawing. The `DrawUseDataCoordinates` method specifies that future drawing commands will draw in a coordinate system that is consistent with the graph's data and axes.
5. Set a color for the graphical "pen" used for drawing subsequent lines. This step is optional. The default color is black.
6. Draw a series of line segments that connect the ordered pairs that are contained in the rows of the **m** matrix. The line segments begin at the point defined by the first row of **m**, continue to the point defined by the second row of **m**, and so on until the last row of **m**.

Notice that the `DrawLine` method draws a line that connects ordered pairs. The arguments to the `DrawLine` method define the order in which the points are connected. If the `SORT` subroutine were not called in the third step, the line segments would connect the points in the order that they appear in the `GLMOut` data set.

Programming Tip: The `Plot.DrawLine(x, y)` method draws line segments connecting the points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, in that order. You will usually want to sort the points by x before you call the `DrawLine` method.

Both [Figure 9.3](#) and the GLM output in [Figure 9.2](#) indicate that the quadratic model fits these data reasonably well. The careful analyst will also want to plot a scatter plot of the residuals versus the `Engine_Liters`, as described in [Section 8.8.2](#).

9.1.2 Graph Coordinate Systems and Drawing Regions

In the previous section, the example program calls the `DrawUseDataCoordinates` method prior to drawing a curve on the screen. The method specifies that future drawing commands will use coordinates consistent with the graph's data and axes. This section discusses the various coordinate systems that you can use with IMLPlus graphics.

9.1.2.1 Drawing in the Coordinate System of the Data

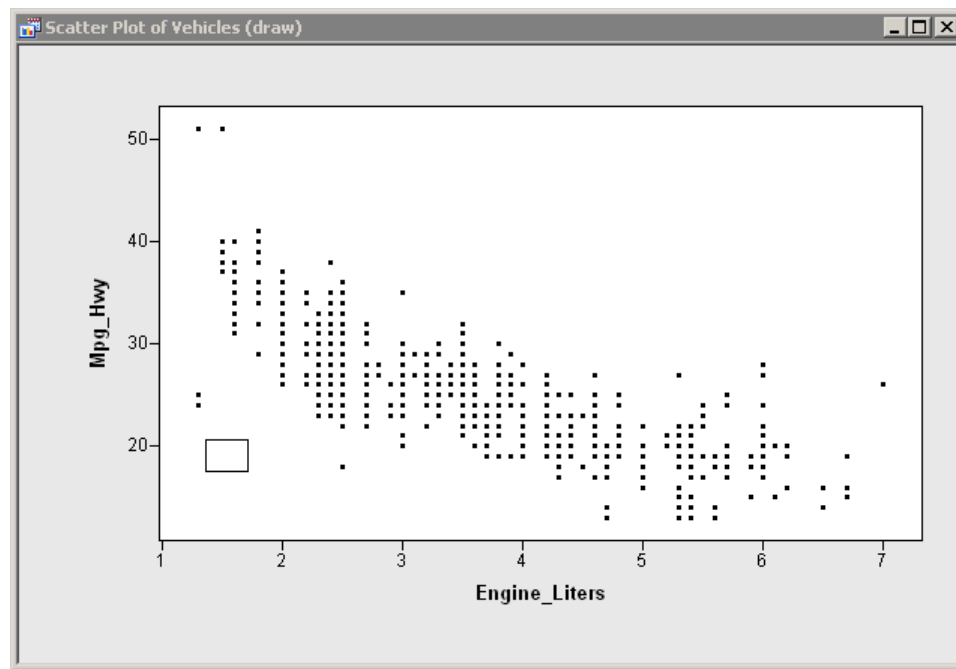
Suppose you want to draw a rectangle on a scatter plot in order to draw attention to certain observations. To be completely concrete, suppose you want to draw a rectangle around the vehicles in [Figure 9.1](#) that get less than 20 miles to the gallon on the highway in order to emphasize that these vehicles are not fuel efficient.

The method in the `Plot` class that draws a rectangle is called `DrawRectangle`. The arguments to the `DrawRectangle` method specify the location of the lower left and upper right corners of the rectangle. Specifically, if you call the method as `DrawRectangle(x1, y1, x2, y2)`, then the graph displays a rectangle with lower left corner (x_1, y_1) and upper right corner (x_2, y_2) . Consequently, you might hastily write the following statements:

```
/* draw a rectangle on a scatter plot */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Vehicles");

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Engine_Liters", "Mpg_Hwy");
p.DrawRectangle(1, 12, 7.2, 20);          /* wrong coordinate system! */
```

The resulting scatter plot is shown in [Figure 9.4](#). The graph *does* contain a rectangle, but notice that the rectangle does not contain the vehicles that get less than 20 miles per gallon on the highway. What happened?

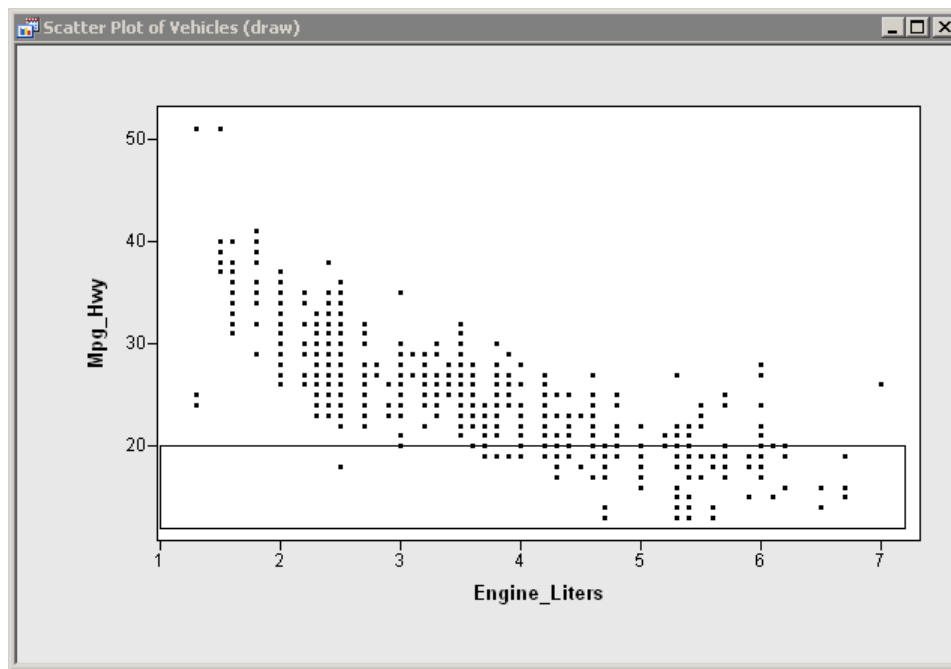
Figure 9.4 An Incorrect Attempt to Draw a Rectangle on a Scatter Plot

What happened is that the `DrawRectangle` method drew the rectangle correctly, but drew it in a different coordinate system than the data uses. The default coordinate system for IMLPlus graphics is a coordinate system with (0,0) near the lower left corner of the plot area, and (100,100) near the upper right corner. In this coordinate system, the coordinates represent a percentage of the data range. (The plot area also contains margins that surround the data, as shown in [Figure 9.15](#).) Consequently, the horizontal dimension of the rectangle in [Figure 9.4](#) starts at 1% of the range for the `Engine_Liters` variable and ends at 7.2% of the range. Similarly, the vertical dimension starts at 12% of the range of the `Mpg_Hwy` variable and ends at 20% of the range.

To correctly draw the rectangle, you need to specify a coordinate system that corresponds to the data ranges of the variables in the graph. The way to do that is to call the `DrawUseDataCoordinates` method prior to calling the `DrawRectangle` method. This is shown in the following statements:

```
/* draw a rectangle on a scatter plot (correct version) */
p.DrawUseDataCoordinates();          /* specify data coordinates */
p.DrawRectangle(1, 12, 7.2, 20);     /* draw rectangle          */
```

The resulting rectangle is now drawn as intended. The scatter plot and rectangle are shown in [Figure 9.5](#).

Figure 9.5 A Rectangle on a Scatter Plot

Programming Tip: When you want to draw in the plot area in the coordinate system defined by the data, remember to call the `Plot.DrawUseDataCoordinates` method before calling any other drawing command.

You can call the `DrawUseDataCoordinates` method on a histogram, a scatter plot, a contour plot, a line plot, and a polygon plot, provided that these plots are not displaying any variables that contain categorical data.

9.1.2.2 Drawing on a Graph That Displays a Categorical Variable

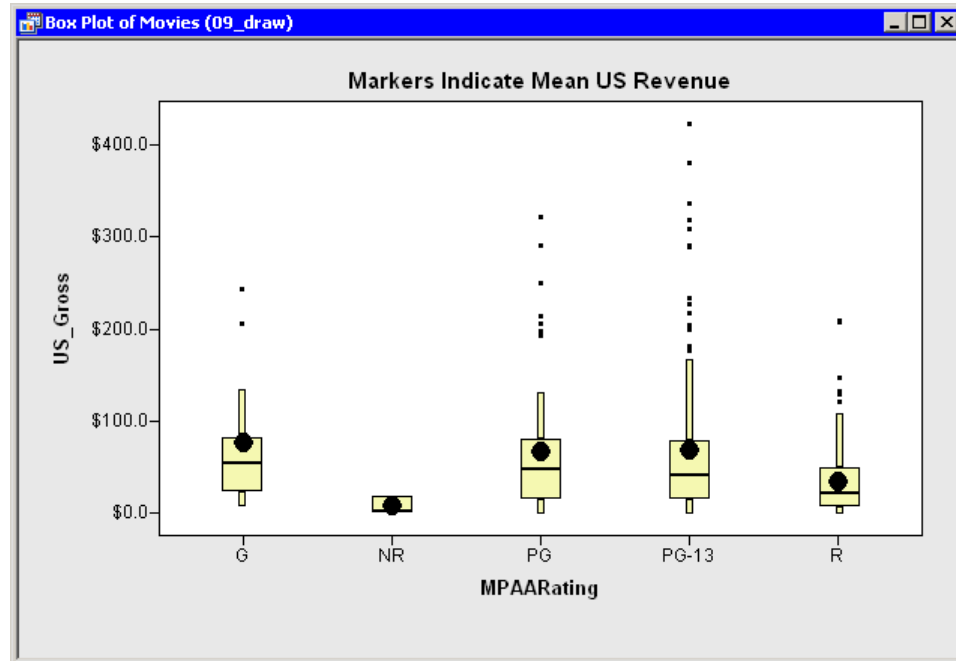
Some graphs plot categorical data; others plot continuous data. Some graphs can handle both types of data. For example, a bar chart plots categorical data, although the vertical axis (which displays frequency or percentage) is continuous. A scatter plot can handle either type of data, although it is most often used to plot two continuous variables. The horizontal axis for a box plot is categorical, whereas the vertical axis is continuous.

If your graph contains a categorical variable, then there is no “natural” coordinate system defined by the data. For example, [Figure 7.13](#) shows a box plot for the `US_Gross` and `MPAARating` variables in the `Movies` data set. The horizontal axis for this plot does not have a numerical minimum or maximum value. There is no intrinsic coordinate value that represents the center of the PG category. Nevertheless, you might want to draw on this box plot or on other graphs with a categorical axis. In this case, you would want to define your own convenient coordinate system for drawing.

For example, suppose you want to model the dependence of `US_Gross` on `MPAARating`. You could modify [Figure 7.13](#) so that it shows the mean revenue under the model for each MPAA rating

category. One way to display that information is to plot a marker for each category, with the vertical position of the marker that represents the expected value of gross US revenues. Such a graph is shown in Figure 9.6.

Figure 9.6 Overlaying Markers on a Box Plot



How can you create such a graph? This section breaks down the creation of Figure 9.6 into the following steps:

1. Create the box plot.
2. Compute the mean of US_Gross accounted for by each MPAA rating category by using the technique that are described in Section 3.3.5.
3. Set up a convenient coordinate system for drawing the markers.
4. Compute the horizontal location of the center of each group and plot each marker at the appropriate horizontal and vertical position.

The following sections describe each step and the related program statements.

Create the box plot

The IMLPlus statements that create the box plot are shown below.

```
/* create a box plot for each MPAA rating */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
```



```
declare BoxPlot box;
box = BoxPlot.Create(dobj, "MPAARating", "US_Gross");
```

Compute the mean for each category

The second step is to compute the mean of the US_Gross for each MPAA rating category.

Section 3.3.5 describes how to write statements that compute quantities for each category. In this case, the quantity of interest is the mean of the US_Gross variable for each rating category. The following statements compute a vector **GrossMean** that contains the relevant values:

```
/* compute mean of revenue for each rating category */
use Sasuser.Movies;           /* or use dobj.GetVarData() */
read all var {"MPAARating" "US_Gross"};
close Sasuser.Movies;

u = unique(MPAARating);       /* find the MPAA categories */
NumGroups = ncol(u);          /* how many categories? */
GrossMean = j(1, NumGroups);  /* allocate a vector for results */
do i = 1 to NumGroups;        /* for each group... */
    idx = loc(MPAARating=u[i]); /* find the movies in that group */
    m = US_Gross[idx];
    GrossMean[i] = m[:];       /* compute group mean */
end;
print GrossMean[c=u];
```

Figure 9.7 Mean US Gross Revenues by MPAA Rating Category

GrossMean				
G	NR	PG	PG-13	R
76.141323	7.9832103	67.479549	69.111818	34.206493

The output from these statements is shown in Figure 9.7.

Define a coordinate system

The third step is to set up a convenient coordinate system.

The vertical axis represents a continuous quantity, so it is convenient to draw in the coordinates of the axis, which is the data range for the US_Gross variable. You can get the minimum and maximum values of an axis by calling the GetAxisViewRange method:

```
box.GetAxisViewRange(YAXIS, YMin, YMax);
```

The horizontal axis is categorical, so there is not a canonical coordinate system. Common choices for coordinates include $[0, 1]$ or $[-1, 1]$ or even $[0, \text{NumGroups}]$. For definiteness, choose $[0, 1]$ for this example. The following statement calls the DrawUseNormalizedCoordinates method to set the coordinate system to $[0, 1] \times [YMin, YMax]$:

```
box.DrawUseNormalizedCoordinates(0, 1, YMin, YMax);
```

Plot the mean markers

The fourth step is to compute the location of the markers and plot them.

The horizontal position of the markers should be at the center of the boxes. There are **NumGroups** boxes, and each has the same width. Because the horizontal coordinates were chosen to be in $[0, 1]$, the distance between each horizontal tick mark is $1/\text{NumGroups}$. It follows that the center of each box is located at the values contained in the **XCenters** vector shown in Figure 9.8. The vectors are computed in the following statements:

```
/* compute coordinates of horizontal centers of boxes */
dX = 1/NumGroups;           /* 1/5, in this example */
BinEdges = (0:NumGroups-1)*dX; /* 0, 1/5, ..., 4/5 */
XCenters = BinEdges + dX/2;   /* 1/10, 3/10, ..., 9/10 */
print BinEdges, XCenters;
```

Figure 9.8 Horizontal Coordinates for Each Category

BinEdges				
0	0.2	0.4	0.6	0.8
XCenters				
0.1	0.3	0.5	0.7	0.9

Figure 9.8 shows the values of the edges and centers of the categories in the $[0, 1]$ coordinate system. Notice that the index creation operator (**:**) has lower precedence than arithmetic operators (**+**, **-**, *****, **/**), so that **0:NumGroups-1** is equivalent to **0:4** for this example.

Programming Tip: Suppose a graph of a categorical variable contains N categories. If you use $[0, 1]$ for the coordinates of that axis, then the center of the i th category is located at $(i - \frac{1}{2})/N$. If you instead use $[0, N]$ for the coordinates of that axis, then the center of the i th category is located at $i - \frac{1}{2}$.

The program is almost complete. The horizontal coordinates of the markers are contained in the **XCenters** vector and the vertical coordinates are contained in the **GrossMean** vector, so you can draw the markers and add the title shown in Figure 9.6 with the following statements:

```
box.DrawMarker(XCenters, GrossMean, MARKER_CIRCLE, 8);
box.SetTitleText("Markers Indicate Mean US Revenue", true);
```

Notice that the **DrawMarker** method accepts vectors for the horizontal and vertical coordinates. Calling the method once is more efficient than writing a loop that plots a marker at **XCenters[i]** and **GrossMean[i]** for each appropriate value of **i**.

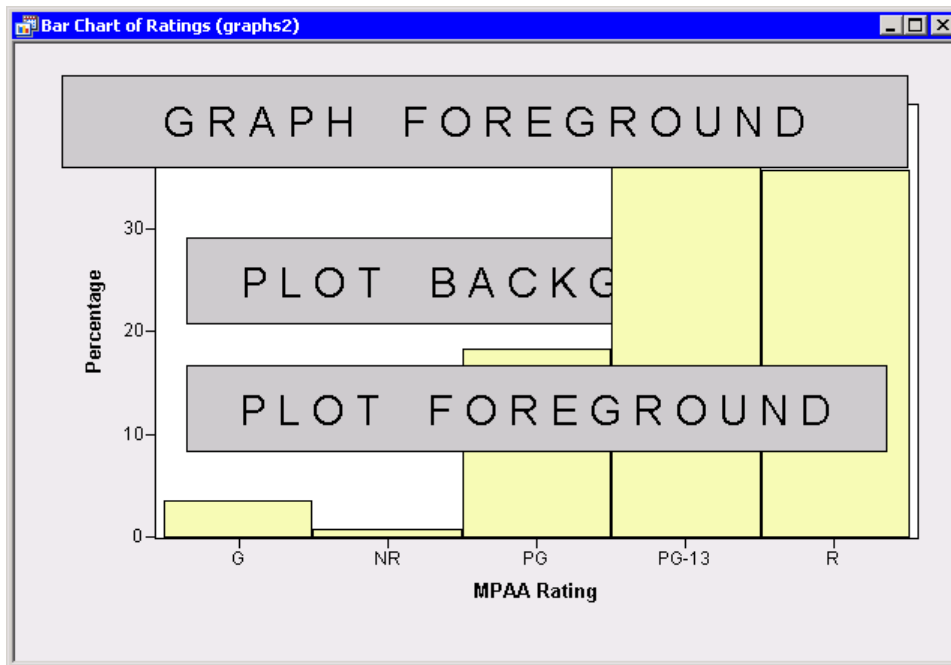
You can use the **SetTitleText** to add a title to a graph. The optional second argument (**true**) causes the **SetTitleText** method to immediately display the title.

9.1.3 Drawing in the Foreground and Background

The IMLPlus graphics enable you to draw in three distinct areas of a graph: the plot area foreground, the plot area background, and the graph area. Whenever possible, you should draw data-related curves and figures in the plot area in the data coordinate system, as shown in the previous sections.

You can change the drawing region by calling the `DrawSetRegion` method. The method takes a single argument; valid values are `PLOTFOREGROUND`, `PLOTBACKGROUND`, and `GRAPHFOREGROUND`. The three regions are illustrated in Figure 9.9.

Figure 9.9 Three Drawing Regions



If you specify the `PLOTFOREGROUND` parameter, anything you draw on the graph appears in the plot area in front of the observations, whether they are represented by markers, bars, or boxes. For example, if you draw a rectangle or polygon in this area, it might obscure observations markers that are behind it. In contrast, the `PLOTBACKGROUND` parameter specifies that drawn objects appear behind observation markers, bars, and boxes. The plot foreground and plot background share a common coordinate system. For example, if you set the coordinate system for the plot foreground by using the `DrawUseDataCoordinates` method, and you then use `DrawSetRegion` to set the drawing area to the plot background, you do not need to call `DrawUseDataCoordinates` a second time.

Programming Tip: The plot foreground and plot background share a common coordinate system.

If you specify the `GRAPHFOREGROUND` parameter as an argument to `DrawSetRegion`, objects can be drawn anywhere in the graph. When you draw in this region, the objects will obscure axes, axis labels, and anything in the plot area. Consequently, it is best to draw in the margins of the graph area. (Margins are described in Section 9.3.) The graph area supports only the normalized coordinate system.

Programming Tip: The graph area supports only a normalized coordinate system.

In general, adhere to the following recommendations:

- Draw data-related lines and curves in the PLOTFOREGROUND. For example, regression curves or density estimates.
- Draw filled polygons in the PLOTBACKGROUND (for example, confidence bands, bivariate confidence ellipses, and regions under a probability curve).
- Display text, notes, and legends in the GRAPHFOREGROUND or PLOTFOREGROUND regions. If necessary, you can make room for drawing by calling the SetGraphAreaMargins or SetPlotAreaMargins methods in the Plot class, as described in [Section 9.3](#).

Programming Tip: Draw lines and curves in the PLOTFOREGROUND region. Draw filled regions in the PLOTBACKGROUND region. Draw legends in the GRAPHFOREGROUND or PLOTFOREGROUND regions.

9.1.4 Case Study: Adding a Prediction Band to a Scatter Plot

To illustrate the different drawing regions, this section describes a program that extends the example in [Section 9.1](#). The program creates a scatter plot of the Mpg_Hwy variable versus the Engine_Liters variable in the Vehicles data set. It then adds a quadratic regression curve. The new features of the program are to add a band in the plot area background that indicates a region of 95% confidence for individual predictions, and to create a simple legend in the graph area foreground.

The following statements re-create the plot and overlay the curve, as previously shown in [Figure 9.3](#):

```
/* create a scatter plot; overlay curve */
submit;
ods exclude ModelANOVA;
proc glm data=Sasuser.Vehicles;
    model Mpg_Hwy = Engine_Liters | Engine_Liters;
    output out=GLMOut P=Pred lcl=Lower ucl=Upper;          /* 1 */
quit;

/* need to sort data by explanatory (X) variable before plotting */
proc sort data=GLMOut;                                     /* 2 */
    by Engine_Liters;
run;
endsubmit;

use GLMOut;
read all var {"Engine_Liters" "Mpg_Hwy"
              "Pred" "Lower" "Upper"};                     /* 3 */
close GLMOut;
```

```

declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Work.GLMOut");

declare ScatterPlot p;                                /* 4 */
p = ScatterPlot.Create(dobj, "Engine_Liters", "Mpg_Hwy");
p.DrawUseDataCoordinates();
p.DrawSetPenColor(BLUE);
p.DrawLine(Engine_Liters, Pred);

```

There are four differences between the statements that created [Figure 9.3](#) and the present example:

1. The GLM OUTPUT statement creates three variables: Pred, Lower, and Upper. The Lower variable contains the lower bound of a 95% confidence interval for an individual prediction. The Upper variable contains the upper bound.
2. The example in the section “[Drawing on a Graph](#)” on page 187 calls the SAS/IML SORT subroutine to sort the variables so that the data can be plotted with the DrawLine method. This example calls the SORT procedure instead.
3. This example reads variables from the sorted GLMOut data set. In particular, the **Engine_Liters** and **Mpg_Hwy** vectors are sorted.
4. The scatter plot is created from the sorted data.

In spite of these differences, the present example creates the same graph and overlays the same curve as shown in [Figure 9.3](#).

The following statements add a light-gray band to the background of the plot area. The drawing is accomplished with the DrawPolygon method:

```

/* draw 95% prediction band */
p.DrawSetRegion(PLOTBACKGROUND);                /* 5 */
p.DrawSetBrushColor(200, 200, 200);              /* 6 */
p.DrawSetPenStyle(OFF);                          /* 7 */

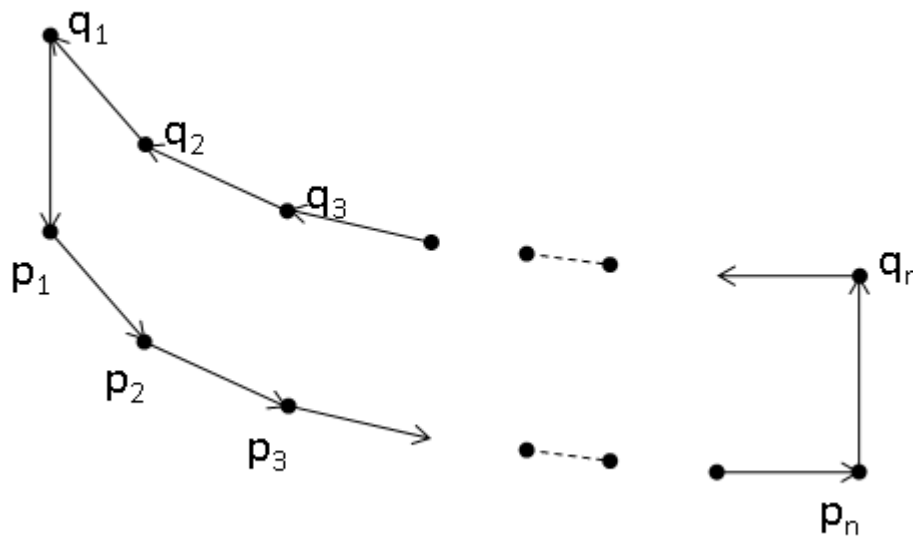
/* create a polygon defined by the upper/lower prediction limits */
n = nrow(Engine_Liters);
x = Engine_Liters // Engine_Liters[n:1];         /* 8 */
y = Lower // Upper[n:1];
p.DrawPolygon(x, y, true);                      /* 9 */

```

5. The default drawing location is the plot area foreground. However, a filled prediction band in the foreground obscures observations, so set the drawing region to the background of the plot area. The foreground and background areas share the same coordinate system, so you do not have to repeat the call to DrawUseDataCoordinates.
6. Set the color of the graphical “brush.” This is the color that is used to fill the prediction band. (The default color is black.) You can specify either a keyword that specifies a predefined color such as GRAY or YELLOW, or you can specify the color by using an RGB specification, as is done for this example. The RGB triple (200, 200, 200) is a light gray color. [Chapter 10](#) describes how to specify colors.

7. When you draw a polygon, the current pen is used to draw the outline of the polygon. This example turns off the pen, so that there is no outline drawn on the prediction band.
8. How can you draw a confidence band if you know the curves that make up its upper and lower boundary? The program shows one approach. The DrawPolygon method requires two arguments: a vector of X values and a vector of Y values. The polygon is drawn by connecting the pairs of points in the order they appear in the vectors. Let p_1, p_2, \dots, p_n name the ordered points on the lower curve and let q_1, q_2, \dots, q_n name the ordered points on the upper curve, as shown in Figure 9.10. You can draw the prediction band by specifying the values of the lower boundary (in increasing X order), followed by the values of the upper boundary *in reverse X order*.
9. The DrawPolygon method draws the polygon. The polygon connects the points $p_1, p_2, \dots, p_n, q_n, q_{n-1}, \dots, q_1$, as shown in Figure 9.10.

Figure 9.10 Creating a Polygon from Upper and Lower Boundaries



The prediction band is drawn in the background of the plot area, as shown in Figure 9.11.

So far, the example in this section has drawn in the plot area foreground and background. For the sake of completeness, the last few statements of this section show how you can draw in the graph area. The graph area is a good place to add a legend, so the following statements draw the text “95% Prediction Band” in the lower right corner of the graph area, and also displays a light-gray rectangle:

```

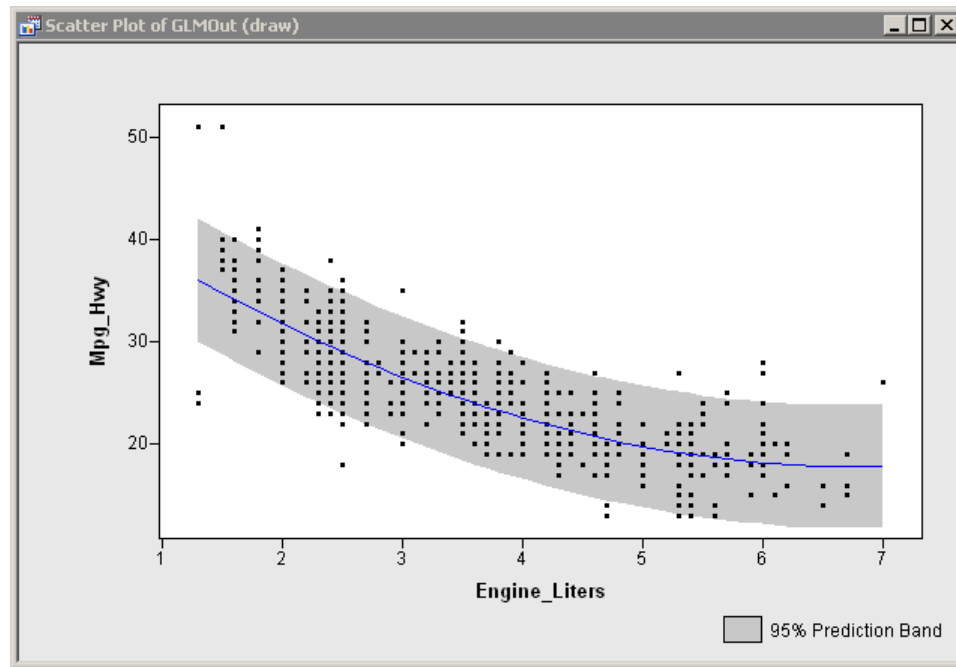
/* draw simple legend */
p.DrawSetPenAttributes(BLACK, SOLID, 1);           /* 10 */
p.DrawSetRegion(GRAPHFOREGROUND);                  /* 11 */
p.DrawSetTextAlignment(ALIGN_LEFT, ALIGN_CENTER);  /* 12 */
p.DrawText(80, 5, "95% Prediction Band");          /* 13 */
p.DrawRectangle(75, 3, 79, 7, true);               /* 14 */

```

10. Recall that the graphical pen is currently turned off. Reset the pen to draw solid black lines one pixel wide.
11. Set the drawing region to be the graph area. Recall that the default coordinate system is $[0, 100] \times [0, 100]$.
12. By default, text is centered at a specified location. For this example, left-justify the text by calling the `DrawSetTextAlignment` method.
13. If a graph is using default margins, the left and right edges of the plot area are 15 and 96, respectively; the coordinates of the bottom and top edges of the plot area have vertical coordinates 20 and 90. Drawing text in the margin sometimes requires trial and error to place the text correctly. For this example, draw the text at the location (80, 5), which corresponds to the lower right corner of the graph area.
14. Draw a filled rectangle preceding the text. The color of the graphical brush is still light gray. The outline of the rectangle is drawn with the characteristics of the current graphical pen.

Programming Tip: There is one graphical pen and one graphical brush that are shared among all drawing regions.

The completed graph is shown in [Figure 9.11](#). The graph shows a quadratic fit drawn in the plot area foreground. The curve is drawn in the data coordinate system. The graph shows a prediction band drawn in the plot area background, which shares the same coordinate system as the plot area foreground. The graph also shows a primitive legend that is drawn in the graph area. The legend is drawn in the default (normalized) coordinate system.

Figure 9.11 A Graph with Features Drawn in Three Drawing Regions

You do not need to draw legends by using low-level drawing commands: SAS/IML Studio distributes several modules that make it easy to create legends. The `DrawInset` and `DrawLegend` modules are discussed in [Section 9.2](#) and are documented in the online Help. You can view the documentation in the chapter titled “IMLPlus Module Reference,” in the section titled “Graphics.”

Programming Tip: Use the `DrawLegend` module to draw legends.

9.1.5 Practical Differences between the Coordinate Systems

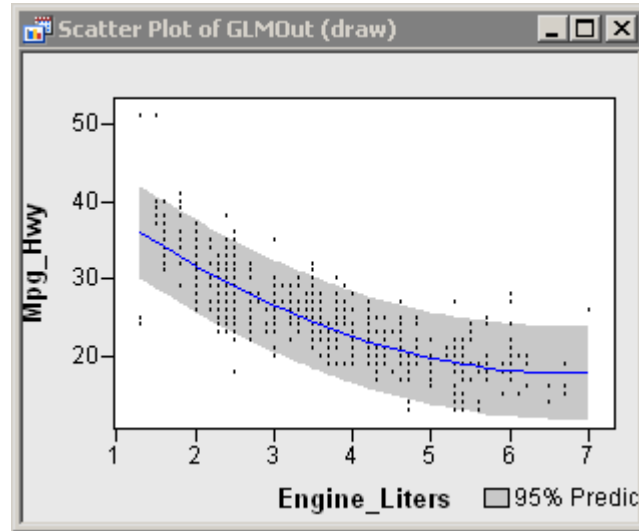
There are some important differences between the data coordinate system and a user-defined coordinate system. Objects drawn in the data coordinate system respond to zooming and panning in the plot area, whereas a user-defined coordinate system does not. (Try panning the box plot in [Figure 9.6](#) and see what happens! To pan the plot area, right-click in the plot area and select **Pan Tool** from the pop-up menu.)

Consequently, the data coordinates are best for plotting regression lines, reference lines, confidence regions, and similar objects. For the same reason, a user-defined coordinate system is best for drawing legends, insets, and similar annotations because you typically do not want a legend to disappear when you zoom or pan in a graph. Unfortunately, there is no way (as of SAS/IML Studio 3.3) to draw in the coordinate system of the data for a graph that displays a categorical variable.

Programming Tip: Whenever possible, use the data coordinate system to display statistical curves and regions.

You should also be aware of the effect of resizing a window on objects drawn in user-defined coordinates. If you resize the window in [Figure 9.11](#), you will observe that the simple legend does not rescale like the rest of the window. This is shown in [Figure 9.12](#).

Figure 9.12 Resizing a Window That Has Text in the Graph Area



The size of the window has shrunk, but the left side of the text is still drawn at a position 80% along the width of the graph. Since the size of the window has changed, the entire text is no longer visible. The same behavior occurs if you change the graph margins. The text is not aware of the graph area margins, it is merely displayed at a certain percentage of the window's width.

The lesson to learn is this: when you draw objects such as legends and insets that are in the GRAPH-FOREGROUND region, they do not rescale as the window resizes. Therefore, you should strive to set the size of a window prior to drawing in a user-coordinate system.

Programming Tip: Adjust the size of a window before drawing a legend or inset. After drawing, avoid resizing the window or changing the graph area margins.

9.2 Drawing Legends and Insets

As mentioned previously, SAS/IML Studio distributes several modules that make it easy to create legends. This section provides an overview of the DrawLegend and DrawInset modules. For more information, see the SAS/IML Studio online Help.

Both of these modules use the IMLPlus drawing subsystem to place text on a graph in a user-defined coordinate system. Consequently, they are affected by the issues discussed in [Section 9.1.5](#).

9.2.1 Drawing a Legend

You can use the DrawLegend module to associate markers or lines in a graph with descriptive text. For example, the following statements extend the ideas in [Section 8.7](#) to create a scatter plot in which hybrid-electric vehicles are represented by markers of one shape and color, whereas traditional gasoline engines are represented by another shape and color:

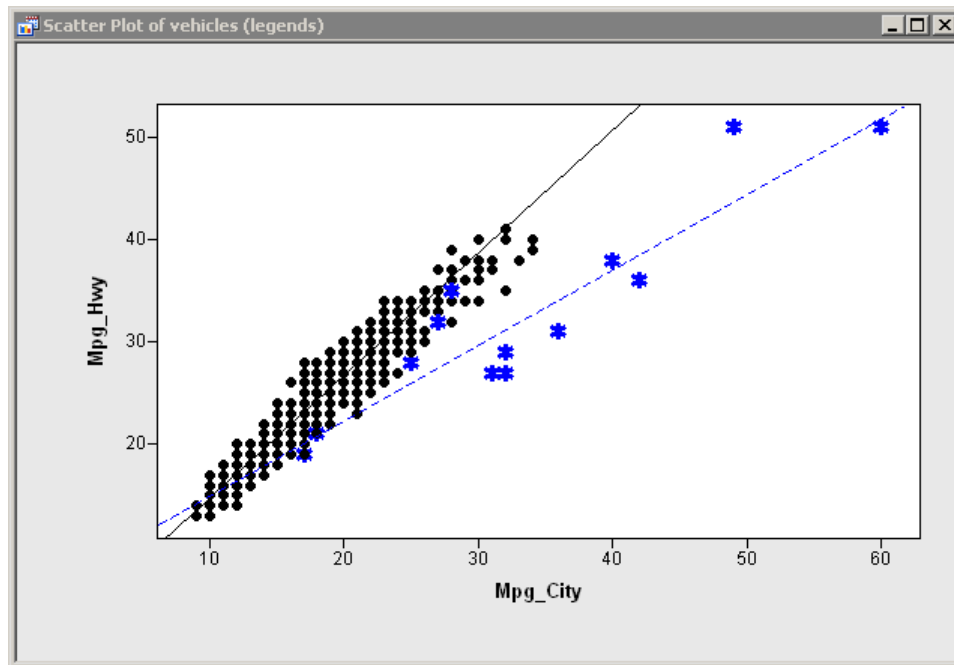
```
/* set markers shape and color based on whether hybrid-electric */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser", "vehicles");

/* traditional = black circle; hybrid = blue star */
dobj.GetVarData("Hybrid", h);
dobj.SetMarkerShape(loc(h=0), MARKER_CIRCLE);
dobj.SetMarkerShape(loc(h=1), MARKER_STAR);
dobj.SetMarkerColor(loc(h=1), BLUE);

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Mpg_City", "Mpg_Hwy");
p.SetMarkerSize(6);
```

You can use the Plot.DrawSetPenAttributes and Plot.DrawLine methods to add regression lines to the graph, as shown in [Section 9.4](#). The final graph is shown in [Figure 9.13](#). Although the person that created this graph would have no problem understanding it, the graph needs a legend if it is going to be understood by others who do not know which observations represent hybrid-electric vehicles.

Figure 9.13 A Scatter Plot with Multiple Marker Shapes and Line Styles



The DrawLegend module has several arguments. The syntax is:

DrawLegend(*Plot plot, Labels, Size, Color, Style, Symbol, BGColor, Loc*);

The first argument specifies the Plot object on which to draw the legend. The *Labels* argument specifies the text entries in the legend, and the *Size* argument determines the size of the text. The next three arguments determine the line color, linestyle, and symbol shape of graphical elements that are associated with each text item. The *BGColor* argument determines the fill-color of the rectangle that defines the legend area, or you can specify -1 to draw only the boundary of the rectangle.

The last argument specifies where the legend should be placed in the graph. The argument is a three-character string. Each character specifies a location according to the following rules:

First Character As described in [Section 9.1.3](#), a graph has two main areas: the plot area and the graph area. The valid options are:

- I** specifies that the legend is drawn inside the plot area.
- O** specifies that the legend is drawn outside the plot area. That is, in the graph area.

Second Character The second character specifies the horizontal placement of the legend. The valid options are:

- L** specifies that the legend is drawn near the left side of the plot or graph area.
- C** specifies that the legend is drawn in the center of the plot or graph area.
- R** specifies that the legend is drawn near the right side of the plot or graph area.

Third Character The third character specifies the vertical placement of the legend. The valid options are:

- T** specifies that the legend is drawn near the top of the plot or graph area.
- C** specifies that the legend is drawn in the center of the plot or graph area.
- B** specifies that the legend is drawn near the bottom of the plot or graph area.

The following statements call the DrawLegend module to create a legend for [Figure 9.13](#). The legend is shown in the lower right of [Figure 9.14](#).

```
/* define arguments for DrawLegend module */
Labels = {"Traditional" "Hybrid-Electric"};
LabelSize = 12;                               /* size of font */
LineColor = BLACK || BLUE;
LineStyle = SOLID || DASHED;
Symbol    = MARKER_CIRCLE || MARKER_STAR;
BGColor = -1;                                  /* -1 means "transparent background" */
Location  = "IRB";                             /* Inside, Right, Bottom */

run DrawLegend(p, Labels, LabelSize, LineColor, LineStyle,
               Symbol, BGColor, Location);
```

9.2.2 Drawing an Inset

An inset is often used to display descriptive statistics: the name of a statistic appears in the left column and the value in the right column. Common statistics suitable for an inset include the number of observations, a correlation coefficient, parameter estimates, and so on.

You can use the `DrawInset` module to draw a rectangle that contains two columns of text. The syntax for the `DrawInset` module is similar to the syntax for the `DrawLegend` module. The syntax is:

```
DrawInset(Plot plot, Labels, Values, Properties, Typeface, BGColor, Loc);
```

The first argument to the module is the `Plot` object on which to draw the inset. The second and third arguments to the `DrawInset` module specify the labels and the values that appear in each column of the inset. The values can be numeric (in which case, an `NLBESTw.` format is applied to the values), or you can specify character strings.

The *Properties* argument is a vector of properties that describe the typeface used in the inset. For this argument, a single missing value means “use default values.” If you want to override some default value, allocate a vector of five missing values and override the element that corresponds to the option that you want to change. For example, the following statements create the `LabelProps` vector as a vector of five missing values, but then override the default font size by assigning a value to the first element:

```
LabelProps = j(5, 1, .);    /* use default values for labels      */
LabelProps[1] = 12;        /* override size of font    */
```

The *Typeface* argument specifies the name of a typeface such as “Arial.” You can specify the default typeface by specifying an empty matrix or a zero-length string for this argument, as shown in the following statements. The remaining arguments for the `DrawInset` module are identical to the final arguments for the `DrawLegend` module.

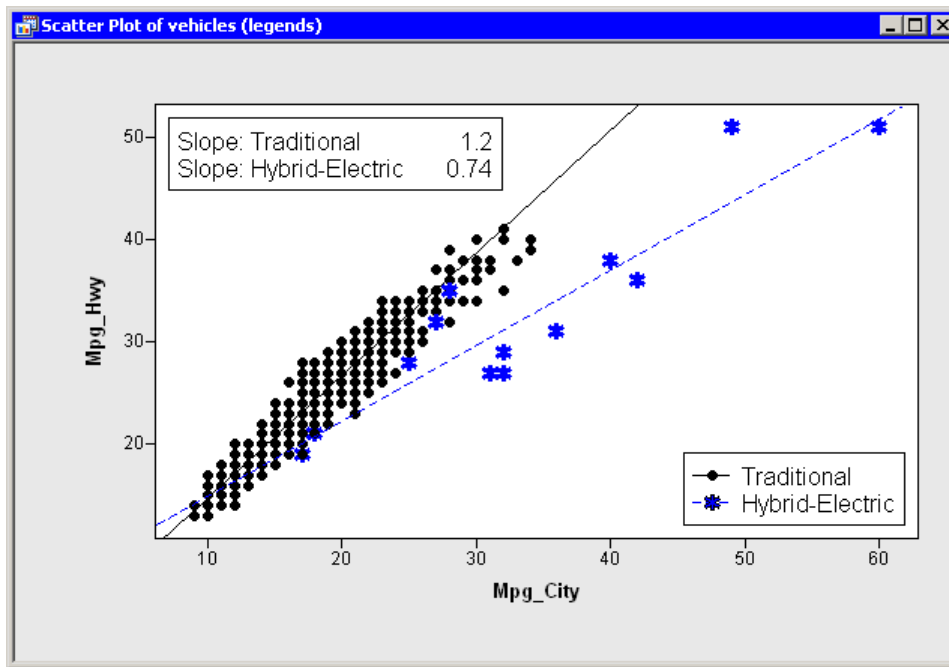
```
/* define arguments for DrawInset module */
Labels = {"Slope: Traditional" "Slope: Hybrid-Electric"};
Values = {1.2 0.74};          /* values associated with labels */
LabelProps = j(5, 1, .);    /* use default values for labels */
LabelProps[1] = 12;        /* override size of font        */

Typeface = "";              /* empty string means "default typeface" */
BGColor = -1;              /* -1 means "transparent background" */
Location = "ILT";          /* Inside, Left, Top */

run DrawInset(p, Labels, Values, LabelProps,
              LabelTypeface, BGColor, Location);
```

The inset is shown in the upper left of [Figure 9.14](#).

Figure 9.14 A Graph with a Legend and Inset



The DrawLegend and DrawInset modules share a common feature: you can pass in missing values or empty matrices in order to obtain default behavior. Some object-oriented programming languages such as Java or C++ enable the programmer to provide default values for unspecified arguments to a function. Unfortunately, the SAS/IML language does not enable you to define a module with optional arguments. However, when you write a module you can—and should—look at the type and value of an argument and use a default value if the argument is an empty matrix or a missing value.

Programming Tip: When you write a module, write it so that the user can pass in a missing value or an empty matrix in order to specify a default value for the argument.

For example, if `arg` is the name of a module argument, you can use the following statements to determine whether to use a default value for the argument:

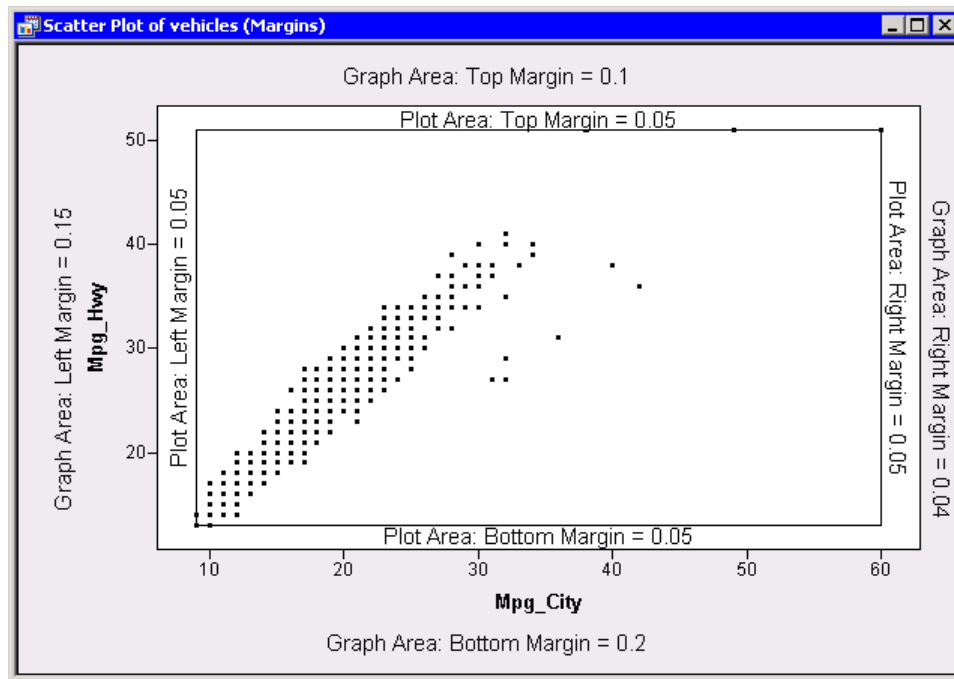
```
/* determine whether to use default value for a module argument */
DefaultArgValue = 12345;          /* define default value for argument */
if type(arg)='U' then              /* arg is empty matrix, use default */
    argValue = DefaultArgValue;
else if arg=. then                 /* arg is missing value, use default */
    argValue = DefaultArgValue;
else
    argValue = arg;                /* use value passed into module */
```

9.3 Adjusting Graph Margins

As discussed previously, there are two main areas in an IMLPlus graph. The plot area displays the data; the graph area displays tick marks, axis labels, and titles. In Figure 9.15, the plot area is the white region and the graph area is the gray area. If you draw a legend that is positioned outside of the plot area, the DrawLegend module automatically increases the graph area margins in order to fit the legend. If you are drawing your own annotations outside of the plot area, you might need to increase the graph margins to make room for the annotations. You can also decrease the graph area margins to suit your preferences. This section describes how to set margins in a graph.

Both the plot area and the graph area contain margins. Plot area margins add space around the data display so that observations are separated from the edge of the plot area. Margins in the graph area add space outside of the plot area so that there is room for axes and titles. Figure 9.15 shows the default margins for a scatter plot. The margins are specified as a proportion of the height or width of the graph. For example, in Figure 9.15 the left margin of the graph area occupies 15% of the width of the graph and the top margin of the plot area occupies 5% of the height of the plot area.

Figure 9.15 Margins in the Plot Area and Graph Area



You can increase or decrease the graph margins by using the SetGraphAreaMargins method in the Plot class. The method takes four arguments: the percentage of the graph that the margin should occupy on the left, right, top, and bottom of the graph, respectively. You can pass in the value -1 for an argument to indicate that the method should not change the corresponding margin. You can use the GetGraphAreaMargins method to retrieve the current settings for the margins. For example, the following statements create a scatter plot, print the default margins, and change three of the margins:

```

/* get and set margins in the graph area */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser", "vehicles");

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Mpg_City", "Mpg_Hwy");

p.GetGraphAreaMargins(left, right, top, bottom);
print left right top bottom;
p.SetGraphAreaMargins(0.1, 0.15, -1, 0.12);

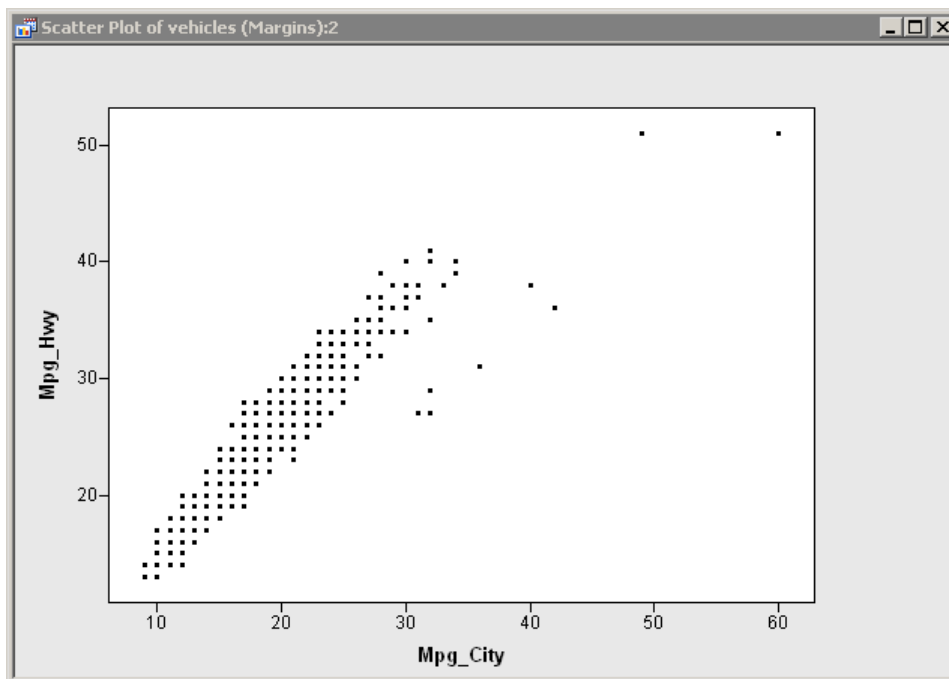
```

Figure 9.16 Default Margin Sizes, as a Fraction of the Graph Area

	left	right	top	bottom
	0.15	0.04	0.1	0.2

The scatter plot that the program creates is shown in [Figure 9.17](#). In a similar way, you can use the `SetPlotAreaMargins` to adjust the margins in the plot area.

Figure 9.17 A Plot with New Margins



9.4 A Module to Add Lines to a Graph

In creating statistical graphs, it is often useful to overlay reference lines. The simplest reference line is a horizontal line. For example, you might want to add a reference line at zero on a residual plot in order to separate positive residuals from negative. It is often useful to display the identity line. In regression diagnostics (see [Chapter 12](#)), it is useful to display lines that have statistical significance, such as lines that indicate large residuals or high-leverage values for observations.

Reference lines are used so frequently that it is worthwhile to construct a SAS/IML subroutine to draw a reference line. A convenient syntax for a module that overlays the line defined by the equation $y = a + bx$ might be as follows:

```
run abline(plot, a, b, attrib);
```

where the arguments are as follows:

<i>plot</i>	is an object of the Plot2D class such as a ScatterPlot, LinePlot, or Histogram. A line will be added to this graph.
<i>a</i>	specifies the intercept of the line. If this argument is a $k \times 1$ vector, then k lines are drawn.
<i>b</i>	specifies the slope of the line. To specify a vertical line, you can adopt the convention that the slope is a missing value and that the <i>a</i> parameter specifies the <i>x</i> -intercept. If this argument is a $k \times 1$ vector, then k lines are drawn.
<i>attrib</i>	specifies the color, style, and width of the line. You can adopt the convention that if this argument is a missing value, default line attributes are used. If this argument is a scalar integer, assume that it specifies a color parameter. If this argument contains four elements, assume that the last element is either PLOT-BACKGROUND or PLOTFOREGROUND.

The following statements define a module that is used in subsequent section to add reference lines to regression diagnostic plots:

```
/* Module to draw a vertical, horizontal, and diagonal lines
 * on a scatter plot, line plot, or histogram.
 * The form of the line is y = a + b*x.
 * INPUT  p: a Plot2D object: ScatterPlot, LinePlot, or Histogram
 *        a: specifies the intercept for the line
 *        b: specifies the slope for the line
 *        attrib: specifies line attributes.
 *           If attrib = ., use default attributes
 *           If attrib = 0xRRGGBB, it specifies a color
 *           if ncol(attrib)=3, attrib = {color,style,width}
 *           if ncol(attrib)=4, attrib = {color,style,width,PlotRegion}
 *
 * For a vertical line, a= x-intercept and b=. (MISSING).
 * To specify multiple lines, the parameters a and b can be column vectors.
 */
```



```

start abline(Plot2D p, a, b, attrib);
  if nrow(a) != nrow(b) then
    Runtime.Error("abline: incompatible intercepts and slopes");

  p.GetAxisViewRange(XAXIS, xMin, xMax);      /* get range of x      */
  x0 = xMin - (xMax-xMin);
  xf = xMax + (xMax-xMin);
  p.GetAxisViewRange(YAXIS, yMin, yMax);      /* get range of y      */
  y0 = yMin - (yMax-yMin);
  yf = yMax + (yMax-yMin);

  p.DrawUseDataCoordinates();
  if attrib=. then
    p.DrawSetPenAttributes(BLUE, DASHED, 1); /* default attributes */
  else if ncol(attrib)=1 then
    p.DrawSetPenColor(attrib);               /* set color          */
  else if ncol(attrib)>=3 then
    p.DrawSetPenAttributes(attrib[1], attrib[2], attrib[3]);
  if ncol(attrib)=4 then
    p.DrawSetRegion(attrib[4]);

  do i = 1 to nrow(a);
    if b[i]=. then                                /* vertical line      */
      p.DrawLine(a[i], y0, a[i], yf);
    else                                           /* horiz or diag line */
      p.DrawLine(x0, a[i]+b[i]*x0, xf, a[i]+b[i]*xf);
    end;
  finish;
  store module=abline;

```

The details of the module are straightforward. Notice that the default line is a thin, blue, dashed line. Those are the line attributes that are used if you pass in a missing value for the *attrib* argument.

Programming Tip: Use the `abline` module to draw lines (in the data coordinate system) on scatter plots, line plots, and histograms.

As an example, consider [Figure 9.13](#) which features two lines. One line is a solid black line; the other is blue and dashed. You can draw those lines on the scatter plot, `p`, by using the following statements:

```

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Mpg_City", "Mpg_Hwy");
p.SetMarkerSize(6);

/* draw regression lines for traditional and hybrid-electric */
attrib = BLACK || SOLID || 1;          /* traditional = solid black */
run abline(p, 2.71, 1.20, attrib);     /* y = 2.71 + 1.20*x        */

attrib = BLUE || DASHED || 1;          /* hybrid = dashed blue     */
run abline(p, 7.36, 0.74, attrib);     /* y = 7.36 + 0.74*x        */

```

9.5 Case Study: A Module to Draw a Rug Plot on a Graph

Section 6.11 shows how you can define a module that accepts an object of the Plot class. Because the Plot class is the base class for multiple graph types, you can call the module on a wide variety of graphs: scatter plots, bar charts, and so on.

It is also possible to write a module that examines the type of the incoming Plot object and calls methods based on whether the object is a member of a specific derived class. For example, suppose you want to draw little tick marks that indicate the distribution of the X variable at the bottom of histograms and scatter plots, as shown in Figure 9.18. (This is sometimes called a “rug plot.”) This sounds relatively simple, but there are some implementation issues to consider:

- How can the module determine if a graph is an object of the Histogram or ScatterPlot class? The answer is that the `IsInstance` method in the `DataView` class returns **true** if the argument to the method is an object of a specified class.
- What should the module do if a variable in the scatter plot is a nominal variable? In this case, the following module returns without drawing the plot.
- The scatter plot has, by default, a nonzero margin at the bottom of the area, whereas the histogram does not. How does that affect the placement of the tick marks for the rug plot? The following module overlays tick marks on the histogram bars, but for a scatter plot it draws the tick marks in the plot area margin.

The following statements implement the `RugPlot` module, which adds a rug plot to a histogram or to a scatter plot:

```
/* argument of module can be scatter plot or histogram */
start RugPlot(Plot2D p);
  /* draw a "rug plot" for the X variable */
  if !Histogram.IsInstance(p) &
    !ScatterPlot.IsInstance(p) then
    return;
    /* 1 */

  declare DataObject dobj;
  dobj = p.GetDataObject();
  p.GetVars(ROLE_X, xVarName);
  dobj.GetVarData(xVarName, x);
  /* 2 */
  /* 3 */
  /* 4 */

  /* for scatter plot, check for nominal variables */
  if ScatterPlot.IsInstance(p) then do;
    p.GetVars(ROLE_Y, yVarName);
    if dobj.IsNominal(xVarName) |
      dobj.IsNominal(yVarName) then
      return;
      /* 5 */
    end;

  p.GetAxisViewRange(YAXIS, yMin, yMax);
  /* 6 */
```

```

y0 = j(nrow(x), 1, yMin);
if Histogram.IsInstance(p) then
    y1 = y0 + 0.05*(yMax-yMin);          /* 7 */
else
    y1 = y0 - 0.05*(yMax-yMin);
p.DrawUseDataCoordinates();
p.DrawLine(x, y0, x, y1);              /* 8 */
finish;

```

The main steps in the module are described in the following list:

1. Use the `IsInstance` method in the `DataView` class to determine whether the argument to the module is a scatter plot or a histogram. If it is neither, the module silently returns. (Alternatively, you could use the `Runtime.Warning` method to display a warning message.)
2. The module requires getting data values for the `X` variable, so use the `GetDataObject` method in the `DataView` class to obtain the data object that is associated with the graph.
3. Use the `GetVars` method in the `Plot` class to obtain the name of the `X` variable for the graph.
4. Get the data for the `X` variable. The `x` vector contains the horizontal positions for the tick marks in the rug plot.
5. If the graph is a scatter plot and either variable is nominal, then the module returns without drawing the rug plot. This is done because the subsequent call to the `DrawUseDataCoordinates` method fails unless both axes are interval variables. (Alternatively, you could set up a user-defined coordinate system; see the documentation for the `Plot.DrawUseNormalizedCoordinates` method.)
6. Get the range of the `Y` axis. For a scatter plot, the minimum and maximum values of the axis are the minimum and maximum values of the `Y` data. For a histogram, the minimum value is zero and the maximum value is the height of the tallest bar.
7. Drawing the rug plot requires drawing many short lines that all begin at the same vertical position and all have the same height. The vector `y0` is a vector of starting vertical positions; it consists of repeated values of `yMin` . The vector `y1` is a vector of ending vertical positions. It differs from `y0` by an amount equal to 5% of the height of the vertical axis. For a histogram, each tick begins at `yMin` and is drawn toward the center of the histogram. For a scatter plot, each tick is drawn away from the plot center (into the plot area margin).
8. Draw the entire rug plot with a single call to the `DrawLine` method. The i th line is drawn from the point $(x[i], y0[i])$ to the point $(x[i], y1[i])$. Notice that the module does *not* loop over all observations and call the `DrawLine` method once for each tick mark.

Programming Tip: The `DrawLine` method in the `Plot` class can draw many line segments in a single call.

The module could be modified to support drawing a rug plot for the `Y` variable of a scatter plot and even a box plot, but that modification is not shown here. The following statements call the module on a scatter plot and a histogram.

```

/* draw rug plot on scatter plot and histogram */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

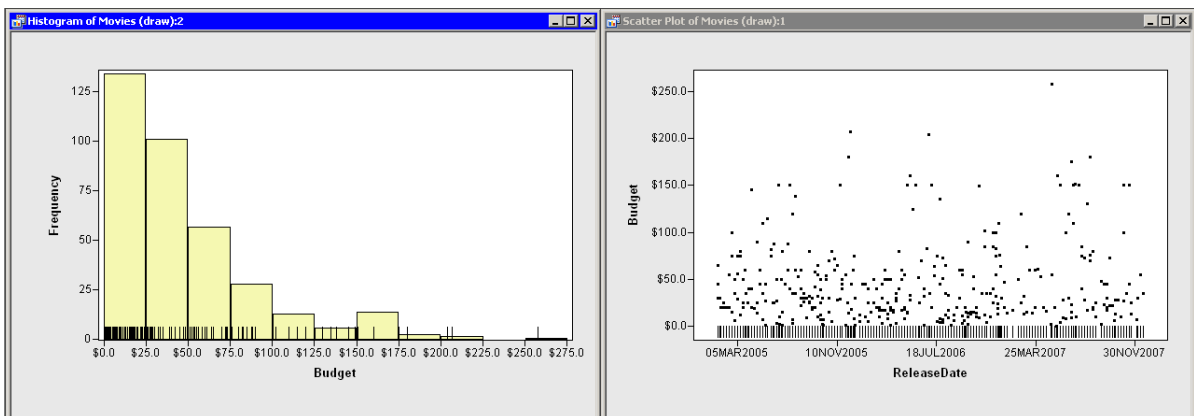
declare ScatterPlot scat;
scat = ScatterPlot.Create(dobj, "ReleaseDate", "Budget");
run RugPlot(scat);

declare Histogram hist;
hist = Histogram.Create(dobj, "Budget");
run RugPlot(hist);

```

The results are shown in Figure 9.18. The rug plot on the histogram clearly shows the nonuniform distribution of the Budget variable. The rug plot on the scatter plot shows a fairly uniform distribution of dates because movies are typically released on Fridays. (However, notice that the rug plot does not indicate how many movies are released on a given day because of overplotting.) The dark bands in the scatter plot are caused by the movies that are released on non-Fridays.

Figure 9.18 Rug Plots in a Histogram and a Scatter Plot



Programming Tip: You can write modules that handle multiple graph types by specifying that the argument to the module is an object of a base class such as the Plot class. You can use the `DataView.IsInstance` method to determine whether the object belongs to a particular derived class. For example, `ScatterPlot.IsInstance(p)` returns **true** if and only if **p** is an object of the ScatterPlot class.

9.6 Case Study: Plotting a Density Estimate

Chapter 4, “Calling SAS Procedures,” describes how to define an IMLPlus module, `ComputeKDE`, that computes a kernel density estimate (KDE) for univariate data. The module is used to create Figure 4.5, which shows a KDE overlaid on a histogram of the `Engine_Liters` variable in the `Vehicles` data set, but no program statements were given.

This section uses the ComputeKDE module to reproduce [Figure 9.19](#). The module either must be defined in the same program window or must be stored in a directory on the IMLPlus module search path, as described in [Section 5.7](#).

Recall that the following statements call the ComputeKDE module:

```
DSName = "Sasuser.Vehicles";
VarName = "Engine_Liters";
Bandwidth = "MISE";
run ComputeKDE(x, f, DSName, VarName, Bandwidth);
```

After the module runs, the vector **x** contains evenly spaced values between the minimum and maximum values of Engine_Liters. The vector **f** contains corresponding values for the density of Engine_Liters. The following statements overlay this information on a histogram of Engine_Liters:

```
/* overlay density estimate on histogram */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet (DSName);      /* 1 */

declare Histogram hist;
hist = Histogram.Create(dobj, VarName);                  /* 2 */

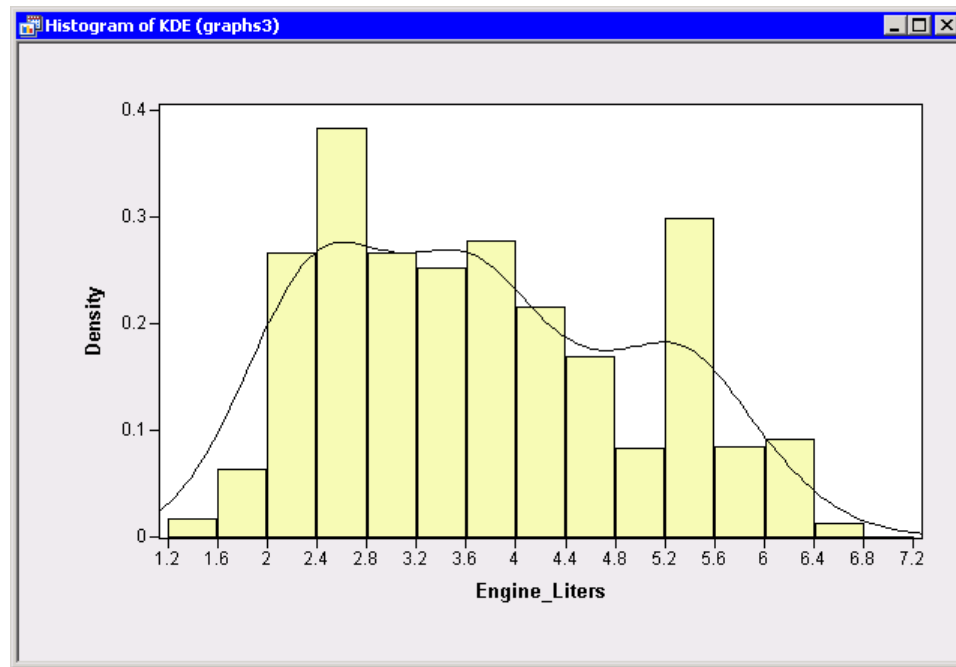
hist.ShowDensity(); /* display density instead of frequency */
hist.DrawUseDataCoordinates();
hist.DrawLine(x, f);                                     /* 3 */
```

The program consists of the following steps:

1. Read the data set. Notice that the string in the **DSName** vector is passed to the DataObject method.
2. Create the histogram. Notice that the variable name is not hard-coded, but is set to the value that is contained in **VarName**.
3. The ShowDensity method is called so that the vertical axis of the histogram is on the same scale as the KDE. Then the KDE is overlaid on the histogram.

The histogram is shown in [Figure 9.19](#). Notice that this program is “reusable” in the following sense: you can change the value of **DSName** and **VarName** and the program creates a histogram and KDE of the new data.

Programming Tip: It is good programming practice to write reusable programs that are independent of the data being analyzed.

Figure 9.19 Histogram with Kernel Density Estimate

The KDE shown in [Figure 9.19](#) indicates that the density of the `Engine_Liters` variable is multi-modal. Most of the engine displacements are in the range 2.4–3.6 liters, with another group of vehicles having displacements near 5.3 liters. The shape of the KDE suggests that there are at least two distinct groups that comprise the 2.4–3.6 range, since there are two peaks in that range. It is possible that the density for the `Engine_Liters` variable is a mixture density: the sum of a set of component densities, with the major components being the densities from the four-, six-, and eight-cylinder vehicles.

9.7 Case Study: Plotting a Loess Curve

The techniques presented in this chapter enable you to create dynamically linked statistical graphs, to call methods that change attributes of the graphs, to overlay curves or regions on the graph, and to draw text and figures in the plot area or in the margins of the graph area. This section combines those techniques to overlay a loess curve on a scatter plot.

The right side of [Figure 9.18](#) shows a scatter plot of `Budget` versus `ReleaseDate` for the `Movies` data set. The graph suggests that movies released in the summer months and in the weeks prior to Christmas day have larger budgets than those released at other times. One way to model the relationship between these two variables is with a nonparametric smoother.

This section calls the LOESS procedure on these data to determine how the mean budget of a movie is related to its release date. A loess model estimates the mean budget for a given release date, t_0 , by local estimation: only the budgets for the k movies released closest to t_0 are used to predict the budget at t_0 . The parameter k can be set explicitly, or the LOESS procedure can choose a value for

k that optimizes a criterion such as generalized cross validation (GCV) or the corrected Akaike's information criterion (AICC).

The following statements create the scatter plot:

```
/* create a scatter plot */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "ReleaseDate", "Budget");
```

The goal of this section is to add a loess fit to this scatter plot. You can call the LOESS procedure in a SUBMIT block. The LOESS procedure does not have an OUTPUT statement; the output data set is created by using a SCORE statement in conjunction with ODS OUTPUT to write a dataset that contains the predicted values, as shown in the following statements:

```
/* compute loess curve for data */
submit;
proc loess data=Sasuser.Movies;
  model Budget = ReleaseDate / select=AICC(presearch); /* 1 */
  score; /* 2 */
  ods output ScoreResults=LoessOut; /* 3 */
  ods select SmoothingCriterion FitSummary;
run;

proc sort data=LoessOut; /* 4 */
  by ReleaseDate;
run;
endsubmit;
```

Figure 9.20 Output from the LOESS Procedure

The LOESS Procedure	
Dependent Variable: Budget	
Optimal Smoothing Criterion	
AICC	Smoothing Parameter
8.44858	0.15181

Figure 9.20 *continued*

The LOESS Procedure	
Selected Smoothing Parameter: 0.152	
Dependent Variable: Budget	
Fit Summary	
Fit Method	kd Tree
Blending	Linear
Number of Observations	359
Number of Fitting Points	65
kd Tree Bucket Size	10
Degree of Local Polynomials	1
Smoothing Parameter	0.15181
Points in Local Neighborhood	54
Residual Sum of Squares	569375
Trace[L]	12.70446
GCV	4.74794
AICC	8.44858

The SUBMIT block consists of the following steps:

1. Call the LOESS procedure to create a model of Budget by ReleaseDate. The SELECT= option is used to select a value of the smoothing parameter that minimizes the AICC statistic. The PRESEARCH suboption helps to find a global minimum of the AICC statistic. This suboption improves the likelihood that the parameter value found by the LOESS procedure is a global minimum of the AICC.
2. The SCORE statement specifies that the loess model be evaluated at the values of the explanatory variables in the input data set.
3. The LOESS procedure does not support an OUTPUT statement, but you can use the ODS OUTPUT statement to create a data set, LoessOut, that contains the predicted values of the loess model at each observation in the input data set. The predicted values are contained in the variable P_Budget. (In general, the LOESS procedure names the variable that contains predicted values P_YVar, where YVar is the name of the response variable.)
4. The output data are sorted according to the values of ReleaseDate in preparation for plotting the predicted values on the scatter plot.

The output from the LOESS procedure is shown in Figure 9.20. The output shows that the AICC statistic is optimized by a smoothing parameter of 0.1518. This implies that approximately 15% of the 359 observations, or 54 points, are used in each local fit.

After the output data are created and sorted, the predicted values are read into SAS/IML vectors and plotted on the scatter plot by using the following statements:

```
/* overlay loess curve on scatter plot */
use LoessOut;
read all var {"ReleaseDate" "P_Budget"};
close LoessOut;
```



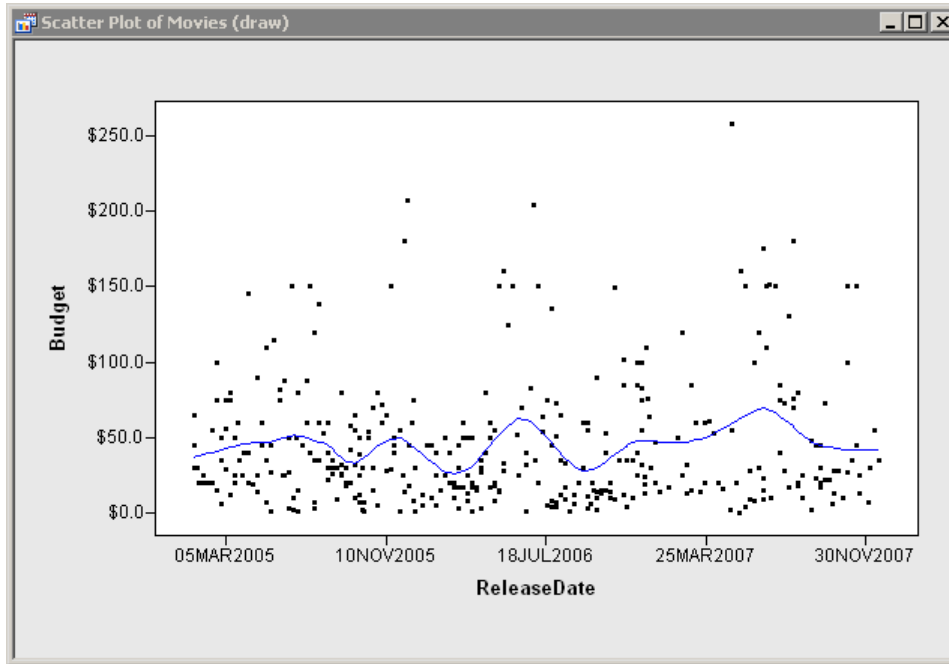
```

p.DrawUseDataCoordinates();
p.DrawSetPenColor(BLUE);
p.DrawLine(ReleaseDate, P_Budget);

```

These statements are discussed in previous sections. The result is shown in Figure 9.21.

Figure 9.21 A Loess Curve



As expected, the smooth curve does exhibit some undulations. The curve tends to be highest in the summer months. It also exhibits peaks in December of 2005 and 2006, but it does not seem to peak in December of 2007.

The data exhibits an interesting feature in the spring of 2007: the smoothed curve does not decrease as much in spring 2007 as it did in spring of 2006. There are two reasons for this. First and most importantly, there were two movies released in the spring of 2007 that had relatively large budgets: one released 16FEB07 had a budget of 120 million dollars and another released 02MAR07 had a budget of 85 million dollars. Secondly, recall that this loess fit uses 54 points in each local neighborhood. Because there were relatively few movies released in the spring of 2007, the local neighborhood for, say, 02MAR07 includes movies more than 100 days before and after that date. In particular, the big-budget movies of December 2006 and May 2007 are included in the local neighborhood for 02MAR07. In contrast, the local neighborhood for 02MAR06 includes movies at most 70 days before and after that date.

9.8 Changing Tick Positions for a Date Axis

This section describes how to modify the placement of ticks on a graph that displays a date variable.

Although [Figure 9.21](#) adequately displays the data and the loess fit of the data, the tick marks for the horizontal axis are less than satisfactory. It is difficult to discern the location of Christmas day and of the summer months—dates that are important for the analysis of the data. This section describes how you can modify the placement of tick marks on an axis by using the `SetAxisTicks` method.

The IMLPlus graphs display numeric axes with evenly spaced tick marks. The two parameters that are used to control the placement of numeric tick marks are the anchor position, x_0 , and the tick unit, Δx . (The IMLPlus methods that set these parameters are `SetAxisTickAnchor` and `SetAxisTickUnit`.) The tick marks are placed at the positions $x_0 \pm k\Delta x$ for $k = 0, 1, 2, \dots$. This approach does not work well for data that represent dates because the months of the year each have a different number of days, so there is no value for Δx that results in, say, tick marks at the beginning of each month.

The resolution to this difficulty is to use irregularly spaced tick marks. You can use the `SetAxisTicks` method to specify the positions and labels for tick marks.

For definiteness, suppose you want to modify [Figure 9.21](#) to display a tick mark at the beginning of each month. How can you accomplish this? Recall that the `ReleaseDate` variable contains the unformatted numeric values, as discussed in [Section 7.10](#). In particular, you can use SAS/IML statements to find the minimum (earliest) and maximum (latest) release date. You can use those values to enumerate each day between those values. You can then apply a `DATEw.` format to those numbers and use string matching functions to find the values that correspond to the first day of each month. The following statements carry out this algorithm:

```
/* plot tick marks at beginning of each month when data has DATE format */
xData = ReleaseDate;                                /* 5 */
fmt = "DATE7.";

allX = min(xData):max(xData);                        /* 6 */
allXText = putn(allX, fmt);                          /* 7 */

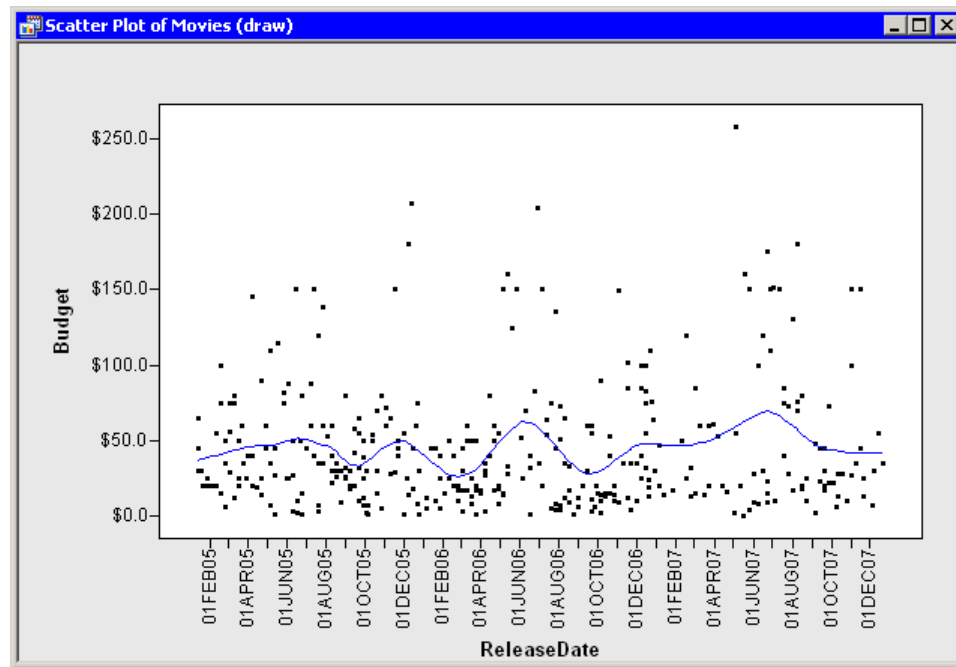
q = substr(allXText, 1, 2);                          /* 8 */
idx = loc(q="01");                                   /* 9 */
pos = allX[idx];
values = allXText[idx];
print (pos[1:5]) [label= "pos"] (values[1:5]) [label= "value"];
p.SetAxisTicks(XAXIS, pos, values);                  /* 10 */
```

Figure 9.22 First Few Values of Tick Marks

pos	value
16468	01FEB05
16496	01MAR05
16527	01APR05
16557	01MAY05
16588	01JUN05

The output from the previous statements is shown in [Figure 9.22](#). The following list explains key steps in the algorithm:

5. For convenience, a new variable **xData** is created as a copy of the **ReleaseDate** variable. This step is not necessary, but makes it easier for the reader to use this technique in his own programs.
6. A numerical vector, **allX**, is created. This vector contains the numeric value for each day between the minimum and maximum values of **xData**.
7. A character vector, **allXText**, is created that contains the formatted values that correspond to **allX**. The Base SAS function PUTN applies the DATE7. format to the values in **allX**.
8. Which dates correspond to the first of the month? The ones whose formatted values start with the string “01”. This step uses the Base SAS function SUBSTR to create a character vector, **q**, that contains the first two characters of the strings contained in **allXText**. The values of **q** are the days of the month: “01”, “02”, and so forth to “31”.
9. The program uses the LOC function to find the indices of **q** that match the string “01”. These indices are the dates that represent the first day of each month. For convenience, the program prints a few of the unformatted and formatted values for these dates, as shown in [Figure 9.22](#).
10. The SetAxisTicks method sets the new positions and values for the tick marks of the horizontal axis. The resulting plot is shown in [Figure 9.23](#).

Figure 9.23 Modified Tick Marks for a Date Variable

Programming Technique: You can use the program statements in this section to find the position and labels for dates that satisfy a certain criterion. In particular, you can use the PUTN function to apply a format to the underlying data, and you can use the SetAxisTicks method to explicitly set the position and labels for tick marks on an axis.

9.9 Case Study: Drawing Arbitrary Figures and Diagrams

Although SAS/IML Studio does not provide a “blank canvas” on which to draw diagrams and other figures, you can use the following technique to simulate a blank drawing area:

1. Define a DataObject with two observations.
2. Set the marker color of those observations to NOCOLOR (that is, invisible).
3. Create a scatter plot of these data.

The result is a blank window on which to draw arbitrary figures. The following statements illustrate this technique by drawing the probability density function for the normal distribution on the interval $[-3, 3]$. The programs also draw lines that correspond to the 5th and 95th quantiles of the distribution. The resulting plot is shown in [Figure 9.24](#).

```

/* create "blank canvas"; draw arbitrary shapes and figures */
coords = {-3 0, 3 0.5};          /* window shows [-3,3]x[0,0.5] */

declare DataObject dobj;
dobj = DataObject.Create("Canvas", {"x" "y"}, coords);
dobj.SetMarkerColor(OBS_ALL, NOCOLOR); /* make markers invisible */

declare ScatterPlot canvas;
canvas = ScatterPlot.Create(dobj, "x", "y");
canvas.DrawUseDataCoordinates(); /* window range: min/max of data */

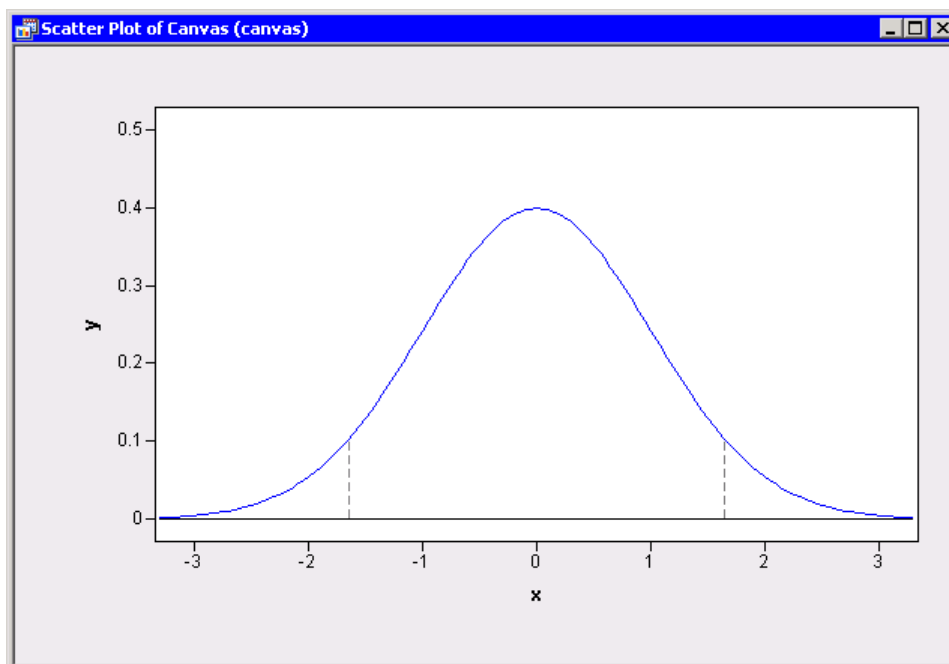
x = do(-3.3, 3.3, 0.05);          /* evenly spaced points */
canvas.DrawLine(x, 0*x);          /* draw reference line at y=0 */
y = pdf("normal", x);             /* evaluate function at x values */
canvas.DrawSetPenColor(BLUE);     /* set pen color to blue */
canvas.DrawLine(x, y);            /* draw normal curve */

/* draw lines at certain locations to indicate quantiles */
canvas.DrawSetPenStyle(DASHED);
canvas.DrawSetPenColor(GRAY);
q = {0.05 0.95};                  /* list of quantiles */
do i = 1 to ncol(q);
    a = quantile("normal", q[i]); /* find quantile */
    canvas.DrawLine(a, 0, a, pdf("normal",a)); /* draw dashed line */
end;

```

Programming Technique: You can use the technique in this section to create a blank window for drawing diagrams and figures.

Figure 9.24 Drawing on an Empty Canvas



9.10 A Comparison between Drawing in IMLPlus and PROC IML

There is not a one-to-one correspondence between IMLPlus drawing methods and the drawing statements in PROC IML. However, the following table lists the PROC IML statements and similar IMLPlus methods. The IMLPlus methods belong to the Plot class. You can investigate these methods further by choosing **Help ► Help Topics** and then by expanding the **IMLPlus Class Reference ► Plot** section.

Table 9.1 Low-Level Drawing Commands in PROC IML

PROC IML Statement	Similar IMLPlus Method
DISPLAY, WINDOW	N/A, although you can display dialog boxes by calling IMLPlus modules. See Section 5.8 .
GOPEN, GCLOSE	Plot.DrawBeginBlock, Plot.DrawEndBlock, Plot.DrawResetState
GDELETE	Plot.DrawRemoveCommands
GPIE	Plot.DrawArc
GDRAW, GDRAWL	Plot.DrawLine
GGRID	Plot.DrawGrid
GPOINT	Plot.DrawMarker
GPOLY	Plot.DrawPolygon
GPORT	Plot.DrawUseNormalizedCoordinates
GPORTPOP	Plot.PopState
GSCRIPT	Plot.DrawText
GSET	Plot.DrawSetBrushColor, Plot.DrawSetBrushStyle, Plot.DrawSetPenAttributes, Plot.DrawSetPenColor, Plot.DrawSetPenStyle, Plot.DrawSetPenWidth, Plot.DrawSetTextSize, Plot.DrawSetTextColor, Plot.DrawSetTextStyle, Plot.DrawSetTextTypeface
GTEXT, GVTEXT	Plot.DrawText, Plot.DrawSetTextAngle
GWINDOW	Plot.DrawUseDataCoordinates
GXAXIS, GYAXIS	Plot axes are drawn automatically. You can change the way that axes are drawn with a number of Plot class methods that start with the “SetAxis” prefix. You can manually draw additional axes with Plot.DrawAxis and Plot.DrawNumericAxis.

Chapter 10

Marker Shapes, Colors, and Other Attributes of Data

Contents

10.1	Overview of Data Attributes	225
10.2	Changing Marker Properties	226
10.2.1	Using Marker Shapes to Indicate Values of a Categorical Variable	226
10.2.2	Using Marker Colors to Indicate Values of a Continuous Variable	229
10.2.3	Coloring by Values of a Continuous Variable	232
10.3	Changing the Display Order of Categories	236
10.3.1	Setting the Display Order of a Categorical Variable	236
10.3.2	Using a Statistic to Set the Display Order of a Categorical Variable	238
10.4	Selecting Observations	241
10.5	Getting and Setting Attributes of Data	244
10.5.1	Properties of Variables	244
10.5.2	Attributes of Observations	246

10.1 Overview of Data Attributes

The `DataObject` class provides methods that can be grouped into two categories: methods that provide access to the data and methods that describe how to represent the data in graphs or analyses. The first category includes methods such as `GetVarData` and `AddVar`. These methods retrieve data or add a variable to the data object; they are described in the previous chapter.

In contrast, the second category of methods affect attributes of the data. Examples of attributes include the color and shape of an observation marker. These properties affect the way that the data are displayed in graphs.

This chapter describes how to use methods in the `DataObject` class to manage attributes of observations and variables.

10.2 Changing Marker Properties

The shape and color of observation markers can be used to visually indicate the value of a third variable that does not appear on the graph. The shape of a marker often encodes the value of a categorical variable with a small number of categories, whereas color often encodes the value of a continuous variable. For example, if you have data about patients in a drug trial, you might want to use marker shape to represent male versus female, or perhaps to distinguish individuals in a test group from individuals in the control group. In the same study, you might use color to indicate a continuous variable such as the age or weight of a patient.

This section describes how to use methods in the `DataObject` class to set properties for markers based on values of other variables.

10.2.1 Using Marker Shapes to Indicate Values of a Categorical Variable

Suppose that you want to set the shape of markers to reflect the MPAA rating of movies in the `Movies` data set. For definiteness, suppose you want to create a scatter plot of the `US_Gross` variable versus the `ReleaseDate` variable for the data in the `Movies` data set. You decide to encode the marker shapes according to the following table:

Table 10.1 Movie Ratings, Marker Symbols, and IMLPlus Constants

MPAA Rating	Symbol	IMLPlus Constant
G	□	MARKER_SQUARE
NR	×	MARKER_X
PG	△	MARKER_TRIANGLE
PG-13	+	MARKER_PLUS
R	▽	MARKER_INVTRIANGLE

The last column in the table is the IMLPlus constant that specifies a marker shape in the `SetMarkerShape` method of the `DataObject` class. The `SetMarkerShape` method enables you to specify the marker shape to use when plotting specific observations. The following program finds the observations that are associated with each MPAA rating category by using the technique presented in [Section 3.3.5](#):

```
/* use marker shape to encode category */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.SetRoleVar(ROLE_LABEL, "Title"); /* 1 */
```



```

dobj.GetVarData("MPAARating", Group);          /* 2 */
u = unique(Group);                             /* 3 */
shapes = MARKER_SQUARE || MARKER_X ||
        MARKER_TRIANGLE || MARKER_PLUS ||
        MARKER_INVTRIANGLE;                   /* 4 */
do i = 1 to ncol(u);                           /* 5 */
    idx = loc(Group = u[i]);
    dobj.SetMarkerShape(idx, shapes[i]);        /* 6 */
end;

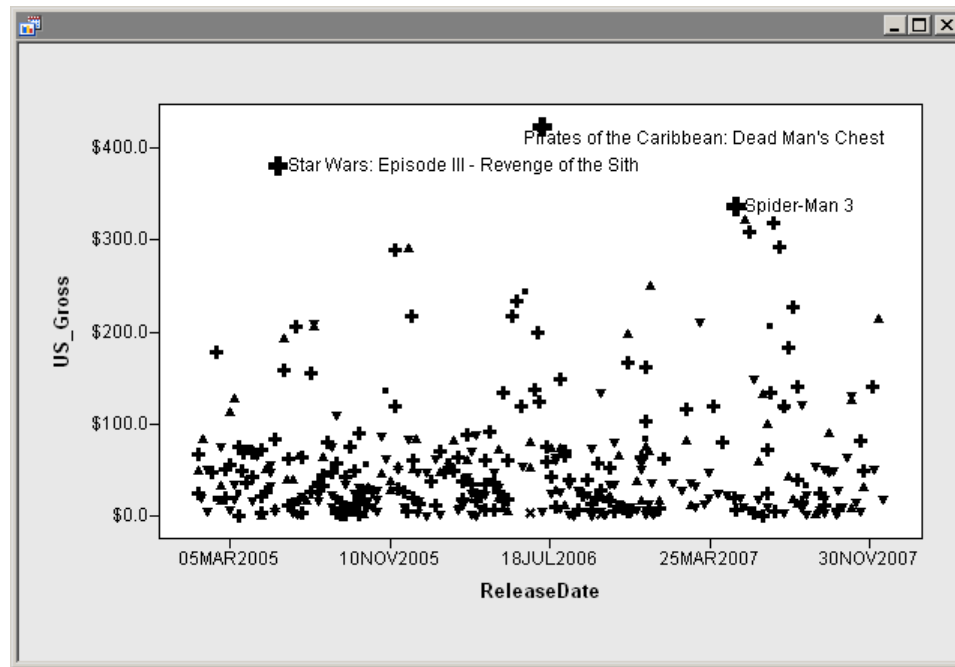
declare ScatterPlot plot;
plot = ScatterPlot.Create(dobj, "ReleaseDate", "US_Gross");
plot.SetMarkerSize(5);                         /* 7 */

```

The program begins by creating a data object from the Movies data set. The program consists of the following main steps:

1. The SetRoleVar method is called. (This statement is optional.) When you click on an observation marker in a scatter plot, the value of the “role variable” is displayed in the graph. For this example, clicking on an observation displays the title of the movie.
2. The GetVarData method in the DataObject class retrieves data from the data object. The SAS/IML vector **Group** is created to hold the values of the MPAARating variable. Notice that, in this case, you could also have gotten these data from Sasuser.Movies by using the SAS/IML USE and READ statements.
3. The UNIQUE function returns a sorted vector of the unique values of the **Group** vector. These values are stored in the vector **u**.
4. A vector, **shapes**, is created. It contains the marker shapes that correspond to [Table 10.1](#).
5. The program loops over the group categories in **u**. The purpose of the loop is to assign the shape in **shape[i]** to all observations whose MPAA rating is equal to **u[i]**.
6. After the LOC function finds the observations in each category, the SetMarkerShape sets the marker shape for those observations. Notice that you do not need to check whether **idx** is empty, because each category in **u** occurs at least once in **Group**.
7. After the scatter plot is created, the SetMarkerSize method increases the size of markers displayed in the graph. If the marker size is too small, it can be difficult to distinguish between different marker shapes.

The scatter plot appears in [Figure 10.1](#). You can click on an observation in a scatter plot to select that observation. In [Figure 10.1](#) several of the largest-grossing movies are selected, which causes the titles of the movies to be displayed. Notice that most of the top-grossing movies are rated PG-13 (+), followed by PG-rated movies (Δ). There are few R-rated movies (▽) that grossed more than \$100 million, which is surprising considering the total number of R-rated movies in the data.

Figure 10.1 Marker Shapes Correspond to MPAA Rating

It is interesting to note that the largest grossing titles are all sequels that are part of successful movie franchises: *Pirates of the Caribbean*, *Star Wars*, and *Spider-Man*. The reader is invited to explore other top-grossing movies in the data and determine the titles of the movies. How many are sequels?

The program in this section used prior knowledge of the data in order to construct the **shapes** vector: the **shapes** vector has five shapes, one for each category of the **MPAARating** variable. However, in general, you might not know how many categories a variable contains. For example, none of the movies included in this data set are rated NC-17, but that is a valid (but rare) MPAA rating.

One solution to this problem is to reuse shapes if there are more categories than shapes. The following statements use the SAS/IML MOD function to ensure that the index into the **shape** vector is always a number between 1 and the number of shapes, regardless of the value of the looping variable **i**:

```
/* if more categories than shapes, reuse shapes */
j = 1 + mod(i-1, ncol(shapes));
shape = shapes[j];
doobj.SetMarkerShape(idx, shape);
```

Recall that the expression $\text{mod}(s, t)$ gives the remainder after dividing s by t . Consequently, the expression $1 + \text{mod}(i - 1, n)$ is a number between 1 and n for all positive integers i .

You can use this programming technique to write a module that sets the marker shapes of observations according to the categories of some variable, as shown in the following statements:

Programming Technique: You can set the marker shapes of observations according to the categories of the variable *VarName* by calling the following module:

```

/* module that uses marker shape to encode category */
start SetMarkerShapeByGroup(DataObject dobj, VarName);
  dobj.GetVarData(VarName, Group);
  u = unique(Group);
  shapes = MARKER_SQUARE || MARKER_X || MARKER_TRIANGLE ||
           MARKER_PLUS || MARKER_INVTRIANGLE ||
           MARKER_CIRCLE || MARKER_DIAMOND || MARKER_STAR;
  do i = 1 to ncol(u);
    idx = loc(Group = u[i]);
    j = 1 + mod(i-1, ncol(shapes));
    shape = shapes[j];
    dobj.SetMarkerShape(idx, shape);
  end;
finish;

```

Recall from [Section 6.10](#) that you can specify objects as arguments to IMLPlus modules. The first argument of the `SetMarkerShapeByGroup` module is an object of the `DataObject` class. To pass objects to a module, you must specify the name of the class in addition to the name of the argument when you list the arguments in the `START` statement. Any arguments that are not preceded by a class name are assumed to be SAS/IML matrices.

10.2.2 Using Marker Colors to Indicate Values of a Continuous Variable

The graph in [Figure 10.1](#) displays a great deal of information. It displays two continuous variables and one categorical variable. However, the markers in the graph are all black. The `SetMarkerColor` method in the `DataObject` class enables you to specify a color for given observations.

The `SetMarkerColor` method can be used directly to color individual markers. It is also used indirectly when you call any of several modules provided with SAS/IML Studio. These modules enable you to manipulate colors and easily color observations according to the value of a variable. The subsequent subsections discuss colors and how to perform common tasks with colors in SAS/IML Studio. This section begins with a description of how to specify colors in IMLPlus.

10.2.2.1 Color Representation in IMLPlus

This section describes how colors are represented in IMLPlus. This is an advanced topic that can be omitted during an initial reading. The key point of this section is that there are two ways of representing colors: as ordered triples or as integers. There are also two modules distributed with SAS/IML Studio that enable you to convert between these representations: the `RGBToInt` and `IntToRGB` modules.

There are many ways to specify colors. IMLPlus represents colors by using the RGB coordinate system, which represents a color as an additive mixture of three primary colors: red, green, and blue.

One way to represent a color in RGB coordinates is as an ordered triple: the first coordinate represents the amount of red in the color, the second coordinate represents the amount of blue, and the

third coordinate represents the amount of green. A popular representation of colors assigns eight bits (one byte) to each of the three colors. This means that each coordinate in the RGB system is an integer between zero and 255. In these coordinates, a value of 0 means the absence of a color and a value of 255 means the color is fully present. Thus (0, 0, 0) represents black and (255, 255, 255) represents white. Red is (255, 0, 0), whereas blue is (0, 0, 255). A mixture of primary colors such as (65, 105, 225) is a bluish shade that some might call “royal blue.”

A second representation of colors packs the three RGB coordinates into a single integer. This is a very compact representation, although not very intuitive! The packing is accomplished by writing the integer in hexadecimal notation: use the lower two place-values to store the amount of blue in the color, use the third and fourth place-values to store the amount of green, and use the fifth and sixth place-values to store the red component.

In this compact representation, a value of 00 in the appropriate place-values means the absence of a color, whereas a value of FF means the color is fully present. In SAS you specify a hexadecimal value by prefixing the number with ‘0’ and appending an ‘x’ to the end of the number. Thus 0000000x represents black, whereas 0FFFFFFx represents white. Red is 0FF0000x, whereas blue is 00000FFx. The royal blue with RGB values (65, 105, 225) is compactly represented as 04169E1x.

SAS/IML Studio comes with two modules that convert colors between ordered triples and integers. The RGBToInt module converts ordered triples to integers, whereas the IntToRGB module converts in the other direction. These modules are demonstrated by the following statements:

```
/* convert colors between different representations */
RoyalBlueRGB = {65 105 225};          /* input as ordered triple */
IntColor = RGBToInt(RoyalBlueRGB);
print IntColor[format=hex6.];          /* print as hexadecimal */

RoyalBlueInt = 04169E1x;               /* input as hexadecimal */
TripletColor = IntToRGB(RoyalBlueInt);
print TripletColor;                    /* print as ordered triple */
```

The output is shown in [Figure 10.2](#). Note that the base-10 representation of the `IntColor` integer (which is 4286945) is never displayed. The hexadecimal representation of the integer is printed by using the `FORMAT=HEX6.` option. Similarly, the integer stored in `RoyalBlueInt` is specified by its hexadecimal value by using the ‘0’ prefix and the ‘x’ suffix.

Figure 10.2 Color Represented as Ordered Triple and Integer

IntColor		
4169E1		
TripletColor		
65	105	225

Programming Tip: Use the `RGBToInt` and `IntToRGB` modules to convert colors between the ordered triples and hexadecimal (integer) representations.

IMLPlus provides a number of predefined colors such as BLACK, RED, and GREEN. These predefined colors are stored as integers. The following program prints the hexadecimal values and ordered triples for a small subset of the predefined colors:

```
/* names and RGB values of predefined colors */
colName = {"BLACK", "RED", "GREEN", "BLUE", "ORANGE", "PINK", "YELLOW", "WHITE"};
color   = BLACK// RED// GREEN// BLUE// ORANGE// PINK// YELLOW// WHITE;
RGB     = IntToRGB(color);
print colName color[format=hex6.] RGB[colname={"Red" "Green" "Blue"}];
```

Figure 10.3 Hexadecimal and RGB Representation of Some Predefined Colors

colName	color	RGB		
		Red	Green	Blue
BLACK	000000	0	0	0
RED	FF0000	255	0	0
GREEN	00FF00	0	255	0
BLUE	0000FF	0	0	255
ORANGE	FF8000	255	128	0
PINK	FF0080	255	0	128
YELLOW	FFFF00	255	255	0
WHITE	FFFFFF	255	255	255

Programming Tip: The predefined colors in IMLPlus are stored as integers. You can print their hexadecimal value by using the FORMAT=HEX6. option of the PRINT statement. You can convert them to an ordered triple of RGB values by using the IntToRGB module.

10.2.2.2 Using Color to Mark Outliers

In [Section 8.8.2](#), you learned how to add residual values to a data object. It is often convenient to change the color or marker of observations whose residual values are far from zero. These observations that are not well-predicted by the model are called *outliers*.

The following statements assume that the residual values are in the data object in a variable called Resid. The statements set the color of any observation with large residuals:

```
/* set color of observations with large residual values */
dobj.GetVarData("Resid", resid);
idx = loc(resid<=-10 | resid>=10);
if ncol(idx)>0 then
    dobj.SetMarkerColor(idx, RED);
```

You can see the colored observations on a scatter plot (such as [Figure 8.6](#)) that includes the Resid variable.

The definition of a “large” residual is data dependent. In this example, an observation is colored red if the actual Mpg_City value differs from the predicted value by ten or more miles per gallon.

Although this example used an arbitrary value (10) to determine which residuals are considered

“large,” you can also detect and color outliers by using less arbitrary criteria. For example, some authors recommend computing the externally studentized residuals and examining an observation when the absolute value of the studentized residual exceeds 2. (The `RSTUDENT=` option in the `GLM OUTPUT` statement computes externally studentized residuals.) Regression diagnostics are discussed in Chapter 12, “Regression Diagnostics.”

10.2.3 Coloring by Values of a Continuous Variable

It is convenient to use marker shapes to indicate levels of a categorical variable. However, colors are better for encoding the values of continuous variables and discrete ordinal variables because you can associate one color (say, blue) with low values and another (say, red) with high values. Often one or more colors are used to represent intermediate colors.

The set of colors that represents values of a variable is called a *color ramp* (or sometimes a color map). Often a linear mapping is used to associate values of a variable to colors: the minimum value of the variable is mapped to the first color, the maximum value is mapped to the last color, and some scheme is used to associate colors with intermediate values. Nonlinear schemes are sometimes used when the distribution of the values is highly skewed.

The next two sections describe ways to associate colors with values. The simplest is for each color to represent a large range of values. (For example, you can divide the data into quartiles.) In this scheme, there is no need to interpolate colors: each observation is placed into one bin, and each bin is assigned a color. The more complicated way to associate colors with values is to specify a small number of colors but to interpolate colors so that only a small range of values is associated with a color.

One Color for a Large Range of Values

Suppose you want to color observations in the `Movies` data set according to values of the `US_Gross` variable. You decide to use four colors to encode the `US_Gross` values: blue for the movies with low values of `US_Gross`, cyan and orange for movies with intermediate values, and red for movies that generated large box-office revenues. To be specific, suppose you want to associate revenue and colors by using the following rules:

Table 10.2 Encoding Colors for Movie Revenue Ranges

US Gross	Color	Comment
(0, 25]	BLUE	A flop
(25, 100]	CYAN	A typical movie
(100, 200]	ORANGE	A hit
> 200	RED	A blockbuster hit

The values that divide one category from another are called *cutoff values* or sometimes *break points*. For this example, the cutoff values are 25, 100, and 200. The cutoff values, together with the minimum and maximum value of `US_Gross`, form endpoints of intervals. All values within an interval are assigned the same color.

A naive implementation would loop over all observations, and use the `SetMarkerColor` method to assign a color for that observation based on the cutoff values for `US_Gross`. This is inefficient. As pointed out in the section “[Writing Efficient SAS/IML Programs](#)” on page 79, you should avoid loops over all observations. It is almost always better to loop over categories and to use the `LOC` function to identify the observations that belong to each category. The following program loops over the four color categories:

```
/* color each observation based on value of a continuous variable */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.GetVarData("US_Gross", v); /* 1 */
EndPts = {0 25 100 200} || max(v); /* 2 */
Color = BLUE || CYAN || ORANGE || RED; /* 3 */
do i = 1 to ncol(Color); /* 4 */
    idx = loc(v>EndPts[i] & v<=EndPts[i+1]); /* 5 */
    if ncol(idx)>0 then
        dobj.SetMarkerColor(idx, Color[i]);
end;
```

The following list describes the main steps of the program:

1. The data in the `US_Gross` variable are copied into the vector `v`.
2. The cutoff values define four intervals. The first interval is $(0, 25]$, the second is $(25, 100]$, and so on. So that the program does not need to treat the last category (“blockbusters”) differently from the previous categories, it is useful to specify a large number (`max(v)`) to use as the right endpoint of the last interval.
3. The corresponding colors are defined as in [Table 10.2](#).
4. The loop is over the categories defined by the four intervals.
5. This is the key step of the program. The `LOC` function finds all observations in the i th interval. The i th interval consists of values greater than `EndPts[i]` and less than or equal to `EndPts[i+1]`. The color of these observations is set to `Color[i]` by calling the `SetMarkerColor` method.

You should carefully study the behavior of the statements inside the loop during the last iteration (when `i=4` in the example). The `LOC` function finds elements of `v` that exceed `EndPts[4]` (200) and are less than or equal to `EndPts[5]`, which corresponds to `max(v)`. If the `EndPts` vector had only four elements, then the program would halt with an index-out-of-bounds error. By using `max(v)` as the fifth element of `EndPts`, the program avoids this error and also avoids having to handle the case $v > 200$ differently than the other cases.

It is left as an exercise for the reader to view the colored observations by creating a scatter plot of any two variables in the `Sasuser.Movies` data set.

Color-Blending: Few Values for each Color

The previous section assigned a single color to a range of values for `US_Gross`. In mathematical terms, the function that maps values to colors is piecewise constant. It is also possible to con-

struct a piecewise linear function that maps values to colors. Defining this mapping requires linear interpolation between colors. Specifically, given two colors, what are the colors between them?

Linear interpolation of real numbers is straightforward: given numbers x and y , the numbers between them are parameterized by $(1 - t)x + ty$ for $t \in [0, 1]$. Linear interpolation of colors works similarly. If two colors, $x = (x_r, x_g, x_b)$ and $y = (y_r, y_g, y_b)$, are represented as triples of integers, then for each $t \in [0, 1]$, the triple $c = (1 - t)x + ty$ is between x and y . In general, c is not a triple of integers, so in practice you must specify a color close to c , typically by using the INT, FLOOR, or ROUND functions in Base SAS software.

Color interpolation makes it easy to color observation markers by values of a continuous variable. Let v be the vector that contains the values. Assume first that you have a color ramp with two colors, a and b (represented as triples). The idea is to assign the color a to any observations with the value $\min(v)$ and assign the color b to any observations with the value $\max(v)$. For intermediate values, interpolate colors between a and b . How can you do this? First, normalize the values of v by applying a linear transformation: for each element of v , let $t_i = (v_i - \min(v)) / (\max(v) - \min(v))$. Notice that the values of t are in the interval $[0, 1]$. Consequently, you can assign the observation with value v_i the color closest to the triple $(1 - t_i)a + t_i b$.

The following program implements these ideas for a two-color ramp that varies between light brown and dark brown. The program uses linear interpolation to color-code observations according to values of a continuous variable.

```
/* linearly interpolate color of each observation */
dobj.GetVarData("US_Gross", v);
a = IntToRGB(CREAM); /* 1 */
b = IntToRGB(BROWN);
t = (v-min(v)) / (max(v)-min(v)); /* 2 */
colors = (1-t)*a + t*b; /* 3 */
dobj.SetMarkerColor(1:nrow(v), colors); /* 4 */
```

The previous statements were not written to handle missing values in v , but can be easily adapted. This is left to the reader. It is also trivial to change the program to use a color ramp that is formed by any other pair of colors. The steps of the program are as follows:

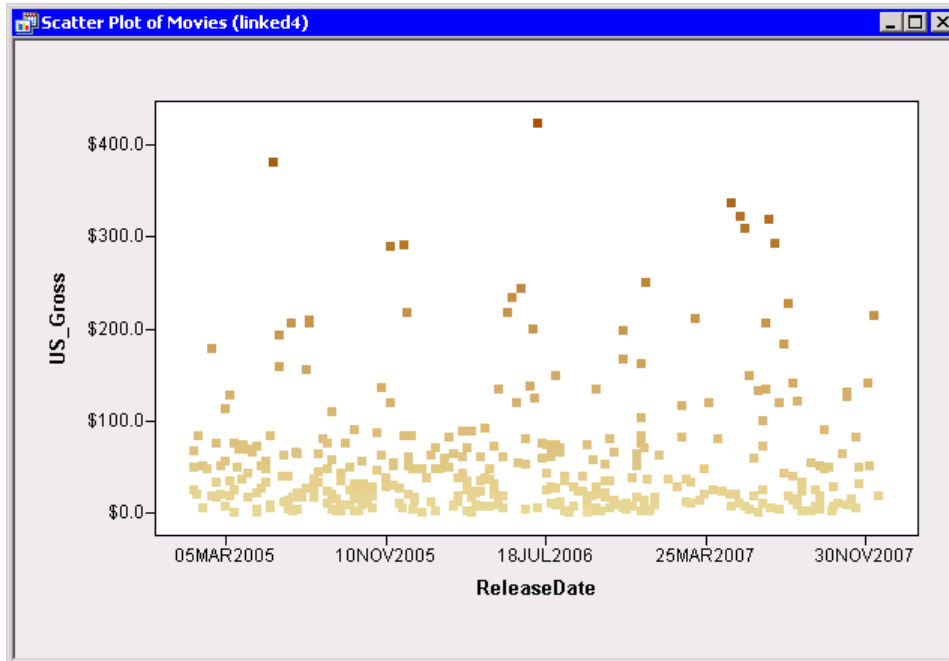
1. Represent the two colors as RGB triples, a and b .
2. Normalize the values of v . The new vector t has values in the interval $[0, 1]$.
3. Linearly interpolate between the colors. The `colors` vector does not contain integer values, but you can apply the INT, FLOOR, or ROUND functions to the `colors` vector. If you do not explicitly truncate the colors, the SetMarkerColor method truncates the values (which is equivalent to applying the INT function).
4. Call the SetMarkerColor method to set the colors of the observations.

It is not initially apparent that Step 3 involves matrix computations, but it does. The vector t is an $n \times 1$ vector, and the vectors a and b are a 1×3 row vectors. Therefore, the matrix product $t*b$ is an $n \times 3$ matrix, and similarly for the product $(1-t)*a$.

Programming Tip: To interpolate colors, represent them as triples of RGB values. The IntToRGB module might be useful for this.

The scatter plot shown in Figure 10.4 displays US_Gross on the vertical axis. Even though the image in this book is monochromatic, the gradient of colors from a light color to a dark color is apparent. In practice, it is common to use color to depict a variable that is *not* present in the graph.

Figure 10.4 A Color Ramp



The ideas of this section can be extended to color ramps with more than two colors. Suppose you have a vector v and want to color observations by using a color ramp with k colors c_1, c_2, \dots, c_k . Divide the range of v into $k - 1$ equal intervals defined by the evenly-spaced values L_1, L_2, \dots, L_k , where $L_1 = \min(v)$ and $L_k = \max(v)$. To the i th interval $[L_i, L_{i+1}]$, associate the two-color ramp with colors c_i and c_{i+1} . For an observation that has a value of v in the interval $[L_i, L_{i+1}]$, color it according to the algorithm for two-color color ramps. In this way, you reduce the problem to one already solved: to handle a color ramp with k colors, simply handle $k - 1$ color ramps with two colors.

SAS/IML Studio distributes several modules that can help you interpolate colors and color-code observations. The modules are briefly described in Table 10.3.

Table 10.3 IMLPlus Modules for Manipulating Colors

IMLPlus Module	Description
BlendColors	Interpolates colors from a color ramp
ColorCodeObs	Uses linear interpolation to color observations according to values of a continuous variable
ColorCodeObsByGroups	Colors observations according to levels of one or more categorical variables
IntToRGB	Converts colors from a hexadecimal representation to ordered triples of RGB values
RGBToInt	Converts colors from ordered triples of RGB values to a hexadecimal representation

10.3 Changing the Display Order of Categories

An *ordinal* variable is a categorical variable for which the various categories can be ordered. For example, a variable that contains the days of the week has a natural ordering: Sunday, Monday, . . . , Saturday.

By default, all graphs display categorical variables in alphanumeric order. This order might not be the most appropriate to display. For example, if you create a bar chart for days of the week or months of the year, the bar chart should present categories in a chronological order rather than in alphabetical order. Similarly, for a variable with values “Low,” “Medium,” and “High,” the values of the variable suggest an order that is not alphabetical.

Sometimes it is appropriate to order categories according to the values of some statistic applied to the categories. For example, you might want to order a bar chart according to the frequency counts, or you might want to order a box plot according to the mean (or median) value of the categories.

This section describes how to change the order in which a graph displays the levels of a categorical variable. For information about using the SAS/IML Studio GUI to set the order of a variable, see the section “Ordering Categories of a Nominal Variable” (Chapter 11, *SAS/IML Studio User’s Guide*).

10.3.1 Setting the Display Order of a Categorical Variable

Suppose you want to create a bar chart showing how many movies in the Movies data set were released for each month, across all years. You can determine the month that each movie was released by applying the MONNAMEw. format to the ReleaseDate variable.

The following program creates a new character variable (Month) in the data object that contains the month in which each movie was released. The program also creates a bar chart (shown in Figure 10.5) of the Month variable.

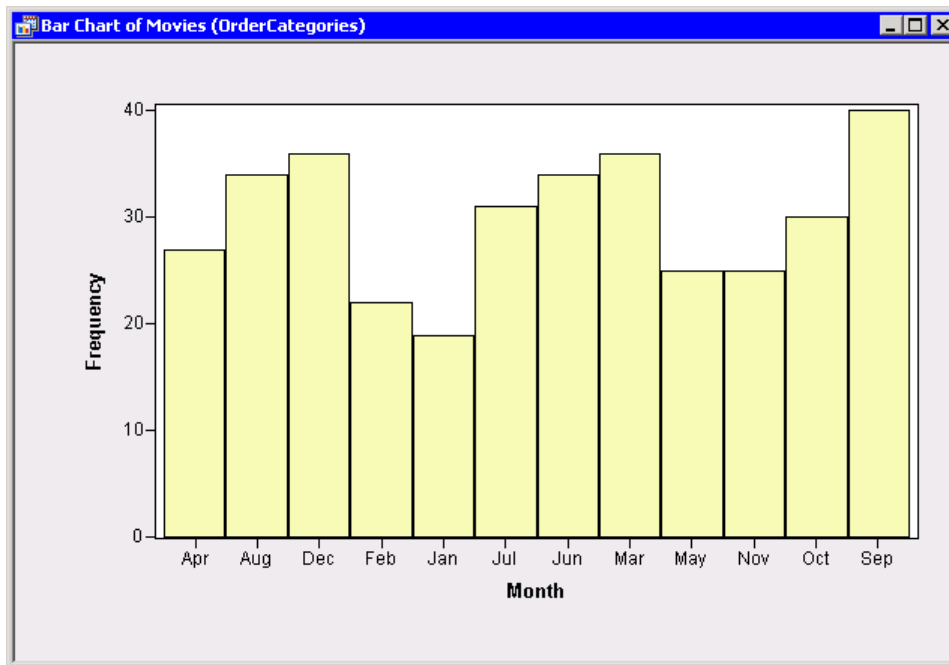
```

/* create a bar chart of variable; bars displayed alphabetically */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.GetVarData("ReleaseDate", r);
month = putn(r, "monname3.");          /* Jan, Feb, ..., Dec      */
dobj.AddVar("Month", month);          /* add new character variable */

declare BarChart bar;
bar = BarChart.Create(dobj, "Month");

```

Figure 10.5 Graph of Categories in Alphabetical Order



The categories in [Figure 10.5](#) are plotted in alphabetical order. This makes it difficult to compare consecutive months or to observe seasonal trends in the data. The months have a natural chronological ordering. You can set the ordering of a variable by using the `SetVarValueOrder` method in the `DataObject` class. This is shown in the following statements:

```

/* change order of bars; display chronologically */
order = {"Jan" "Feb" "Mar" "Apr" "May" "Jun"
         "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"};
/* alternatively: order = putn(mdy(1:12,1,1960), "monname3."); */
dobj.SetVarValueOrder("Month", order);

```

All graphs that display the variable will update to reflect the new ordering, as shown in [Figure 10.6](#).

Programming Tip: You can set the ordering of a nominal variable by using the `SetVarValueOrder` method in the `DataObject` class.

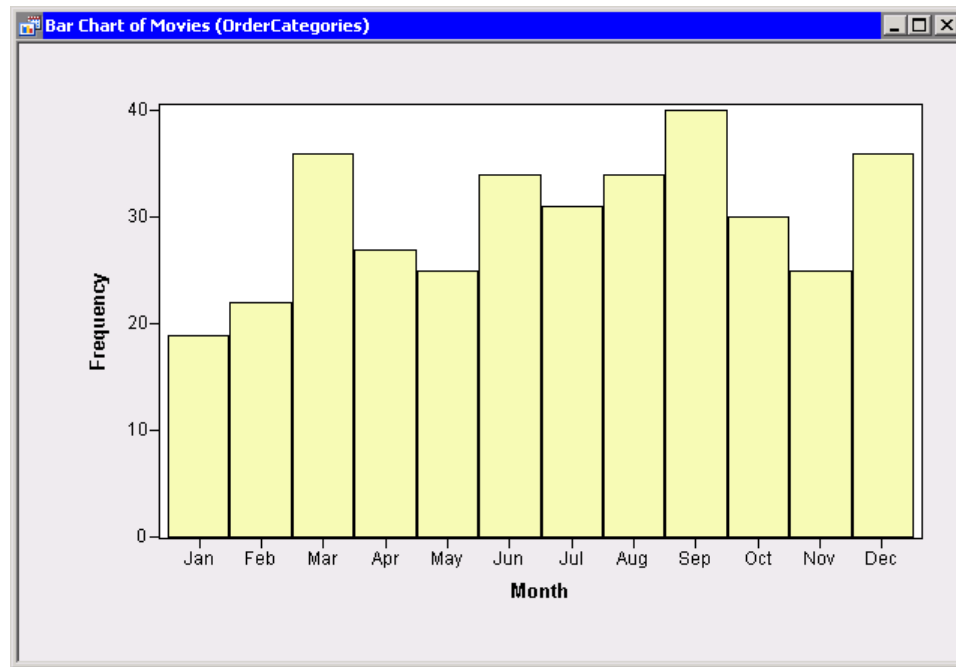
Figure 10.6 Graph of Categories in Chronological Order

Figure 10.6 shows the relative changes from month to month. Notice that the distributors release relatively few movies in January and February; perhaps the large number of December movies are still playing in the theaters? It is also curious that there are more releases in the months of March and September than in the summer months.

10.3.2 Using a Statistic to Set the Display Order of a Categorical Variable

Sometimes you might want to emphasize differences between groups by ordering categories according to some statistic calculated for data in each category. For example, you might want to order a bar chart according to the frequency counts, or you might want to order a box plot according to the mean (or median) value of the categories.

The following program demonstrates this idea by creating a box plot of the `US_Gross` variable versus the month that the movie was released. The `Month` variable is ordered according to the mean of the `US_Gross` variable for movies that are released during that month (for any year).

The previous section describes how you can create a new character variable in the data object (named `Month`) that contains the name of the month in which the movie was released. The following program continues the example. The program uses the technique of [Section 3.3.5](#) to compute the mean gross revenue for each month:

```
/* compute a statistic for each category */
GroupVar = "Month";          /* variable that contains categories */
dobj.GetVarData(GroupVar, group);
dobj.GetVarData("US_Gross", y);
```

```

u = unique(group);          /* find the categories          */
numGroups = ncol(u);        /* number of categories      */
stat = j(1, numGroups);    /* allocate a vector for results */
do i = 1 to numGroups;     /* for each group...        */
    idx = loc(group=u[i]);  /* find the observations in that group */
    m = y[idx];             /* extract the values        */
    stat[i] = m[:];         /* compute statistic for group */
end;
print stat[colname=u];

```

The categories are stored in the **u** vector in alphanumeric order. The mean of the *i*th group is stored in the **stat** vector, as shown in [Figure 10.7](#).

Figure 10.7 Statistics for Each Category in Alphabetical Order

	Apr	Aug	stat Dec	Feb	Jan	Jul
ROW1	39.439534	46.851001	66.429999	49.910561	36.849156	90.680443
	Jun	Mar	stat May	Nov	Oct	Sep
ROW1	72.549527	45.179194	107.31437	68.917781	30.683314	26.638737

The means are known for each group, so the following statements order the months according to their means:

```

/* sort categories according to the statistic for each category */
r = rank(stat);
print r[colname=u];
sorted_u = u;          /* copy data in u          */
sorted_u[r] = u;        /* permute categories into sorted order */
print sorted_u;

dobj.SetVarValueOrder(GroupVar, sorted_u);
declare BoxPlot box;
box = BoxPlot.Create(dobj, GroupVar, "US_Gross");

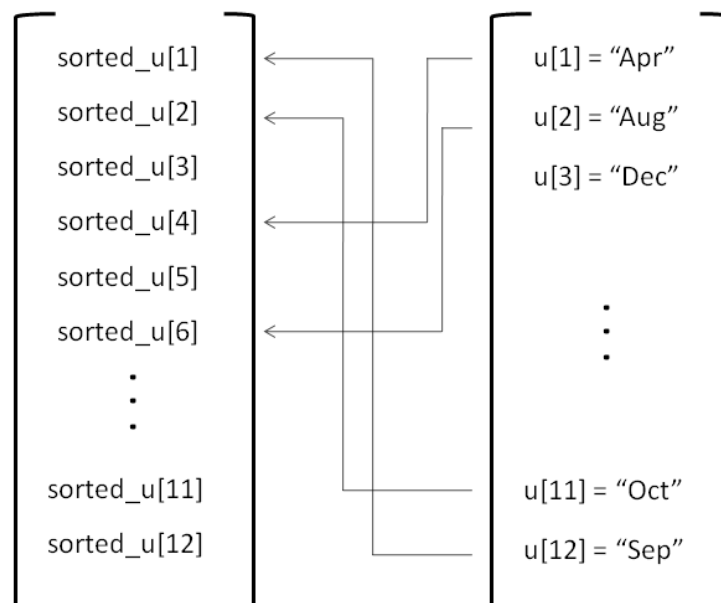
```

The program shows how the RANK function is used to sort the **u** vector according to the values of the **stat** vector. The output from the program is shown in [Figure 10.8](#). The smallest mean is stored in **stat[12]** which corresponds to movies released in September. Consequently, **r[12]** contains the value 1, which indicates that September needs to be the first entry when the vector is sorted. The second entry should be October, and so on.

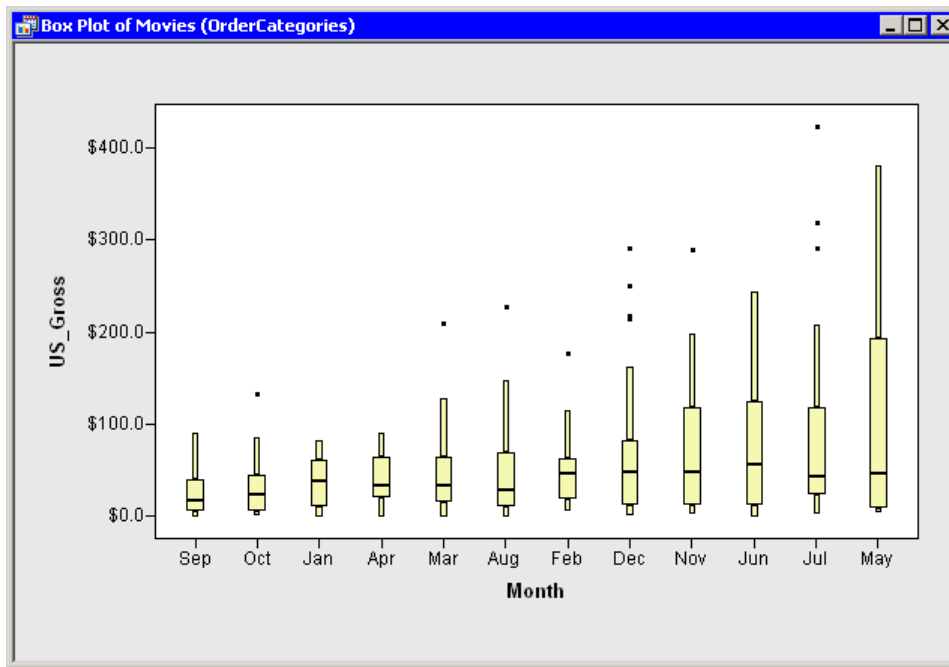
Figure 10.8 Ranks of Categories and the Sorted Categories

	Apr	Aug	r Dec	Feb	Jan	Jul
ROW1	4	6	8	7	3	11
	Jun	Mar	r May	Nov	Oct	Sep
ROW1	10	5	12	9	2	1
sorted_u						
Sep Oct Jan Apr Mar Aug Feb Dec Nov Jun Jul May						

The actual sorting is accomplished by using the r vector to permute the entries of u . First, the vector **sorted_u** is created as a copy of u . This ensures that **sorted_u** is the same size and type (character or numeric) as u . Then the permutation occurs. This is shown schematically in Figure 10.9. The first entry in r is 4, so **sorted_u**[4] is assigned the contents of u [1] which is “Apr”. The second entry in r is 6, so **sorted_u**[6] is assigned the contents of u [2] which is “Aug”. This process continues for each row. Finally, the last (twelfth) entry in r is 1, so **sorted_u**[1] is assigned the contents of u [12] which is “Sep”. In this way, the rows of **sorted_u** are sorted according to the values of **stat**.

Figure 10.9 Permuting Rows to Sort Categories

The last statements of the program reorder the categories of the Month variable by using the SetVar-ValueOrder method in the DataObject class. All graphs that include the Month variable display the months in the order given by the **sorted_u** vector, as shown in the box plot in Figure 10.10.

Figure 10.10 Box Plots Ordered by the Mean of Each Category

The program in this section is written so as to work in many situations. For example, if you assign “MPAARating” to **GroupVar** then the program creates a box plot of the MPAA ratings sorted according to the mean US gross revenues for each ratings category. (If **GroupVar** contains the name of a numeric variable, the program still runs correctly provided that you comment out the PRINT statements since the COLNAME= option expects a character variable.)

Programming Technique: The following statements sort the values of a vector **u** according to the values of a numeric vector **x**:

```

r = rank(x);          /* compute permutation that sorts x      */
sorted_u = u;         /* same dim and attributes (num/char) as u */
sorted_u[r] = u;      /* permute categories into sorted order  */

```

The statements correctly sort **u** regardless of whether **u** is numeric or character, or whether it is a row vector or a column vector.

10.4 Selecting Observations

Selecting observations in IMLPlus graphics is a major reason to use SAS/IML Studio in data analysis. You can discover relationships among variables in your data by selecting observations in one graph and seeing those same observations highlighted in other graphs.

You can select observations interactively by clicking on a bar in a bar chart or by using a selection rectangle to select several observations in a scatter plot. However, sometimes it is useful to select observations by running a program. Selecting observations with a program can be useful if the criterion that specifies the selected observations is complicated. It is also useful for selecting observations during a presentation or as part of an analysis that you run frequently, or for highlighting observations that have missing values in a certain variable.

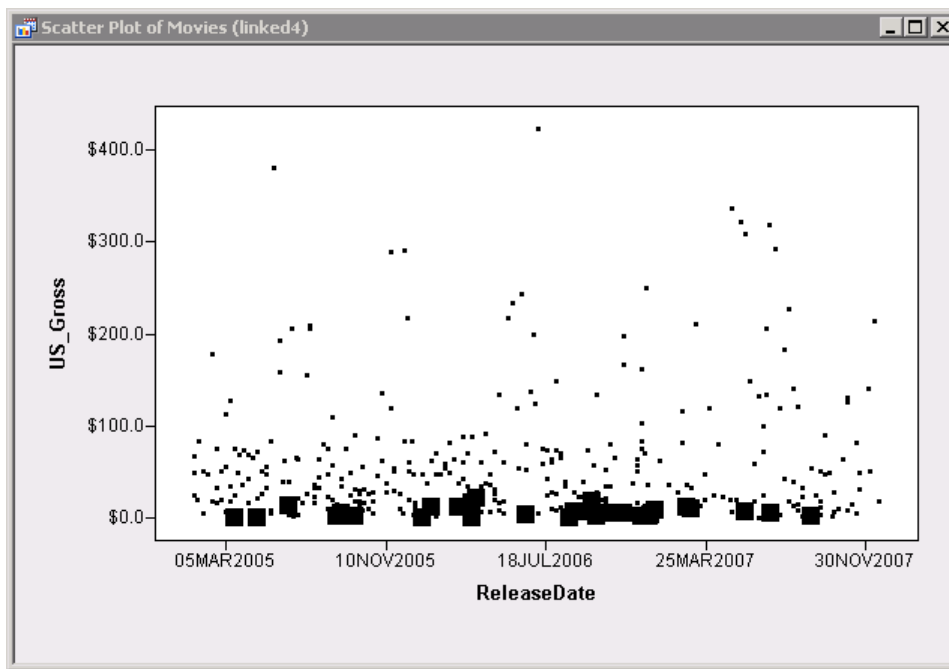
The following program selects observations in a data object created from the Movies data set for which the World_Gross variable has a missing value:

```
/* select observations that contain a missing value for a variable */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.GetVarData("World_Gross", wGross);
/* find observations with missing values */
idx = loc(wGross=.);
if ncol(idx)>0 then
    dobj.SelectObs(idx);
```

The SelectObs method selects observations. The observation numbers are specified by the first argument to the method. The SelectObs method has an optional second argument that specifies whether to deselect all observations prior to selecting new observations. By default, new selections do not deselect any existing selections, so you can call the SelectObs method several times to build up the union of observations that satisfy any of several criteria. For example, the following statements (which continue the preceding program) find all movies for which the world gross revenues are more than ten times the US revenue. These movies (which did far better internationally than in the US) are added to the set of selected observations:

```
/* select observations that satisfy a formula */
dobj.GetVarData("US_Gross", usGross);
jdx = loc(wGross>10*usGross);
if ncol(jdx)>0 then
    dobj.SelectObs(jdx);                /* add to previous selection */
```

The results of the previous statements are shown in [Figure 10.11](#). The size of the selected observations is increased relative to the unselected observations so that they are more visible. You can increase the size difference between selected and unselected observations by clicking ALT+UP ARROW in the plot. As of SAS/IML Studio 3.3, there is no method to increase this size difference.

Figure 10.11 Observations That Satisfy Either of Two Criteria

If you want to replace existing selected observations, you can call the `SelectObs` method with the second argument equal to **false**. If you want to clear existing selected observations, you can call the `DeselectAllObs` method in the `DataObject` class. Similarly, the `DeselectObs` method allows you to deselect specified observations. You can use the `DeselectObs` method to specify the intersection of criteria. You can also use the `XSECT` function to form the intersection of criteria.

The following table lists the frequently used `DataObject` methods that select or deselect observations:

Table 10.4 Methods in the `DataObject` Class That Select or Deselect Observations

Method	Description
<code>DeselectAllObs</code>	Deselects all observations
<code>DeselectObs</code>	Deselects specified observations
<code>SelectObs</code>	Selects specified observations
<code>SelectObsWhere</code>	Selects observations that satisfy a criterion

In addition to viewing selected observations in IMLPlus graphs, there is a second reason you might want to select observations in a data object: many `DataObject` methods operate on the selected observations unless a vector of observation numbers is explicitly specified. The following table lists `DataObject` methods that, by default, operate on selected observations:

Table 10.5 DataObject Class Methods That Can Operate on Selected Observations

Method	Description
GetSelectedObsNumbers	Gets the observation numbers for the selected observations
GetVarSelectedData	Gets the data for the selected observations and for a list of specified variables
IncludeInAnalysis	Specifies whether an observation should be included in statistical analyses
IncludeInPlots	Specifies whether an observation should be included in IMLPlus graphs
SetMarkerColor	Sets the color of the marker that represents an observation
SetMarkerShape	Sets the shape of the marker that represents an observation

10.5 Getting and Setting Attributes of Data

Previous sections describe methods such as `SetRoleVar`, `SetMarkerColor`, and `SelectObs`, which all affect attributes of data. This section gives further examples of using `DataObject` methods to change attributes for observations and properties for variables.

10.5.1 Properties of Variables

SAS data sets contain some properties of variables: the length of a variable (more often used for character variables than for numeric) and the variable's label, format, and informat. Even the name of the variable is a property and can be changed by using the `RENAME` statement in a `DATA` step. The `DataObject` class has methods that get or set all of these properties.

In addition, the `DataObject` class maintains additional properties. For example, a variable in a data object can be nominal. All character variables are nominal; numerical variables are assumed to represent continuous quantities unless explicitly set to be nominal. A variable can also be selected or unselected. Selected variables are used by analyses chosen from the SAS/IML Studio GUI to automatically fill in fields in some dialog boxes. The analyses built into SAS/IML Studio are accessible from the **Analysis** menu; they are described in the *SAS/IML Studio User's Guide*.

The following program calls several `DataObject` methods to set variable properties. It then displays a data table.

```
/* call DataObject methods related to variable properties */
obsNum = t(1:5);                               /* 5 x 1 vector      */
x = obsNum / 100;
declare DataObject dobj;
dobj = DataObject.Create("Properties", {"ObsNum", "x"}, obsNum || x);
```

```

/* call some "Set" methods for variable properties */
/* standard SAS variable properties */
dobj.SetVarFormat("x", "Percent6.1");          /* set format          */
dobj.SetVarLabel("x", "A few values");          /* set label           */

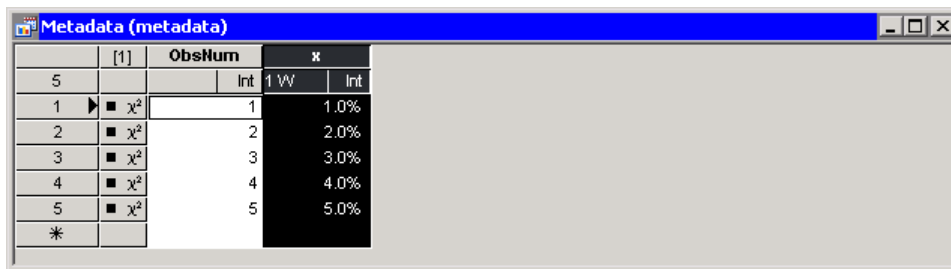
/* IMLPlus properties */
dobj.SetRoleVar(ROLE_WEIGHT, "x");              /* set role (=WEIGHT) */
dobj.SelectVar("x");                            /* select variable     */

/* displays data table in front of any other SAS/IML Studio windows */
DataTable.Create(dobj).ActivateWindow();

```

The data table created by the program is shown in Figure 10.12. You can see from the data table that the x variable is selected (notice the highlighting), is assigned the “weight” role (notice the ‘W’ in the column header), and is assigned the PERCENT6.1 format (notice the values).

Figure 10.12 The Result of Calling DataObject Methods



	[1]	ObsNum	x
5		Int	1 W Int
1	■ x ²	1	1.0%
2	■ x ²	2	2.0%
3	■ x ²	3	3.0%
4	■ x ²	4	4.0%
5	■ x ²	5	5.0%
*			

You can also retrieve variable properties. The following example continues the previous program statements:

```

/* call some "Get" methods for variable properties */
dobj.GetVarNames(varNames);                    /* get name of all vars */
dobj.GetSelectedVarNames(varName);              /* get name of sel. vars */
/* standard SAS variable properties */
format = dobj.GetVarFormat(varName);           /* get format           */
label = dobj.GetVarLabel(varName);             /* get label            */
informat = dobj.GetVarInformat(varName);       /* get informat         */
if informat="" then                          /* return empty string  */
    informat = "None";                        /* if no informat      */
/* IMLPlus properties */
weightName = dobj.GetRoleVar(ROLE_WEIGHT);    /* get name of WEIGHT var */
isNominal = dobj.IsNominal(varName);          /* is variable nominal? */

print varName format informat label;           /* SAS properties      */
print weightName isNominal;                    /* IMLPlus properties  */

```

The output from the program is shown in Figure 10.13. The program is not very interesting; its purpose is to demonstrate how to set and get frequently used properties of variables. Notice that the DataObject class contains methods that get and set standard SAS properties (variable name, format, and informat) in addition to IMLPlus properties (role, selected state). For information on how to set and examine variable properties by using the SAS/IML Studio GUI, see Chapter 4, “Interacting with the Data Table” (*SAS/IML Studio User’s Guide*).

Figure 10.13 Variable Properties

varName	format	informat	label
x	PERCENT6.1	None	A few values
weightName isNominal			
x			0

10.5.2 Attributes of Observations

Although SAS data sets do not contain attributes for each observation, the `DataObject` class does. The `DataObject` class has methods to set and retrieve attributes for observations. For example, you can set the marker color and marker shape for an observation, and specify whether the observation is included in plots and included in analyses. You can also select or deselect an observation.

The following program continues the previous program by calling `DataObject` methods to set attributes of observation for the data object:

```
/* call DataObject methods related to observation attributes */
/* call some "Set" methods for observation attributes */
dobj.DeselectAllVar();           /* remove selected var */
dobj.SetMarkerColor(1:2, RED);   /* set marker color */
dobj.SetMarkerShape(2:3, MARKER_STAR); /* set marker shape */
dobj.IncludeInAnalysis(3:4, false); /* set analysis indicator */
dobj.IncludeInPlots(4:5, false); /* set plot indicator */
dobj.SelectObs({1,3,5});         /* select observations */
```

The data table created by the program is shown in [Figure 10.14](#). Notice that the selected observations are highlighted, and that the row headers have icons that show the shape for each observation, in addition to whether the observation is included in plots or in analyses. The icons also show the color for each observation, although the colors are not apparent in the figure.

Figure 10.14 The Result of Calling `DataObject` Methods

ObsNum		x	
Int	vWeight	Int	
1	1	1.0%	
2	2	2.0%	
3	3	3.0%	
4	4	4.0%	
5	5	5.0%	
*			

In the same way, you can call methods that retrieve attributes for observations. This is shown in the following statements:

```

/* call some "Get" methods for observation attributes */
dobj.GetMarkerFillColor(colorIdx);      /* get all colors          */
dobj.GetMarkerShape(shapeIdx);          /* get all shapes          */
dobj.GetObsNumbersInAnalysis(analIdx);  /* get only obs numbers with */
dobj.GetObsNumbersInPlots(plotsIdx);    /* a certain indicator      */
dobj.GetSelectedObsNumbers(selIdx);     /* get selected obs numbers */

/* compute some combinations */
selColor = colorIdx[selIdx];             /* color of selected obs    */
shapePlots = shapeIdx[plotsIdx];        /* shapes in plots          */
selAnal = xsect(analIdx, selIdx);       /* selected obs in analyses */

/* printing convenience:
   create a format that associates marker names with marker values */
submit;
proc format;
  value shape
    0='Square' 1='Plus'      2='Circle'      3='Diamond'
    4='X'       5='Triangle' 6='InvTriangle' 7='Star';
run;
endsubmit;

print selColor[format=hex6.] shapePlots[format=shape.], selAnal;

```

The output from the program is shown in [Figure 10.15](#).

Figure 10.15 Observation Attributes

selColor shapePlots	
FF0000	Square
000000	Star
000000	Star
selAnal	
1	5

There are several statements in the program that require further explanation. Notice that there are two kinds of “Get” methods for observation attributes. Some (such as `GetMarkerShape`) create a vector that always has n rows, where n is the number of observations in the data set. The elements of the vector are colors, shapes, or some other attribute. In contrast, other methods (such as `GetSelectedObsNumbers`) create a vector that contains k rows where k is the number of observations that satisfy a criterion. The elements of the vector are observation numbers (the indices of observations) for which a given criterion is true. Consequently, you use `colorIdx[selIdx]` in order to compute the colors of selected observations, whereas you use the `XSECT` function to compute the indices that are both selected and included in analyses.

Notice how the `SUBMIT` block is used to call the `FORMAT` procedure in order to print names for the values returned by the `GetMarkerShape` method. The `FORMAT` procedure creates a new format which associates the values of IMLPlus keywords with explanatory text strings. For example, the

keyword **MARKER_STAR** has the numeric value 7 as you can determine by using the PRINT statement. The newly created format (named SHAPE.) is used to print the values of the **shapePlots** vector.

Part III

Applications

Chapter 11

Calling Functions in the R Language

Contents

11.1 Overview of Calling Functions in the R Language	251
11.2 Introduction to the R Language	252
11.3 Calling R Functions from IMLPlus	252
11.4 Data Frames and Matrices: Passing Data to R	254
11.4.1 Transferring SAS Data to R	254
11.4.2 What Happens to the Data Attributes?	256
11.4.3 Transferring Data from R to SAS Software	257
11.5 Importing Complicated R Objects	259
11.6 Handling Missing Values	261
11.6.1 R Functions and Missing Values	261
11.6.2 Merging R Results with Data That Contain Missing Values	262
11.7 Calling R Packages	264
11.7.1 Installing a Package	264
11.7.2 Calling Functions in a Package	264
11.8 Case Study: Optimizing a Smoothing Parameter	267
11.8.1 Computing a Loess Smoother in R	268
11.8.2 Computing an AICC Statistic in R	269
11.8.3 Encapsulating R Statements into a SAS/IML Module	270
11.8.4 Finding the Best Smoother by Minimizing the AICC Statistic	271
11.8.5 Conclusions	272
11.9 Creating Graphics in R	273
11.10 References	278

11.1 Overview of Calling Functions in the R Language

Just as you can call SAS statements from an IMLPlus program, you can also call functions in the R language. This chapter shows how to do the following:

- transfer data between R and SAS software
- import the results of a statistical analysis that was computed in R

- reconcile differences between the ways that R and SAS software handle missing values
 - incorporate R code into an IMLPlus module
 - create graphics in R from an IMLPlus program
-

11.2 Introduction to the R Language

R is an open-source language and environment for statistical computing. The R language has many similarities with SAS/IML: it supports matrix-vector computations, it has a rich library of statistical functions, and it is intended primarily for researchers and statistical programmers. R supports the creation and distribution of *packages*: user-written functions, data, and documentation that can be added to the R library.

These packages are sometimes written by leading researchers in the academic statistical community. In some fields of statistics, it is common for authors of journal articles and books to create an R package that accompanies the publication. The R package typically includes functions that implement the main ideas of the publication and also provides data used in the publication. In this way, researchers disseminate not only their statistical ideas but also a particular implementation of their ideas.

There are over 2,400 R packages. Some packages are general purpose libraries that implement functions in a particular statistical area. For example, the **boot** package (Canty and Ripley 2009) contains “functions and datasets for bootstrapping from the book *Bootstrap Methods and Their Applications* by A. C. Davison and D. V. Hinkley” (Davison and Hinkley 1997). Other packages contain a few functions (or sometimes just one!) that implement particular computations from a journal article or from someone’s class project, masters thesis, or PhD dissertation.

SAS/IML Studio 3.2 (released in July 2009) was the first SAS product to offer an interface to the R language. Beginning in SAS/IML 9.22, you can also call R from PROC IML. This chapter describes how to exchange data between R and various SAS data formats, and how to call functions in the R language. This chapter does not teach you how to program in R, but rather how to incorporate knowledge of R into your SAS programs.

The chapter assumes that a 32-bit Windows version of R is installed on the same PC that runs SAS/IML Studio. You can download R from CRAN: The Comprehensive R Archive Network at <http://cran.r-project.org>. The SAS/IML Studio online Help lists versions of R that are supported.

11.3 Calling R Functions from IMLPlus

You can call R functions from an IMLPlus program by using the SUBMIT and ENDSUBMIT statements. These are the same statements that are used to run SAS statements and procedures. The

difference is that to call R functions, you use the R option in the SUBMIT statement as shown in the following example:

```
/* call R functions */
submit / R;
  n <- 100                      # 100 obs
  set.seed(1)                  # seed for pseudorandom number generation
  x <- runif(n)                 # x is pseudorandom uniform of length n
  y <- 3*x + 2 + rnorm(n)      # y is linearly related to x
  model <- lm(y~x)             # model y as a function of x
  summary(model)               # print summary of linear model
endsubmit;
```

The statements in the SUBMIT block are sent to R for interpreting. The left-pointing arrow, `<-`, is the R assignment operator. The first four R statements create a vector **y** that is linearly related to **x**, but that contains errors randomly generated from the standard normal distribution. The `set.seed` function sets a seed value for the random number generator used by the `rnorm` and `runif` functions. (See the section “Using Functions to Create Matrices” on page 24 for analogous SAS/IML statements.) The `lm` function uses the method of least squares to model the response **y** as a function of the explanatory variable **x**. The `summary` function displays relevant statistics for the model, as shown in Figure 11.1. In SAS/IML Studio, R output appears in the current output window; in PROC IML, R output is sent to all open ODS destinations.

Figure 11.1 Output from R Statements

```
Call:
lm(formula = y ~ x)

Residuals:
    Min       1Q   Median       3Q      Max
-1.84978 -0.56222 -0.08707  0.52427  2.51661

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.8207     0.2058   8.846 3.85e-14 ***
x              3.3123     0.3535   9.371 2.80e-15 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9411 on 98 degrees of freedom
Multiple R-squared:  0.4726, Adjusted R-squared:  0.4672
F-statistic: 87.82 on 1 and 98 DF, p-value: 2.804e-15
```

Programming Tip: Use the SUBMIT and ENDSUBMIT statements with the R option to call R functions from an IMLPlus program.

11.4 Data Frames and Matrices: Passing Data to R

In most cases, an R function requires data in the form of an R matrix or an R *data frame*. A data frame is an R object that stores variables and observations. Like a SAS data set or an IMLPlus data object, a data frame can contain mixed types of variables (numeric or character), whereas a matrix can contain only a single type of data. This section describes how to transfer SAS data sets, data objects, or matrices to an R matrix or data frame.

11.4.1 Transferring SAS Data to R

You can transfer data from three sources: a SAS data set in a SAS library (such as `Work`, `Sasuser`, and so on), a data object, and a SAS/IML matrix.

You can transfer a SAS data set by calling the `ExportDataSetToR` module. The first argument to the module is the two-level name (*libref.filename*) of the SAS data set. The second argument is the name of a data frame in R to contain the data. For example, the following statements create a data frame called `MovieFrame` that contains a copy of the data in the `Sasuser.Movies` data set:

```
/* transfer data from SAS data set to R data frame */
run ExportDataSetToR("Sasuser.Movies", "MovieFrame");
submit / R;
    names(MovieFrame)
    summary(MovieFrame$Budget)
endsubmit;
```

The `names` function displays the names of variables in a data frame. As the example indicates, you can use a dollar sign (\$) to refer to a particular variable within a data frame. The `summary` function prints some descriptive statistics about the variable, as shown in [Figure 11.2](#).

Figure 11.2 Statistics from R (Data from a SAS Data Set)

[1]	"Title"	"MPAARating"	"ReleaseDate"	"Budget"
[5]	"US_Gross"	"World_Gross"	"SexScore"	"ViolenceScore"
[9]	"ProfanityScore"	"NumAANomin"	"NumAAWon"	"AANomin"
[13]	"AAWon"	"Notes"	"Distributor"	"Year"
[17]	"Sex"	"Violence"	"Profanity"	
	Min.	1st Qu.	Median	Mean 3rd Qu. Max.
	0.15	16.50	30.00	44.83 60.00 258.00

Programming Tip: The R language is case-sensitive. If you transfer a SAS data set (or data object) to an R data frame, you must refer to variables in the correct case. You can use the `names` function to display the names of variables in a data frame.

Programming Tip: Similarly, you must refer to the names of R matrices, data frames, and other objects in the correct case. The `ls` function in R displays the names of objects.

If your data are in a data object, you can transfer all variables in a `DataObject` by calling the `ExportToR` method in the `DataObject` class. The method takes a single argument: the name of the data frame in R to contain the data. This is shown in the following statements:

```
/* transfer data from data object to R data frame */
declare DataObject dobj;
Dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.ExportToR("MovieFrame");          /* write all variables */
submit / R;
    summary(MovieFrame$ReleaseDate)
endsubmit;
```

Figure 11.3 Statistics from R (Data from a Data Object)

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
"2005-01-14"	"2005-09-09"	"2006-04-28"	"2006-05-21"	"2006-12-21"	"2007-12-21"

Lastly, you can transfer data from a SAS/IML matrix by calling the `ExportMatrixToR` module. The following statements transfer the data from a SAS/IML vector into an R matrix:

```
/* transfer data from SAS/IML matrix to R matrix */
dobj.GetVarData("World_Gross", w);      /* get data into vector w */
run ExportMatrixToR(w, "WorldG");       /* send matrix to R      */
submit / R;
    summary(WorldG)
endsubmit;
```

The statements create a SAS/IML vector `w` that contains the data in the `World_Gross` variable. Those data are then transferred to the R matrix named `WorldG`. The output is shown in [Figure 11.4](#). Notice that the output is column-oriented (compare with [Figure 11.3](#)) because the `summary` function behaves differently for a matrix than for a variable in a data frame.

Figure 11.4 Statistics from R (Data from a SAS/IML Matrix)

A1	
Min.	: 0.3418
1st Qu.	: 26.3060
Median	: 70.5941
Mean	: 125.5068
3rd Qu.	: 143.3581
Max.	: 1065.6598
NA's	: 25.0000

Notice also in [Figure 11.4](#) that there are 25 `NA`s in the `World` vector. The `NA` represents a missing value in R. A SAS missing value is automatically converted to `NA` when data are transferred to R.

The following table summarizes the frequently used methods and modules that transfer SAS data to R. For details of the data transfer, see the online Help chapter titled “Accessing R.”

Table 11.1 Methods and Modules for Transferring Data from SAS Software to R

Method/Module	Data Source	Data Destination
DataObject.ExportToR	DataObject	R data frame
ExportDataSetToR	SAS data set in libref	R data frame
ExportMatrixToR	SAS/IML matrix	R matrix

11.4.2 What Happens to the Data Attributes?

Although SAS data (numbers and strings) can be sent to R, the properties that are associated with variables and the attributes associated with observations are not, in general, transferred to R. R does not support the same properties of variables that SAS software uses. For example, SAS data sets associate formats, lengths, and labels with variables, but R does not. Similarly, an IMLPlus data object maintains attributes for observations, including marker color, shape, and whether the observation is selected. In general, these attributes are not transferred to R when the data are transferred.

However, there are two variable properties that receive special handling: variables with date or time formats, and IMLPlus nominal variables.

When the ExportDataSetToR module or the ExportToR method transfers a variable with a DATE_w., TIME_w., or DATETIME_w. format, the corresponding R variable is automatically assigned a special R class that indicates that the variable contains time or date information. A SAS variable with the DATE_w. format is transferred to an R variable with the “Date” class. A SAS variable with a TIME_w. or DATETIME_w. format is transferred to an R variable with the “POSIXct” class.

Programming Tip: Use the DataObject.ExportToR method to create an R data frame from an IMLPlus data object. Nominal variables in the data object are automatically converted to R factors. A variable with a DATE_w., TIME_w., or DATETIME_w. format is automatically assigned to an appropriate R class.

Similarly, when the ExportToR method of the DataObject class transfers a nominal variable to R, the corresponding R variable is automatically assigned the “factor” class. A *factor* is a storage class in R that is used for categorical variables. For statistical modeling, using a factor in R to represent a variable is similar to listing the variable on a CLASS statement in a SAS procedure. Factors are treated as classification variables in analyses.

For example, in the Movies data set, the variable ReleaseDate has a DATE9. format. When the Movies data set is used to create an R data frame, the corresponding ReleaseDate variable in R is automatically assigned the “Date” class. Similarly, the NumAANomin variable is a numeric nominal variable in the IMLPlus data object; the corresponding R variable is assigned the “factor” class in R. This is shown in the following statements and in [Figure 11.5](#):

```

/* show that data transfer preserves some variable properties */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
dobj.ExportToR("MovieFrame");          /* write all variables */
submit / R;
    class (MovieFrame$ReleaseDate)
    class (MovieFrame$NumAANomin)
endsubmit;

```

Figure 11.5 Classes of R Objects Reflect Variable Properties

```

[1] "Date"
[1] "factor"

```

Note that not all formats are preserved when data are transferred to R. For example, the Budget variable has a DOLLAR8.1 variable in the SAS data set, but [Figure 11.2](#) shows that the corresponding R variable is just an ordinary numeric variable with no special characteristics.

The SAS/IML Studio online Help provides details about the differences between the ways that SAS and R software handle date and time values.

11.4.3 Transferring Data from R to SAS Software

You can also transfer data from R objects to SAS software. The source (an R data frame or matrix) and destination (a data object, SAS dataset, or SAS/IML matrix) determine which module or method you should use.

If the data are contained in an R data frame (or any expression that can be converted to a data frame), you can create a data object by using the `CreateFromR` method of the `DataObject` class, as shown in the following statements.

```

/* transfer data from R data frame to data object */
declare DataObject dobjR;
dobjR = DataObject.CreateFromR("Old Faithful Data", "faithful");
dobjR.GetVarNames(VarNames);
dobjR.GetDimensions(NumVar, NumObs);
print VarNames NumVar NumObs;

```

The `CreateFromR` method has two arguments: the first names the data object and the second gives the name of an R data frame (or an expression that can be converted to a data frame). The previous example creates a data object from the `faithful` data frame that contains data about eruptions of the Old Faithful geyser in Yellowstone National Park. After the data object is created, the `GetVarNames` and `GetDimensions` methods are called to get basic information about the data. This information is displayed in [Figure 11.6](#).

Figure 11.6 Attributes of a Data Object Created from a Data Frame

VarNames	NumVar	NumObs
eruptions	2	272
waiting		

You can also transfer R data directly to a SAS data set by using the `ImportDataSetFromR` module. You can transfer data that are contained in an R data frame, an R matrix, or any expression that can be converted to either of these objects. For example, the following statement transfers the Old Faithful data directly to a SAS data set in the Work library:

```
run ImportDataSetFromR("Work.OldFaithful", "faithful");
```

You can also create a SAS/IML matrix from either an R data frame or an R matrix by using the `ImportMatrixFromR` module, as shown in the following statements:

```
/* transfer data from R matrix to SAS/IML matrix */
run ImportMatrixFromR(w, "faithful$waiting");
q = quartile(w);
print q[rowname={"Min" "Q1" "Median" "Q3" "Max"}];
```

The output from this example is shown in [Figure 11.7](#) and shows summary statistics for the waiting variable in the `faithful` data frame. In particular, the median waiting time between eruptions for Old Faithful was 76 minutes for the eruptions recorded in these data.

Figure 11.7 Summary of Data in a SAS/IML Matrix Created from R Data

q	
Min	43
Q1	58
Median	76
Q3	82
Max	96

Since SAS/IML matrices are either character or numeric, you need to be careful if you transfer data from an R data frame into a SAS/IML matrix. The `ImportMatrixFromR` module requires that the R data be either all character or all numeric. If `df` is a data frame, you can always extract the numeric variables with the command `df[sapply(df, is.numeric)]` and the character variables with `df[sapply(df, is.factor)]`.

Finally, it is possible to add a new variable to an existing `DataObject` by using the `AddVarFromR` method in the `DataObject` class. For example, the R expression `faithful$waiting>70` creates a logical vector that indicates which eruptions of Old Faithful occurred more than 70 minutes after the previous eruption. The following statement uses this R expression to create an indicator variable in the `dobjR` data set that was created earlier in this section:

```
dobjR.AddVarFromR("wait70", "faithful$waiting>70");
```


In order for the `AddVarFromR` method to succeed, the length of the R vector must be the same as the number of observations in the data object.

The following tables summarizes the frequently used methods that transfer data between SAS and R. For details of the data transfer and a full list of methods to transfer data, see the chapter titled “Accessing R” in the SAS/IML Studio online Help.

Table 11.2 Methods and Modules for Transferring Data from R to SAS Software

Method/Module	Data Source	Data Destination
<code>DataObject.CreateFromR</code>	R expression	DataObject
<code>DataObject.AddVarFromR</code>	R expression	DataObject variable
<code>ImportDataSetFromR</code>	R expression	SAS data set in libref
<code>ImportMatrixFromR</code>	R expression	SAS/IML matrix

11.5 Importing Complicated R Objects

The result returned by an R function is often an R object. Not all objects can be converted to a data frame. If `m` is an object in R, you can make sure it is convertible by typing `as.data.frame(m)`. If this call fails, use `str(m)` to examine the structure of the object. Often the object is composed of simpler objects (or *components*) such as a vector, matrix, or list. These components can be readily imported into SAS/IML matrices by using the methods and modules that are described in the previous section.

For example, consider the object returned by the `lm` function in R. The object is complicated, as partially shown in Figure 11.8. The figure is created by the following statements:

```
/* create R object */
submit / R;
  lm.obj <- lm(waiting~eruptions, data=faithful)
  str(lm.obj)
endsubmit;
```

Figure 11.8 A Partial Listing of an R Data Structure

```

List of 12
 $ coefficients : Named num [1:2] 33.5 10.7
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "eruptions"
 $ residuals    : Named num [1:272] 6.9 1.21 4.76 4.03 2.89 ...
  ..- attr(*, "names")= chr [1:272] "1" "2" "3" "4" ...
 $ effects      : Named num [1:272] -1169.26 201.6 4.36 3.59 2.54 ...
  ..- attr(*, "names")= chr [1:272] "(Intercept)" "eruptions" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:272] 72.1 52.8 69.2 58 82.1 ...
  ..- attr(*, "names")= chr [1:272] "1" "2" "3" "4" ...

... more lines ...

- attr(*, "class")= chr "lm"

```

The first line in the output from the `str` function is “List of 12.” This tells you that the `lm.obj` object contains 12 components. The description of each component begins with a dollar sign (\$) followed by the name of the component. For example, the first object is `lm.obj$coefficients`, so you could get these regression coefficients into a SAS/IML matrix by using the following statements:

```

/* access coefficients of R object. First technique. */
submit / R;
    coef <- lm.obj$coefficients      # direct access of coefficients
endsubmit;
run ImportMatrixFromR(c, "coef");

```

Or, more concisely, you can omit the entire SUBMIT block, as shown in the following statement:

```

/* access coefficients of R object. Second technique. */
run ImportMatrixFromR(c, "lm.obj$coefficients");

```

Because the second argument to the `ImportMatrixFromR` module can be any R expression, both of the previous examples produce the same result.

Programming Tip: When an R object cannot be directly converted to a data frame, you can use the `str` function to determine the components of the object. Then you can use R expressions and functions to extract the components of the object.

This technique works, but the R documentation discourages the direct access of components for some analysis objects such as the object of the “lm” class in the example. Instead, each class is supposed to overload certain “helper functions” (R calls them *generic functions*), and you are supposed to call the helper functions to extract relevant components. For example, the “lm” object overloads the `coef`, `residuals`, `fitted`, and `predict` functions, among others. The following statement shows the preferred technique for obtaining the coefficients of the linear model:

```

/* access coefficients of R object. Third technique. */
run ImportMatrixFromR(c, "coef(lm.obj)");

```

11.6 Handling Missing Values

When you transfer a SAS data set to R, missing values are automatically converted to the missing value in R, which is represented by **NA**. Similarly, an **NA** value is converted to a SAS missing value when you transfer data from R into SAS software. However, there are some differences between the way that R statistical functions and SAS procedures handle missing values. This section describes these differences.

11.6.1 R Functions and Missing Values

The treatment of missing values in R varies from function to function. For example, the following statements compare the computation of a mean in SAS/IML to the same computation in R:

```
/* compute a mean in SAS/IML and in R */
x = {., 2, 1, 4, 3};
meanIML = x[:];
print meanIML;
run ExportMatrixToR(x, "x");
submit / R;
    mean(x)          # compute mean in R; this also prints the result
endsubmit;
```

Figure 11.9 Default Computation of Mean for Data with Missing Values

```
meanIML
      2.5

[1] NA
```

Note that the result of the R **mean** function is **NA** because one of the elements of the **x** vector was missing. You can override this default behavior, as shown in the following statements:

```
/* compute mean in R; omit missing values */
submit / R;
    mean(x, na.rm=TRUE)
endsubmit;
```

Figure 11.10 Mean Computed by R after Removing Missing Values

```
[1] 2.5
```

The **na.rm=** option specifies whether to remove missing values before the computation occurs. In addition to the **mean** function, the same option occurs for **var**, **sum**, **min**, **max**, and many other functions.

Programming Tip: Many R functions that compute basic descriptive statistics return **NA** if any data value is missing. See the R documentation to determine how to override the default handling of missing values.

11.6.2 Merging R Results with Data That Contain Missing Values

By convention, an output data set created by a SAS regression procedure contains the same number of observations as the input data. In fact, the output data set created by most **OUTPUT** statements also contains a copy of the modeling variables from the input data set. If some observation contains a missing value for any of the explanatory variables in the model, then the predicted value for that observation is also missing.

This is not the default behavior for R regression functions. Instead, the default behavior is to drop observations that contain missing values. Consequently, if your explanatory variables contain missing values, the number of observations for vectors of fitted values and residuals is smaller than the number of observations in the data. This can be a problem if you want to merge the results from R into a SAS data set.

The following statements demonstrate this behavior.

```
/* compute predicted values in R when the data contain missing values */
x = {., 2, 1, 4, 3};
y = {1, 2, 2, 4, 3};
run ExportMatrixToR(x, "x");
run ExportMatrixToR(y, "y");

submit / R;
  lm.obj <- lm(y~x)           # default behavior: omit obs with NA
  pred <- fitted(lm.obj)     # predicted values from linear model
  print(pred)
endsubmit;
```

The output, shown in [Figure 11.11](#), shows that the result of the **fitted** function is a vector of four numbers, whereas the input data contains five observations. The first row of numbers in [Figure 11.11](#) represents the observations with nonmissing values. The predicted values for these observations are given by the second row of numbers.

Figure 11.11 Predicted Values from R Computation

2	3	4	5
2.4	1.7	3.8	3.1

If you want to merge these results with data in SAS, there are two techniques that you can use. You can use the observation numbers to help you merge the results, or you can rerun the **lm** function with an option that overrides the default behavior.

For the first technique, you need to get the nonmissing observation numbers. You can obtain a char-

acter vector that contains the first row of [Figure 11.11](#) by using the `names` function in R. You then need to convert the character names to observation numbers. The following IMLPlus statements retrieve the fitted values and the observations to which they correspond:

```
/* adjust for missing values */
run ImportMatrixFromR(p0, "pred");
run ImportMatrixFromR(nonMissing, "as.numeric(names(pred))");
p = j(nrow(x), 1, .);          /* allocate vector with missing values */
p[nonMissing] = p0;            /* plug in nonmissing values */
print p;
```

Figure 11.12 Merging Results with Missing Values

p	
	.
	2.4
	1.7
	3.8
	3.1

The nonmissing fitted values are transferred into the vector `p0` while the nonmissing observation numbers are transferred into the vector `nonMissing`. The observation numbers are used to create a vector `p` that contains the predicted values in the correct locations.

The alternative to this technique is to specify an option in R so that predicted (and residual) values are always the same length as the input data and contain missing values where applicable. You can specify the option by using the `options` function as indicated in the following statements:

```
/* tell R to pad results with missing values (global option) */
submit / R;
options(na.action="na.exclude")
# continue with other R statements...
endsubmit;
```

The `na.action=` option specifies the name of a function that determines how the statistical models handle missing values. The function “`na.exclude`” specifies that models should pad residual and predicted values with missing values where appropriate. You can also explicitly override the default value of this option each time you call a statistical model, as shown in the following statements:

```
/* tell R to pad results with missing values (each function) */
submit / R;
  lm.obj <- lm(y~x, na.action="na.exclude")    # pad obs with NA
  pred <- fitted(lm.obj)                      # predicted values
endsubmit;

run ImportMatrixFromR(p, "pred");
print p;
```

The results are identical to [Figure 11.12](#). In particular, there are five fitted values in the vector `p`.

Programming Tip: Many R functions support the `na.action=` option. If you intend to merge the results of these functions with SAS data sets, it is often convenient to specify the “na.exclude” value for the option.

11.7 Calling R Packages

Because the SAS/IML language (like the R language) is a matrix-vector language with a rich library of statistical functions, the SAS/IML programmer will not typically need to use R for basic computations. Both languages enable the programmer to iterate, index, locate certain elements in a matrix, and so on. However, as mentioned previously, R supports packages that add functionality in certain specialized areas of statistics. The SAS programmer might want to call these packages from a SAS/IML program in order to take advantage of those computational methods.

This section describes how to call functions in a package from SAS/IML programs. The package used in this section is the **KernSmooth** package by M. Wand, which implements kernel smoothing functions that are associated with the book *Kernel Smoothing* (Wand and Jones 1995).

In the 1970s, “kernel regression” became a popular method for smoothing a scatter plot. In kernel regression, the prediction at a point x is determined by a weighted mean of y values near x , with the weight given by a kernel function such as the normal distribution. The bandwidth of the kernel determines how much weight is given to points near x . This is similar to loess regression, except that kernel regression uses all observations within a certain distance from x , whereas loess uses a fixed number of neighbors regardless of their distance.

SAS/STAT software does not have a procedure that computes kernel regression, so this is a good candidate for calling an R package to compute a kernel estimator. Furthermore, the **KernSmooth** package is distributed with the core R distribution as a “recommended” package, so you do not need to download this package from CRAN.

11.7.1 Installing a Package

For packages that are not contained in the core R distribution, the R Project Web site (<http://cran.r-project.org/web/packages/>) gives instructions for installing packages. The R software contains commands that automatically download and install packages. You can also install a package manually if the automatic installation does not succeed.

11.7.2 Calling Functions in a Package

Calling a function in a package is easy. First, load the package into R by using the **library** function in R. Then call any function in the package. The example in this section uses the **KernSmooth** package, which is distributed with the core R software. Be aware that the **KernSmooth** functions

do not support missing values, so you might want to use a different package if your data contain missing values.

The goal of this section is to create a figure similar to [Figure 9.7](#). However, instead of calling a SAS procedure to compute the scatter plot smoother, this section describes how to call R functions to compute the smoother.

As explained in [Section 11.4.2](#), the `ReleaseDate` variable in the `Movies` data set is transferred to a variable in R that is assigned the “Date” class. However, the functions in **KernSmooth** do not handle date values (and neither do many other modeling functions in R). There are two ways to handle this problem: you can remove the format from the SAS variable before transferring the data to R, or you can remove the “Date” class from the R variable after the transfer. This example removes the format from the `ReleaseDate`, as shown in the following statements; see the R documentation for the `unclass` function to learn about the second method.

```
/* remove format on ReleaseDate variable since some R
   functions cannot handle dates for explanatory variables */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

oldFormat = dobj.GetVarFormat("ReleaseDate");      /* save format */
dobj.SetVarFormat("ReleaseDate", "");              /* remove format */
dobj.ExportToR("df");                               /* export data to R as a data frame */
```

Notice that the variable’s format (DATE9.) is saved in `oldFormat` so that it can be restored later in the program. A variable’s format is cleared by specifying the empty string in the `SetVarFormat` method.

A copy of the data is now in the R data frame named `df`. The following statements load the **KernSmooth** package and compute the smoother:

```
/* compute kernel regression by calling an R package */
YVar = "Budget";
XVar = "ReleaseDate";

/* note: KernSmooth does not handle missing values */
submit XVar YVar / R;
  library(KernSmooth)                # load package
  attach(df)                         # make names of vars available
  h <- dpill(&XVar, &YVar)            # compute plug-in bandwidth
  model <- locpoly(&XVar, &YVar, bandwidth=h) # fit smoother
  detach(df);
endsubmit;
```

To make the program more general, the matrices `YVar` and `XVar` are defined to contain the name of the response and explanatory variables. These matrices are listed in the `SUBMIT` statement and their values are substituted into the R statements as discussed in [Section 4.4](#). As explained in that section, the values of the matrices are referred to by preceding the matrix name with an ampersand (&). The substitution is made by `IMLPlus` before the `SUBMIT` block is sent to R.

Programming Technique: To pass the contents of a SAS/IML matrix, **m**, into R statements, list the matrix in the SUBMIT statement and reference the matrix inside the SUBMIT block by using an ampersand (&**m**).

The SUBMIT block contains R statements. The first statement loads the **KernSmooth** package. The next statement uses the **attach** statement to add **df** to a search path so that variables in the data frame can be referred to by their name. (Alternatively, you can refer to a variable as **df\$&xVar**, and so forth.) The **dpill** and **locpoly** functions are provided by the **KernSmooth** package. The first function uses a “plug-in” method to return a bandwidth for the kernel regression method. The bandwidth is stored in the variable **h**. The value is used in the **locpoly** function, which actually computes the kernel regression estimate. The last statement in the SUBMIT block is a call to the **detach** function, which removes the **df** data frame from the search path.

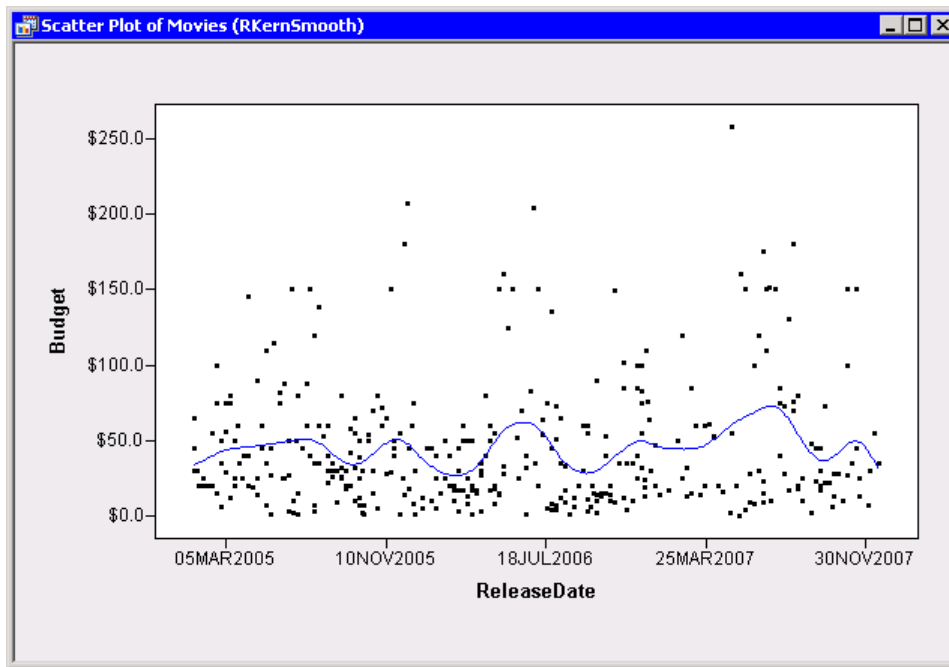
The **model** object is a simple structure that contains two vectors: **model\$x** and **model\$y**. The next statements read the R structure directly into a SAS/IML matrix and plot the model on a scatter plot of the data:

```
/* read simple R structure and overlay results */
run ImportMatrixFromR(m, "model");          /* x=m[,1]; pred=m[,2]; */
dobj.SetVarFormat("ReleaseDate", oldFormat);/* replace format      */

declare ScatterPlot p1;
p1 = ScatterPlot.Create(dobj, XVar, YVar);
p1.DrawUseDataCoordinates();
p1.DrawSetPenColor(BLUE);
p1.DrawLine(m[,1], m[,2]);
```

The data in **model** is transferred into a SAS/IML matrix named **m**. (Alternatively, the program could have transferred **model\$x** and **model\$y** with two separate calls.) The program then restores the format for the **ReleaseDate** variable, creates a scatter plot of the data, and overlays the kernel smoother. The scatter plot is shown in [Figure 11.13](#). A comparison with [Figure 9.7](#) shows that the two smoothers are qualitatively similar, although [Figure 11.13](#) exhibits a bump near November 2007 that is not present in the loess model.

Figure 11.13 Kernel Smoother Computed in R



11.8 Case Study: Optimizing a Smoothing Parameter

Section 9.7 describes how to call the LOESS procedure and add a smoother onto a scatter plot. The LOESS procedure used the `SELECT=` option to automatically select a smoothing parameter in the interval $[0, 1]$ that minimizes the AICC statistic.

There is also a `loess` function in R, but that function does not contain an option to automatically choose a bandwidth that optimizes some criterion. However, Section 3.6 describes a SAS/IML module for optimizing a function of one variable over an interval. Is it possible to use the golden section search module that is developed in that section to choose a smoothing parameter for the R `loess` function?

The answer is yes. Recall that the previously defined GoldenSection module requires that you define the module to be minimized. For the current example, the function must evaluate the loess model for a given parameter and then compute the AICC statistic for that model.

This section demonstrates how SAS and R software can work together. The section consists of four subsections.

1. Section 11.8.1 describes how to call the `loess` function in R.
2. Section 11.8.1 describes how to compute the AICC statistic from the object returned by the `loess` function.
3. Section 11.8.3 describes how to wrap (or encapsulate) the R statements into a SAS/IML module.

4. [Section 11.8.4](#) describes how to call the GoldenSection module developed in an earlier chapter to find the smoothing parameter that minimizes the AICC statistic for a loess fit.

11.8.1 Computing a Loess Smoother in R

The first step in the example is to figure out how to call the `loess` function in R. Assume that the Movies data set has already been copied to a data frame named `df` in R as in [Section 11.7.2](#). Then the following statements fit a loess model that is similar to the model from the LOESS procedure that is shown in [Figure 9.20](#):

```
/* fit loess model in R */
XVar = "ReleaseDate";
YVar = "Budget";

submit XVar YVar / R;
  loess.obj <- loess(&YVar ~ &XVar, data=df, span=0.15,
                    na.action="na.exclude", degree=1, normalize=FALSE,
                    control=loess.control(iterations=1));
  summary(loess.obj)
endsubmit;
```

Figure 11.14 Summary of a Loess Model in R

```
Call:
loess(formula = Budget ~ ReleaseDate, data = df, na.action = "na.exclude",
      span = 0.15, degree = 1, normalize = FALSE,
      control = loess.control(iterations = 1))

Number of Observations: 359
Equivalent Number of Parameters: 11.16
Residual Standard Error: 40.63
Trace of smoother matrix: 13.26

Control settings:
  normalize: FALSE
    span    : 0.15
   degree   : 1
   family   : gaussian
  surface   : interpolate  cell = 0.2
```

The `loess` function in R uses different default parameters than the LOESS procedure, but the options specified in the preceding statements force the `loess` model to be fairly similar to the model shown in [Figure 9.20](#). (For an even closer match, use the `INTERP=CUBIC` option in the `MODEL` statement for the LOESS procedure.) Note that the smoothing parameter used in the R model is 0.15. This value is chosen because [Figure 9.20](#) shows that 0.1518 is the value of the smoothing parameter that optimizes the AICC statistic as computed by the LOESS procedure.

11.8.2 Computing an AICC Statistic in R

The second step in this example is to compute the AICC statistic. The documentation for the LOESS procedure specifies that the procedure uses the following formula for the AICC statistic:

$$\text{AICC} = \log(\hat{\sigma}^2) + 1 + \frac{2(\text{Trace}(L) + 1)}{n - \text{Trace}(L) - 2}$$

where n is the number of nonmissing observations, $\hat{\sigma}^2$ is the error mean square, and L is the loess smoothing matrix. Fortunately, the `loess.obj` object returned by the `loess` function contains all of the necessary information required to compute the AICC statistic.

To compute the AICC statistic, look at the contents of the `loess.obj` object by using the `str` function, as shown in the following statements. A partial listing of the structure is shown in [Figure 11.15](#).

```
/* display structure of an R object */
submit / R;
    str(loess.obj)
endsubmit;
```

Figure 11.15 Partial Listing of the Structure of a Loess Object in R

```
List of 17
 $ n          : int 359
 $ fitted     : num [1:359] 36.9 36.9 36.9 37.5 37.7 ...
 $ residuals: Named num [1:359] 8.15 28.15 -6.85 -7.49 -17.74 ...
 ..- attr(*, "names")= chr [1:359] "1" "2" "3" "4" ...
 $ enp       : num 11.2
 $ s         : num 40.6
 $ one.delta : num 344
 $ two.delta : num 343
 $ trace.hat : num 13.3
 ... more lines ...
```

The structure contains 17 components, including three components that can be used to construct the AICC statistic. The `n` component gives the number of nonmissing observations. The `residuals` component can be used to form the expression $\hat{\sigma}^2 = \sum_i r_i^2 / n$, where r_i is the i th residual. The trace of the smoothing matrix, $\text{Trace}(L)$, is given by `trace.hat`. Consequently, the following statements create an R function named `aicc` that computes the AICC statistic for a loess object:

```

/* create an R function that computes AICC for a loess model */
submit / R;
  aicc <- function(loess.obj)
  {
    n <- loess.obj$n
    r <- loess.obj$residuals;
    TraceL <- loess.obj$trace.hat
    sigma.hat2 <- sum(r^2, na.rm=TRUE) / n
    aicc <- log(sigma.hat2) + 1 + (2*(TraceL+1)) / (n-TraceL-2)
    return(aicc)
  }
endsubmit;

```

Notice that defining a function in R is very similar to defining a function module in SAS/IML. The following statements call the function and display the AICC value for the loess model with the smoothing parameter 0.15:

```

submit / R;
  aicc(loess.obj)
endsubmit;

```

The value computed by the `aicc` function (shown in [Figure 11.16](#)) is close to the value reported in [Figure 9.20](#).

Figure 11.16 AICC Statistic Computed by an R Function

```
[1] 8.448075
```

11.8.3 Encapsulating R Statements into a SAS/IML Module

The third step in this example is to write a SAS/IML module that encapsulates the R code. Recall that the `GoldenSection` module (defined in [Chapter 3](#)) calls a SAS/IML function module called `Func`. The `Func` module takes a single parameter and returns the function evaluated at that parameter. In this example, the parameter is the smoothing parameter for the loess fit. The function is the AICC statistic for the fit. Consequently, the following statements define a SAS/IML module that uses R to fit a loess model and to compute the AICC statistic for that model:

```

/* module that calls R to compute a model and statistic */
start Func(alpha) global (XVar, YVar);
  /* Fits a loess model in R for a value of the smoothing parameter, alpha.
   * Returns the AICC statistic. */
  submit alpha XVar YVar / R;
    loess.obj <- loess(&YVar ~ &XVar, data=df, span=&alpha,
                      na.action="na.exclude", degree=1, normalize=FALSE,
                      control=loess.control(iterations=1));
    crit <- aicc(loess.obj);
  endsubmit;
  run ImportMatrixFromR(aicc, "crit");
  return ( aicc );
finish;

```

The Func module has a single argument, **alpha**, but also uses two global variables, **xVar** and **yVar**, which should be defined in the scope of the main program. These global variables give the names of the two variables used in the model. All three of these variable values are passed to R by listing the variables in the SUBMIT statement and referring to the values of the variables by preceding the names with an ampersand (&). The R statements merely evaluate a loess model for the given smoothing parameter and compute the AICC statistic for that fit by calling the **aicc** function that you defined in the previous section. Lastly, the value of the AICC statistic is transferred into a SAS/IML matrix, and this value is returned by the module.

You can call the module to see the AICC value for the loess model with the smoothing parameter 0.15, as shown in the following statements:

```
AICC = Func(0.15);
print AICC;
```

The AICC statistic is shown in [Figure 11.17](#). Note that this is the same value shown in [Figure 11.16](#).

Figure 11.17 Result of Calling the Func Module

AICC
8.4480752

Programming Tip: You can encapsulate R statements into an IMLPlus module to make it easier to call and reuse the statements.

11.8.4 Finding the Best Smoother by Minimizing the AICC Statistic

The last step of this example is to call the GoldenSection module that you defined in [Section 3.6](#). Assuming that the GoldenSection module is on the SAS/IML Studio search path, you can find the loess smoothing parameter that leads to a minimum value of the AICC statistic by calling the module as shown in the following statements:

```
/* find parameter that minimizes the AICC on [0, 0.9] */
Alpha = GoldenSection(0, 0.9, 1e-3);
minimumAICC = Func(Alpha);
print Alpha minimumAICC;
```

Figure 11.18 Smoothing Parameter for the Minimum Value of the AICC Statistic

Alpha	minimumAICC
0.1503696	8.4480752

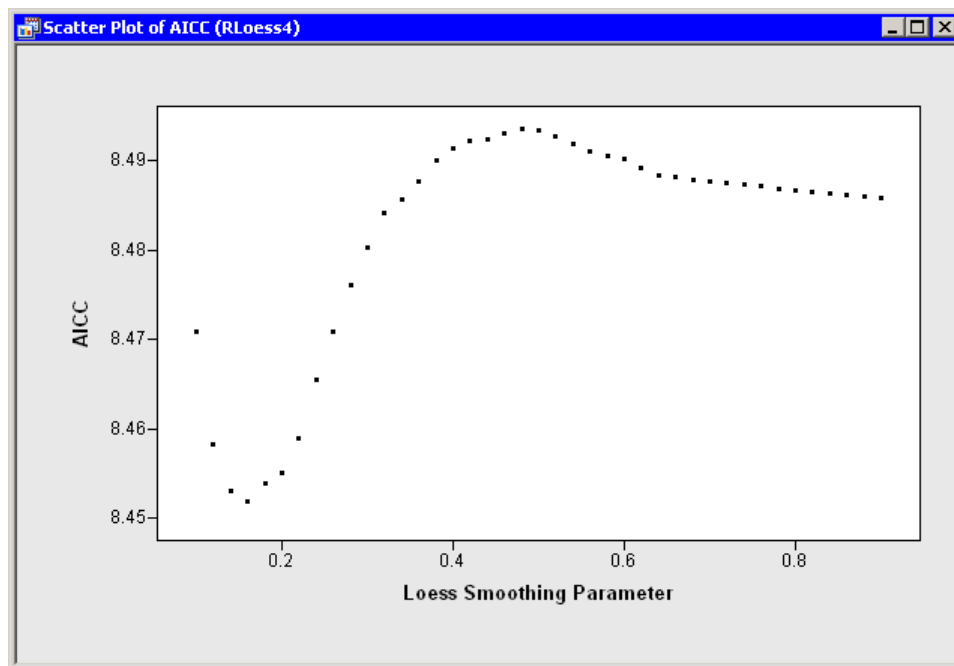
The minimum value, shown in [Figure 11.18](#), is close to 0.1518, which is the optimal value found by the LOESS procedure. The differences between the two values can be explained by noting that

the algorithms are not identical.

Given that the AICC function is not a continuous function of the smoothing parameter and that the AICC function can have multiple local minima, it is surprising that the GoldenSection algorithm works as well as it does for this example. It turns out that for these data the AICC function is particularly well-behaved. The following statements graph the AICC statistic as a function of the smoothing parameter. The resulting graph is shown in Figure 11.19.

```
/* graph AICC statistic as a function of the smoothing parameter */
x = do(0.1, 0.9, 0.02);          /* row vector of sequential values */
x = t(x);                        /* transpose to column vector */
y = j(nrow(x), 1);
do i = 1 to nrow(x);
    y[i] = Func(x[i]);
end;
declare ScatterPlot p2;
p2 = ScatterPlot.Create("AICC", x, y);
p2.SetAxisLabel(XAXIS, "Loess Smoothing Parameter");
p2.SetAxisLabel(YAXIS, "AICC");
```

Figure 11.19 The AICC Statistic as a Function of the Smoothing Parameter



11.8.5 Conclusions

The example in this section demonstrates how SAS and R software can work together. The SAS/IML program controls the flow of the algorithm, but you can choose to carry out some computation in R.

The interface to R in SAS/IML Studio enables you to pass data between the two statistical systems and to call R functions from SAS/IML programs. As the example in this section shows, you can

have data in a SAS data set but choose to call an analysis in R. The loess analysis in R lacks a feature found in the LOESS procedure, namely the automatic selection of a smoothing parameter that minimizes some criterion. However, you can use a previously written SAS/IML module to optimize the smoothing parameter for the R model. You could also have written the AICC function in the SAS/IML language instead of in R.

What else can be learned from this example? First, by calling R functions from SAS/IML programs you can, in effect, increase the SAS/IML library of functions. Second, by wrapping R statements in a SAS/IML module, you can encapsulate the R code; the program that calls the module can ignore the fact that R is used internally by the module. Third, the interface to R provided by SAS/IML Studio gives you tremendous flexibility to take advantage of algorithms written in either software language. You can continue to use all of your existing knowledge about SAS programming even while you evaluate statistical techniques implemented in R packages.

11.9 Creating Graphics in R

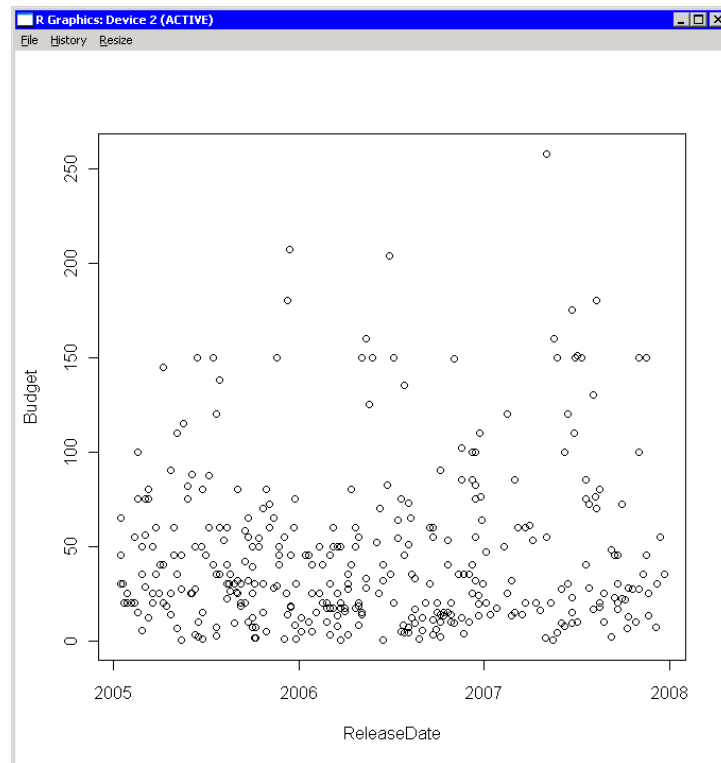
Advocates of R software often point to the graphics as a major reason for using R. In R, there is a graphical window that you can partition in order to create multiple plots in a single window. There are built-in high-level functions such as `plot` and `hist` that create graphs, and other functions such as `lines` and `points` that overlay markers or curves onto an existing graph. There are also several graphical packages that you can use to create special graph types or to add functionality to the R graphics.

You do not have to do anything special to use R graphics from SAS/IML Studio. When you call an R graphical function, that function creates a graphical window and draws to it. The graphical window is not part of the SAS/IML Studio workspace: it is owned and managed by the R process.

The following statements create an R graph from an IMLPlus program:

```
/* create an R graph */
run ExportDataSetToR("Sasuser.Movies", "MovieFrame");
submit / R;
    plot(Budget ~ ReleaseDate, data=MovieFrame)
endsubmit;
```

A window appears that contains the R graph, as shown in [Figure 11.20](#). In contrast to the SAS/IML Studio graphics, the default graphics in R are not dynamically linked—neither to each other nor to IMLPlus graphics of the same data. In particular, as explained in [Section 11.4.2](#), the R graphs do not reflect any attributes that are associated with marker shape, color, or selection state, even if the data are transferred to R from an IMLPlus data object.

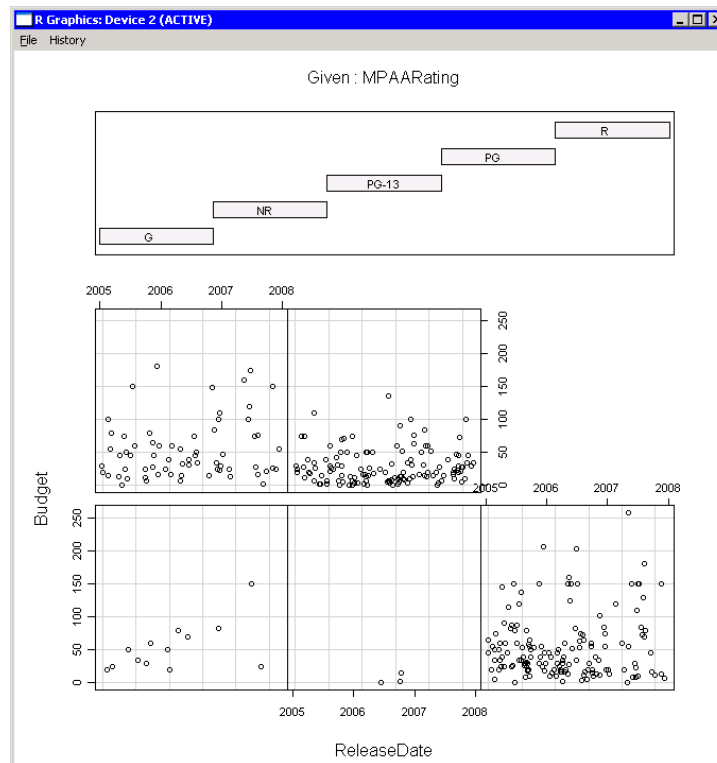
Figure 11.20 An R Graph Created from an IMLPlus Program

Programming Tip: You can create R graphics from IMLPlus programs.

There is little need to use R to create a basic scatter plot, but R also provides some sophisticated graphs that you might want to use as part of an analysis. For example, R has a function called `coplot` that creates a *conditioning plot*. A conditioning plot is a panel of scatter plots for the variables X and Y , given particular values of a third variable C (called the conditioning variable). When C is a nominal variable, the conditioning plot is similar to BY-group processing in SAS: `coplot` produces a scatter plot for each level of C . The `coplot` function also graphically indicates the levels of C next to the panel of scatter plots. This is shown in Figure 11.21, which is produced by the following statements:

```
submit / R;
  coplot(Budget ~ ReleaseDate | MPAARating, data=MovieFrame)
endsubmit;
```


Figure 11.21 An R Coplot Graph



The scatter plot of Budget versus ReleaseDate, given that the movie is rated G, is located in the leftmost column of the bottom row of Figure 11.21. The scatter plot, given that the movie is not rated, is in the second column of the bottom row, and so forth. This panel of scatter plots enables you to see all BY groups of the MPAARating variable in a single glance. (You can create a similar panel of plots in SAS/IML Studio; see the chapter “Plotting Subsets of Data” in the *SAS/IML Studio User’s Guide*.)

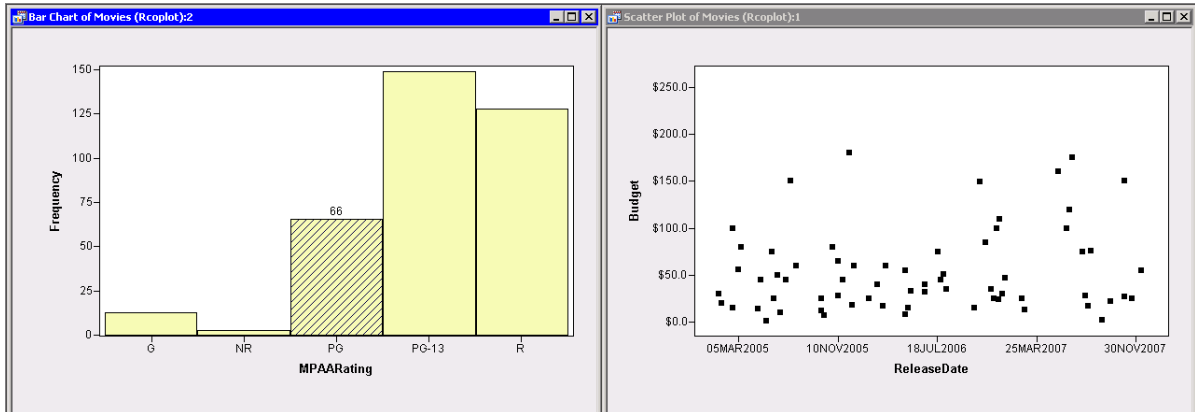
The conditioning plot panel in R is easy to make, and it can be very informative. In SAS/IML Studio you can create each plot in the panel by using the interactive and dynamically linked IMLPlus graphics. The following statements create a single scatter plot of Budget versus ReleaseDate and also create a bar chart of the MPAARating variable:

```
/* show coplot functionality in IMLPlus */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");
declare ScatterPlot cp;
cp = ScatterPlot.Create(dobj, "ReleaseDate", "Budget");
cp.ShowObs(false);          /* show observations only when selected */
declare BarChart bar;
bar = BarChart.Create(dobj, "MPAARating");
```

The ShowObs method in the Plot class is used to show only those observations that are selected. When you click on a bar in the bar chart, the scatter plot displays one of the five scatter plots shown in the conditioning plot. For example, Figure 11.22 shows the scatter plot for PG-rated movies. The IMLPlus formulation uses the interactive nature of the IMLPlus graphics to enable you to see the scatter plot, given a particular movie rating. (You can even see a scatter plot for a union of rating

categories, such as for movies that are rated either PG or PG-13.) Both approaches have advantages: the interactive display is more flexible, but the paneled display enables you to more easily compare data across different categories, such as comparing movies rated PG with those rated PG-13.

Figure 11.22 An Interactive Coplot in SAS/IML Studio



As shown in [Section 11.8.3](#), it is sometimes helpful to create an IMLPlus module that encapsulates a call to R. For example, the following statements create an IMLPlus module that calls the `coplot` function in R:

```
/* define module that creates a graph in R */
start CallRCoplot(DataObject dobj, XVar, YVar, CoVar);
  DSName = dobj.GetServerDataSetName();
  dobj.ExportToR(DSName);
  submit XVar YVar CoVar DSName / R;
    coplot(&YVar ~ &XVar | &CoVar, data = &DSName)
  endsubmit;
finish;

run CallRCoplot(dobj, "ReleaseDate", "Budget", "MPAARating");
```

The module takes four arguments: a data object and three character matrices that contain, respectively, the name of the *X*, *Y*, and conditioning variables. The first statement in the module illustrates a clever technique: the `GetServerDataSetName` returns a string that can be used as the name of the R data frame. (In this example, the string is “Movies.”) You could hard-code this string by writing `DSName = "MyData"`, but calling the `GetServerDataSetName` method enables you to make sure that the R data frame has a name related to the data. The second statement in the module creates the R data frame from the data object. The `SUBMIT` statement then calls the `coplot` function in R, with the arguments to the function substituted from the SAS/IML matrices `XVar`, `YVar`, `CoVar`, and `DSName`. After the module is defined, you can call it as you would any IMLPlus module. The results from the previous program is identical to [Figure 11.21](#).

You can even use the following technique to modify the module so that it skips the creation of the R graphics window. (Thanks to Simon Smith for this idea.) The graphic is written to the Windows clipboard and pasted from the clipboard directly into the SAS/IML Studio output document.

```

/* paste R graphic into the SAS/IML Studio output window */
start CallRCoplot2(DataObject dobj, XVar, YVar, CoVar);
  DSName = dobj.GetServerDataSetName();
  dobj.ExportToR(DSName);
  submit XVar YVar CoVar DSName / R;
    win.metafile(); # 1. write graphics to Windows clipboard
    coplot(&YVar ~ &XVar | &CoVar, data = &DSName)
    rc <- dev.off() # 2. close graphics device (the metafile)
  endsubmit;
  /* 3. paste graphics from clipboard to output document */
  OutputDocument.GetDefault().PasteGraphic(GRAPHICTYPE_WMF);
finish;

run CallRCoplot2(dobj, "ReleaseDate", "Budget", "MPAARating");

```

The output from the module is shown in [Figure 11.23](#). The module is identical to the CallRCoplot module except for three statements.

1. The `win.metafile` function writes the subsequent graph to a file in Windows Metafile Format (WMF). If no filename is specified, as in this example, the output is copied to the Windows clipboard. No writing occurs until the file is closed.
2. The `dev.off` function closes the current graphics device, which results in the graph being written to the clipboard.
3. The contents of the clipboard are pasted into the output document. This is accomplished by using the `OutputDocument` class to get the default output document. The `PasteGraphic` method is then used to paste from the clipboard in WMF format. (The `OutputDocument` class is described in [Section 12.8](#).)

Programming Tip: You can write an R graph to the Windows clipboard and paste the graph into the SAS/IML Studio output window.

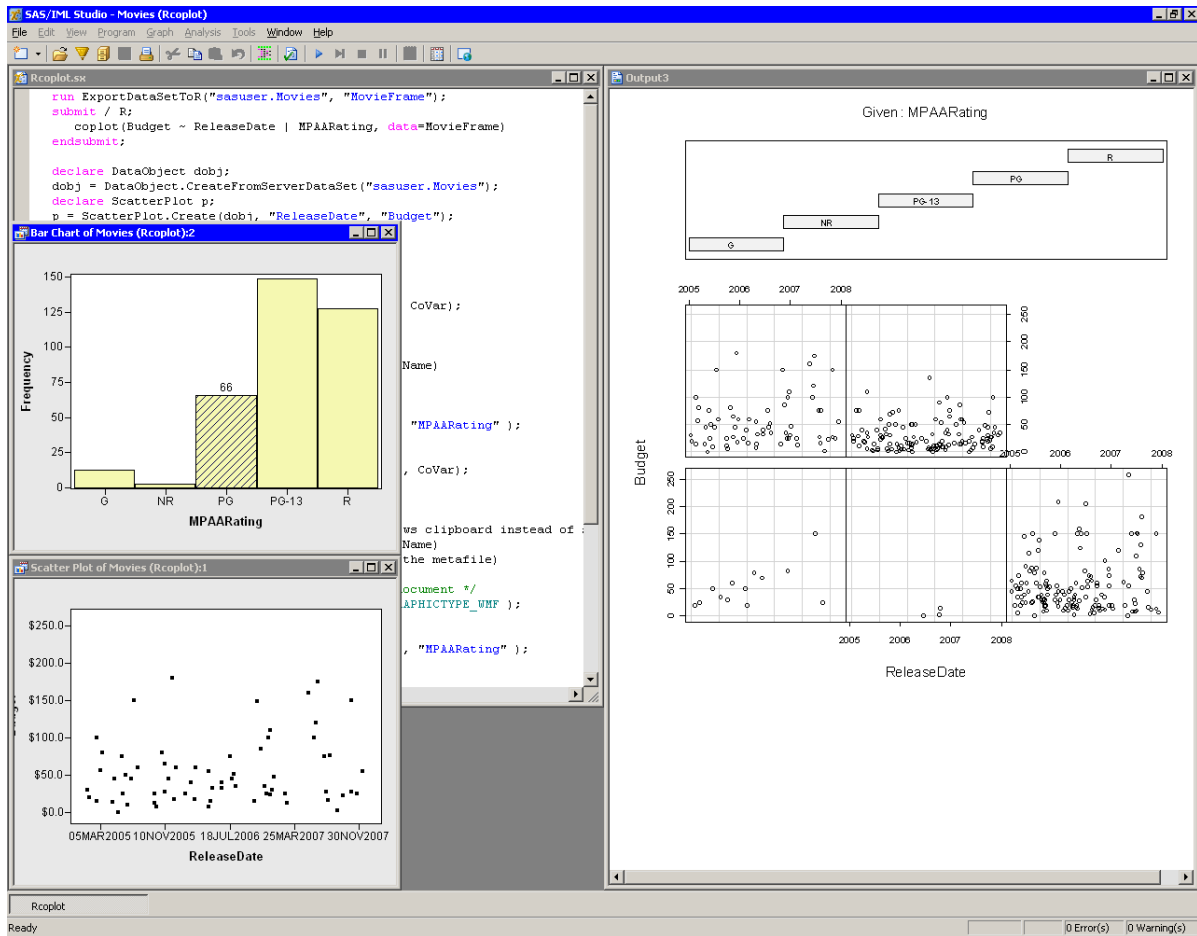
The statement that uses the `OutputDocument` class might look strange to readers who are not experienced with object-oriented programming techniques. The following statements might look more familiar, but are equivalent:

```

declare OutputDocument doc;
doc = OutputDocument.GetDefault();
doc.PasteGraphic(GRAPHICTYPE_WMF);

```

This second formulation explicitly declares that `doc` is an object of the `OutputDocument` class. (The `OutputDocument` class enables you to programatically control a SAS/IML Studio output window.) The `GetDefault` method returns the current output document if it exists, or creates a new output document if one does not exist. The second formulation is preferred when you intend to use the output document several times. The shorthand used in the module is sometimes used to save typing when you call only a single `OutputDocument` method.

Figure 11.23 Copying the R Graphic to the SAS/IML Studio Output Window

11.10 References

- Canty, A. and Ripley, B. D. (2009), *boot: Bootstrap R (S-Plus) Functions*, R package version 1.2-37.
- Davison, A. C. and Hinkley, D. V. (1997), *Bootstrap Methods and Their Application*, Cambridge, UK: Cambridge University Press.
- Wand, M. P. and Jones, M. C. (1995), *Kernel Smoothing*, London: Chapman & Hall.

Chapter 12

Regression Diagnostics

Contents

12.1 Overview of Regression Diagnostics	279
12.2 Fitting a Regression Model	280
12.3 Identifying Influential Observations	284
12.4 Identifying Outliers and High-Leverage Observations	286
12.5 Examining the Distribution of Residuals	287
12.6 Regression Diagnostics for Models with Classification Variables	289
12.7 Comparing Two Regression Models	292
12.7.1 Comparing Analyses in Different Workspaces	293
12.7.2 Comparing Analyses in the Same Workspace	294
12.8 Case Study: Comparing Least Squares and Robust Regression Models	296
12.9 Logistic Regression Diagnostics	303
12.10 Viewing ODS Statistical Graphics	307
12.11 References	310

12.1 Overview of Regression Diagnostics

A regression diagnostic identifies observations that are either unusual or highly influential in a regression analysis. It is a good idea to plot regression diagnostics when you perform such an analysis.

The SAS/STAT regression procedures use ODS Graphics to create diagnostic plots. However, it is also useful to create diagnostic plots by using the statistical graphics in SAS/IML Studio, as shown in this chapter. Because the graphics in SAS/IML Studio are interactive, it is easy to identify and investigate outliers and high-leverage points for a regression model. For example, you can click on an outlier in a graph. This labels the observation and highlights it in other graphs. You can double-click on an observation to examine the values of other covariates. If there is a large group of outliers, you can select all of the outliers and determine whether those observations share some characteristic such as a high (or low) value of some covariate. If so, you can revise your model by incorporating that covariate into the regression model.

If used correctly, these plots can assist you in building statistical models and investigating outliers. For an explanation of the statistics that are associated with regression diagnostics, see Belsley, Kuh, and Welsch (1980).

Programming Tip: Use interactive graphics to create diagnostic plots. Use the plots to identify and examine unusual or influential observations in a regression model.

This chapter is an applied chapter that describes how to do the following:

- Fit a regression model.
- Add reference lines to a scatter plot.
- Identify influential observations such as outliers and high-leverage observations.
- Compare two regression models.

12.2 Fitting a Regression Model

This section uses the techniques that are described in [Chapter 4](#) to call the GLM procedure to perform a linear regression and to create graphs of the analysis in SAS/IML Studio. Techniques from [Chapter 9](#) are used to add prediction bands and reference lines to the graphs in order to make the graphs more interpretable.

The regression model for this example uses variables from the Movies data set. As seen in previous chapters, there appears to be a relationship between the Budget of a movie and the US_Gross variable (see [Figure 7.7](#)). However, neither variable is normally distributed (see [Figure 7.6](#)), so Atkinson (1985) and others suggest applying a normalizing transformation to these variables. For these data, it makes sense to apply a logarithmic transformation to each variable (see [Figure 8.5](#)) and to model the log-transformed variables.

The analysis that begins in this section is continued in subsequent sections. The following list is an outline of the main steps in the analysis. The numbers used in the list are also used in comments in the program.

1. Apply a logarithmic transformation to the US_Gross and Budget variables.
2. Call PROC GLM to run a regression analysis.
3. Read predicted values and confidence limits into SAS/IML vectors.
4. Create a data object that contains the data and results.
5. Create a plot of the data.
6. Overlay the results of the regression analysis.
7. Create Cook's D diagnostic plot ([Section 12.3](#)).
8. Overlay reference lines on Cook's D plot.

9. Create a plot of studentized residuals versus leverage (see [Section 12.4](#)).
10. Overlay reference lines on the residual-leverage plot.

The following statements transform the variables and use PROC GLM to fit a regression model:

```
/* create regression diagnostic plots in SAS/IML Studio */
submit;
/* 1. log transform of US_Gross and Budget variables */
data movies;
    set Sasuser.Movies;
    LogUS_Gross = log(US_Gross);
    LogBudget = log(Budget);
    label LogUS_Gross = "log(US Gross)"  LogBudget = "log(Budget)";
run;

/* 2. run classic regression analysis */
proc glm data=movies;
    model LogUS_Gross = LogBudget;
    output out=GLMOut
        P=P LCLM=LCLM UCLM=UCLM          /* predictions and CLM */
        R=R RSTUDENT=ExtStudR           /* residuals */
        COOKD=CookD H=Leverage;          /* influence diagnostics */
quit;
endsubmit;
```

The DATA step computes the transformed variables named LogBudget and LogUS_Gross. Then the GLM procedure is called to model LogBudget by LogUS_Gross. The output data set (called GLMOut) contains the input data in addition to six new variables created by the procedure:

P contains the predicted values for the model

LCLM contains the lower 95% confidence limits for the mean of the predicted value

UCLM contains the upper 95% confidence limits for the mean of the predicted value

R contains the residual values for the model

ExtStudR contains externally studentized residuals, which are studentized residuals with the current observation deleted. (Recall that a studentized residual is a residual divided by its standard error.)

CookD contains Cook's D influence statistic

Leverage contains the leverage statistic

The output from the GLM procedure is shown in [Figure 12.1](#).

Figure 12.1 Regression Summary

The GLM Procedure					
Number of Observations Read			359		
Number of Observations Used			359		
The GLM Procedure					
Dependent Variable: LogUS_Gross		log(US Gross)			
Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	243.4183267	243.4183267	181.38	<.0001
Error	357	479.1034740	1.3420265		
Corrected Total	358	722.5218007			
R-Square	Coeff Var	Root MSE	LogUS_Gross Mean		
0.336901	35.06699	1.158459	3.303559		
Source	DF	Type I SS	Mean Square	F Value	Pr > F
LogBudget	1	243.4183267	243.4183267	181.38	<.0001
Source	DF	Type III SS	Mean Square	F Value	Pr > F
LogBudget	1	243.4183267	243.4183267	181.38	<.0001
Parameter	Estimate	Standard Error	t Value	Pr > t	
Intercept	0.8982088871	0.18877580	4.76	<.0001	
LogBudget	0.7234640923	0.05371813	13.47	<.0001	

The parameter estimates table shown in [Figure 12.1](#) provides estimates for the coefficients of the linear model. The estimate for the LogBudget coefficient indicates that each unit of LogBudget can be expected to generate an additional 0.72 units of LogUS_Gross. In terms of the original variables, $US_Gross \approx e^{0.9}(\text{Budget})^{0.72}$. Thus if you have two movies, one with twice the budget of the other, the more expensive movie is expected to bring in $2^{0.72} \approx 1.64$ times the revenue of the less expensive movie. The fact that the coefficient for LogBudget is less than 1 implies that movies with very large budgets have predicted gross revenues that are *less* than the budget! For this model, the “break-even” point is a budget of about $\exp(0.9/(1 - 0.72)) \approx 24.9$ million dollars; movies with larger budgets have predicted US gross revenues that are less than their budgets.

As the previous analysis demonstrates, it can be difficult to interpret a model whose variables are log-transformed. It can be useful, therefore, to invert the log-transformation and plot the predicted values and explanatory variables in the original scale units, as shown in [Figure 12.2](#). The following statements read some of the output variables from the GLMOut dataset into SAS/IML matrices and use the EXP function to transform the variables from the log scale back into the original scale:


```

/* 3a. read variables from output data set into vectors */
use GLMOut;
read all var {LogBudget P UCLM LCLM} into logFit; /* log scale      */
read all var {CookD};                          /* read for Step 8 */
close GLMOut;

/* 3b. invert transformation for predicted values and CLM */
fit = exp(LogFit);

```

You can also read the GLMOut data set into a data object so that you can create dynamically linked graphics. The following statements create a data object and add new columns that contain the predicted values in the original scale. The new columns come from data in the `LogFit` matrix. Observation numbers are also added. These variables will be used in subsequent diagnostic plots.

```

/* 4. create data object that contains data and results */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Work.GLMOut");
dobj.SetRoleVar(ROLE_LABEL, "Title");

/* 4a. add transformed variable to the data object */
dobj.AddVar("Pred", "Predicted US Gross (million $)", fit[,2]);
dobj.AddVar("ObsNumber", "Observation Number", T(1:nrow(fit)));

```

The `SetRoleVar` method specifies a label for selected observations in subsequent plots. The `Title` variable contains the name of each movie in the data.

After the data object is created, you can create graphs that are linked to the data object. The following statements create a line plot of the `Budget` and `Pred` variables, and override a few default characteristics of the line plot:

```

/* 5. create line plot */
declare LinePlot line;
line = LinePlot.Create(dobj, "Budget", "Pred");
line.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
line.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);
line.ShowObs(false);
line.SetLineWidth(2);

```

It is useful to add confidence limits to the line plot of the predicted values. The following statements sort the `fit` matrix by the `Budget` variable and use the `DrawPolygon` method to add confidence limits as described in [Section 9.1.4](#):

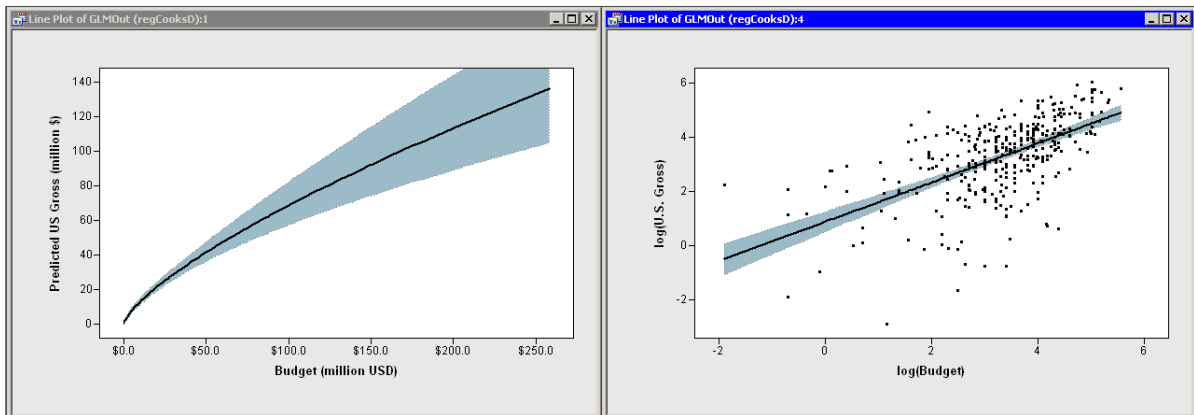
```

/* 6. Use drawing subsystem to draw CLM band. */
line.DrawUseDataCoordinates();
line.DrawSetRegion(PLOTBACKGROUND);
LightBlue = 0x9BBBC9; /* RGB value: (155, 187, 201) */
line.DrawSetBrushColor(LightBlue);
line.DrawSetPenAttributes(GRAY, DOTTED, 1);
call sort(fit, 1); /* sort CLM by Budget */
x = fit[,1]; LCLM = fit[,3]; UCLM = fit[,4];
N = nrow(fit);
line.DrawPolygon(x//x[N:1], LCLM//UCLM[N:1], true);

```

The resulting graph is shown in Figure 12.2. This plot (shown on the left in the figure) enables you to focus on the predicted values without seeing the observed values. Alternatively, you could create a scatter plot of the observed values, and add the predicted curve with the `Plot.DrawLine` method as described in Section 9.1. The second plot (shown on the right in the figure) displays the log-transformed data and predicted values. The program statements that produce this second graph are left as an exercise.

Figure 12.2 Regression Shown in the Original Scale (left) and the Log-Transformed Scale (right)



12.3 Identifying Influential Observations

An influential observation is one that, according to some measure, has a large effect on a regression analysis. Intuitively, an influential observation is one for which the analysis would change substantially if the observation were deleted. There are many statistics that attempt to quantify various aspects of the influence of an observation. For details, see Belsley, Kuh, and Welsch (1980) or see the documentation for the `INFLUENCE` option in the `MODEL` statement of the `REG` procedure in the *SAS/STAT User's Guide*.

In a regression analysis, there are two types of influential observations: outliers in the response variable and outliers in the space of the explanatory variables. The outliers in the response variable (often simply called “outliers”) are typically determined by the size of a residual or studentized residual. The outliers in the space of the explanatory variables are often called “high-leverage points.” Cook’s D is a popular statistic that incorporates both of these notions of influence. An observation is often judged to be *influential* if the Cook’s D statistic exceeds $4/n$, where n is the number of nonmissing observations (Rawlings, Pantula, and Dickey 1998).

The program statements in this section are a continuation of the analysis in Section 12.2 and use the `abline` module defined in Section 9.4. The standard Cook’s D plot is a graph of the Cook’s D statistic versus the observation number. The Cook’s D statistic is contained in the `CookD` variable in the `dobj` data object. The data object also contains the `ObsNumber` variable. The Cook’s D statistic is also contained in the SAS/IML vector named `CookD`, so you can count the number of nonmissing values and use that value to compute the cutoff value for the Cook’s D statistic. You can use the `abline` module to display the cutoff, as shown in the following statements:

```

/* 7. create Cook's D plot */
declare ScatterPlot cook;
cook = ScatterPlot.Create(dobj, "ObsNumber", "CookD");
cook.SetMarkerSize(5);

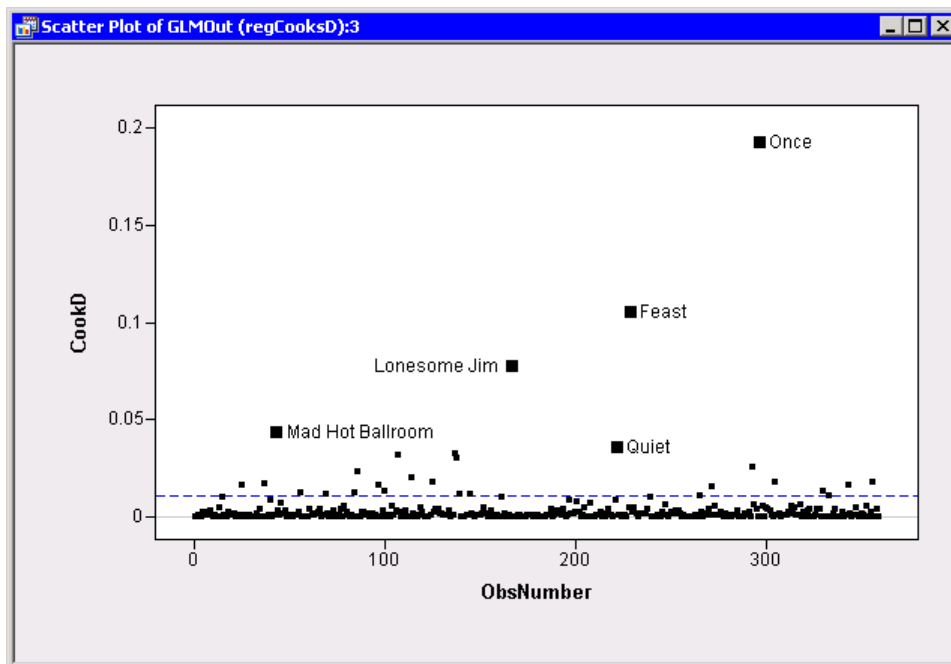
/* 8. draw reference lines on Cook's D plot */
solidGray = 0xc8c8c8|| SOLID|| 1; /* use solid gray for y=0 */
run abline(cook, 0, 0, solidGray); /* horiz line at y=0 */

nonMiss = sum(CookD^=.); /* number of nonmissing values */
cookCutoff = 4 / nonMiss;
run abline(cook, cookCutoff, 0, .); /* high Cook's D: horiz line */

```

The plot of Cook's D is shown in Figure 12.3. As discussed in Section 12.1, the reason for creating these plots in SAS/IML Studio is that you can interactively click on influential observations to identify the observation. This is shown in Figure 12.3 where the five movies with the largest Cook's D value are selected. Upon further investigation of the selected movies, you can discover that the movies have similar characteristics. They all have extremely low budgets (most less than one million dollars), which makes them high-leverage points. They also have US_Gross values that are either much higher or much lower than predicted by the model, which makes them outliers or nearly outliers. Two of the movies (*Once* and *Mad Hot Ballroom*) were commercial successes that earned many times more than predicted by the regression model; the others were commercial flops.

Figure 12.3 Plot of Cook's D Statistic



12.4 Identifying Outliers and High-Leverage Observations

The Cook's D plot attempts to identify observations that either have high leverage or are outliers in the response variable. However, if an observation has a large Cook's D value, it is not clear (without further investigation) whether that point has high leverage, is an outlier, or both. A plot that complements the Cook's D plot is a plot of the studentized residuals versus the leverage statistic for each observation.

These residual and leverage statistics were created in the output data set by the GLM procedure in [Section 12.2](#). The program statements in this section are a continuation of the analysis in the preceding sections. For the residual-leverage plot, Belsley, Kuh, and Welsch (1980) suggests adding a horizontal reference line at ± 2 to identify observations with large studentized residuals, and a vertical line at $2p/n$ to identify points with high leverage, where p is the total number of parameters in the model, including the intercept. The following statements create the plot and use the `abline` module defined in [Section 9.4](#) to add reference lines:

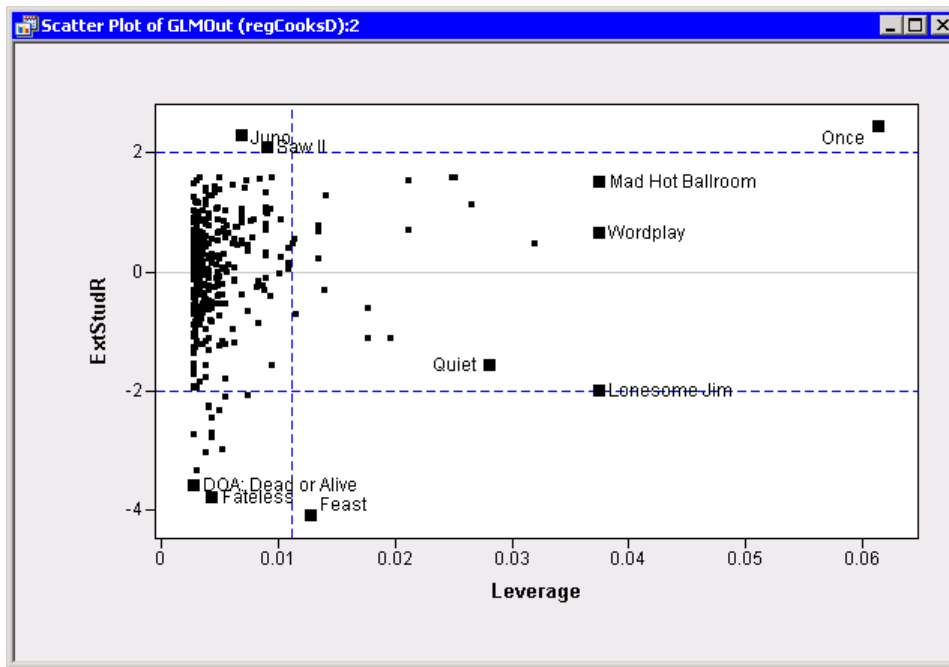
```
/* 9. create plot of studentized residuals versus leverage */
declare ScatterPlot resid;
resid = ScatterPlot.Create(dobj, "Leverage", "ExtStudR");
resid.SetMarkerSize(5);

/* 10. draw reference lines on residual-leverage plot */
run abline(resid, 0, 0, solidGray); /* horiz line at y=0 */

run abline(resid, {2,-2}, {0,0}, .); /* outliers: horiz lines at +/-2 */
p = 2; /* number of params */
leverage = 2*p / nonMiss;
run abline(resid, leverage, ., .); /* high leverage: vertical line */
```

The resulting plot is shown in [Figure 12.4](#). The graph shows 10 points that have been manually selected and labeled:

- The five movies (also shown in [Figure 12.3](#)) that have the largest Cook's D : *Once*, *Feast*, *Lonesome Jim*, *Mad Hot Ballroom*, and *Quiet*. It is now apparent that all are high-leverage points and that *Once* and *Feast* are outliers.
- Two movies with large positive residuals that are not high-leverage points: *Juno* and *Saw II*.
- Two movies with large negative residuals: *Fateless* and *DOA: Dead or Alive*.
- A movie with high leverage that has a modest positive residual: *Wordplay*.

Figure 12.4 Studentized Residuals versus Leverage

You can select all observations with large Cook's D statistics by using the mouse to create a selection rectangle in the Cook's D plot (not shown). If you do this, then the residual-leverage plot updates and reveals that those observations have either high leverage or are outliers.

12.5 Examining the Distribution of Residuals

A quantile-quantile (Q-Q) plot shows whether univariate data approximately follows some distribution. The most common Q-Q plot is the normal Q-Q plot, which graphically compares quantiles of the data to quantiles of a normal distribution. The normal Q-Q plot is often used to check whether the residuals of a regression are normally distributed.

There are several ways to compare the quantiles of data with the quantiles of a theoretical distribution. The UNIVARIATE procedure follows Blom (1958) who recommends the following approach:

1. Order the n nonmissing data values: $x_{(1)} \leq x_{(2)} \leq \dots x_{(n)}$. These are quantiles of the empirical cumulative distribution function (ECDF).
2. For each $x_{(i)}$, compute the standardized quantity $v_i = (i - 0.375)/(n + 0.25)$.
3. Compute the associated quantiles from the theoretical distribution: $q_i = F^{-1}(v_i)$, where F is the theoretical distribution function with unit scale and zero location parameter.
4. Plot the ordered pairs (q_i, v_i) for $i = 1, \dots, n$.

The SAS documentation for the UNIVARIATE procedure includes a section on how to interpret Q-Q plots. Briefly, if the Q-Q plot lies along a straight line, then there is graphical evidence to believe that the data are well-described by the theoretical distribution.

The following statements define a module that uses Blom's approach to compute the normal quantiles that are associated with univariate data:

```
/* define module to compute normal quantiles associated with data */
start GetNormalQuantiles(x);
  nonMissing = loc(x^=.);          /* locate nonmissing values */
  numNM = ncol(nonMissing);        /* count the nonmissing values */

  if numNM=0 then                  /* all values are missing */
    return ( j(nrow(x), ncol(x), .) );

  /* rank the observations; handle any missing values */
  r = j(nrow(x), ncol(x), .);
  r[nonMissing] = ranktie(x[nonMissing]);

  v = (r-0.375) / (numNM+0.25);    /* transform quantiles of ECDF */
  q = quantile("NORMAL", v);       /* associated normal quantiles */
  return ( q );
finish;
```

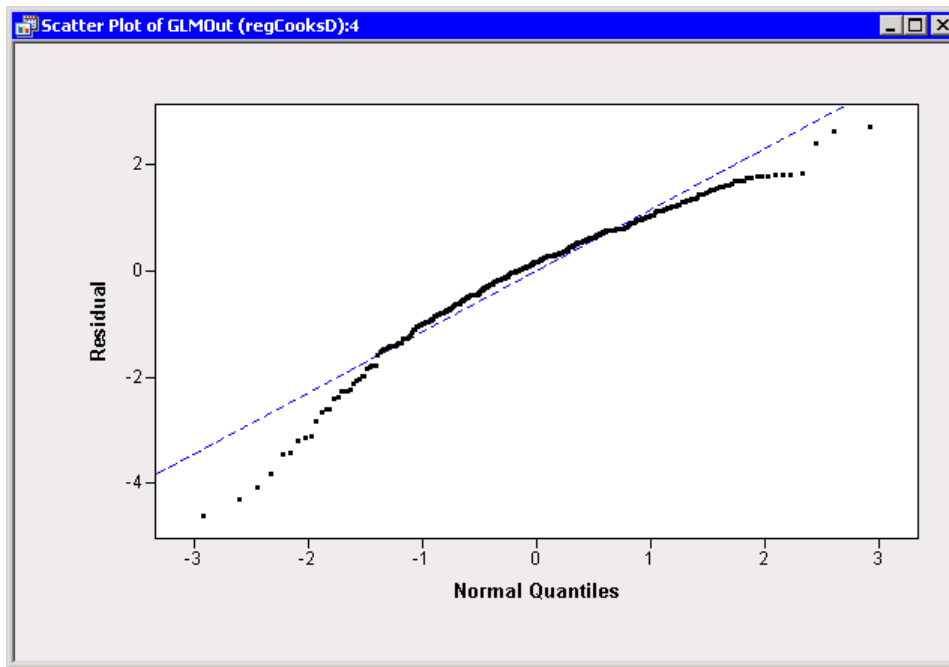
You can call the GetNormalQuantiles module on the residual values computed with the R= option in [Section 12.2](#). These residuals are contained in the data object in the R variable. The following statements retrieve the R variable and compute the normal quantiles that are associated with the residual values. The DataObject.AddVar method adds those normal quantiles to the data object. After adding the normal quantiles, you can create a Q-Q plot that is linked to the other plots.

```
/* create QQ plot of residuals */
dobj.GetVarData("R", r);
q = GetNormalQuantiles(r);
dobj.AddVar("QN_R", "Normal Quantiles", q);

declare ScatterPlot qq;
qq = ScatterPlot.Create(dobj, "QN_R", "R");
qq.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);

mu = r[:];
sigma = sqrt(var(r));
run abline(qq, mu, sigma, .);
```

The graph is shown in [Figure 12.5](#). The abline module adds a line to the plot. The slope of the line is the sample standard deviation of the residuals; the intercept is the mean of the residuals, which is zero.

Figure 12.5 A Quantile-Quantile Plot of Residuals

Because the points in the Q-Q plot in Figure 12.5 do not lie along a straight line, you can conclude that the residual values are not well-described by a normal distribution. In fact, the curved pattern of the points indicates that the distribution of the residuals is skewed to the left.

12.6 Regression Diagnostics for Models with Classification Variables

There have been many studies that attempt to determine whether movies that are nominated for (or win) Academy Awards also enjoy greater box-office revenue as a result of the increased publicity (for example, Simonoff and Sparrow (2000); Nelson et al. (2001)). Most conclude that an Academy Award nomination does, in fact, result in increased revenue.

The Movies data set contains an indicator variable, `AANomin`, that has the value 1 if the movie was nominated for an Academy Award. This section modifies the regression analysis in the previous sections to include this classification variable. The complete program is available from this book's companion Web site

The first step in the program (transforming variables) remains the same. However, in Step 2 of the program, you need to modify the call to the GLM procedure. Specifically, you need to add a `CLASS` statement to specify that `AANomin` is a classification variable, and also add `AANomin` and any interaction effects to the `MODEL` statement. It turns out that the interaction between `LogBudget` and `AANomin` is not significant (the p -value for the Type 3 statistic is 0.1223), so the following statements use only main effects:

```

/* 2. run classic regression analysis: class variable, main effects */
proc glm data=movies;
  class AANomin;
  model LogUS_Gross = LogBudget AANomin / solution;
  output out=GLMOut2
    P=P LCLM=LCLM UCLM=UCLM          /* predictions and CLM */
    RSTUDENT=ExtStudR                /* studentized residuals */
    COOKD=CookD H=Leverage;          /* influence diagnostics */
quit;

```

Notice that you need to include the SOLUTION option on the MODEL statement if you want to see the parameter estimates table.

In Step 3, it is useful to also read the AANomin variable into a SAS/IML vector:

```
read all var {AANomin};
```

In Step 4, after creating a data object from the GLM output data set, it is convenient to use marker shapes to represent whether a movie was nominated for an Academy Award. The following statements set all marker shapes to 'x', and then set the markers for award-nominated movies to triangles:

```

/* 4. create data object that contains data and results */
declare DataObject dobj1;
dobj1 = DataObject.CreateFromServerDataSet("Work.GLMOut2");
... lines omitted ...

/* 4b. use marker shapes to represent AANomin levels */
dobj1.SetMarkerShape(OBS_ALL, MARKER_X);
dobj1.SetMarkerShape(loc(AANomin), MARKER_TRIANGLE);
dobj1.SetNominal("AANomin");

```

The call to the DataObject.SetNominal method is recommended because it enables you to create a bar chart of the AANomin variable. If you do not call the SetNominal method, SAS/IML Studio will use a histogram to plot the distribution of the AANomin variable.

In Step 5, you can use the LinePlot.CreateWithGroup method to create the line plot of the predicted values. When you overlay the confidence limits, you need to iterate over the levels of the classification variable, as shown in the following statements:

```

/* 5. create line plot */
declare LinePlot line1;
line1 = LinePlot.CreateWithGroup(dobj1, "Budget", "P", "AANomin");
... lines omitted ...

u = unique(AANomin);
do i = 1 to ncol(u);
  idx = loc(AANomin=u[i]);
  N = ncol(idx);
  z = fit[idx,];
  call sort(z, 1);
  x = z[,1]; LCLM = z[,3]; UCLM = z[,4];
  line1.DrawPolygon(x//x[N:1], LCLM//UCLM[N:1], true);
end;

```


The only additional modification to the program is in Step 10 where you need to adjust the value of `p`, the number of parameters in the model. You can manually code `p=3` for this example, but that is prone to error if you decide to add additional parameters to the model at a later date. A better idea is to obtain the degrees of freedom from one of the ODS tables created by the GLM procedure. For example, prior to calling the GLM procedure, you can add the following statement:

```
ods output OverallAnova = Anova;
```

This writes the ANOVA table to a SAS data set. The first element of the `DF` variable contains the degrees of freedom for the model. You can add 1 to the model degrees of freedom to obtain the number of variables in the design matrix, as shown in the following statements:

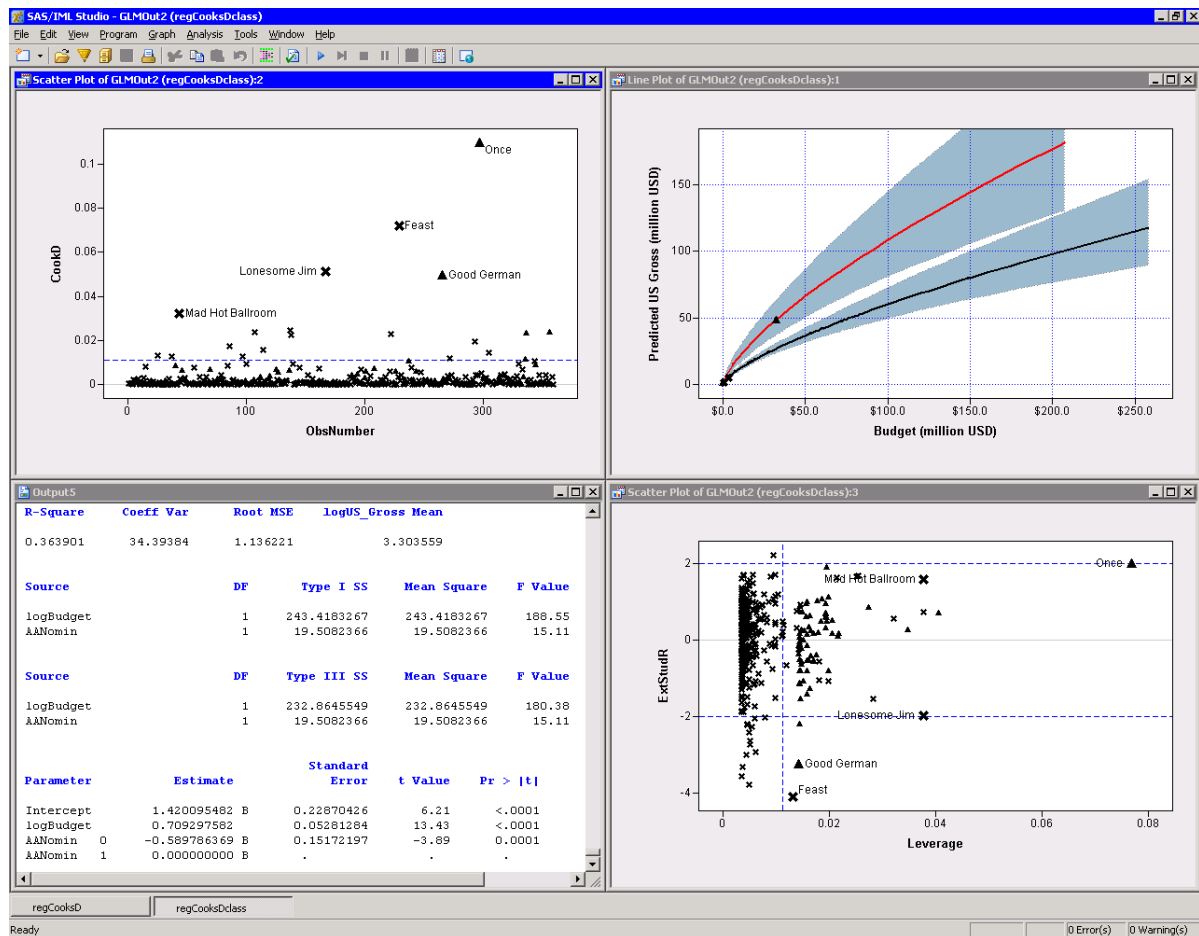
```
/* 10. draw reference lines on residual plot */
use Anova;
read point 1 var {DF};          /* first element is Model DF          */
close Anova;
p = DF + 1;                     /* number of params (+1 for intercept) */
```

You can use same technique (that is, reading relevant information from an ODS table created by PROC GLM) to read in the number of nonmissing observations from the NObs table. This is left as an exercise.

Programming Technique: To ensure a robust analysis, read ODS tables created by procedures in order to obtain parameters used by the model, such as the number of observations, the model degrees of freedom, and so on. If you later change the model, the graphs and reference lines remain correct and do not require modification.

The results of the program that models `LogUS_Gross` by `LogBudget` and `AANomin` is shown in Figure 12.6. The upper-right graph shows the predicted values and 95% confidence limits for the model. Given a value for the `Budget` variable, the expected value for the `US_Gross` variable is higher for movies that are nominated for an Academy Award. For example, for movies with 100-million-dollar budgets, the expected `US_Gross` is 50 million dollars higher for award-nominated movies.

Figure 12.6 Regression Model with a Classification Variable



The upper-left graph in Figure 12.6 is Cook's D plot. The five movies with the largest Cook's D statistic were manually selected. Four of the movies are the same as shown in Figure 12.3. One movie (*Good German*) was not in the top five for the previous model; it has more leverage under the new model because it was a small-budget movie that was nominated for an Academy Award. The residuals-leverage plot is shown in the lower-right graph in Figure 12.6 (compare with Figure 12.4). In several cases, the movies classified as outliers or as high-leverage points under the new model are different than those classified under the previous model.

12.7 Comparing Two Regression Models

When you are faced with two (or more) regression models on the same data, it is useful to compare details of the models in order to determine which model is better. This section briefly describes techniques you can use to compare regression models. Some of the ideas in this section are implemented in Section 12.8.

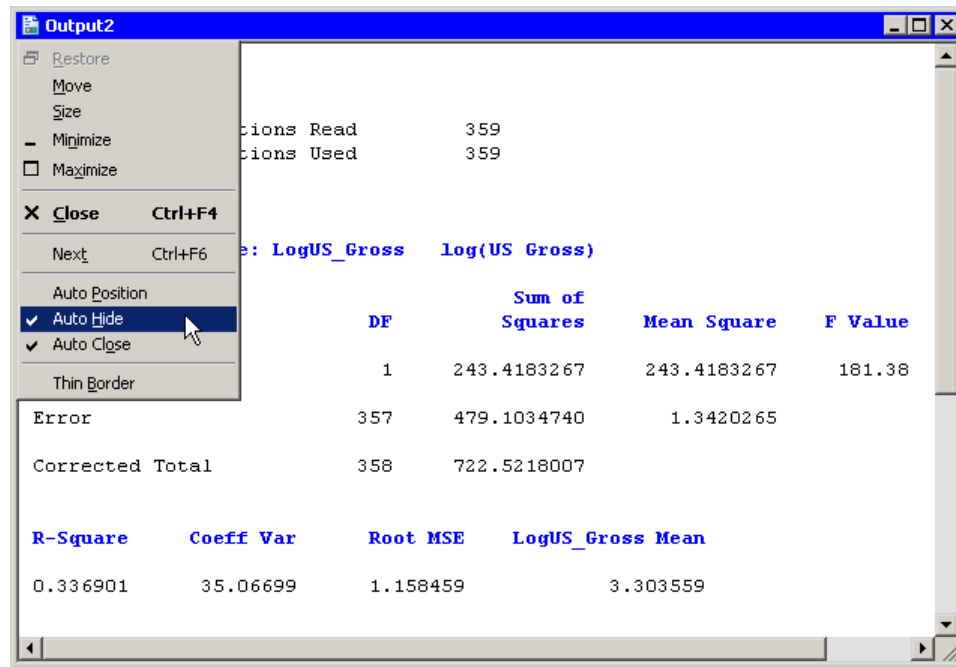
Although this section presents techniques in the context of regression models, most of the ideas in this section apply equally well to other comparisons such as comparing density estimates or comparing multivariate analyses.

12.7.1 Comparing Analyses in Different Workspaces

SAS/IML Studio uses the concept of *workspaces* to help you organize your work. Every time you create a new program or open a new data set, the SAS/IML Studio application creates a new workspace and adds a new button to the workspace bar at the bottom of the main SAS/IML Studio window. You can see the workspace bar at the bottom of [Figure 12.6](#).

When you click a button in the workspace bar, all windows that are associated with the chosen workspace become visible. By default, windows that are associated with other workspaces are hidden. This makes it difficult to compare, for example, the output from two different programs. However, it is easy to get SAS/IML Studio to display the output from two different workspaces at the same time. Assume one of the workspaces is named **A** and another is named **B**. The following steps enable you to compare the output window in Workspace **A** with the output window in Workspace **B**:

1. In Workspace **A**, click on the icon on the left side of the window title bar. A menu appears, as shown in [Figure 12.7](#). (When the output window is active, you can also press ALT+HYPHEN to display this control menu.)
2. Select **Auto Hide** from the menu. This toggles the **Auto Hide** property for the window. When the property is not selected, the window is visible regardless of the active workspace.
3. Position the output window for easy comparison. For example, move it so that it occupies the entire left side of the SAS/IML Studio application.
4. Click **B** on the workspace bar to activate Workspace **B**. Notice that the output window from Workspace **A** is still visible.
5. On the output window for Workspace **B**, display the control menu and select **Auto Hide**. Now the output window for Workspace **B** is visible regardless of the active workspace.
6. Position the output window for Workspace **B** for easy comparison. For example, move it so that it occupies the entire right side of the SAS/IML Studio application.

Figure 12.7 Turning Off the Auto Hide Property

When the **Auto Hide** property is turned off for the output windows, you can scroll both output windows in order to compare parameter estimates, adjusted R square values, and other statistics.

The same technique works for comparing the contents of program windows, data tables, and graphs.

Programming Tip: To compare the results of analyses in different workspaces, turn off the **Auto Hide** property in the windows you want to compare. This enables you to compare output windows, graphs, data tables, and even program statements from two different workspaces.

12.7.2 Comparing Analyses in the Same Workspace

The previous section describes how to compare results that are in different workspaces. However, SAS/IML Studio also enables you to use a single workspace to directly compare two different analyses of the same data.

The main technique to compare two different analyses of the same data is to graph similar quantities against each other. For example, if you are comparing two regression analyses, you can create a scatter plot that graphs the residuals of one regression model against the residuals of the other model. Similarly, you can plot predicted values against each other or, in the case of one regressor, overlay both models' predicted values on a scatter plot of the data. This technique is implemented in [Section 12.8](#).

Programming Tip: To compare the results of two analyses on the same data, create a scatter plot that graphs similar statistics against each other. For example, plot the residuals of one regression model against the residuals of a second regression model.

In order to graph two quantities against each other in SAS/IML Studio, those quantities should be in a common data object. Therefore, you should add predicted values, residual values, and other observation-wise statistics to a single data object. You can use the DATA step to merge the results from several analyses into a single data set, and then create a data object from that merged data set.

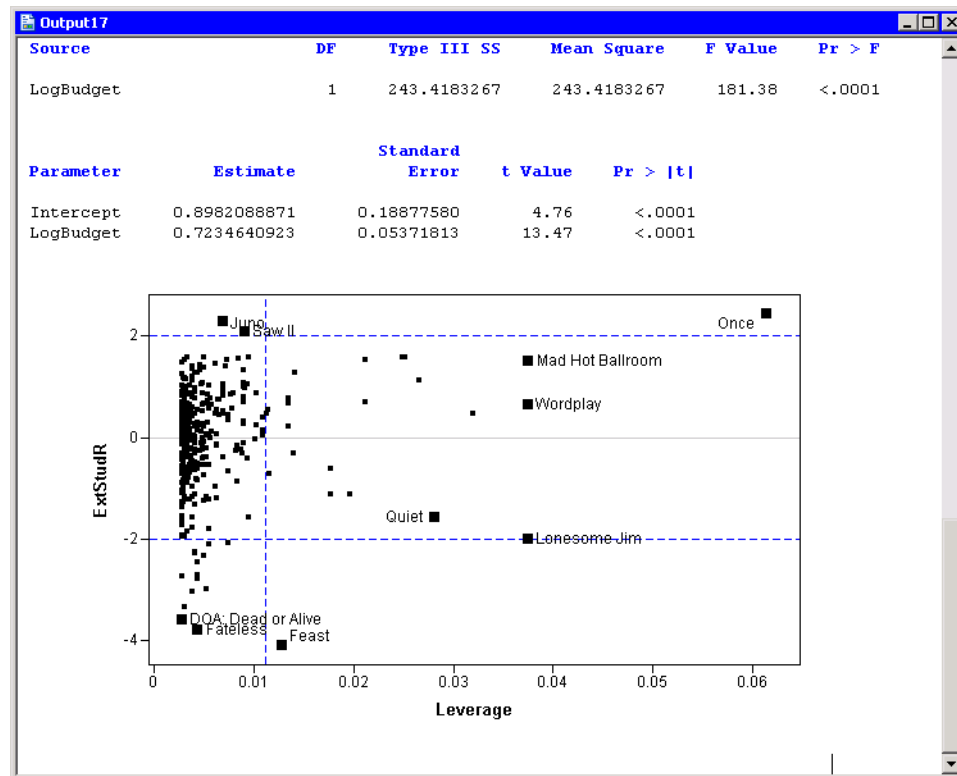
The previous sections describe how to compare the contents of output windows that are in different workspaces. You can also use IMLPlus programming techniques to direct the output from a program to two (or more!) separate output windows in the same workspace. This enables you to compare results (for example, parameter estimates) side-by-side instead of scrolling an output window up and down to compare the output of two procedures. You can use the `OutputDocument` class in SAS/IML Studio to create multiple output windows and to control which window receives the output from an analysis. This technique is implemented and described in [Section 12.8](#).

In addition to sending text to an output window, you can also paste IMLPlus graphics into an output window. The pasted graphic is not dynamically linked; it is merely a static image of an IMLPlus graph. You can copy and paste a graphic manually by using the standard Windows CTRL+C and CTRL+V as described in the section “Copying Plots to the Windows Clipboard” (Chapter 11, *SAS/IML Studio User’s Guide*). You can also write program statements to copy and paste a graph. For example, suppose you want to copy the graph shown in [Figure 12.4](#) and paste it into the output document for the program in [Section 12.2](#). You can accomplish this with the following statement, which continues the program that creates [Figure 12.4](#):

```
resid.CopyToOutputDocument();
```

A static version of the image is pasted into the output window, as shown in [Figure 12.8](#). You can resize the graph in the output window, drag it to a different position within the output window, or copy and paste it to other Windows applications.

Programming Tip: Use the `Plot.CopyToOutputDocument` method to output (static) graphs, especially if you want to easily print multiple graphs as part of the output of some analysis.

Figure 12.8 A Graph Image Pasted into an Output Window

12.8 Case Study: Comparing Least Squares and Robust Regression Models

This section shows how you can compare a robust regression model to the least squares regression model that is described in [Section 12.2](#). This comparison is useful if you know that there are outliers or high-leverage points in the data and you want to investigate how the regression changes if you use robust statistics instead of classical least squares. If the robust regression model is similar to the least squares model, you can confidently use the least squares model, which is faster to compute.

The program in this section consists of the following main steps:

1. Clear and position an output window that will receive output from the classical least squares regression analysis.
2. Run a least squares regression analysis by using the GLM procedure.
3. Create a second output window that will receive output from the robust regression analysis.
4. Run a robust regression on the same data by using the ROBUSTREG procedure.
5. Create a single data object that contains the data and the results from both analyses.

6. Create a scatter plot of the data and overlay regression lines for both models.
7. Create a scatter plot to compare residuals.
8. Create other interactive plots as needed for further analysis.

The program begins by programatically positioning the default output window and by clearing any previous output it contains. This is accomplished by using the `OutputDocument` class that is described in the SAS/IML Studio online Help. The program then proceeds as in [Section 12.2](#) by transforming the data and running the GLM procedure, as shown in the following statements:

```
/* compare regression models */
/* 1. first window is for least squares regression */
declare OutputDocument LSDoc = OutputDocument.GetDefault();
LSDoc.SetWindowPosition(0, 50, 50, 50);
LSDoc.Clear(); /* clear any previous content */

/* 2. transform variables and run least squares regression analysis */
submit;
data movies;
    set Sasuser.Movies;
    LogUS_Gross = log(US_Gross);
    LogBudget = log(Budget);
    label LogUS_Gross = "log(US Gross)" LogBudget = "log(Budget)";
run;

proc glm data=movies;
    model LogUS_Gross = LogBudget;
    output out=GLMOut P=P R=Resid;
quit;
endsubmit;

printnow; /* flush any pending output */
```

The last statement is the `PRINTNOW` statement, which is an IMLPlus statement that ensures that any output buffered on the SAS server is transferred to and displayed on the SAS/IML Studio client. Usually you do not need the `PRINTNOW` statement, since all output is flushed to the client when the program ends. However, the next step in the current program is to create a new output window to receive the output from a robust regression model, so it is a good idea to make sure that all output related to `PROC GLM` is displayed in the first output window prior to creating a second output window.

The following statements create and position a new output window, and run the `ROBUSTREG` procedure for the same model used by the `GLM` procedure:

```
/* 3. set up second output window */
declare OutputDocument RobustDoc = OutputDocument.Create();
RobustDoc.SetWindowPosition(50, 50, 50, 50);

/* 4. run robust regression on the same data */
submit;
proc robustreg data=movies method=LTS;
    model LogUS_Gross = LogBudget / leverage;
```

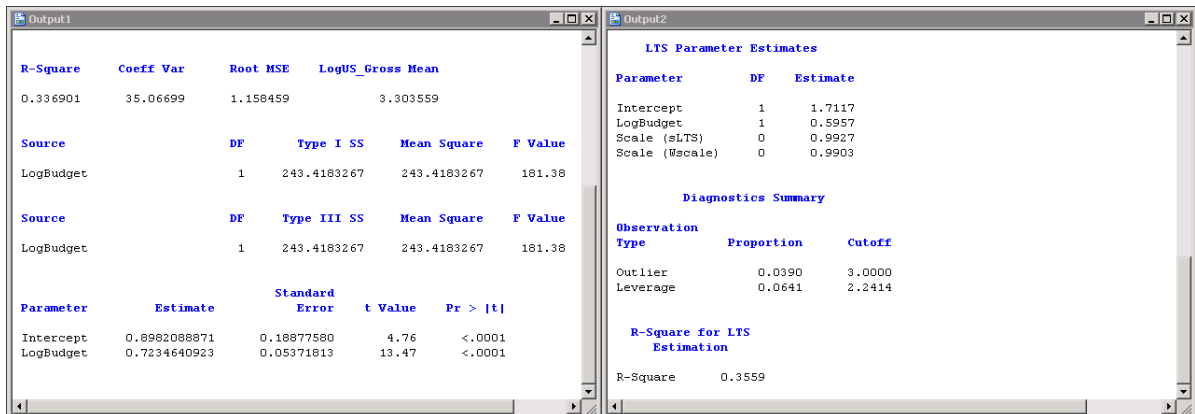
```

output out=RROut P=RobP R=RobResid
      RD=RobDist MD=MahalDist
      outlier=RobOutFlag leverage=RobLevFlag;
ods output DiagSummary = RobustDiagnostics;
quit;
endsubmit;

```

The output from the ROBUSTREG procedure appears in the second output window. This makes it easy to compare parameter estimates and other statistics side-by-side, as shown in [Figure 12.9](#). The statements also save the DiagSummary table, which shows cutoff values for outliers and leverage points

Figure 12.9 Separate Output Windows for Two Analyses



The parameter estimates for each model are also displayed in [Figure 12.10](#). The intercept estimate is smaller for the least squares model (0.90) than for the robust model (1.71), and the slope of the least squares model (0.72) is greater than for the robust model (0.60). This occurs because the least squares model for the logged data is affected by a number of small-budget movies that did not earn much money. This is shown graphically in [Figure 12.11](#).

Figure 12.10 Parameter Estimates from Two Regression Models

The GLM Procedure				
Dependent Variable: LogUS_Gross log(US Gross)				
Parameter	Estimate	Standard Error	t Value	Pr > t
Intercept	0.8982088871	0.18877580	4.76	<.0001
LogBudget	0.7234640923	0.05371813	13.47	<.0001

Figure 12.10 *continued*

The ROBUSTREG Procedure		
LTS Parameter Estimates		
Parameter	DF	Estimate
Intercept	1	1.7117
LogBudget	1	0.5957
Scale (sLTS)	0	0.9927
Scale (Wscale)	0	0.9903

Programming Tip: Use two output windows to easily compare two related analyses side-by-side. You can create an output window by using the `OutputDocument` class in SAS/IML Studio.

This example does not produce any further output. However, you can call `LSDoc.SetDefault()` to direct further output to the first output window and `RobustDoc.SetDefault()` to direct output to the second output window.

The next step in the analysis is to create a single data object that contains the data and the results from both analyses. There are several ways to accomplish this. One way is to use the DATA step (inside of a SUBMIT block) to merge the output from the procedures, and then to create a data object from that merged data. A second way, shown in the following statements and described in [Section 8.8.2](#), is to create a data object from, say, the output data set created by the ROBUSTREG procedure, and then use the `CopyServerDataToDataObject` module to add certain variables from the GLM output to the same data set:

```
/* 5. create data object that contains data and results */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Work.RROut");
dobj.SetRoleVar(ROLE_LABEL, "Title");

rc = CopyServerDataToDataObject("Work", "GLMOut", dobj,
    {"P" "Resid"},
    {"P" "Resid"},
    {"Predicted log(US_Gross)" "Residual"},
    true);
```

The data object contains all of the variables in the Movies data set, the log-transformed variables, and the predicted values, residuals, and statistics from the GLM and ROBUSTREG procedures. You can therefore explore the results interactively by using the SAS/IML Studio GUI, or you can create plots programmatically.

The next step is to create a scatter plot of the data and to overlay regression lines for both models in order to compare them. The following statements use the techniques that are described in [Section 9.1](#) to overlay each regression curve on the data:

```

/* 6. create scatter plot that compares fits */
declare ScatterPlot plot;
plot = ScatterPlot.Create(dobj, "LogBudget", "LogUS_Gross");
plot.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
plot.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);

/* 6a. use drawing subsystem to add fit lines */
dobj.GetVarData({"LogBudget" "P" "RobP"}, fit);
call sort(fit, 1);                                     /* sort by X */
x = fit[,1]; P = fit[,2]; RobP = fit[,3];

plot.DrawUseDataCoordinates();
plot.DrawSetPenAttributes(BLUE, DASHED, 2);
plot.DrawLine(x, RobP);
plot.DrawSetPenAttributes(BLACK, SOLID, 2);
plot.DrawLine(x, P);

/* 6b. add legend */
Labels = {"Robust LTS" "Least Squares"};
Color = BLUE || BLACK;
Style = DASHED || SOLID;
run DrawLegend(plot, Labels, 12, Color, Style, _NULL_, -1, "ILT");

```

The previous statements use a dashed blue line to represent the robust regression model, and a solid black line to represent the least squares model. In order to make the graph easier to understand, the statements also call the DrawLegend module to add a legend to the upper left portion of the plot area, as shown in Figure 12.11. The DrawLegend module is described in Section 9.2.

Figure 12.11 Comparison of Two Regression Models

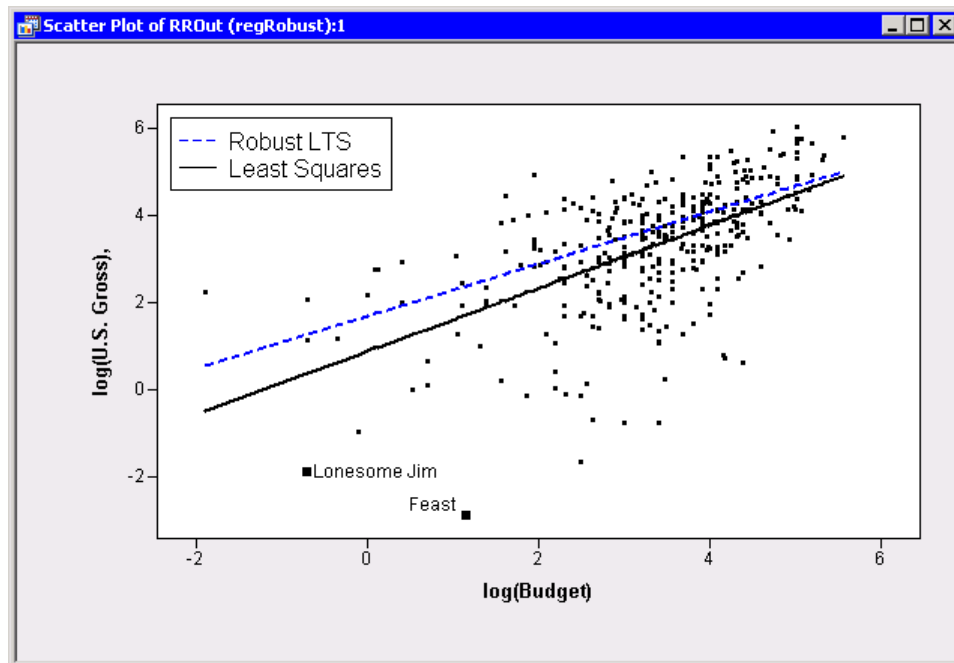


Figure 12.11 makes it apparent that the least squares regression line is “pulled down” by the high-leverage points in the lower left of the figure. The two movies that have the most negative `LogUS_Gross` values are *Lonesome Jim* and *Feast*. Those two movies are among the movies labeled in Figure 12.4; it is evident from Figure 12.4 that they are high-leverage points with large negative residuals for the least square model.

When comparing two different models, it is often useful to compare the residual values for the models. It is sometimes the case that there are observations that have large residuals for one model but have small residuals or negative residuals for the other model. (This happens frequently with nonparametric models.) The following statements create a scatter plot of the residuals for each model and use the `abline` module (see Section 9.4) to add a reference line:

```
/* 7. compare residuals */
declare ScatterPlot residPlot;
residPlot = ScatterPlot.Create(dobj, "Resid", "RobResid");
residPlot.SetMarkerSize(5);
residPlot.ShowReferenceLines();
solidGray = 0xc8c8c8 || SOLID || 1;      /* define line attributes */
run abline(residPlot, 0, 1, solidGray);    /* identity line          */
```

The results are shown in Figure 12.12. All pairs of residuals from these two models lie near the identity line, which indicates that the two models are fairly similar. If a point in Figure 12.12 were far from the identity line, it would indicate an observation for which the predicted value from one model was vastly different than the predicted value from the other model. Notice that all of the points in Figure 12.12 are *below* the identity line. This indicates that every residual value for the least squares model is less than the corresponding residual value for the robust model. This is obvious from Figure 12.11, because the predicted values for the robust model are larger than the predicted values for the least squares model for every value of the explanatory variable.

Figure 12.12 Comparison of Residuals for Each Model

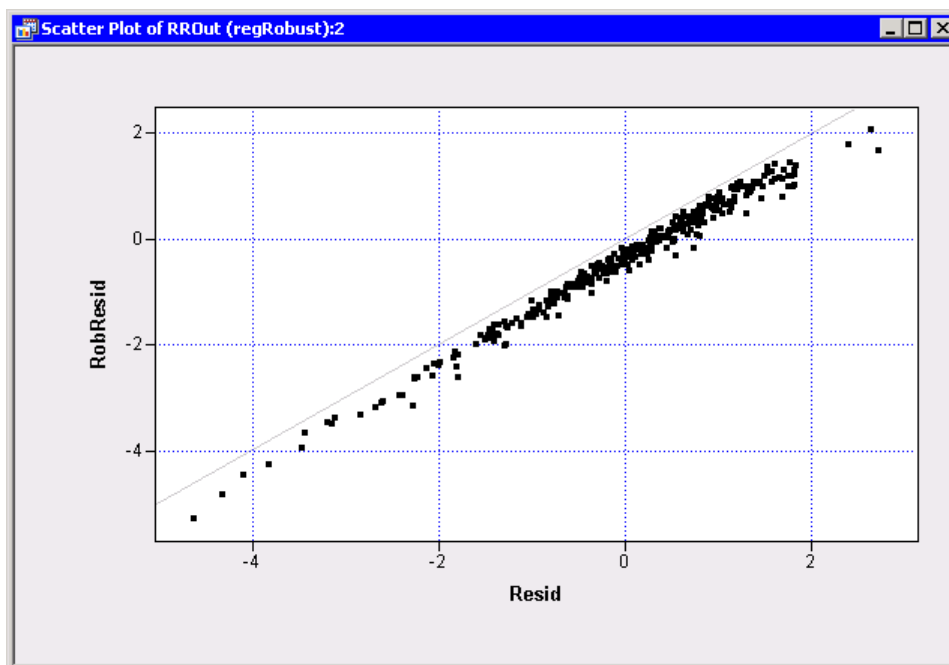


Figure 12.12 compares residual values and enables you to examine whether an observation that is an outlier for one model is also an outlier for the other model. You can do a similar comparison for the leverage statistic displayed in Figure 12.4. The robust analogue to the leverage statistic is the robust distance statistic (created by the RD= option on the ROBUSTREG OUTPUT statement) because both statistics indicated how far an observation is from the center of the data in the space of the explanatory variables. However, the leverage statistic is proportional to a squared distance, so it is desirable to directly compare these quantities. However, recall that the square root of the leverage is proportional to the Mahalanobis distance (created by the MD= option on the ROBUSTREG OUTPUT statement). Therefore, the distance-distance plot shown in Figure 12.13 compares the leverage of the least squares model (measured by the Mahalanobis distance) to the leverage of the robust model (measured by the robust distance). The following statements create Figure 12.13:

```
/* 8. compare leverage */
declare ScatterPlot distPlot;
distPlot = ScatterPlot.Create(dobj, "MahalDist", "RobDist");
distPlot.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
distPlot.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);
distPlot.SetMarkerSize(5);
run abline(distPlot, 0, 1, solidGray);          /* identity line      */
use RobustDiagnostics;
    read point 2 var {Cutoff};                  /* cutoff for RobDist */
close RobustDiagnostics;
run abline(distPlot, Cutoff, 0, solidGray);     /* high-leverage line */
```

Figure 12.13 Comparison of Classical and Robust Leverage

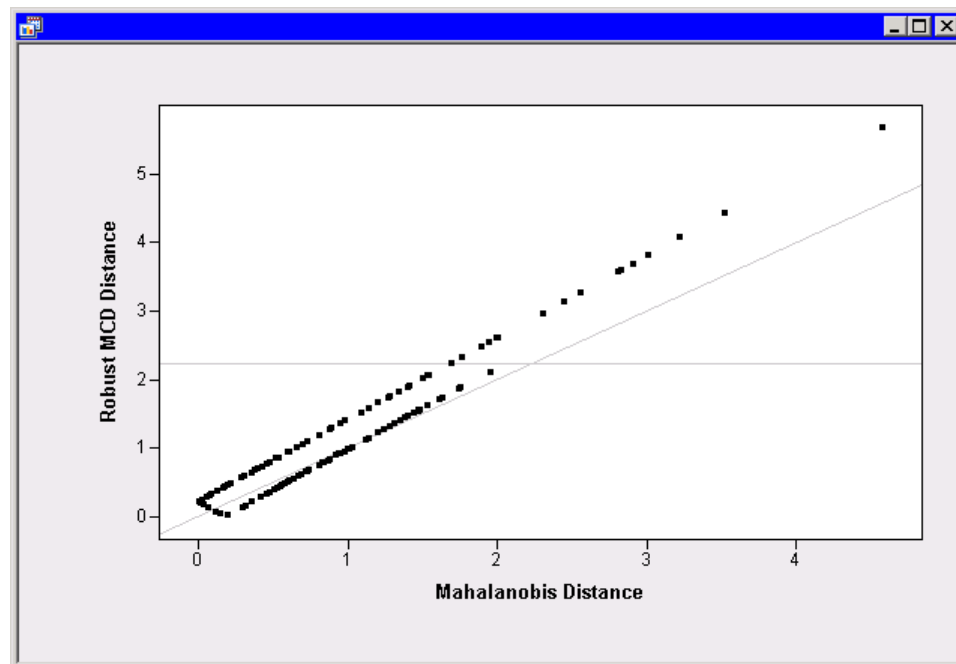


Figure 12.13 includes a horizontal line that indicates the cutoff location for high-leverage points. The value for the cutoff is read from the RobustDiagnostics data set that was created from the DiagSummary ODS table. If you use the mouse to drag out a selection rectangle that selects all observations above the cutoff line, those high-leverage movies are highlighted in all other graphs that were created in this section.

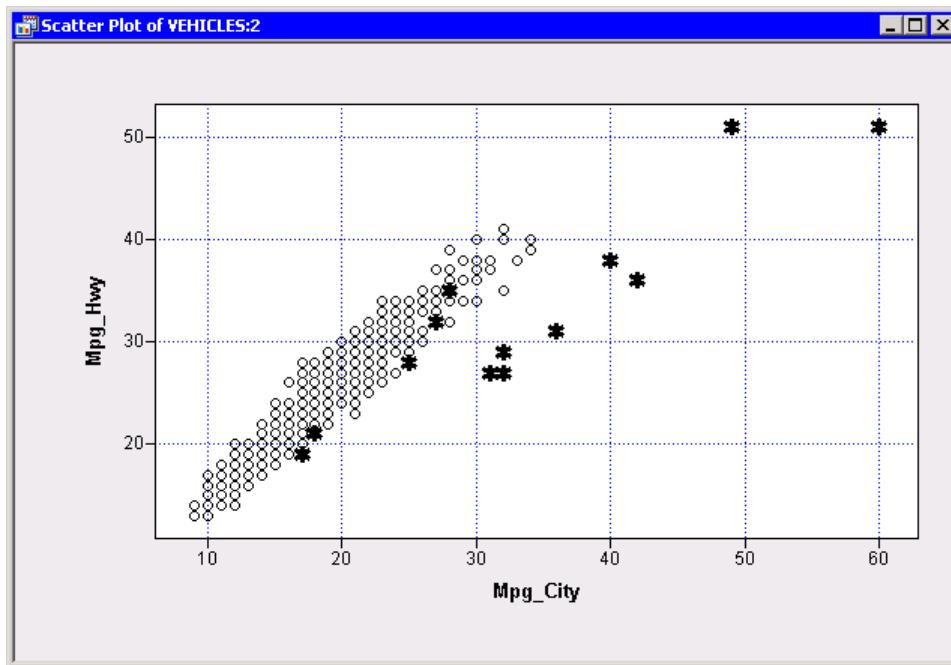
12.9 Logistic Regression Diagnostics

As shown in the previous sections, the dynamically linked graphics in SAS/IML Studio make it easy to identify influential observations. After you create graphs of diagnostic statistics (Cook's D , leverage, and so forth), you can click on the highly influential observations in order to reveal more information about those observations. This technique is not limited to least squares regression. You can also apply the ideas to logistic models, mixed models, nonparametric models, and so forth. This section briefly describes how to create and examine diagnostic statistics for a logistic model.

This section uses the `Vehicles` data set, which contains an indicator variable named `Hybrid`. The `Hybrid` variable has the value 1 for hybrid-electric vehicles and the value 0 for other vehicles.

A distinguishing characteristic of many hybrid-electric vehicles is that they achieve better gas mileage during city driving than on the highway. This is shown in Figure 12.14, where the hybrid-electric vehicles are indicated by stars (*). The graph might lead you to suspect that you can predict whether a vehicle is hybrid-electric based on comparing values of the `Mpg_Hwy` and `Mpg_City` variables.

Figure 12.14 Mileage for Various Vehicles. Stars Represent Hybrid-Electric Vehicles.



A particularly simple model is to consider only the difference between the `Mpg_Hwy` and `Mpg_City` variables. This quantity is contained in the `Mpg_Hwy_Minus_City` variable in the `Vehicles` data set.

The process of fitting a logistic model, displaying the results in SAS/IML Studio, and using linked graphics to identify influential observations is very similar to the process described in the section “Fitting a Regression Model” on page 280. The main steps of the process are as follows:

1. Fit a logistic regression model.
2. Read the results into a data model.
3. Set the observation markers to indicate values of the response variable.
4. Create plots of predicted values, residual values, and/or influence diagnostics.
5. Use dynamically linked graphics to identify outliers and influential observations.

The following statements call the LOGISTIC procedure to build a predictive model of hybrid-electric vehicles that is based on the difference between the mileage on the highway and in the city:

```
/* create logistic regression diagnostic plots */
submit;
/* 1. run logistic analysis */
proc logistic data=Sasuser.Vehicles;
  model Hybrid(event='1') = Mpg_Hwy_Minus_City;
  output out=LogOut P=Prob c=C
          reschi=ResChiSq difchisq=DifChiSq;
quit;

data LogOut;          /* add ObsNumber variable and revise labels */
  set LogOut;
  ObsNumber = _N_;
  label Prob      = "Estimated Probability of Being Hybrid"
        C        = "CI Displacement C"
        DifChiSq  = "Influence on Chi-Square"
        ObsNumber = "Observation Number";

run;
endsubmit;
```

The OUTPUT statement for PROC LOGISTIC creates an output data set that contains all of the variables in the Vehicles data in addition to four variables based on the model:

Prob contains the estimated probability that a vehicle is a hybrid.

C contains an influence diagnostic statistic (called the confidence interval displacement statistic) which is similar to Cook's *D* statistic.

ResChiSq contains the Pearson chi-square residuals.

DifChiSq contains a statistic that measures the effect of deleting an observation on the chi-square statistic.

After using the DATA step to add the ObsNumber variable and to revise the labels of several variables, the next step is to read the data and output statistics into a data object and to set the marker characteristics for the observations. The following statements set the markers for hybrid-electric vehicles to be stars, and the markers for traditional vehicles to be hollow circles:

```

/* 2. create data object that contains data and results */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Work.LogOut");
dobj.SetRoleVar(ROLE_LABEL, "Model");

/* 3. set observation markers to encode Hybrid=1 */
use LogOut; read all var {Hybrid}; close LogOut;
dobj.SetMarkerShape(loc(Hybrid=0), MARKER_CIRCLE);
dobj.SetMarkerFillColor(loc(Hybrid=0), NOCOLOR);
dobj.SetMarkerShape(loc(Hybrid=1), MARKER_STAR);

```

The markers are used for all plots of the data.

Programming Tip: If your data contain a binary response variable, it is often useful to set marker shapes that correspond to the levels of the response variable.

The following statements create a graph that shows the estimated probability of being a hybrid, based on a model that uses only the Mpg_Hwy_Minus_City variable:

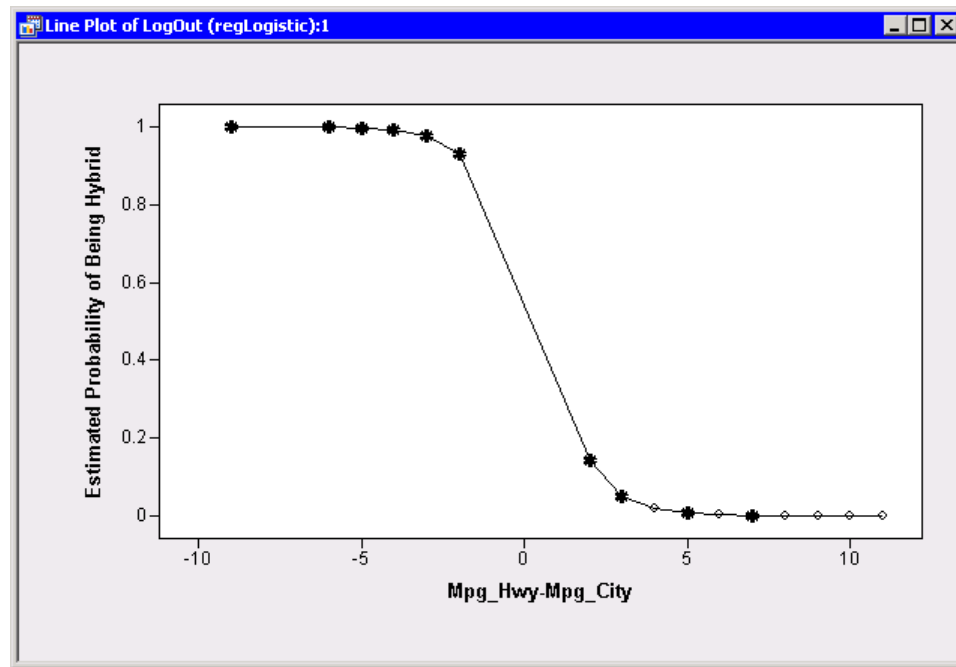
```

/* 4. create line plot of estimated probability */
declare LinePlot line;
line = LinePlot.Create(dobj, "Mpg_Hwy_Minus_City", "Prob");
line.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
line.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);
line.SetMarkerSize(5);

```

The graph is shown in [Figure 12.15](#). Of the 19 hybrid-electric vehicles in the data, the model assigns a high probability of being a hybrid to the 10 that have negative values for the Mpg_Hwy_Minus_City variable. The 1,177 vehicles that have positive values for the Mpg_Hwy_Minus_City variable are assigned a low probability of being a hybrid. Nine of those vehicles are, in fact, hybrid-electric. Consequently, these misclassified vehicles have relatively large Pearson chi-square residuals.

Figure 12.15 Plot of Estimated Probability

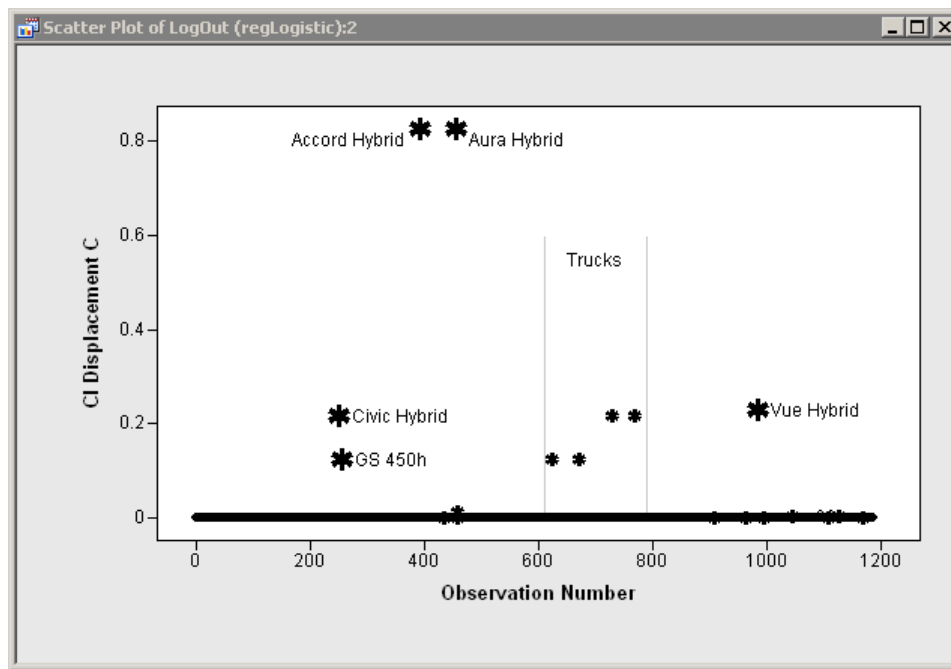


You can create many influence diagnostics plots, as described in the documentation for the LOGISTIC procedure. The following statements create two plots. One is a graph of the confidence interval displacement statistic (see Figure 12.16), whereas the other (not shown) is a plot that shows the effect of deleting an observation on the Pearson chi-square statistic.

```
/* 5. create influence diagnostic plot */
declare ScatterPlot influencePlot;
influencePlot = ScatterPlot.Create(dobj, "ObsNumber", "C");
influencePlot.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
influencePlot.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);
influencePlot.SetMarkerSize(5);

/* 6. create other diagnostic plots */
declare ScatterPlot difChiSqPlot;
difChiSqPlot = ScatterPlot.Create(dobj, "Prob", "DifChiSq");
difChiSqPlot.SetAxisLabel(XAXIS, AXISLABEL_VARLABEL);
difChiSqPlot.SetAxisLabel(YAXIS, AXISLABEL_VARLABEL);
difChiSqPlot.SetMarkerSize(5);
```

Figure 12.16 shows nine observations with large values of the C statistic. Five of those observations were selected by clicking on them. By using the interactive graphs in SAS/IML Studio, you can come to a better understanding of why certain vehicles are outliers for the models. For example, in the Vehicles data set, observations numbered 610–790 are pickup trucks, and four pickup trucks are hybrid-electric. These four trucks are shown in Figure 12.16 as vehicles with large C values, which indicates that the simple logistic model developed in this section is not adequate to distinguish hybrid-electric pickup trucks from the other pickup trucks. If correctly classifying hybrid-electric pickup trucks is important, then you need to revise the logistic model.

Figure 12.16 Change in the Deviance Resulting from Deleting an Individual Observation

12.10 Viewing ODS Statistical Graphics

Many of the SAS statistical procedures can produce their own diagnostic plots by using ODS statistical graphics. These plots are not dynamically linked to each other, but still can help you to analyze the fit of a statistical model.

The output window in SAS/IML Studio displays the listing destination for ODS tables. As of SAS/IML Studio 3.3, there is not a way to display HTML output from SAS procedures within SAS/IML Studio. However, *if SAS/IML Studio and SAS Foundation are both running on the same PC*, you can easily write the HTML output to a file, and then use a Web browser to view the tables and graphs.

The following program demonstrates this approach. Suppose you want to create and view the ODS graphics that are produced by the REG procedure. One way to do this is to specify a Windows path and file name (say, *Output.html*) for the ODS output on the ODS HTML statement. You can then run one or more procedures and view the HTML output (including any graphs that you have requested) by opening the *Output.html* file in a Web browser. You can use the Windows operating system to navigate to the file and open it, or you can programmatically launch a browser to view the file, as shown in the following example:

```
/* write ODS output (including graphics) to HTML */
run GetPersonalFilesDirectory(path);
htmlfile = "Output.html";
```

/* 1 */

```

submit path htmlfile;                                /* 2 */
ods graphics on;
ods html body="&htmlfile" path="&path";              /* 3 */

proc reg data=Sasuser.Vehicles;                       /* 4 */
    model MPG_Hwy = MPG_City;
quit;

ods html close;                                       /* 5 */
ods graphics off;
endsubmit;

/* launch browser to display HTML */
pgm = "C:\Program Files\Internet Explorer\iexplore.exe";
ok = ExecuteOSProgram(pgm, path+htmlfile, false);    /* 6 */

```

The program contains the following main steps:

1. Define a directory in which to store the HTML and image files. You must have write permission to this directory. One portable way to get such a directory is to use the GetPersonalFilesDirectory module, which is distributed with SAS/IML Studio. By default, the personal files directory corresponds to one of the following Windows directories:

Windows XP	C:\Documents and Settings\userid\My Documents\My IML Studio Files
Windows Vista	C:\Users\userid\Documents\My IML Studio Files
Windows 7	C:\Users\userid\Documents\My IML Studio Files
2. Use the SUBMIT statement to set ODS options and to call PROC REG. The Windows path and the name of the HTML file are passed into the SUBMIT block as parameters.
3. Use the ODS HTML statement to specify that PROC REG should write output to the ODS destination. The PATH= option specifies the directory in which to store the HTML file and image files. The BODY= option specifies the name of the HTML file that you can view with a browser in order to display the output.
4. Call PROC REG. Because this program specifies the ODS GRAPHICS ON statement, PROC REG creates ODS graphics as part of its output.
5. The HTML output is written when the ODS HTML CLOSE statement is executed.
6. Use the ExecuteOSProgram module (which is distributed with SAS/IML Studio) to launch a browser to view the HTML output. The first argument to the module is the name of the program to execute. The second argument contains command-line arguments to the program. In this case, Internet Explorer needs to know the name and location of the *Output.html* file. The third argument specifies whether or not SAS/IML Studio should pause until Internet Explorer exits. By using **false** for this argument, the IMLPlus program continues to execute while Internet Explorer is running.

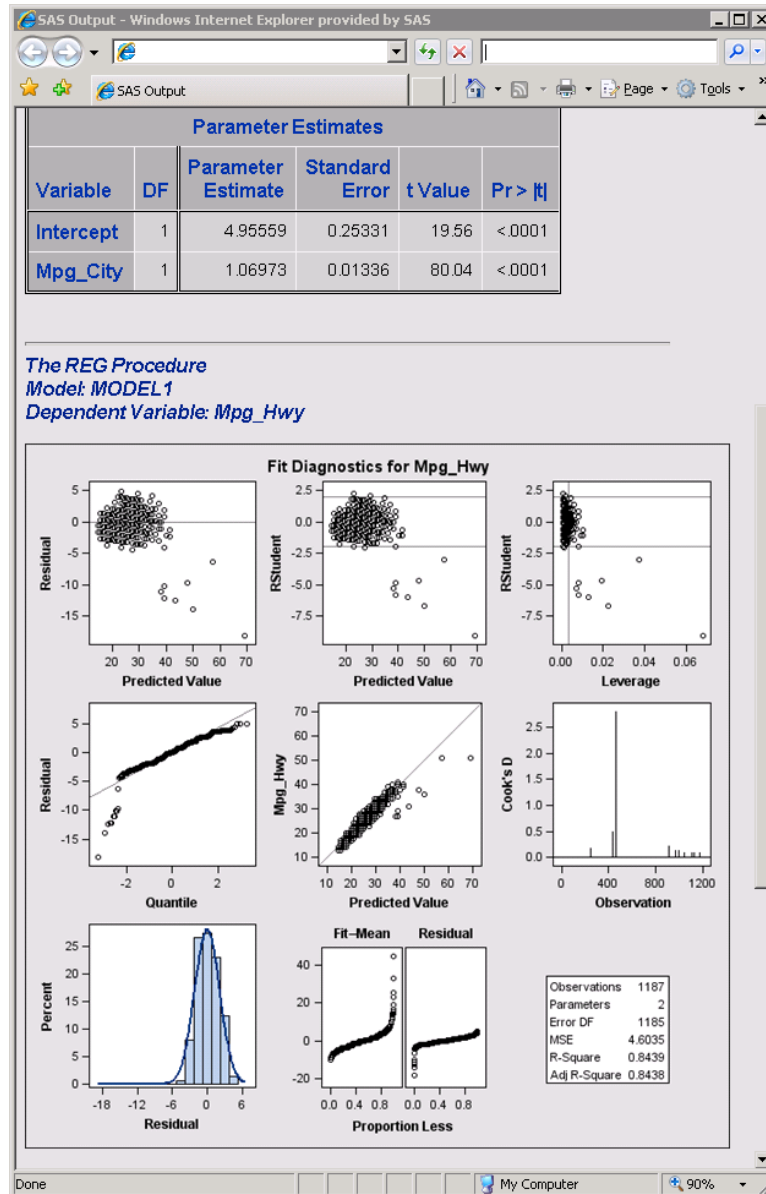
The output from the program is shown in [Figure 12.17](#). You can modify the program to use a different browser, or to store the HTML and image files in a different location. After running the

program, the HTML and image files are not automatically deleted, so you should manually delete those files when you are finished viewing them.

If you have configured SAS/IML Studio to communicate with a SAS Workspace Server that is running on a remote computer, then there is no simple way to display ODS graphics on the client PC.

For more information on using ODS statistical graphics, see the chapter “Statistical Graphics Using ODS” in the *SAS/STAT User’s Guide*.

Figure 12.17 ODS Graphics from SAS Procedures



12.11 References

- Atkinson, A. C. (1985), *Plots, Transformations, and Regression*, New York: Oxford University Press.
- Belsley, D. A., Kuh, E., and Welsch, R. E. (1980), *Regression Diagnostics*, New York: John Wiley & Sons.
- Blom, G. (1958), *Statistical Estimates and Transformed Beta Variables*, New York: John Wiley & Sons.
- Nelson, R. A., Donihue, M. R., Waldman, D. M., and Wheaton, C. (2001), “What’s an Oscar Worth?” *Economic Inquiry*, 39(1), 15–24.
URL <http://ssrn.com/abstract=253082>
- Rawlings, J. O., Pantula, S. G., and Dickey, D. A. (1998), *Applied Regression Analysis: A Research Tool*, Springer Texts in Statistics, Second Edition, New York: Springer-Verlag.
- Simonoff, J. S. and Sparrow, I. R. (2000), “Predicting Movie Grosses: Winners and Losers, Blockbusters and Sleepers,” *CHANCE*, 13(3), 15–24.

Chapter 13

Sampling and Simulation

Contents

13.1	Overview of Sampling and Simulation	311
13.2	Simulate Tossing a Coin	312
13.3	Simulate a Coin-Tossing Game	314
13.3.1	Distribution of Outcomes	314
13.3.2	Compute Statistics for the Simulation	316
13.3.3	Efficiency of the Simulation	319
13.4	Simulate Rolling Dice	321
13.5	Simulate a Game of Craps	322
13.5.1	A First Approach	323
13.5.2	A More Efficient Approach	324
13.6	Random Sampling with Unequal Probability	327
13.7	A Module for Sampling with Replacement	329
13.8	The Birthday Matching Problem	332
13.8.1	A Probability-Based Solution for a Simplified Problem	332
13.8.2	Simulate the Birthday Matching Problem	335
13.9	Case Study: The Birthday Matching Problem for Real Data	338
13.9.1	An Analysis of US Births in 2002	338
13.9.2	The Birthday Problem for People Born in 2002	340
13.9.3	The Matching Birth Day-of-the-Week Problem	340
13.9.4	The 2002 Matching Birthday Problem	344
13.10	Calling C Functions from SAS/IML Studio	346
13.11	References	347

13.1 Overview of Sampling and Simulation

This chapter describes how to use the techniques in this book to generate random samples from a specified distribution. In particular, the chapter describes how to use SAS/IML software to generate random samples from a set of outcomes with known probabilities.

Previous chapters used the RANDGEN subroutine to sample from continuous distributions such as the normal or uniform distribution. This chapter samples from discrete distributions. In particular,

assume that some event can have k outcomes: outcome X_1 occurs with probability p_1 , outcome X_2 occurs with probability p_2 , and so on, where $\sum_{i=1}^k p_i = 1$.

For example, the outcomes of tossing a coin can be “Heads” or “Tails,” with associated probabilities $p_1 = p_2 = 0.5$. This is known as sampling with equal probability. Alternatively, you could have a jar that contains marbles of various colors. The probability of pulling a particular color is equal to the proportion of marbles of that color. This is known as sampling with unequal probabilities.

A *random sample* is a sample of n elements that is chosen randomly from a population according to some scheme. If a given element can be selected more than once, the scheme is known as random sampling with replacement; otherwise, the scheme is known as random sampling without replacement. The term “with replacement” means that drawing a particular element does not change the probability of choosing that element for the next draw. A classic example is pulling colored marbles from a jar. When sampling with replacement, each marble that you pull from the jar is examined, the color is recorded, the marble is replaced in the jar, and the jar is well shaken prior to pulling another marble. The examples in this chapter all use sampling with replacement. A widely used text on survey sampling is Lohr (2009), which includes a section on simple random sampling and an appendix on probability concepts that are used in sampling.

A simulation is a model of a real-world outcome that involves randomness. Some simulations presented in this chapter include the following:

- simulating a game that involves tossing a coin
- simulating the game of craps, which involves rolling a pair of dice
- simulating selecting items when some items have a higher probability of being selected than other items
- simulating the probability of two people in a room having the same birthday

13.2 Simulate Tossing a Coin

The simplest game of chance is the coin toss. For a fair coin, the probability of a flipped coin landing on a given side (“heads”) is 0.5. You can simulate a coin toss by generating a pseudorandom number from the uniform distribution on the open interval $(0, 1)$. You can classify the number as “heads” if it is in the range $[0.5, 1)$ and as “tails” otherwise. Alternatively, you can use the CEIL or FLOOR function to produce integers that represent heads and tails, as shown in the following statements:

```
/* simulate a coin toss */
call randseed(1234);
N = 10;
c = j(N, 1);
call randgen(c, "uniform");
c = floor(2*c);
print c;

/* set random number seed */
/* allocate Nx1 vector */
/* fill with values in (0,1) */
/* convert to integers 0 or 1 */
```

Figure 13.1 Simulated Coin Flips

c
1
1
0
0
1
0
0
1
0
1

Notice that no loop is used to generate the pseudorandom numbers: a vector is obtained by setting the size of the vector `c` and then calling the `RANDGEN` subroutine a single time. The results of the simulation are shown in [Figure 13.1](#). Heads are represented by ones and tails by zeros.

Programming Technique: Do not use a loop to generate random values. Instead, allocate a vector of the desired size and call the `RANDGEN` subroutine.

It is often useful to encapsulate a simulated event into a module, as shown in the following statements:

```
/* Module to simulate the flip of a fair coin.
 * Call the RANDSEED function before calling this module.
 * The module returns a vector of ones (heads) and zeros (tails).
 */
start TossCoin(N);
    c = j(N, 1);                /* allocate Nx1 vector      */
    call randgen(c, "uniform"); /* fill with values in (0,1) */
    c = floor(2*c);             /* convert to integers 0 or 1 */
    return ( c );
finish;
```

The module can be used to simulate the event while hiding the specifics of the simulation. For example, the following statements simulate tossing a coin 100 times and count the total number of heads and tails:

```
/* simulate many coin tosses */
call randseed(1234);          /* set random number seed   */
N = 100;
g = TossCoin(N);              /* flip a coin N times      */
NumHeads = sum(g);            /* count the number of heads */
NumTails = N - NumHeads;      /* the other tosses are tails */
PropHeads = g[:];             /* proportion of heads       */
print NumHeads NumTails PropHeads[format=percent8.3];
```

Figure 13.2 Simulated Coin Tosses

NumHeads	NumTails	PropHeads
51	49	51.00%

As expected, roughly half of the simulated coin tosses result in heads.

Programming Tip: A statistic computed from simulated outcomes is an estimate for a corresponding probability.

13.3 Simulate a Coin-Tossing Game

In a game of chance, you can sometimes use probability theory to compute the expected gain (or loss) for playing the game. Sometimes the strategy you use and the choices you make while playing affect the game's outcome; other times the outcome is completely determined by the rules of the game. Even if you cannot calculate the probabilities exactly, you can estimate the expected gain by simulating the game many times and averaging the gains and losses of the simulated games.

This section analyzes a coin game with the following rules:

1. Toss a fair coin.
2. If the coin comes up heads, toss again.
3. If the coin comes up tails, the game is over.

In a multiplayer version of this game, the player with the longest string of tosses wins. The next section simulates the game. Subsequent sections estimate the expected gain for playing the game when there are certain costs and rewards that are associated with each outcome.

13.3.1 Distribution of Outcomes

The simplest way to analyze this game is to determine the probability for each outcome, as shown in Table 13.1. In general, the probability of the game ending after k tosses is 2^{-k} , $k = 1, 2, \dots$

Table 13.1 Outcomes and Associated Probabilities for a Coin Game

Outcome	Tosses	Probability
T	1	1/2
HT	2	1/4
HHT	3	1/8
HHHT	4	1/16
\vdots	\vdots	\vdots
H...HT	k	$1/2^k$

A simulation of this game is straightforward and uses the TossCoin module developed in the previous section.

```

/* Simulate a game: flip a coin until tails appears.
   Keep count of the number of tosses.
   First approach: not optimally efficient. */
call randseed(1234);                /* set random number seed */
NumGames = 1e5;                      /* repeat the game many times */
Results = j(NumGames, 1, .);         /* number of tosses per game */
do i = 1 to NumGames;
    count = 1;                        /* number of tosses so far */
    toss = TossCoin(1);               /* toss the coin */
    do while(toss=1);                 /* continue until tails appears */
        count = count + 1;
        toss = TossCoin(1);
    end;
    Results[i] = count;                /* record the results */
end;

```

The program allocates the **Results** vector prior to the main loop. This vector holds the results of the simulation. The i th element of the vector contains the number of tosses for the i th game. If the simulation is accurate, then the distribution of simulated tosses should approximate the distribution shown in Table 13.1.

Programming Technique: You can program a simple simulation with the following pseudocode:

```

NumSims = 1e5;                        /* number of simulations */
Results = j(NumSims, 1, .);           /* allocate results vector */
do i = 1 to NumSims;
    /* simulate */
    /* save results of i_th simulation in Results[i] */
end;

```

The simulation uses classic language features and functions of SAS/IML software. One way to visualize the distribution of outcomes is to use the following IMLPlus statements to create a bar chart:

```

/* visualize the distribution of simulated outcomes */
declare BarChart bar;
bar = BarChart.Create("coin", Results);
bar.SetOtherThreshold(0);
bar.ShowPercentage();
bar.ShowBarLabels();

```

Figure 13.3 Outcomes of Simulated Coin Flips

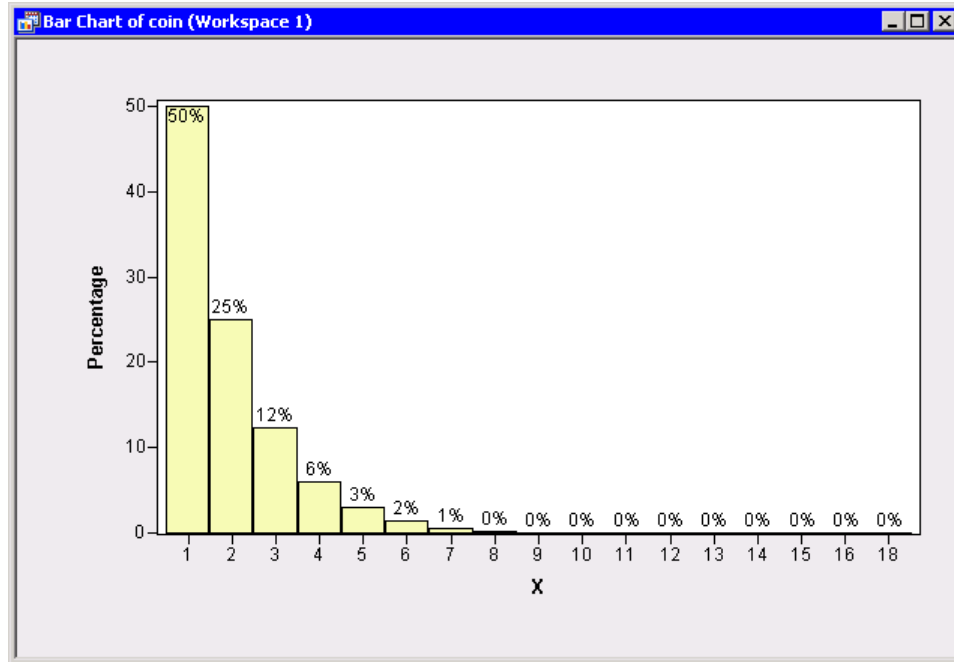


Figure 13.3 shows that the distribution of simulated outcomes for the coin game is very close to the theoretical distribution shown in Table 13.1.

13.3.2 Compute Statistics for the Simulation

The previous section simulated many possible outcomes for the coin-tossing game. If the simulation is correct, a statistic computed from the simulated outcomes is an estimate for a corresponding probability.

13.3.2.1 What is the expected number of coin tosses in a game?

If you play the coin game, what is the expected number of coin tosses in a game?

For this problem it is possible to use elementary probability theory to compute the expected value. As stated earlier, the probability of a game that consists of exactly k flips is 2^{-k} . Therefore, the expected value for the number of flips is $\sum_{k=1}^{\infty} k2^{-k}$. You can use techniques from calculus to show that this series converges to the value 2.

For readers who prefer computations more than calculus, the following statements estimate the expected number of coin tosses per game by numerically summing a finite number of terms:

```
/* compute expected number of tosses per game: theoretical result */
flips = 1:50;                /* approximate series by first 50 terms */
prob = 2##(-flips);          /* probability for each game */
e = sum(flips#prob);          /* expected number of coin flips */
print e;
```

Figure 13.4 Numerical Summation of Probabilities

```
e
2
```

For this game, it is not hard to show that the expected number of coin tosses for a game is 2. However, in more complicated scenarios, it might not be possible to compute the probability exactly. If that is the case, you can estimate the expected value by computing the mean of the (simulated) number of tosses for a large number of games. The following statements compute the average number of tosses per game in the simulated data:

```
/* calculate average number of tosses per game in simulated data */
AvgFlipsPerGame = Results[:];          /* mean over all games */
print AvgFlipsPerGame;
```

Figure 13.5 Average Number of Flips per Game

```
AvgFlipsPerGame
2.00075
```

The results, shown in [Figure 13.5](#), indicate that the average number of coin tosses in the simulated data is about 2, which is in close agreement with the theoretical value.

Notice that the average number of coin tosses in the simulated data depends on the random number seed and on the number of simulated games. If you rerun the simulation with different seeds (but still use 100,000 games for each simulation), the mean number of coin tosses will vary. Perhaps one simulation produces 2.006 for the mean, another produces 1.992, and yet another produces 1.998. You can imagine that if you rerun the simulation many times, you might produce a set of means that are distributed as shown in [Figure 13.6](#). [Chapter 14](#) shows how you can use this distribution of statistics to obtain a standard error for the mean statistic and confidence intervals for the mean.

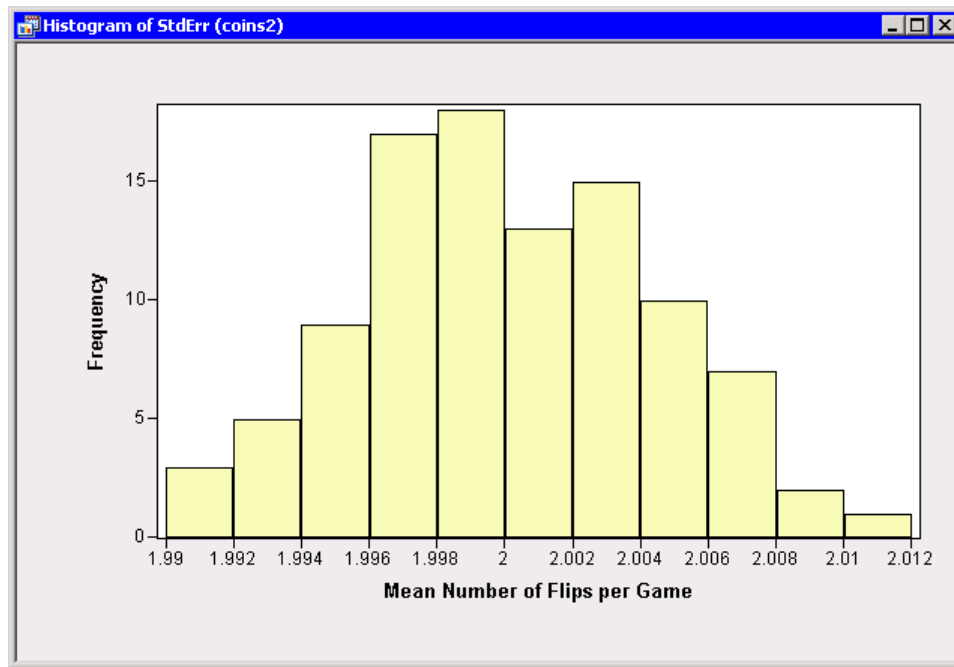
Figure 13.6 Mean Number of Coin Tosses for 100 Simulations

Figure 13.6 shows that a simulation that uses 100,000 games results in a mean estimate that does not vary very much. Of course, if you use a smaller number of games for each simulation, you would expect greater variation in the statistic.

13.3.2.2 What is the expected gain (loss) for playing the game?

For many games of chance, it is interesting to calculate the expected gain (or loss) for the game, given a particular cost to play and a reward for certain outcomes.

Suppose a huckster approaches you on a street corner, describes the coin game, and offers you the following proposition:

- It costs one dollar to play the game.
- You keep tossing until the coin comes up on the tails side.
 - If the coin shows tails on the first or second toss, you lose.
 - If your toss shows heads two or more times before it shows tails, you win a dollar for each head.

Should you accept the bet? If the game outcome is T or HT, you lose a dollar. If the outcome is HHT, you get two dollars—but only one dollar of that is a gain, since it costs a dollar to play. The outcome HHHT results in a net gain of two dollars, and so on, as shown in Table 13.2.

Table 13.2 Gain or Loss for Each Outcome

Outcome	Probability	Gain
T	1/2	-1
HT	1/4	-1
HHT	1/8	1
HHHT	1/16	2
⋮	⋮	⋮

You can compute the expected gain (loss) by using the simulated data. It is convenient to examine the number of heads in each game, which is simply one less than the number of tosses. You can use the `CHOOSE` function (see [Section 3.3.3](#)) to compute the gain of each game in the simulation. After you have computed the gain for each simulated outcome, you can compute the mean of these quantities, which is an estimate for the expected gain. The following statements perform these computations:

```
/* compute gain = payout for game - cost to play */
NumHeads = Results - 1;           /* each game ends with tails */
Gain = choose(NumHeads<2, 0, NumHeads)-1; /* proposed gain/loss */
AvgGain = Gain[:];               /* average gain for many games */
print AvgGain;
```

Figure 13.7 Evaluation of a Betting Scheme on Simulated Games

AvgGain
-0.25038

The `CHOOSE` function enables you to return a vector of results based on the value of each element of the `NumHeads` vector. For each value less than 2, the `CHOOSE` function returns zero. For each value greater than or equal to 2, it returns the corresponding element of the `NumHeads` vector. Consequently, the `CHOOSE` function returns the amount of money that the huckster pays you for each simulated game. Since it costs a dollar to play the game, that amount is subtracted from the payout. Therefore, the vector `Gain` contains the gain or loss for each simulated game. The average gain is shown in [Figure 13.7](#). For this betting scheme, the expected loss is twenty-five cents per game. You would be foolish to accept the bet!

13.3.3 Efficiency of the Simulation

As a general rule, it is better to call a function one time to compute a vector of values, as compared with calling the same function many times. This rule also holds for generating random numbers.

Programming Tip: It is usually more efficient to use a single call to generate and store a large number of pseudorandom numbers, as compared to making many calls that each generate a few pseudorandom numbers.

An alternative to the algorithm in the previous section is to simulate many coin tosses, store those results in a vector, and then analyze the tosses to determine the outcomes of the games. This is the approach used in this section. With this approach, you call the `TossCoin` module with an argument that is large enough to complete a desired number of games. (Knowing how many coin tosses are needed might be difficult.) The following statements use this approach:

```
/* Simulate a game: flip a coin until tails appears.
   Keep count of the number of tosses.
   Second approach: more efficient. */
call randseed(1234);          /* set random number seed      */
NumGames = 1e5;
N = int(2.5 * NumGames);      /* 1. Determine number of rolls */
toss = TossCoin(N);           /* 2. generate all random numbers */
Results = j(NumGames, 1, .);  /* 3. Allocate results vector   */
tossIdx = 1;
do i = 1 to NumGames;
    count = 1;                /* number of tosses so far      */
    do while(toss[tossIdx]=1); /* continue until tails         */
        count = count + 1;
        tossIdx = tossIdx + 1; /* 4. Move to next toss         */
    end;
    Results[i] = count;        /* 5. Record the results        */
    tossIdx = tossIdx + 1;
end;
```

The algorithm in this section has a few differences from the previous algorithm:

1. The purpose of the program is to simulate a certain number of games, but how can you determine how many tosses you will need for the simulation? The previous section showed that the average number of tosses per games is 2. To be on the safe side, increase this number to 2.5 (or more if you are ultra-conservative or intend to simulate a small number of games) and estimate that `2.5*NumGames` tosses are sufficient to simulate the requested number of games.
2. Generate all random numbers with a single call. The `toss` vector is a long vector of zeros and ones.
3. Allocate a vector to hold the results (the number of heads for each game). The allocation is outside of the main loop.
4. Use the integer `tossIdx` to keep track of the current toss. The integer is an index into the `toss` vector that contains the simulated coin tosses. Every time the program needs a new toss, it increments `tossIdx`.
5. Every time that a game is determined, the number of heads for that game is recorded. At the end of the simulation, the `Results` vector contains the number of tosses for each game.

The algorithm is only slightly more complicated than the algorithm in the previous section, but the increased efficiency results in the revised program being about 40% faster than the previous approach. This is quite an improvement for a slight reorganization of the statements.

Programming Tip: The straightforward way to implement a simulation might not be the most efficient. You might need to reorganize the algorithm to obtain better performance.

13.4 Simulate Rolling Dice

You can use the ideas in the previous section to simulate rolling a pair of six-sided dice. The main idea is simple: generate random uniform variates and transform those values into integers in the range one through six. The following statements present a module that uses the RANDGEN subroutine to simulate rolling a pair of dice a specified number of times:

```
/* Module to roll two six-sided dice N times.
 * Call the RANDSEED function before calling this module.
 * The return value is the sum of the values of the roll.
 * This module calls the uniform distribution.
 */
start RollDice(N);
    d = j(N, 2);                /* allocate Nx2 vector (2 dice) */
    call randgen(d, "uniform"); /* fill with values in (0,1) */
    d = ceil(6*d);              /* convert to integers in 1-6 */
    return ( d[,+] );          /* values of the N rolls */
finish;
```

The RollDice module takes a single argument: the number of dice rolls to simulate. The RANDGEN subroutine creates an $N \times 2$ matrix of values in the range (0, 1), so **d** consists of pairs of integers with the values 1–6. The module returns the sum of the values showed on each roll.

Notice that the vector returned by the RollDice module has **N** elements. The following statements simulate the results of rolling dice several times:

```
/* simulate rolling a pair of six-sided dice */
call randseed(54321);
rolls = RollDice(10);
print rolls;
```

Figure 13.8 Simulated Rolls for a Pair of Six-Sided Dice

rolls
7
7
8
6
7
3
5
7
11
8

Actually, there is an even more efficient way to simulate these dice rolls. The RANDGEN subroutine supports a “Table” distribution that enables you to specify a table of probabilities for each of k outcomes. In this case, $k = 6$ and the probability for each outcome is $1/6$. The following statements use the RANDGEN function to sample from six outcomes with equal probability:

```

/* Module to roll two six-sided dice N times.
 * Call the RANDSEED function before calling this module.
 * The return value is the sum of the values of the roll.
 * This module calls the Table distribution with equal probability.
 */
start RollDice(N);
    d = j(N, 2);                /* allocate Nx2 vector          */
    prob = j(6, 1, 1)/6;        /* equal prob. for six outcomes */
    call randgen(d, "Table", prob); /* fill with integers in 1-6    */
    return ( d[,+] );           /* values of the N rolls        */
finish;

```

This second definition of the RollDice module is about 40% faster than the first definition.

Programming Tip: Use the RANDGEN subroutine with the “Table” distribution to sample from a discrete set of outcomes with a given set of probabilities.

13.5 Simulate a Game of Craps

If you can simulate the rolling of dice, you can simulate games that use dice. A well-known dice game is the gambling game of craps. This section presents a program that simulates the game of craps and analyzes the results.

The rules of craps are straightforward: a player (called the “shooter”) rolls two six-sided dice and adds their values.

- If the first roll sums to 7 or 11, the shooter wins. (This is called a “natural.”)
- If the first roll sums to 2, 3, or 12, the shooter loses. (This is called “craps.”)
- If the first roll sums to anything else (4, 5, 6, 8, 9, or 10), that sum becomes the player’s “point.” The shooter continues rolling the dice until he does either of the following:
 - “makes his point” (that is, rolls the point again) and wins, or
 - rolls a 7 and loses.

As noted in the previous section, there are two approaches to simulate the game. The first is to loop over a large number of games, rolling (simulated) dice for each game until the game is decided. This mimics the way that the game is physically played. The second approach is to simulate a large number of dice rolls, and then loop over the rolls to determine how many games were decided and which ones were won or lost. The first approach is easier to program and explain, but is less efficient because it requires a large number of calls to a random number routine. The second approach is more efficient because it requires a single call to a random number routine, but it is slightly harder to program and requires more memory to store the results.

13.5.1 A First Approach

The following algorithm is one that many programmers would use to simulate playing many games of craps. It simulates each game by calling the RollDice module with an argument of 1.

```
/* simulate the game of craps: first approach
   Not optimally efficient: many calls to the random number routine */
call randseed(54321);
NumGames = 1e5;
Results = j(NumGames, 1, 0);          /* win/loss status of each game */
do i = 1 to NumGames;
  roll = RollDice(1);
  if roll=7 | roll=11 then do;
    Results[i] = 1;                    /* won by a "natural"          */
  end;
  else if roll=2 | roll=3 | roll=12 then do;
    Results[i] = 0;                    /* lost (rolled "craps")      */
  end;
  else do;
    game = .;                          /* game is not yet determined */
    thePoint = roll;                   /* establish the point         */
    do while (game=.);                 /* keep rolling                */
      roll = RollDice(1);
      if roll=7 then
        game = 0;                      /* roll seven? Lost!          */
      else if roll=thePoint then
        game = 1;                      /* make the point? Won!       */
      end;
      Results[i] = game;               /* record results (won/lost)  */
    end;
  end;
end;
```

The algorithm is straightforward. The program allocates the (**Results**) vector prior to the main loop. This vector holds the results of the simulation. For each game, the appropriate entry for the **Results** vector is set to zero or one according to whether the game was lost or won. The following statements use the values in this vector to show simple statistics about the game of craps. The results are shown in Figure 13.9.

```
won = sum(Results);           /* count the number of 1's */
lost = NumGames-won;
pctWon = Results[:];
print won lost pctWon[format=PERCENT8.3];
```

Figure 13.9 Simulated Wins and Losses

won	lost	pctWon
49279	50721	49.28%

Notice that the average number of games won is slightly less than 50%. Assuming that the simulation accurately reflects the true probabilities in the game, this indicates that the shooter's chance of winning is roughly 1.4% less than the chance of losing. You can use probability theory to show that the exact odds are $244/495 \approx 0.4929$ (Grinstead and Snell 1997). The estimate based on the simulated data is very close to this value.

You could also allocate and fill other vectors that record quantities such as the number of rolls for each game, the value of the "point," and whether each game was determined by a "natural," "craps," or by rolling for a "point." These extensions are examined in the next section.

13.5.2 A More Efficient Approach

An alternative to the algorithm in the previous section is to simulate many rolls of the dice, store those results, and then analyze the rolls to determine which games were won or lost. With this approach, you call the RollDice module with an argument that is large enough to complete the desired number of games. (Again, the potential problem is that it might not be obvious how many rolls are sufficient.) The following statements use this approach:

```
/* simulate the game of craps: second approach
   More efficient: one call to the random number routine */
call randseed(54321);
NumGames = 1e5;
N = 4*NumGames;           /* 1. Determine number of rolls */
rolls = RollDice(N);

Results = j(NumGames, 1, 0);           /* 2. Allocate results vectors */
rollsInGame = j(NumGames, 1, 1);       /* win/loss status of each game */
Points = j(NumGames, 1, .);            /* how many rolls in each game? */
howEnded = j(NumGames, 1, "natural", "craps", ...); /* "point" for each game? */
```

```

rollIdx = 1;
do i = 1 to NumGames;
  count = 1;
  roll = rolls[rollIdx];          /* 3. Keep track of current roll*/
  if roll=7 | roll=11 then do;
    game = 1;                     /* won by a "natural"          */
    howEnded[i] = "natural";
  end;
  else if roll=2 | roll=3 | roll=12 then do;
    game = 0;                     /* lost (rolled "craps")      */
    howEnded[i] = "craps";
  end;
  else do;
    game = .;                     /* game is not yet determined */
    thePoint = roll;              /* establish the point        */
    do while (game = .);          /* keep rolling               */
      rollIdx = rollIdx + 1;      /* examine next roll          */
      roll = rolls[rollIdx];
      if roll = 7 then
        game = 0;                /* roll seven? Lost!          */
      else if roll = thePoint then
        game = 1;                /* make the point? Won!       */
      count = count + 1;
    end;
    Points[i] = thePoint;         /* 4. Record results (won/lost) */
    howEnded[i] = choose(game, "point-Won", "point-Lost");
  end;
  Results[i] = game;
  rollsInGame[i] = count;
  rollIdx = rollIdx + 1;
end;

```

The revised program is about 40% faster than the previous approach. The algorithm in this section has a few differences from the previous algorithm:

1. How can you determine how many rolls you will need for the simulation? You can either use probability theory or you can simulate a small number of games using the algorithm in the previous section and estimate the average number of rolls per game. In either case, you can determine that the expected number of rolls per game is about 3.37. To be on the safe side, round this up to 4 and estimate that **4*NumGames** rolls is sufficient to simulate the requested number of games.
2. Allocate vectors to hold the results. This is done outside of the main loop. In addition to keeping track of the wins and losses, this program also counts how many rolls were required to determine the game (in the vector **rollsInGame**), what the “point” (if any) was for each game (in **Points**), and whether the game was determined by rolling a natural, craps, winning a point, or losing a point (in **howEnded**).
3. Use the integer **rollIdx** to keep track of the current roll.
4. Every time that a game is determined, the various results for that game are recorded in the relevant vectors.

You can estimate the probability of winning the game by calculating the proportion of games won in the simulation. The expected number of rolls is estimated by the mean number of rolls per game in the simulation. These estimates are computed by the following statements and are shown in Figure 13.10.

```
PctGamesWon = Results[:];           /* percentage of games won */
uHow = unique(howEnded);           /* "natural", "craps", ... */
pct = j(1, ncol(uHow));             /* percentage for howEnded */
do i = 1 to ncol(uHow);
    pct[i] = sum(howEnded = uHow[i]) / NumGames;
end;
AvgRollsPerGame = rollsInGame[:];   /* average number of rolls */
print PctGamesWon, pct[colname=uHow], AvgRollsPerGame;
```

Figure 13.10 Simulation-Based Estimates

PctGamesWon			
0.49279			
pct			
craps	natural	point-Lost	point-Won
0.11049	0.22321	0.39672	0.26958
AvgRollsPerGame			
3.37848			

Similarly, for each value of the point, you can estimate the conditional expectation that a shooter will make the point prior to throwing a seven and losing the game. The following statements estimate the conditional expectations:

```
/* examine conditional expectations: Given that the point
   is (4, 5, 6, 8, 9, 10), what is the expectation to win? */
pts = {4 5 6 8 9 10};           /* given the point... */
PctWonForPt = j(1, ncol(pts));   /* allocate results vector */
do i = 1 to ncol(pts);
    idx = loc(Points=pts[i]);     /* find the associated games */
    wl = Results[idx];           /* results of those games */
    PctWonForPt[i] = wl[:];       /* percentage of games won */
end;
print PctWonForPt[colname=(char(pts,2)) format=6.4
    label="Pct of games won, given point"];
```

The previous statements use the LOC function to find all games for which a given point was established, and then compute the percentage of those games that were won. The results are shown in Figure 13.11.

Figure 13.11 Estimates for Conditional Probabilities

Pct of games won, given point					
4	5	6	8	9	10
0.3296	0.3959	0.4578	0.4559	0.3965	0.3284

For comparison, the exact odds of making each point is given in [Table 13.3](#).

Table 13.3 Probabilities of Making a Given Point

Point	4 or 10	5 or 9	6 or 8
Exact Probability	3/9	4/10	5/11
Approximation	0.333	0.4	0.455

13.6 Random Sampling with Unequal Probability

Previous sections have simulated events in which each event had the same probability of occurrence. For tossing a coin, the probability of either side appearing is $1/2$; for rolling a single six-sided die, the probability of any face appearing is $1/6$. These are examples of random sampling with equal probability.

However, sometimes the probability that is associated with one event is greater than the probability of another event. For example, suppose that there are ten socks in a drawer: five are black, two are brown, and three are white. If you draw a sock at random, the probability of the sock being black is 0.5, the probability of brown is 0.2, and the probability of white is 0.3. This is an example of random sampling from the set $\{Black, Brown, White\}$ with unequal probability.

If you are not a sock lover, you can mentally substitute other scenarios. For example, the ideas in this section also apply to political affiliation (for example, Democrats, Republicans, and Independents) and marital status (for example, single, married, separated/divorced).

The following steps simulate the process of drawing socks (with replacement) from a drawer:

1. Choose a sock at random. The probability of choosing a particular color is proportional to the number of socks with that color.
2. Record the color of the sock.
3. Replace the sock in the drawer.
4. Repeat the process.

How can you sample from the set $\{Black, Brown, White\}$ with unequal probability? As shown in [Section 13.4](#), you can use the RANDGEN subroutine to generate outcomes with specified probabilities. In this case, there are three outcomes with probabilities $p = \{0.5, 0.2, 0.3\}$. The RANDGEN

subroutine generates integers 1, 2, or 3 according to those probabilities. This is shown in the following statements:

```
/* simulate drawing a sock from a drawer */
call randseed(123);
outcome = {"Black" "Brown" "White"};
p = {0.5 0.2 0.3}; /* probability of events */
N = 1e5; /* number of simulated draws */

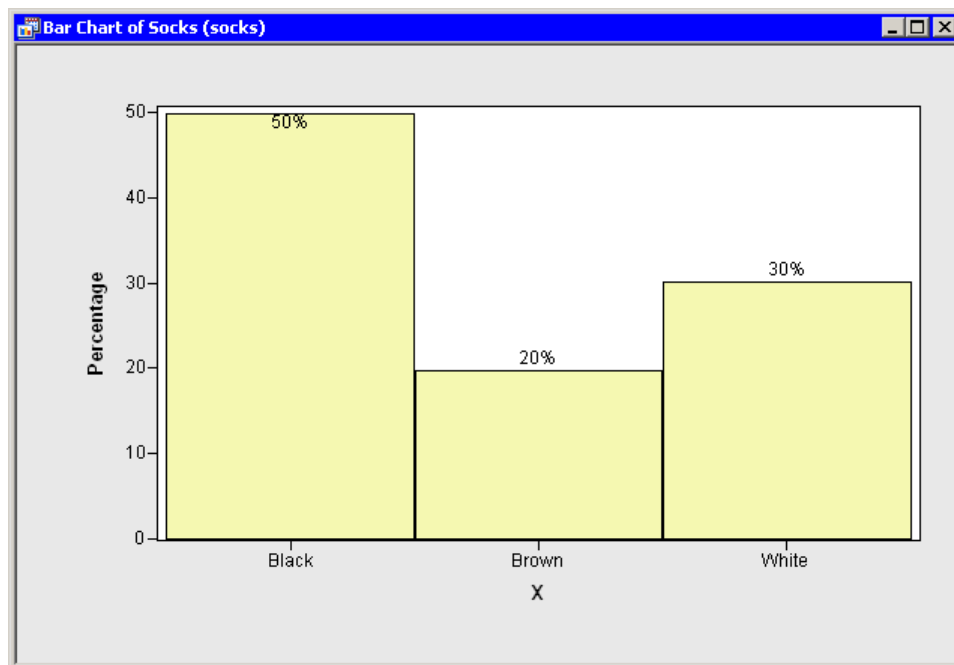
results = j(N, 1);
call randgen(results, "table", p); /* fill with values 1..ncol(p) */
```

Notice that the **results** vector is numeric. When the program ends, the **results** vector contains indices into the **outcome** vector. That is, **results[i]** is an integer (1, 2, or 3) that indicates the color of the *i*th sock.

Figure 13.12 shows the results for one run of the algorithm. The frequencies of the simulated draws are very close to the theoretical probabilities. The figure is created by the following IMLPlus statements:

```
declare BarChart bar;
bar = BarChart.Create("Socks", outcome[results]);
bar.ShowPercentage();
bar.ShowBarLabels();
```

Figure 13.12 Outcomes of Drawing from a Set with Unequal Probability



You can analyze statistics of the simulated results to estimate probabilities such as the probability that three draws in a row result in the same color, or the expected number of draws before you obtain a matching pair in some color. These are left as exercises for the reader.

13.7 A Module for Sampling with Replacement

In each of the previous sections, you used SAS/IML statements to choose from some set of events with some probability. For simulating a coin toss, you sample from the set $\{Heads, Tails\}$ with equal probability $1/2$. For simulating a roll of two dice, you sample twice (independently) from the set $\{1, 2, 3, 4, 5, 6\}$ with equal probability $1/6$. For simulating a choice of socks, you sample from the set $\{Black, Brown, White\}$ with associated probabilities $\{0.5, 0.2, 0.3\}$.

Each of these examples is an instance of the general problem of sampling with replacement from a set of events. It is both useful and instructive to construct a SAS/IML module that samples with replacement from a given set. The module will be a function that returns a matrix that contains random samples from a given set of events. A convenient syntax for the module might be

```
sample = SampleWithReplace(A, numSamples, prob);
```

where the arguments are as follows:

<i>A</i>	is the set of events from which to sample.
<i>NumSamples</i>	specifies the number of times to sample.
<i>prob</i>	specifies the probabilities for sampling. Because sampling with equal probability is an important subcase, it is useful to adopt the convention that if this argument is a missing value, then the module should assume that each element of <i>A</i> has equal probability of being selected.

For example, to simulate tossing a coin 1000 times, you could call the module as follows:

```
EQUAL = .;          /* convention: missing implies uniform probability */
tosses = SampleWithReplace({"H" "T"}, 1000, EQUAL);
```

In this call, **EQUAL** is explicitly set to be a missing value. The module must check whether the third argument is a missing value. If so, it should sample from the events with equal probability.

The following statements define the module `SampleWithReplacement`:

```
/* Random sampling with replacement. The arguments are:
 * A          events (sample space)
 * numSamples number of times to sample from A.
 *           numSamples[1] specifies the number of rows returned.
 *           If numSamples is a vector, then numSamples[2]
 *           specifies the number of repeated draws from A
 *           that are contained in each sample.
 * prob       specifies the probabilities that are associated with
 *           each element of A. If prob=. (missing), then equal
 *           probabilities are used.
 * The module returns a matrix that contains elements of A. The matrix
 * has numSamples[1] rows. It has either 1 or numSamples[2] columns.
 */
```

```

start SampleWithReplace(A, numSamples, prob);
    x = rowvec(A);                                /* 1 */
    k = ncol(x);

    if prob = . then
        p = j(1, k, 1) / k;                        /* 2 */
    else do;
        p = rowvec(prob);
        if ncol(p) ^= k then do;                    /* 3 */
            msg = "ERROR: The probability vector must have the same
                  number of elements as the set being sampled.";
            /* Runtime.Error(msg); */                /* use in SAS/IML Studio */
            reset log; print msg; reset nolog; /* use in PROC IML */
            stop;
        end;
    end;

    /* overload the numSamples argument:
       if a vector, then sumSamples[2] is a repetition factor */
    if nrow(numSamples)=1 & ncol(numSamples)=1 then do;
        nSamples = numSamples;                      /* 4 */
        nRep = 1;
    end;
    else do;
        nSamples = numSamples[1];
        nRep = numSamples[2];
    end;

    results = j(nSamples, nRep);                    /* 5 */
    call randgen(results, "Table", p);

    return (shape(A[results], nSamples));            /* 6 */
finish;

```

This module includes basic error checking and logic to handle both equal and unequal probabilities. The main steps of the module are as follows:

1. The ROWVEC module (in IMLMLIB) transforms the first argument (**A**, the matrix of events) to a row vector. This enables the module to uniformly handle the events without knowing whether **A** was passed in as a row vector or as a column vector. For example, the next statement defines **k** to be the number of events, and this statement is independent of the shape of **A**.
2. The **prob** argument is examined. If the **prob** argument is missing, the module defines **p** to be a row vector of equal probabilities. Otherwise, **p** is defined by the probabilities passed into the module.
3. Is the **prob** argument properly specified? It should have the same number of elements as there are events. If not, you can print a message to the SAS log and use the STOP statement to stop execution of the module. (If you are running the program in SAS/IML Studio, you can use the Runtime.Error method to accomplish the same thing.) You could also add additional

error checking, such as using the CUSUM function to check whether the probabilities sum to unity, but that is not done in this module.

4. The astute reader might wonder how the module will simulate rolling two (or more) dice. After all, there is only a single argument (**numSamples**) that specifies the number of times to roll the dice, but no argument to specify the number of dice. However, the **numSamples** argument is not constrained to be a scalar! You can *overload* the argument: if **numSamples** is vector, then use the second element to specify how many times to repeatedly toss additional coins, roll additional dice, or draw additional socks. Within the module, the local variables **nSamples** and **nRep** contain this information.
5. The samples are generated as described in Section 13.6. A **results** matrix is created and filled with integers in the range $[1, k]$, according to the specified probabilities.
6. The **results** matrix contains integers in the range $[1, k]$. Therefore, the matrix **A[results]** is a column vector that contains the corresponding elements of **A**. The SHAPE function reshapes that matrix so that it has **nSamples** rows and **nRep** columns. This reshaped matrix is returned to the caller of the module.

Programming Tip: Use the ROWVEC and COLVEC modules (defined in IMLMLIB) to transform arguments in a module. This simplifies the process of handling arguments, and the person calling the argument can pass in either row vectors or column vectors.

Programming Tip: Remember that arguments to modules can be vectors. Instead of passing in many scalar arguments, consider passing in vector arguments.

This one module can do all of the work of the programs in the previous sections of this chapter. For example, the following statements call the SampleWithReplace module to simulate a few coin tosses, dice rolls, and sock draws:

```
call randseed(321);
N = 5;
EQUAL = .;
coinTosses = SampleWithReplace({"H" "T"}, N, EQUAL);
diceRolls = SampleWithReplace(1:6, N||2, EQUAL);      /* two dice */
p = {0.5 0.2 0.3};
sockDraws = SampleWithReplace({"Black" "Brown" "White"}, N, p);
print coinTosses, diceRolls, sockDraws;
```

Figure 13.13 Results of Calling the SampleWithReplace Module

coinTosses
H
T
H
T
H

Figure 13.13 *continued*

diceRolls	
3	1
1	3
6	6
3	4
4	6
sockDraws	
Black	
Brown	
Black	
White	
Black	

By encapsulating the complexities of sampling into a module, programs become simpler to support, modify, and maintain. The following statement stores the module for later use:

```
store module=SampleWithReplace;
```

The SampleWithReplace module will be used in the rest of the chapter and in Chapter 14, “[Bootstrap Methods](#).” (If you intend to use the module from PROC IML, you might want to use the RESET STORAGE statement to store the module to a permanent library, as described in [Section 3.4.6](#).)

13.8 The Birthday Matching Problem

The birthday matching problem is a well-known problem in elementary probability. A statement of the problem is, “If there are $N > 2$ people in a room, what is the probability that two or more share the same birthday?” This problem has been studied and presented in many places. The ideas in this section are inspired by the excellent presentation by Trumbo, Suess, and Schupp (2005).

One of the reasons this problem is often discussed is because many people find the probabilities in the problem to be surprising. For example, if $N = 25$, the chance of at least two people sharing a birthday is about 57%!

13.8.1 A Probability-Based Solution for a Simplified Problem

The birthday matching problem is usually presented under the following assumptions:

- The people in the room are randomly chosen
- A year consists of 365 days; no one in the room was born on February 29
- Birthdays are uniformly distributed throughout the year

This last assumption is called the “uniformity assumption.” Under these three assumptions, you can use elementary probability theory to compute the probability of matching birthdays. (Actually, it is simpler to compute the complement: the probability that no one in the room has a matching birthday.) Number the people in the room $1, \dots, N$. The probability that the second person does not share a birthday with the first person is $364/365$. The probability that the third person’s birthday does not coincide with either of the first two people’s birthdays is $363/365$. In general, the probability that the i th person’s birthday does not coincide with any of the birthdays of the first $i - 1$ people is $(365 - i + 1)/365 = 1 - (i - 1)/365$, $i = 1 \dots 365$.

Let Q_N be the probability that no one in the room has a matching birthday. By the assumption that the people in the room are chosen randomly from the population, the birthdays are independent of each other, and Q_N is just the product of the probabilities for each person:

$$Q_N = \prod_{i=1}^N (1 - (i - 1)/365)$$

If R_N is the probability that two or more people share a birthday, then $R_N = 1 - Q_N$.

You can use the index operator ($:$) to create a vector whose i th element is the probability that there is not a common birthday among the first i individuals. It is then easy to use the subscript reduction product operator ($\#$) to form the product of the probabilities. For example, if there are $N = 25$ people in a room, then the following statements compute R_{25} , the probability that at least two people share a birthday:

```
/* compute probability that at least two people share a birthday
   in a room that contains N people */
N = 25;                               /* number of people in room */
i = 1:N;
iQ = 1 - (i-1)/365;                   /* individual probabilities */
Q25 = iQ[#];                           /* product: prob of no match */
R25 = 1 - Q25;                         /* probability of match */
print R25;
```

Figure 13.14 Probability of a Shared Birthday ($N = 25$)

R25
0.5686997

For 25 people in a room, the computation shows that there is about a 57% chance that two or more share a birthday.

You can compute the quantity R_N for a variety of values for N . Notice that the value of Q_i is simply $(1 - (i - 1)/365)$ times the value of Q_{i-1} . Therefore, you can compute the value of Q_i iteratively, as shown in the following statements:

```

/* compute vector of probabilities: R_N for N=1,2,3... */
maxN = 50;                                /* maximum value of N */
Q = j(maxN, 1, 1);                        /* note that Q[1] = 1 */
do i = 2 to maxN;
    Q[i] = Q[i-1] * (1 - (i-1)/365);
end;
ProbR = 1 - Q;

```

The vector **Q** contains the value of Q_N and the vector **ProbR** contains the value of R_N for $N = 1, 2, \dots, 50$. You can query the **ProbR** vector to find a specific probability or to determine how many people need to be in the room before the probability of a shared birthday exceeds some value. For example, the following statements show that if there are at least 32 people in a room, then the chance of a matching birthday exceeds 75%:

```

/* Compute how many people need to be in the room before the
   probability of two people having the same birthday exceeds 75% */
N75 = loc(ProbR > 0.75) [1];
print N75[label="N such that probability of match exceeds 75%"];

```

Figure 13.15 75% Chance of a Match When 32 People Are in a Room

N such that probability of match exceeds 75%	
	32

You can also create an IMLPlus graph that plots elements of the **ProbR** vector versus the number of people in the room. Figure 13.16 shows how the probability depends on N . The following statements create Figure 13.16:

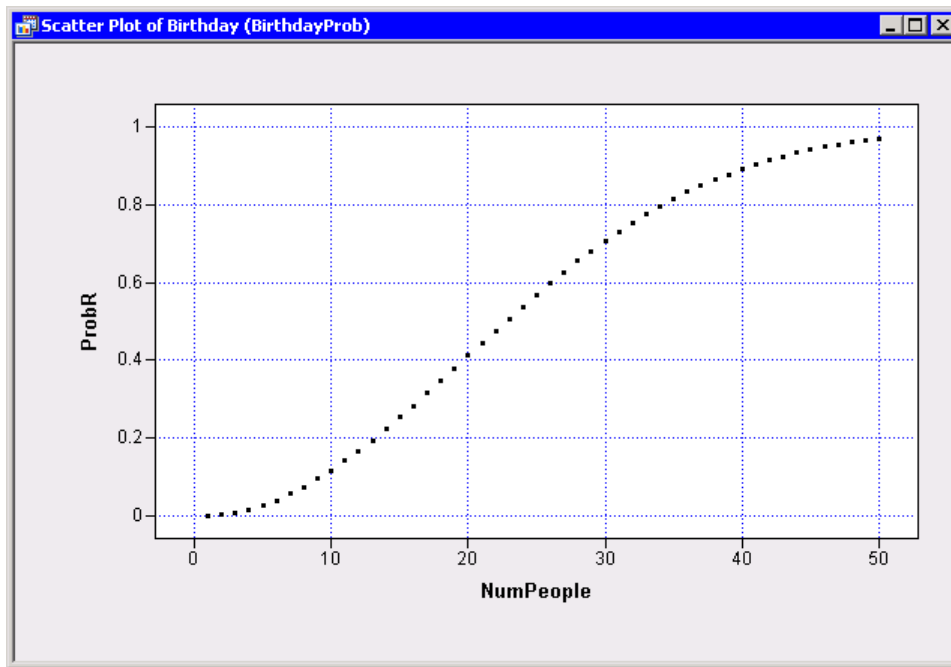
```

/* create plot of probability versus number of people in the room */
np = t(1:maxN);                          /* number in the room (t=transpose) */
declare DataObject dobj;
dobj = DataObject.Create("Birthday", {"NumPeople" "ProbR"}, np||ProbR);

declare ScatterPlot plot;
plot = ScatterPlot.Create(dobj, "NumPeople", "ProbR");
plot.ShowReferenceLines();

```

The figure shows that the probability increases rapidly as the number of people in a room increases. By the time there are 23 people in a room, the chance of a common birthday is more than 50%. With 40 people in a room, the chance is more than 90%.

Figure 13.16 Probability of a Matched Birthday versus the Number of People in a Room

13.8.2 Simulate the Birthday Matching Problem

The birthday matching problem readily lends itself to simulation. To simulate the problem, represent the birthdays by the integers $\{1, 2, \dots, 365\}$ and do the following:

1. Choose a random sample of size N (with replacement) from the set of birthdays. That sample is a “room.”
2. Determine whether any of the birthdays in the room match.
3. Repeat this process for thousands of rooms and tabulate the results.

The following statements create a single “room” with 25 people in it. The 25 birthdays are randomly generated with uniform probability by using the `SampleWithReplace` module that is described in Section 13.7:

```
/* create room with 25 people; find number of matching birthdays */
call randseed(123);
load module=SampleWithReplace;          /* not required in IMLPlus */
N = 25;                                /* number of people in room */
EQUAL = .;                             /* convention */
bday = SampleWithReplace(1:365, 1||N, EQUAL);
u = unique(bday);                       /* number of unique birthdays */
numMatch = N - ncol(u);                 /* number of common birthdays */
print bday, numMatch;
```

The vector that contains all birthdays of people in the room is shown in Figure 13.17, along with the number of matching birthdays. The vector `birthday` contains 25 elements. The `UNIQUE` function removes any duplicate birthdays and returns (in `u`) a sorted row vector of the unique birthdays. As pointed out in Trumbo, Suess, and Schupp (2005), this can be used to determine the number of matching birthdays: simply use the `NCOL` function to count the number of columns in `u` and subtract this number from 25.

Figure 13.17 Probability of a Shared Birthday ($N = 25$)

			bday				
	COL1	COL2	COL3	COL4	COL5	COL6	COL7
ROW1	213	14	29	142	121	132	124
			bday				
	COL8	COL9	COL10	COL11	COL12	COL13	COL14
ROW1	62	21	30	338	83	174	357
			bday				
	COL15	COL16	COL17	COL18	COL19	COL20	COL21
ROW1	275	309	124	296	7	160	335
			bday				
		COL22	COL23	COL24	COL25		
	ROW1	39	29	53	260		
			numMatch				
				2			

For this sample, there are actually two pairs of matching birthdays. As shown in Figure 13.17, the third and twenty-third persons share a birthday on day 29 (that is, 29JAN), and the seventh and seventeenth persons share a birthday on day 124 (that is, 04MAY).

As seen in this example, it is possible to have a sample for which there is more than one birthday in common. How often does this happen? Or, said differently, in a room with N people, what is the distribution of the number of matching birthdays?

Simulation can answer this question. You can create many “rooms” of 25 people. For each room, record the number of matching birthdays in that room. The following statements present one way to program such a simulation:

```

/* simulation: record number of matching birthdays for many samples */
N = 25;                               /* number of people in room */
NumRooms = 1e5;                       /* number of simulated rooms */
bday = SampleWithReplace(1:365, NumRooms||N, EQUAL);
match = j(NumRooms, 1);               /* allocate results vector */
do j = 1 to NumRooms;
    u = unique(bday[j,]);              /* number of unique birthdays */
    match[j] = N - ncol(u);           /* number of common birthdays */
end;

```

You can use the contents of the `match` vector to estimate the probability of certain events. For example, the following statements estimate the probability that a room with 25 people contains at least one pair of matching birthdays, and the probabilities of exactly one and exactly two matching birthdays. The results are shown in [Figure 13.18](#).

```
/* estimate probability of a matching birthday */
SimP = sum(match>0) / NumRooms;           /* any match          */
print SimP[label="Estimate of match (N=25)"];

SimP1 = sum(match=1) / NumRooms;           /* exactly one match    */
print SimP1[label="Estimate of Exactly One Match (N=25)"];

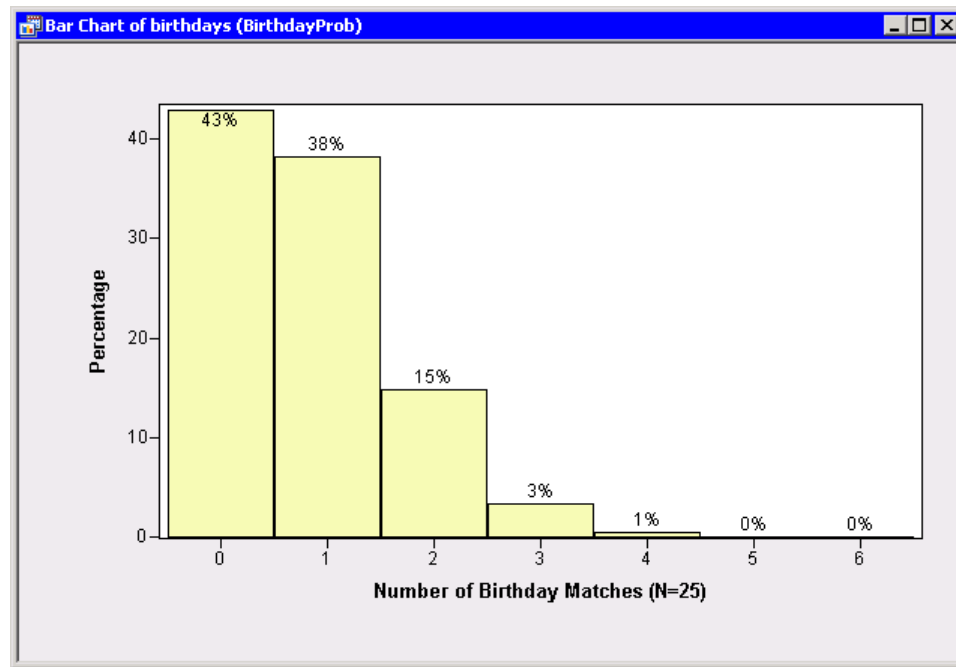
SimP2 = sum(match=2) / NumRooms;           /* exactly two matches  */
print SimP2[label="Estimate of Exactly Two Matched (N=25)"];
```

Figure 13.18 Estimated Probabilities of Matching Birthdays ($N = 25$)

Estimate of match (N=25)	
	0.57117
Estimate of Exactly One Match (N=25)	
	0.38187
Estimate of Exactly Two Matched (N=25)	
	0.14926

Recall from the theoretical analysis that there is about a 57% chance that two or more persons share a birthday when there are 25 people in a room. The estimate based on simulation is very close to this theoretical value. According to the simulation, there is exactly one match about 38% of the time, whereas there are two matches about 15% of the time.

[Figure 13.19](#) shows a bar chart of the elements in the `match` vector. This graph uses the simulated outcomes to estimate the distribution of the number of matching birthdays in a room with 25 people.

Figure 13.19 Distribution of Matching Birthdays in Simulation ($N = 25$)

13.9 Case Study: The Birthday Matching Problem for Real Data

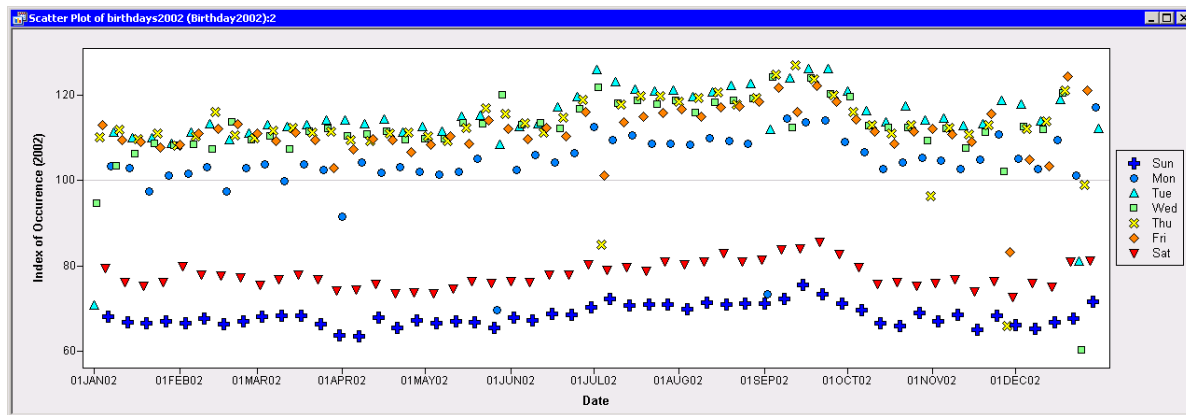
It is interesting to examine one of the main assumptions of the birthday matching problem, namely, are birthdays really distributed uniformly throughout the year?

This is discussed briefly in Trumbo, Suess, and Schupp (2005), and the article notes that data for US births by day are available from the National Center for Health Statistics (NCHS) at the Centers for Disease Control and Prevention (CDC) Web site. The data for live US births by day in 2002 is available in the data set Birthdays2002.

Programming Tip: Simulation enables you to estimate probabilities for which an exact solution is difficult or complicated.

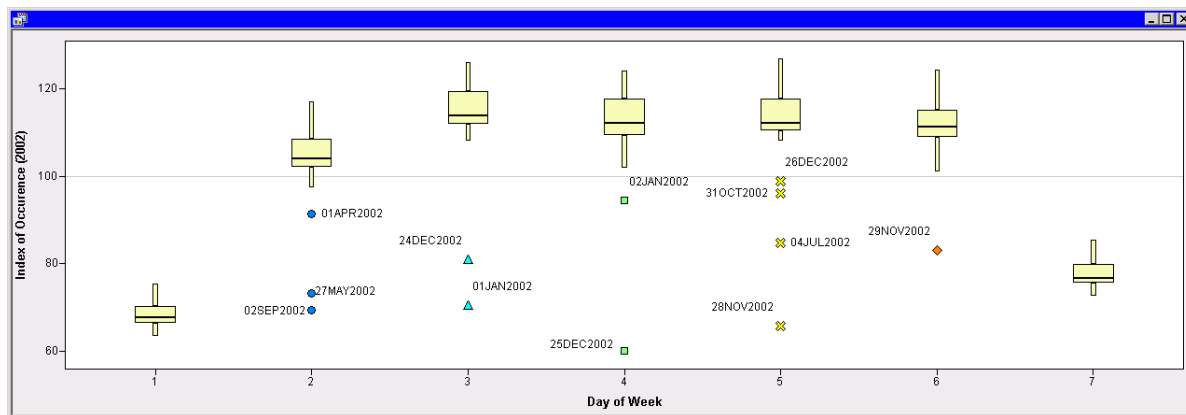
13.9.1 An Analysis of US Births in 2002

Figure 13.20 shows data for the 4,021,726 live births for the year 2002. The mean number of live births each day in 2002 was about 11,018, and the plot shows the percentage of births for each day, relative to the mean: $100 \times \text{births}/(\text{mean births})$. The NCHS calls this quantity the *index of occurrence*. The data range from a minimum value of 6,629 births (an index of about 60) on 25DEC2002, to a maximum value of 13,982 births (an index of 127) on 12SEP2002.

Figure 13.20 Data for US Births (2002)

The shapes of the plot markers in Figure 13.20 correspond to days of the week. These data indicate that the number of births varies according to the day of the week. Tuesday through Friday is when most US babies are born. Mondays account for slightly fewer births, compared with the other weekdays. The graph shows that the fewest babies are born on weekends.

Figure 13.21 presents the same data as a box plot, with separate boxes for each day of the week. Again, the graph indicates a sizeable difference between the mean number of births for each day of the week.

Figure 13.21 Distribution of Births by Day of Week (2002)

The graphs indicate that the day of the week has a large effect on the number of births for that day. Researchers at the NCHS commented on this effect in their analysis of 2004 data, which shows the same qualitative features as the 2002 data (Martin et al. 2006, p. 10):

Patterns in the average number of births by day of the week may be influenced by the scheduling of induction of labor and cesarean delivery.... The relatively narrow range for spontaneous vaginal births contrasts sharply with that of ... cesarean deliveries that ranged from 32.7 on Sunday to 130.5 on Tuesday.

In other words, the NCHS researchers attribute the day-of-week effect to induced labor and cesarean deliveries, as contrasted with “spontaneous” births. The numbers quoted are all indices of occurrence. The 2002 data also indicate a “holiday effect” (notice the outliers in [Figure 13.21](#)) and seasonality with a peak rate during the months of July, August, and September.

13.9.2 The Birthday Problem for People Born in 2002

Suppose we have a room full of randomly chosen people who were born in the US in 2002. The previous section demonstrated that the birthdays of the people in the room are not uniformly distributed throughout the year. There is a strong effect due to the day of the week. This section simulates the birthday problem for a room of people who were born in 2002. Rather than assume that every birthday is equally probable, the analysis uses the actual birth data from 2002 to assign the probability that a person in the room has a given birthday.

The analysis consists of two parts. [Section 13.9.3](#) presents and analyzes the matching “birth day-of-the-week” problem; [Section 13.9.4](#) analyzes the matching birthday problem. For each problem, the analysis consists of two different estimates:

- a probability-based estimate
 - assumes a uniform distribution of birthdays
 - solves the matching problem exactly under the assumption of uniformity
- a simulation-based estimate
 - uses the actual distribution of US birthdays in 2002
 - uses simulation and unequal sampling to estimate the probability of matching

13.9.3 The Matching Birth Day-of-the-Week Problem

Before trying to simulate the birthday matching problem with a nonuniform distribution of birthdays, consider a simpler problem. In light of the day-of-the-week effect noted in [Figure 13.20](#) and [Figure 13.21](#), you can ask: “If there are N people in a room ($N = 2, \dots, 7$), what is the probability that two or more were born on the same day of the week?”

13.9.3.1 A Probability-Based Estimate

[Section 13.8.1](#) uses elementary probability theory to compute the probability of matching birthdays under the assumption of a uniform distribution of birthdays. The same approach can be used for matching the day of week on which someone was born. For simplicity, let the acronym BDOW mean the “birth day of the week.” A person’s BDOW is one of the days Sunday, Monday, ..., Saturday. Number the people in the room $1 \dots N$. The probability that the i th person’s BDOW does not coincide with any of the BDOW of the first $i - 1$ people is $1 - (i - 1)/7$, $i = 1, \dots, 7$.

Let S_N be the probability that no one in the room has a matching BDOW. Then

$$S_N = \prod_{i=1}^N (1 - (i-1)/7)$$

If R_N is the probability that two or more people share a BDOW, then $R_N = 1 - S_N$.

As explained in [Section 13.8.1](#), you can compute the quantity R_N for a variety of values for N , as shown in the following statements:

```
/* compute vector of probabilities: R_N for N=1..7 */
maxN = 7;                               /* maximum value of N */
S = j(maxN, 1, 1);                       /* note that S[1] = 1 */
do i = 2 to maxN;
    S[i] = S[i-1] * (1 - (i-1)/7);
end;
ProbR = 1 - S;
rlabel = 'N1': 'N7';
print ProbR[rowname=rlabel];
```

Figure 13.22 Probability of a Shared BDOW, Assuming Uniform Distribution

ProbR	
N1	0
N2	0.1428571
N3	0.3877551
N4	0.6501458
N5	0.8500625
N6	0.9571607
N7	0.9938801

[Figure 13.22](#) shows how the probability of a matching BDOW depends on N , the number of people in the room, under the assumption that BDOWs are uniformly distributed. These estimates are lower bounds for the more general matching BDOW problem with a nonuniform distribution of BDOWs (Munford 1977).

13.9.3.2 A Simulation-Based Estimate

As shown in [Figure 13.21](#), the BDOWs are *not* uniformly distributed. You can use simulation to determine how the departure from uniformity changes the probabilities shown in [Figure 13.22](#).

The first step is to compute the proportion of the 2002 birthdays that occur for each day of the week. The following statements read the data and compute the empirical proportions:

```
/* compute proportion of the birthdays that occur for each DOW */
use Sasuser.Birthdays2002;
read all var {Births} into y;
read all var {DOW} into Group;
close Sasuser.Birthdays2002;
```

```

u = unique(Group);
GroupMeans = j(1, ncol(u));
do i = 1 to ncol(u);
    idx = loc(Group = u[i]);
    GroupMeans[i] = (y[idx])[:]; /* mean births for each day of week */
end;

proportions = GroupMeans/GroupMeans[+]; /* rescale so the sum is 1 */
labl = {"Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"};
print proportions[colname=labl format=5.3];

```

Figure 13.23 Proportion of Birthdays for each Day of the Week

proportions						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
0.098	0.149	0.163	0.160	0.160	0.159	0.111

You can use the proportions shown in [Figure 13.23](#) as the values for the *prob* argument of the `SampleWithReplace` module. That is, you can generate samples where the probability that a person has a birthday on Sunday is proportional to the number of babies who were born on Sunday in the 2002 data, and similarly for Monday, Tuesday, and so on.

The following statements use the `proportions` vector to simulate the probability of a matching BDOW in a room of N people who were born in the US in 2002, for $N = 2, 3, \dots, 7$:

```

/* simulate matching BDOW by using empirical proportions from 2002 */
load module=(SampleWithReplace); /* not required in IMLPlus */
p = proportions; /* probability of events */
call randseed(123);

NumRooms = 1e5; /* number of simulated rooms */
NumPeople = 1:7; /* number of people in room */

SimR = j(7, 1, 0); /* est prob of matching BDOW */
match = j(NumRooms, 1); /* allocate results vector */

do N = 2 to 7; /* for rooms with N people... */
    bdow = SampleWithReplace(1:7, NumRooms||N, p); /* simulate */
    do j = 1 to NumRooms;
        u = unique(bdow[j,]); /* number of unique BDOW */
        match[j] = N - ncol(u); /* number of common BDOW */
    end;
    SimR[N] = (match>0) [:]; /* proportion of matches */
end;

rlabel = "N1":"N7";
print SimR[rowname=rlabel label="Estimate"];

```

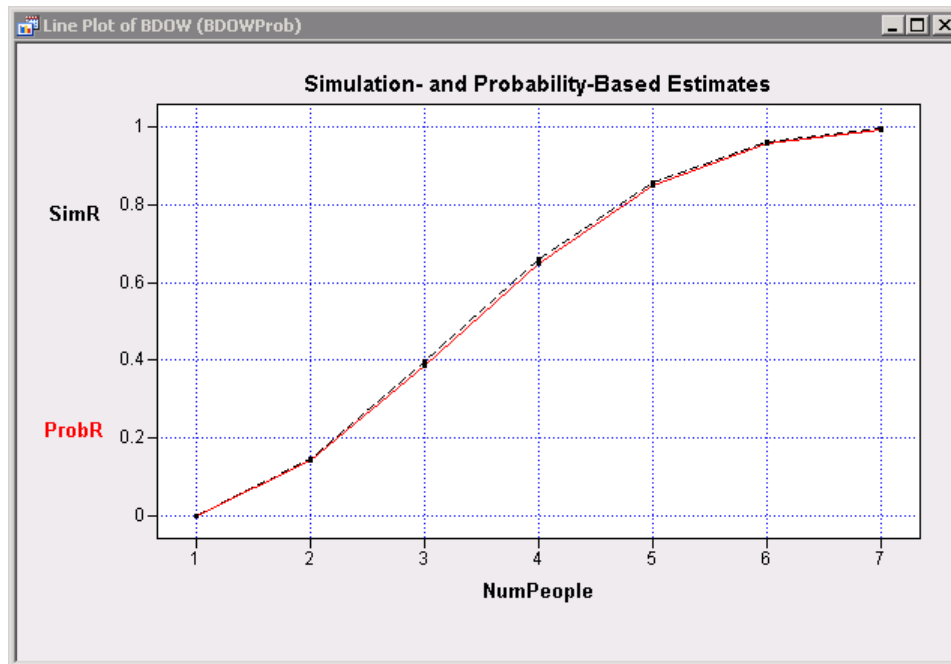
The results are shown in [Figure 13.24](#). The i th row of the `SimR` vector contains the estimated probability that there is at least one matching BDOW in a room of i people who were born in 2002.

Figure 13.24 Estimated Probability of Matching BDOWs (2002 Proportions)

Estimate	
N1	0
N2	0.14557
N3	0.39739
N4	0.6615
N5	0.85777
N6	0.9611
N7	0.99442

13.9.3.3 Compare the Probability-based and Simulation-Based Estimates

You can compare the probability-based and simulation-based estimates of a matching BDOW. The simulation-based estimate is contained in the **SimR** vector; the probability-based estimate is contained in the **ProbR** vector shown in Figure 13.22. Figure 13.25 is a line plot that compares these quantities. The results are almost identical. The solid line (which represents the probability-based estimates) is slightly below the dashed line (the simulation-based estimate).

Figure 13.25 Comparison of Matching BDOW for Equal and Unequal Probabilities (2002)

This close agreement is somewhat surprising in view of Figure 13.21, which shows stark differences between births on weekdays and on weekends. Munford (1977) showed that any nonuniform distribution increases the likelihood of a matching BDOW. Consequently, the curve of probability-based estimates shown in Figure 13.24 is lower than the simulation-based curve. The difference between the two quantities is shown in Figure 13.26. The largest difference occurs in a room with four people. However, the difference in the estimated chance of a matching BDOW is at most 1.1%.

```
diff = SimR - ProbR;
print diff[rowname=rlabel];
```

Figure 13.26 Difference between Estimates

	diff
N1	0
N2	0.0027129
N3	0.0096349
N4	0.0113542
N5	0.0077075
N6	0.0039393
N7	0.0005399

In conclusion, although the probability-based estimates assume a uniform distribution of BDOWs, the probability estimates are not much different from the simulation-based estimates that use a distribution of BDOWs that is based on actual 2002 data.

13.9.4 The 2002 Matching Birthday Problem

Suppose a room contains N randomly chosen people born in the US in 2002. What is the probability of a matching birthday? The values shown in [Figure 13.16](#) are estimates for the probabilities. Recall that [Figure 13.16](#) was created for a probability-based model that assumes a uniform distribution of birthdays.

You can also simulate the matching birthday problem. The ideas and the programs that are described in the previous section extend directly to the matching birthday problem:

- The probabilities used for the simulation are the proportions of the total birthdays that occurred for each day in 2002.
- Samples are drawn (with replacement) from the set $\{1, 2, \dots, 365\}$.
- The number of people in the room varies from 2 to 50.

The algorithm for simulating matching birthdays is the same as for simulating matching BDOWs. However, the birthday problem is much bigger. The sample space contains 365 elements, and you need to generate many rooms that each contain between 2 and 50 people. Whereas each simulation that generates [Figure 13.25](#) is almost instantaneous, the corresponding simulations for the matching birthday problem take longer. Consequently, the following program uses only 10^4 rooms for each of the 49 simulations:

```

/* simulate matching birthdays by using empirical proportions from 2002 */
use Sasuser.Birthdays2002;
read all var {Births};
close Sasuser.Birthdays2002;
call randseed(123);
load module=SampleWithReplace;      /* not required in IMLPlus */
p = Births/Births[+];               /* probability of events */
NumRooms = 1e4;                     /* number of simulated rooms */
maxN = 50;
NumPeople = 1:maxN;                 /* number of people in room */

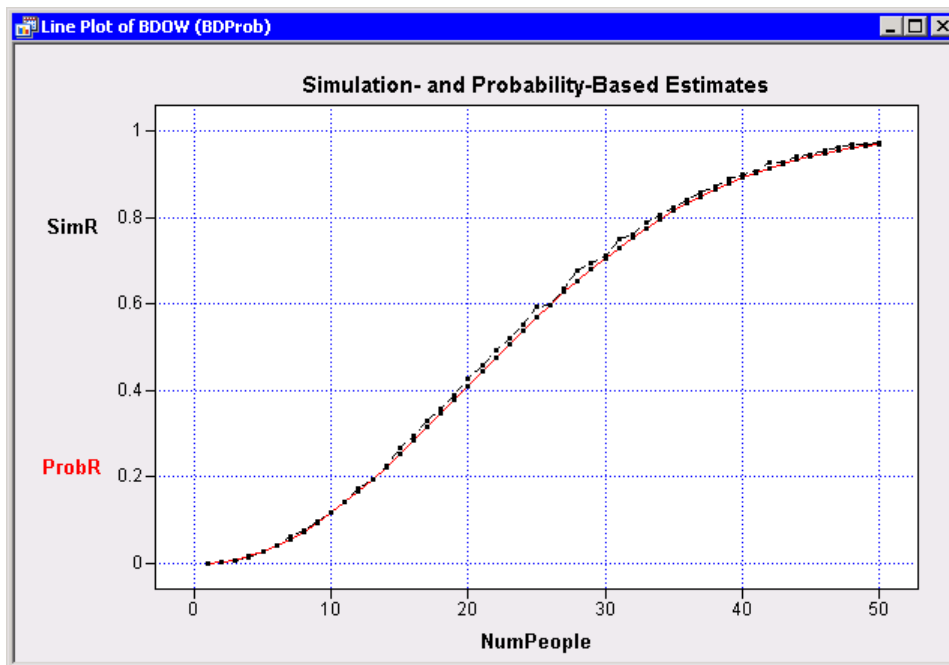
SimR = j(maxN, 1, 0);
match = j(NumRooms, 1);             /* allocate results vector */

do N = 2 to maxN;                   /* for rooms with N people... */
  bday = SampleWithReplace(1:365, NumRooms||N, p); /* simulate */
  do j = 1 to NumRooms;
    u = unique(bday[j,]);           /* number of unique birthdays */
    match[j] = N - ncol(u);         /* number of common birthdays */
  end;
  /* estimated prob of >= 1 matching birthdays */
  SimR[N] = (match>0)[:];
end;

```

You can plot the estimates from the following simulation on the same graph as the probability-based estimates shown in [Figure 13.16](#). The results are shown in [Figure 13.27](#).

Figure 13.27 Comparison of Probability Estimates for Matching Birthdays



The results are similar to the matching BDOW problem. Once again, the solid line (which represents the probability-based estimates) is slightly below the dashed line (the simulation-based estimate). The largest difference occurs in the room with 25 people; the difference is 2.6%. This close agreement between the simulation-based estimates and the probability-based estimates is surprising, given the magnitude of the departures from uniformity shown in [Figure 13.20](#). The dashed curve for the simulation-based estimates is jagged because each point on the curve is based on only 10^4 simulated rooms.

13.10 Calling C Functions from SAS/IML Studio

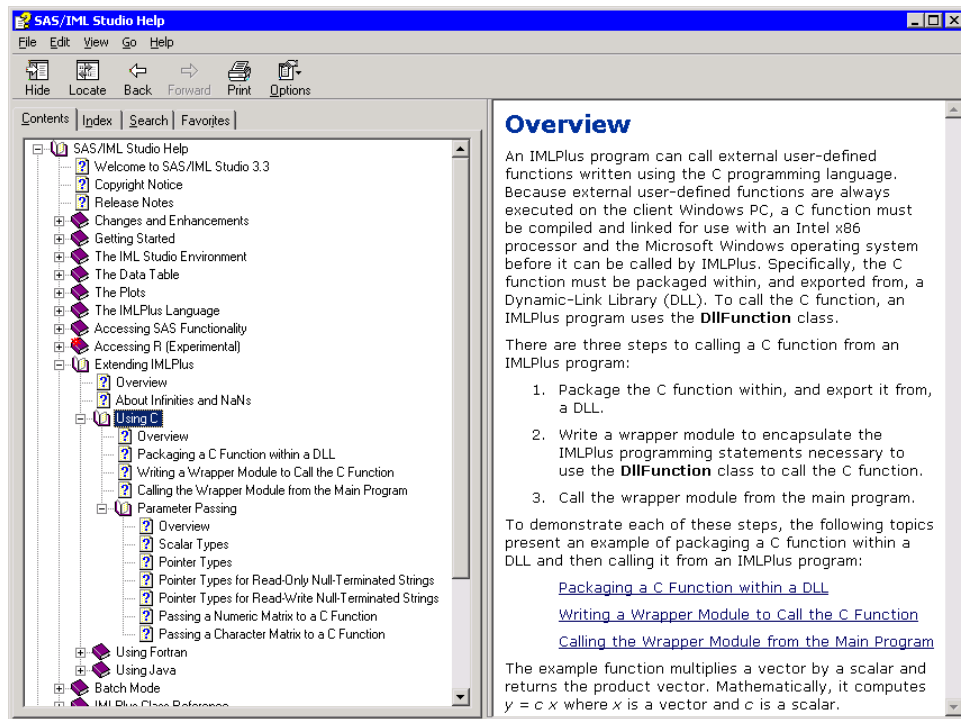
In most cases, you can use the techniques presented in this chapter to implement simulations that are written entirely in the SAS/IML language. As the examples in this chapter demonstrate, the SAS/IML language enables you to simulate complex events such as dice games and the matching birthday problem. However, as indicated in [Section 13.9.4](#), some simulation problems require generating many millions of samples. Furthermore, the analysis that you perform on each sample might be very time consuming. (In [Section 13.9.4](#), the “analysis” consists of simply counting the number of matching birthdays.) In circumstances such as these, you might decide to write a portion of your program in the C programming language, compile the code into a Windows DLL, and call the C function from within SAS/IML Studio.

The process of how to compile and call a C function from a Windows DLL is explained in the SAS/IML Studio online Help. To view the documentation in the SAS/IML Studio Help:

1. Display the help by selecting **Help►Help Topics** from the SAS/IML Studio main menu.
2. Expand the link for the SAS/IML Studio Help topic.
3. Expand the link for the chapter “Extending IMLPlus.”
4. Expand the link for the section “Using C.”

[Figure 13.28](#) shows the topics in the “Using C” portion of the documentation. Notice that there is also a section entitled “Using FORTRAN,” which explains how to compile a FORTRAN function into a DLL and how to call it from an IMLPlus program.

Figure 13.28 The SAS/IML Studio Help



13.11 References

- Grinstead, C. M. and Snell, J. L. (1997), *Introduction to Probability*, Second revised Edition, Providence, RI: American Mathematical Society.
- Lohr, S. L. (2009), *Sampling: Design and Analysis*, Second Edition, Pacific Grove, CA: Duxbury Press.
- Martin, J. A., Hamilton, B. E., Sutton, P. D., Ventura, S. J., Menacker, F., and Kirmeyer, S. (2006), "Births: Final Data for 2004," *National Vital Statistics Reports*, 55(1).
URL http://www.cdc.gov/nchs/data/nvsr/nvsr55/nvsr55_01.pdf
- Munford, A. G. (1977), "A Note on the Uniformity Assumption in the Birthday Problem," *The American Statistician*, 31(3), 119.
- Trumbo, B., Suess, E., and Schupp, C. (2005), "Computing the Probabilities of Matching Birthdays," *STATS: The Magazine for Students of Statistics*, Spring(43), 3–7.

Chapter 14

Bootstrap Methods

Contents

14.1	An Introduction to Bootstrap Methods	349
14.2	The Bootstrap Distribution for a Mean	350
14.2.1	Obtaining a Random Sample	350
14.2.2	Creating a Bootstrap Distribution	351
14.2.3	Computing Bootstrap Estimates	353
14.3	Comparing Two Groups	356
14.4	Using SAS Procedures in Bootstrap Computations	358
14.4.1	Resampling by Using the SURVEYSELECT Procedure	359
14.4.2	Computing Bootstrap Statistics with a SAS Procedure	361
14.5	Case Study: Bootstrap Principal Component Statistics	363
14.5.1	Plotting Confidence Intervals on a Scree Plot	366
14.5.2	Plotting the Bootstrap Distributions	368
14.6	References	369

14.1 An Introduction to Bootstrap Methods

The bootstrap method is a general technique for estimating unknown parameters, computing standard errors for statistics, and finding confidence intervals for parameters. Introductory references for bootstrap techniques include Efron and Tibshirani (1993) and Davison and Hinkley (1997).

This chapter is an applied chapter that describes how to use the bootstrap method in SAS/IML and IMLPlus. IMLPlus has several features that make the bootstrap method easier to implement. The chapter introduces resampling techniques and explains estimating variability with the bootstrap standard error and bootstrap confidence intervals.

The basic bootstrap algorithm is as follows. Given data $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and a statistic $s(\mathbf{x})$, the bootstrap method generates a large number of independent bootstrap samples $\mathbf{x}^{*1}, \mathbf{x}^{*2}, \dots, \mathbf{x}^{*B}$. Each sample is typically of size n . For each bootstrap sample, you compute $s(\mathbf{x}^{*i})$, the statistic of the i th sample. In the simplest cases, the bootstrap samples are formed by randomly sampling from \mathbf{x} with replacement.

The union of the statistics forms the *bootstrap distribution* for s . The bootstrap distribution is an estimate for the sampling distribution of the statistic. The mean of the bootstrap distribution is

an estimate of $s(\mathbf{x})$ (which is itself an estimate of the corresponding population parameter). The standard deviation of the bootstrap distribution is the bootstrap estimate of the standard error of s . Percentiles of the bootstrap distribution are the simplest estimates for confidence intervals. There are also more sophisticated estimates for confidence intervals (see Efron and Tibshirani (1993)), but these are not described in this book.

14.2 The Bootstrap Distribution for a Mean

One of the simplest statistics is the mean. Because the sampling distribution of the mean and the standard error of the mean (SEM) are well understood, the mean is a good choice for a statistic to use in an introduction to bootstrap methods.

This section consists of three parts. The first part creates a data set to be analyzed. The second part resamples from the data many times and computes the mean of each resample. The union of the means is the bootstrap distribution for the mean statistic. The third part uses the bootstrap distribution to estimate the SEM and confidence intervals for the mean of the underlying population.

14.2.1 Obtaining a Random Sample

Suppose you are interested in statistics related to movies released in the US during the time period 2005–2007. The Movies data set contains a sampling of movies that were also rated by the kids-in-mind.com Web site. Strictly speaking, these movies are not a random sample from the population of all US movies, since the data includes all major movies with nationwide release. (In fact, the data essentially equal the population of US movies; the data set is lacking only minor releases that had limited distribution.) In discussing bootstrap statistics, it is important to differentiate between a random sample and the underlying population. To make this distinction perfectly clear, the following DATA step extracts a random sample of approximately 25% of the movies into the Sample data set:

```
/* extract random sample of movies for 2005–2007 */
submit;
data Sample;
    set Sasuser.Movies;
    if (ranuni(1)<0.25);
run;
endsubmit;
```

Figure 14.1 shows the distribution of the budgets for movies in the Sample data set. The histogram is created by the following statements:

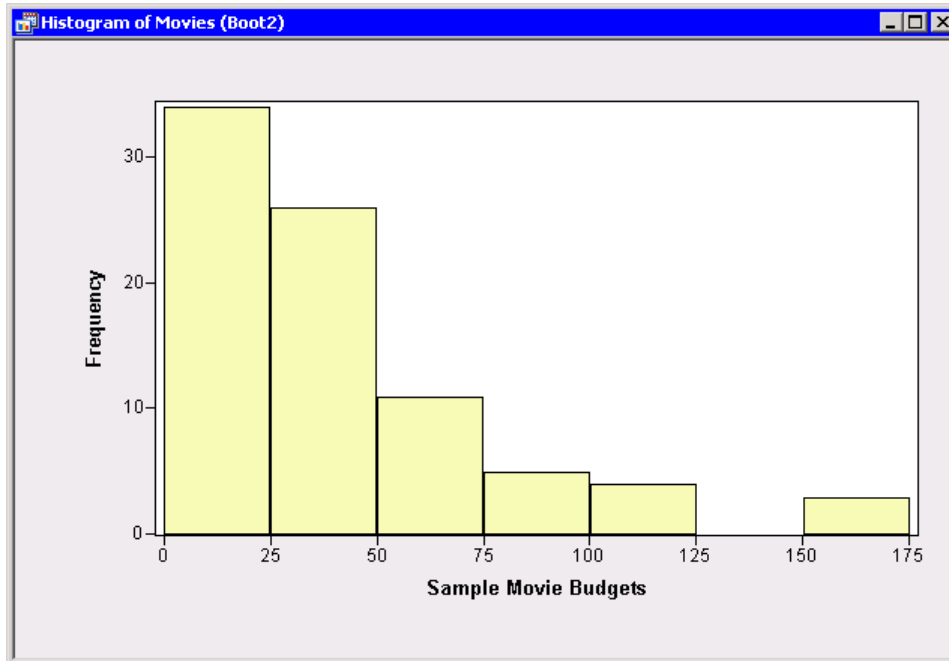
```

/* read in movies and draw histogram */
use Sample;
read all var {Budget} into x;
close Sample;

declare Histogram hist;
hist = Histogram.Create("Movies", x);
hist.SetAxisLabel(XAXIS, "Sample Movie Budgets");

```

Figure 14.1 Distribution of Movie Budgets in Sample



The data are not normally distributed. The sample mean of the budgets is \$39.2 million. There are several movies with very large budgets, although the complete *Movies* data set has three movies with budgets greater than \$200 million that are not included in the *Sample* data. (For comparison, [Figure 7.4](#) shows the distribution for all of the budgets in the *Movies* data set.) The remainder of this chapter analyzes the *Sample* data.

14.2.2 Creating a Bootstrap Distribution

Implementing a bootstrap estimate often includes the following steps:

1. Compute the statistic of interest on the original data.
2. Resample B times from the data (with replacement) to form B bootstrap samples.
3. Compute the statistic on each resample.
4. If desired, graph the bootstrap distribution.

5. Compute standard errors and confidence intervals.

For the movies in the Sample data set, you can use bootstrap methods to examine the sampling distribution of the mean (or any other statistic). Since the first step in the bootstrap method is to resample (with replacement) from the data, it is convenient to use the `SampleWithReplace` module that was developed in Chapter 13, “[Sampling and Simulation](#).”

The following statements use the `SampleWithReplace` module to carry out a simple bootstrap on the budgets for movies:

```
/* bootstrap method: the statistic to bootstrap is the mean */
/* 1. Compute the statistic on the original data */
Mean = x[:];                               /* sample mean          */

/* 2. Resample B times from the data (with replacement)
   to form B bootstrap samples. */
call randseed(12345);
load module=SampleWithReplace;             /* not required in IMLPlus */
B = 1000;
n = nrow(x);
EQUAL = .;
xBoot = SampleWithReplace(x, B||n, EQUAL);

/* 3. compute the statistic on each resample */
s = xBoot[:, :];
```

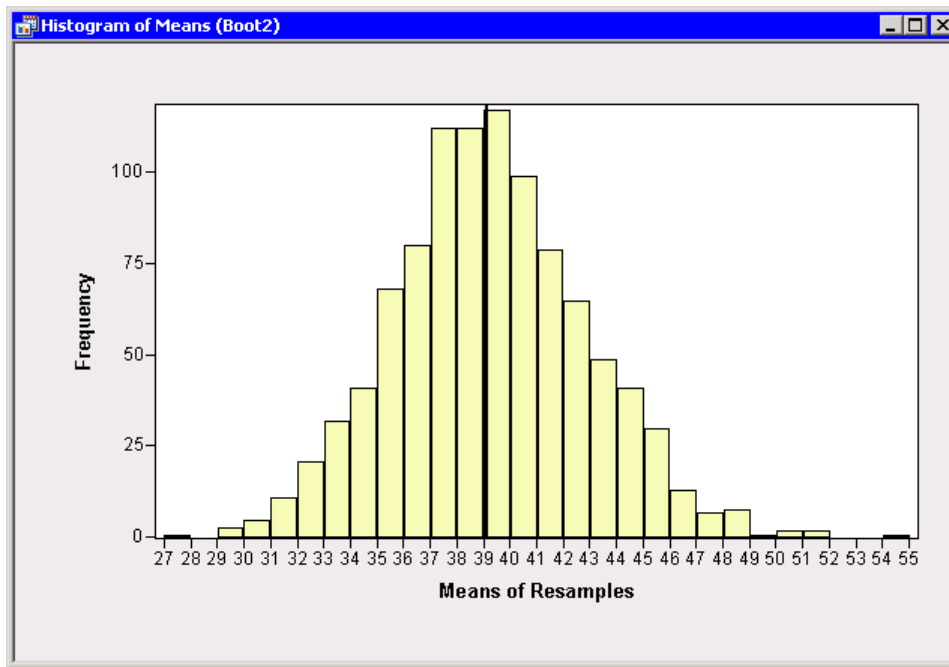
As indicated previously, the main steps of the bootstrap algorithm are as follows:

1. Compute the statistic (in this case, the sample mean) on the data.
2. Use the `SampleWithReplace` module to resample from the data. The module returns a matrix, `xBoot`, which consists of B rows and n columns, where n is the number of observations in the data. Each row represents a resampling from the data.
3. Compute the statistic on each resample. In this case, compute the mean of each row of `xBoot`. The resulting vector, `s`, contains B statistics (means).

Programming Tip: You can use the `SampleWithReplace` module in a bootstrap analysis to resample from the data.

The vector `s` in the previous program contains B statistics. The union of the statistics is the bootstrap distribution for the statistic. It is an estimate for the sampling distribution of the statistic. A histogram of `s` is shown in [Figure 14.2](#) and is created with the following IMLPlus statements:

```
/* 4. If desired, graph the bootstrap distribution */
declare Histogram hBoot;
hBoot = Histogram.Create("Means", s);
hBoot.SetAxisLabel(XAXIS, "Means of Resamples");
hBoot.ReBin(40, 1);
attrib = BLACK || SOLID || 2;
run abline(hBoot, Mean, ., attrib);
```

Figure 14.2 Estimate of the Sampling Distribution of the Mean

The figure shows that most of the means for the resampled data are between \$35 and \$43 million per movie. A small number were as small as \$27 million per movie, or as large as \$54 million per movie. The sample statistic has the value 39.2, which is shown as a vertical line in Figure 14.2. This figure enables you to see the variation in the mean of the resampled data.

14.2.3 Computing Bootstrap Estimates

This section shows how you can use the bootstrap distribution shown in Figure 14.2 to compute bootstrap estimates for familiar quantities: the mean, the standard error of the mean (SEM), and the confidence intervals for the mean. The mean of the bootstrap distribution is an estimate of the mean of the original sample. The standard deviation of the bootstrap distribution is the bootstrap estimate of the standard error of s . Percentiles of the bootstrap distribution are the simplest estimates for confidence intervals. The following statements (which continue the program in the previous section) compute these quantities:

```
/* 5. Compute standard errors and confidence intervals. */
MeanBoot = s[:];          /* a. mean of bootstrap dist */
StdErrBoot = sqrt(var(s)); /* b. std error */
alpha = 0.05;
prob = alpha/2 || 1-alpha/2; /* lower/upper percentiles */
call qntl(CIBoot, s, prob); /* c. quantiles of sampling dist */
print MeanBoot StdErrBoot CIBoot;
```

Figure 14.3 Bootstrap Estimates for the Sampling Distribution of the Mean

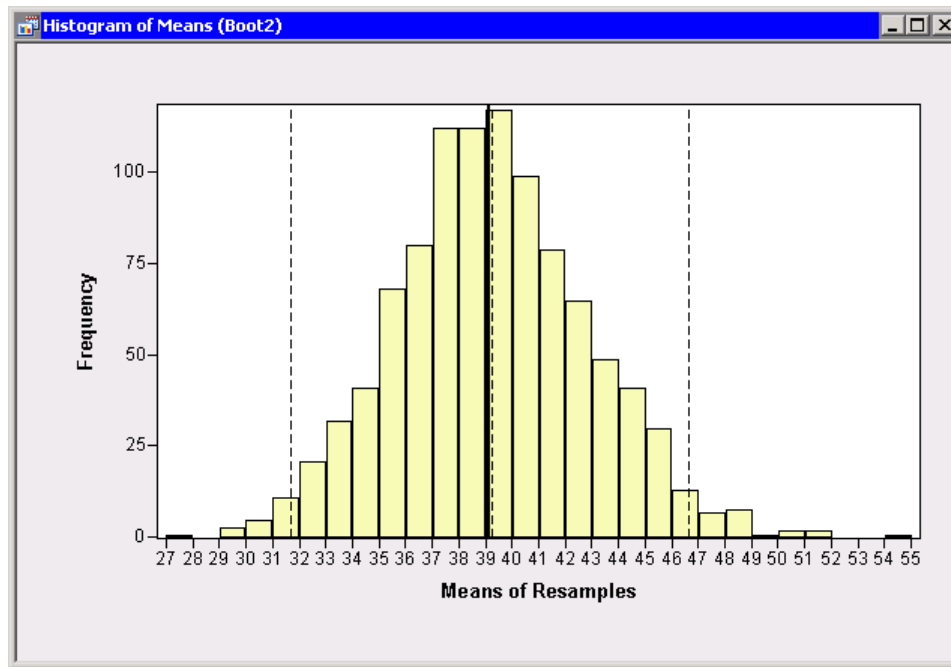
MeanBoot	StdErrBoot	CIBoot
39.270441	3.6625928	32.410843
		46.673494

The output is shown in [Figure 14.3](#). The following list describes the main steps:

- Recall that the bootstrap distribution for the mean is contained in **s**. The mean of the bootstrap distribution is an estimate of the sample mean. In this case, [Figure 14.2](#) shows that the bootstrap distribution is symmetric and unimodal (as expected by the central limit theorem), so the mean of **s** represents the central tendency of the bootstrap distribution.
- The standard deviation of the bootstrap distribution is an estimate of the SEM. In this case, the standard deviation **s** represents the spread of the bootstrap distribution. The VAR function is available in SAS/IML 9.22. If you are using an earlier version, you can use the Var module that is defined in [Appendix D](#).
- The percentiles of the sampling distribution provide estimate of a 95% confidence interval. The QNTL function is implemented in SAS/IML 9.22. If you are using an earlier version, you can use the Qntl module that is defined in [Appendix D](#).

You can visualize these quantities by adding vertical lines to the histogram of the sampling distribution. The following statements draw lines on [Figure 14.2](#). The resulting graph is shown in [Figure 14.4](#).

```
attrib = BLACK || DASHED || 1 || PLOTFOREGROUND;
run abline(hBoot, MeanBoot//CIBoot, {.,.,.}, attrib);
```


Figure 14.4 Estimate of the Mean and Confidence Intervals for the Sampling Distribution

In Figure 14.4, the solid line is the original statistic on the data. The dashed lines are bootstrap estimates.

The power of the bootstrap is that you can create bootstrap estimates for *any* statistic, s , even those for which there are no closed-form expression for the standard error and confidence intervals. However, because the statistic in this example is the mean, you can use elementary statistical theory to compute classical statistics (based on the sample) and compare the bootstrap estimates to the classical estimates.

The sampling distribution of the mean is well understood. The central limit theorem says that the sampling distribution of the mean is approximately normally distributed with parameters (μ, σ) . The sample mean, \bar{x} , is an estimate for μ . An estimate for σ is the quantity $\sqrt{s_x^2/n}$, where s_x^2 is the sample variance and n is the size of the sample. Furthermore, the two-sided $100(1 - \alpha)\%$ confidence interval for the mean has upper and lower limits given by $\bar{x} \pm t_{1-\alpha/2, n-1} \sqrt{s_x^2/n}$, where $t_{1-\alpha/2, n-1}$ is the $1 - \alpha/2$ percentile of the Student's t distribution with $n - 1$ degrees of freedom. The following program computes these estimates:

```
/* Compute traditional estimates for the sampling distribution of
   the mean by computing statistics of the original data */
StdErr = sqrt(var(x)/n);          /* estimate SEM */
t = quantile("T", prob, n-1);    /* percentiles of t distribution */
CI = Mean + t * StdErr;          /* 95% confidence interval */
print Mean StdErr CI;
```

Figure 14.5 Estimates for the Sampling Distribution of the Mean

Mean	StdErr	CI
39.151807	3.7595334	31.672898 46.630717

The output is shown in Figure 14.5. The sample mean is 39.15. The SEM is estimated by 3.76 and the 95% confidence interval by [31.7, 46.6].

A comparison of Figure 14.5 with Figure 14.3 shows that the classical estimates are similar to the bootstrap estimates. However, note that the classical confidence interval is symmetric about the mean (by definition), whereas the bootstrap confidence interval is usually not symmetric. The bootstrap estimate is based on percentiles of the statistic on the resampled data, not on any formula. This fact implies that bootstrap confidence intervals might be useful for computing confidence intervals for a statistic whose sampling distribution is not symmetric.

It is interesting to note that the mean budget in the Movies data set is \$44.8 million, which falls within the 95% confidence limits for either method. The Movies data set contains essentially all (major) movies released in 2005-2007, so you can take \$44.8 million to be the mean of the population.

14.3 Comparing Two Groups

In the previous example, you can replace the mean statistic with any other statistic such as the variance or a trimmed mean. If you choose a set of variables in the Sample data set, you can bootstrap the standard errors and confidence intervals for regression coefficients or correlation coefficients.

You can also use the bootstrap method to compare statistics between two subpopulations. For example, you can compare the mean budgets of PG-13 and the R-rated movies in the Sample data set and ask whether they are significantly different. The computations are straightforward:

1. Given two groups, resample B times from each group (with replacement) to form B bootstrap samples.
2. Compute the difference between the means for each of the B resamples. This creates the bootstrap distribution for the difference of means.
3. Determine whether the 95% percentile confidence intervals contain zero. If so, conclude that there is no evidence that the means of the two groups are different.

The following statements compute the bootstrap estimate:

```
/* compute bootstrap estimate for difference between means of two groups */
use Sample where (MPAARating="PG-13");    /* read data from group 1 */
read all var {Budget} into x1;
use Sample where (MPAARating="R");        /* read data from group 2 */
read all var {Budget} into x2;
close Sample;

/* 1. compute bootstrap distribution for difference between means */
call randseed(12345);
load module=SampleWithReplace;            /* not required in IMLPlus */
B = 1000;
```

```

n1 = nrow(x1);
n2 = nrow(x2);
EQUAL = .;
Boot1 = SampleWithReplace(x1, B||n1, EQUAL); /* resample B times */
Boot2 = SampleWithReplace(x2, B||n2, EQUAL); /* from each group */

/* 2. difference between the B means computes for each resample */
s1 = Boot1[, :]; /* means of B resample from x1 */
s2 = Boot2[, :]; /* means of B resample from x2 */
s = s1 - s2; /* difference of means */

/* 3. Compute bootstrap estimate for 95% C.I. */
alpha = 0.05;
prob = alpha/2 || 1-alpha/2;
call qnt1(CIBoot, s, prob);
print CIBoot;

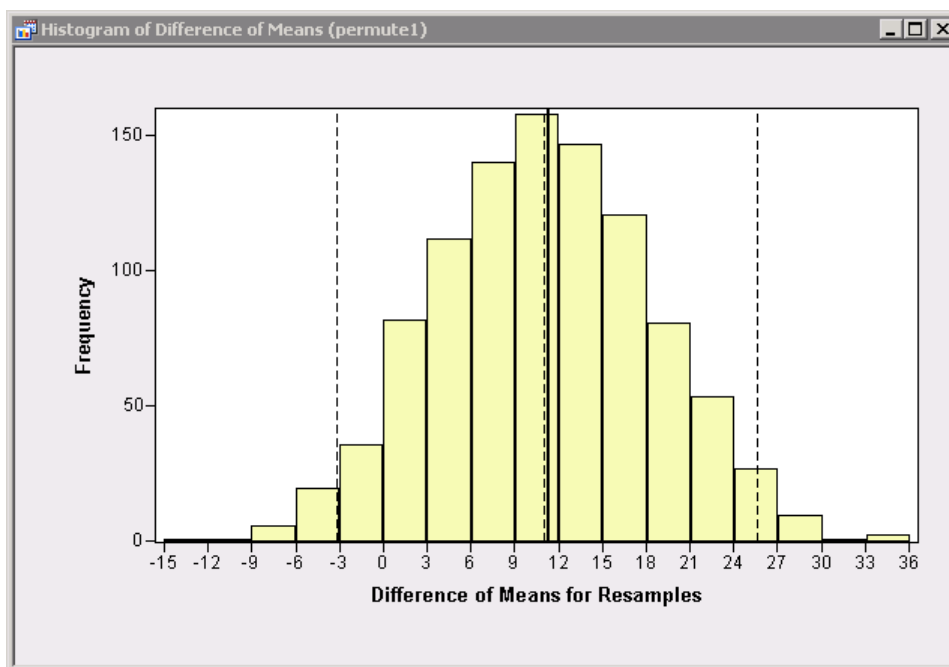
```

Figure 14.6 Bootstrap Confidence Intervals for Difference of Means

CIBoot
-3.171053
25.638467

As shown in Figure 14.6, the 95% confidence interval for the mean difference includes zero, so the observed difference in the mean budgets between PG-13 and R-rated movies is not significant. Figure 14.7 shows a bootstrap distribution for the difference of means. Vertical lines on the figure indicate the 2.5% and 97.5% percentiles of the distribution.

Figure 14.7 A Bootstrap Distribution for the Difference of Means



14.4 Using SAS Procedures in Bootstrap Computations

In many ways, the SAS/IML language is a natural language for programming simple bootstrap algorithms on small or medium-sized data sets. As shown in the previous sections, it is fast and easy to resample the data and to compute a mean or another descriptive statistic.

However, it is also possible to implement bootstrap methods in IMLPlus by combining SAS/IML statements with calls to SAS procedures. Procedures offer several advantages over implementing complicated bootstrapping algorithms entirely in SAS/IML software:

More Resampling Methods The bootstrap resampling is supposed to mimic the design that produced the original data. For random sampling with replacement, this is straightforward, as indicated in Chapter 13, “[Sampling and Simulation](#).” For complicated sampling schemes, it might be useful to use the SURVEYSELECT procedure to carry out the bootstrap resampling, as opposed to implementing the resampling entirely in SAS/IML statements.

More Statistical Computations Most of the statistical functionality in SAS software is contained in procedures. The functionality available in SAS/STAT, SAS/ETS, and other products is much greater than the built-in functionality of the SAS/IML statistical library. For example, consider the problem of estimating the accuracy of parameter estimates for a complicated generalized linear model. It is straightforward to call the GENMOD procedure in SAS/STAT, whereas SAS/IML does not have an analogous built-in function or module.

Procedures Handle Missing Values Not all SAS/IML operations support missing values (for example, matrix multiplication), but SAS procedures handle missing values without any effort from you.

Procedures Handling Degenerate Data Writing robust SAS/IML statements to perform a computation means you have to handle errors that arise in analyzing all possible data sets. For example, a SAS/IML module for linear regression should correctly handle singular data such as collinear variables and constant data. SAS procedures do this already.

Procedures Save Time Writing your own SAS/IML computation means spending time verifying that your program is computing the correct answers. A SAS/IML program that you write yourself might have a subtle (or not so subtle!) error, whereas the SAS procedures have been tested and validated. Furthermore, SAS procedures (which are implemented in compiled and optimized C code) can compute statistics more efficiently than an interpreted language such as the SAS/IML language.

Procedures Handle Large Data SAS/IML software requires that all vectors and matrices fit into RAM. One of the advantages of SAS procedures is the ability to compute with data sets so large that they do not fit in RAM. While bootstrapping a mean is not memory-intensive, computing more complex statistics on large data sets might require more memory than is available on your computer.

For all of these reasons, this section describes how you can call SAS procedures to generate resamples and to compute statistics on the resamples. You can use the SURVEYSELECT procedure in

SAS/STAT software to carry out the resampling portion of the bootstrap method. You can call any SAS procedure to compute the statistics.

Programming Tip: You can call SAS procedures as part of the bootstrap computation. This is especially useful when the statistic that you are bootstrapping was itself computed by SAS procedures.

14.4.1 Resampling by Using the SURVEYSELECT Procedure

Some SAS programmers use the DATA step for everything, including generating random samples from data. The DATA step can sample reasonably fast. But it is even faster to use the SURVEYSELECT procedure for generating a random sample.

The SURVEYSELECT procedure can generate B random samples of the input data, and is therefore ideal for bootstrap resampling (Cassell 2007, 2010; Wicklin 2008). It implements many different sampling techniques, but this section focuses on random sampling with replacement. The SURVEYSELECT procedure refers to “sampling with replacement” as “unrestricted random sampling” (URS).

As an example, suppose your data consists of four values: A, B, C, and D. The following statements create a data set named `BootIn` that contains the data:

```
/* create small data set */
x = {A,B,C,D};
create BootIn var {"x"};
append;
close BootIn;
```

Suppose you want to generate B bootstrap resamples from the data in the `BootIn` data set. The following statements use the SUBMIT statement to call the SURVEYSELECT procedure and to pass in two parameters to the procedure. The procedure reads data from `BootIn` and writes the resamples into the `BootSamp` data set.

```
/* Use the SURVEYSELECT procedure to generate bootstrap resamples */
N = nrow(x);
B = 5;

submit N B;
proc surveyselect data=BootIn out=BootSamp noprint
    seed    = 12345          /* 1 */
    method  = urs           /* 2 */
    n       = &N            /* 3 */
    rep     = &B            /* 4 */
    OUTHITS;                /* 5 */
run;
endsubmit;
```

The following list describes the options to the SURVEYSELECT procedure:

1. Set the seed for the random number generator used by the SURVEYSELECT procedure.

2. Specify the method used for resampling. This example uses unrestricted random sampling (URS).
3. Specify the size of the resampled data. For bootstrap resampling this is often the size of the input data set. This value is passed in on the SUBMIT statement.
4. Specify the number of times, B , to resample. This value is also passed in on the SUBMIT statement.
5. Specify the OUTHITS option on the PROC SURVEYSELECT statement when you want the output data to consist of nB observations, where the first n observations correspond to the first resample, the next n observations correspond to the next resample, and so on.

The output data set created by the SURVEYSELECT procedure is displayed by the following statements, and is shown in Figure 14.8:

```
submit;
proc print data=BootSamp;
  var Replicate x;
run;
endsubmit;
```

Figure 14.8 Bootstrap Resamples Using OUTHITS Option

Obs	Replicate	x
1	1	B
2	1	B
3	1	C
4	1	D
5	2	A
6	2	A
7	2	C
8	2	C
9	3	A
10	3	B
11	3	C
12	3	D
13	4	A
14	4	B
15	4	B
16	4	D
17	5	A
18	5	A
19	5	A
20	5	B

Programming Tip: You can use the SURVEYSELECT procedure to generate bootstrap resamples. This is especially useful if the sampling scheme is complex.

It is not necessary to specify the OUTHITS option. In fact, it is more efficient to omit the OUTHITS option. If you omit the OUTHITS option, the NumberHits variable in the output data set contains the

frequency that each observation is selected. Omitting the OUTHITS option and using NumberHits as a frequency variable when computing statistics is typically faster than using the OUTHITS option because there are fewer observations in the BootSamp data set to read. Most, although not all, SAS procedures have a FREQ statement for specifying a frequency variable. Figure 14.9 shows the output data set created by the SURVEYSELECT procedure when you omit the OUTHITS option:

Figure 14.9 Bootstrap Resamples without Using OUTHITS Option

Obs	Replicate	x	Number Hits
1	1	B	2
2	1	C	1
3	1	D	1
4	2	A	2
5	2	C	2
6	3	A	1
7	3	B	1
8	3	C	1
9	3	D	1
10	4	A	1
11	4	B	2
12	4	D	1
13	5	A	3
14	5	B	1

The BootSamp data set contains a copy of all variables in the input data set and contains the variable Replicate, which identifies the bootstrap resample that is associated with each observation. The data set also contains the NumberHits variable, which can be used on a FREQ statement to specify the frequency of each observation in the resample.

Programming Tip: You can omit the OUTHITS option on the PROC SURVEYSELECT statement. The output data set will contain fewer observations. You can compute statistics on the bootstrap resamples by using the FREQ statement of SAS procedures.

14.4.2 Computing Bootstrap Statistics with a SAS Procedure

The preceding section described how to use the SURVEYSELECT procedure to generate B bootstrap resamples. This section shows how to use the BY statement (and, if desired, the FREQ statement) of a SAS procedure to generate the bootstrap distribution of a statistic.

This section duplicates the example in Section 14.2.2 by creating a bootstrap distribution for the mean statistic. However, instead of computing the means of each resample in the SAS/IML language, this section uses the MEANS procedure to do the same analysis.

Assume that the data set Sample contains the Budget variable for a subset of movies, as described in Section 14.2.2. The following program uses the SURVEYSELECT procedure to generate bootstrap resamples and uses the MEANS procedure to generate the bootstrap distribution for the mean statistic:

```

/* use SURVEYSELECT to resample; use PROC to compute statistics */
DSName = "Sample";                      /* 1 */
VarName = "Budget";
B = 1000;

submit DSName VarName B;                 /* 2 */
/* generate bootstrap resamples */
proc surveyselect data=&DSName out=BootSamp noprint
    seed=12345 method=urs rep= &B
    rate = 1;                           /* 3 */
run;

/* use procedure to compute statistic on each resample */
proc means data=BootSamp noprint;        /* 4 */
by Replicate;                          /* a */
freq NumberHits;                       /* b */
var &VarName;
output out=BootDist mean=s;            /* c */
run;
endsubmit;

```

The following list describes the main steps of the program:

1. Define SAS/IML variables that contain parameters for the analysis. These include the name of the input data set, the name of the variable to use in the bootstrap analysis, and the number of bootstrap resamples to generate.
2. Pass the parameters to SAS procedures by listing the names of the SAS/IML variables on the SUBMIT statement.
3. Generate the bootstrap resamples by calling the SURVEYSELECT procedure. The options for the procedure are the same as in the previous section, except that the program uses the RATE= option instead of the N= option. The option RATE=1 specifies that each resample should contain the same number of observations as the input data set, but does not require that you specify how many observations are in the input data set. Notice that the OUTHITS option is omitted from the PROC SURVEYSELECT statement.
4. The MEANS procedure computes the mean of the Budget variable for each resample.
 - a) The BY statement specifies that the *B* resamples correspond to values of the Replicate variable.
 - b) The FREQ statement specifies that the NumberHits variable contains the frequency of each value of the Budget variable.
 - c) The mean for each BY group is saved in the BootDist data set, in the variable *s*.

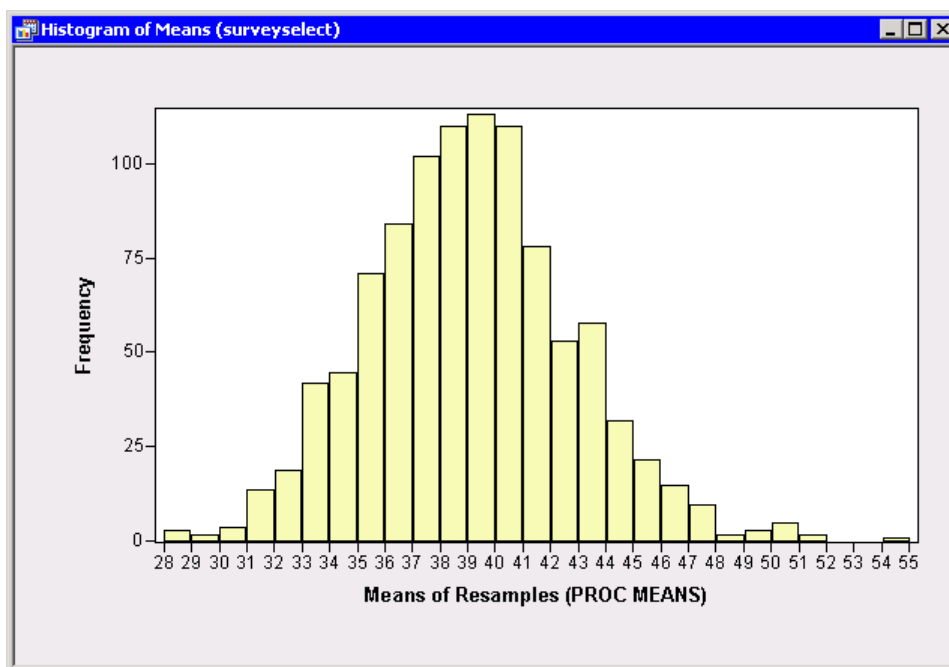
Programming Tip: You can use the BY statement in SAS procedures to compute a statistic on each bootstrap resample.

You can visualize the bootstrap distribution by using IMLPlus graphics, as shown in the following statements and in Figure 14.10:

```
/* create histogram of bootstrap distribution */
use BootDist;
read all var {s};
close BootDist;

declare Histogram hBoot;
hBoot = Histogram.Create("Means", s);
hBoot.SetAxisLabel(XAXIS,"Means of Resamples (PROC MEANS)");
hBoot.ReBin(40, 1);
```

Figure 14.10 Estimate of the Sampling Distribution of the Mean



You can compare Figure 14.10, which was computed by calling the SURVEYSELECT and MEANS procedures, with Figure 14.4, which was computed entirely in the SAS/IML language. The distributions are essentially the same. The small differences are due to the fact that the random numbers that were used to generate the resamples in SAS/IML are different from the random numbers used to generate the resamples in the SURVEYSELECT procedure.

14.5 Case Study: Bootstrap Principal Component Statistics

The bootstrap examples in this chapter so far have been fairly simple: the statistic was either a mean or the difference of means. This section uses the bootstrap method to estimate confidence intervals for a statistic that occurs as part of a principal component analysis.

The starting point for the bootstrap example is a principal component analysis for several variables in the Sample subset of the Movies data set. The following program calls the PRINCOMP procedure to compute a principal component analysis of the variables Budget, US_Gross, Sex, Violence, and Profanity:

```
/* compute principal component analysis of data */
DSName = "Sample";
VarNames = {"Budget" "US_Gross" "Sex" "Violence" "Profanity"};

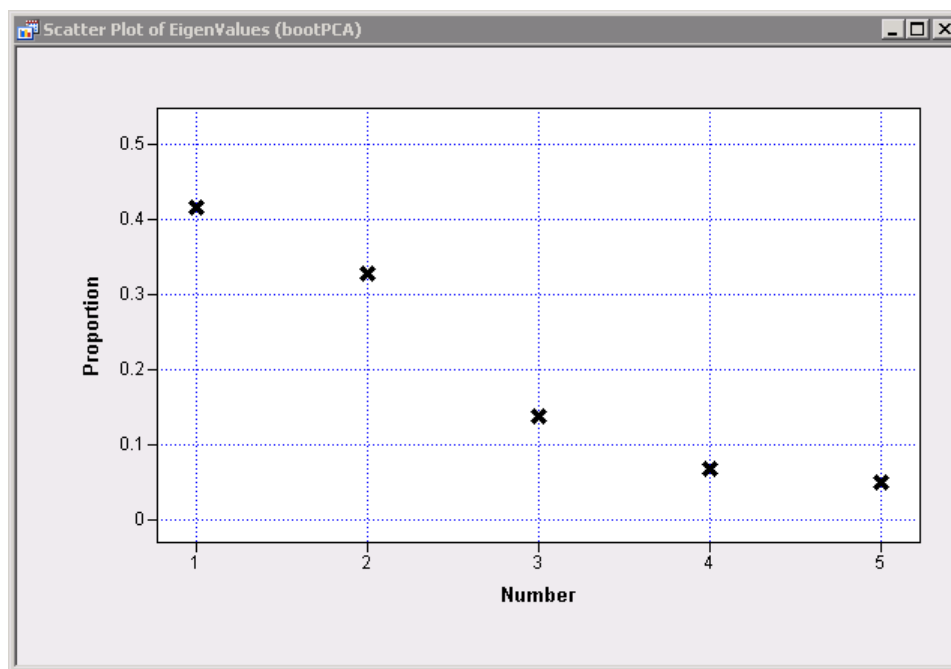
submit DSName VarNames;
ods select NObsNVar EigenValues Eigenvectors;
proc princomp data=&DSName;
    var &VarNames;
    ods output Eigenvalues=EigenValues;
run;
endsubmit;
```

Figure 14.11 Principal Component Analysis

The PRINCOMP Procedure				
Observations		83		
Variables		5		
Eigenvalues of the Correlation Matrix				
	Eigenvalue	Difference	Proportion	Cumulative
1	2.08094154	0.44170278	0.4162	0.4162
2	1.63923876	0.94566916	0.3278	0.7440
3	0.69356960	0.35620697	0.1387	0.8827
4	0.33736262	0.08847516	0.0675	0.9502
5	0.24888747		0.0498	1.0000

Let P_i be the proportion of variance explained (PVE) by the i th principal component, $i = 1 \dots 5$. This quantity is shown in the third column of the “Eigenvalues” table, shown in Figure 14.11. According to Figure 14.11, the first principal component explains 41.6% of the variance. But how much uncertainty is in that estimate? Is a 95% confidence interval for the PVE small (such as [0.414, 0.418]) or large (such as [0.32, 0.52])? There is no analytic formula for estimating the confidence intervals for the PVE, but bootstrap methods provide a computational way to estimate the sampling distribution of the P_i .

A plot of the P_i versus the component number, i , is called a *scree plot*. A scree plot for these data is shown in Figure 14.12. The goal of this section is to use bootstrap methods to estimate the variation in the Y coordinates of the scree plot.

Figure 14.12 Scree Plot of Eigenvalues for PCA

The following program builds on the bootstrap method implemented in [Section 14.4.2](#). The SURVEYSELECT procedure is used to generate the bootstrap resamples, and the call to the PRINCOMP procedure that generates [Figure 14.11](#) is modified to include a BY statement that generates the sampling distribution for the PVE for each principal component.

```

/* use SURVEYSELECT to resample; use PROC to compute statistics */
B = 1000;                               /* number of bootstrap samples */

submit DSName VarNames B;
/* generate bootstrap resamples */
proc surveyselect data=&DSName out=BootSamp noprint
    seed=12345 method=urs rate=1 rep=&B;
run;

/* Compute the statistic for each bootstrap sample */
ods listing exclude all;
proc princomp data=BootSamp;
    by Replicate;
    freq NumberHits;
    var &VarNames;
    ods output Eigenvalues=BootEvals(keep=Replicate Number Proportion);
run;
ods listing exclude none;
endsubmit;

```

The sampling distribution of the PVE can be visualized in two ways: as a scree plot and as a panel of five dynamically linked histograms.

14.5.1 Plotting Confidence Intervals on a Scree Plot

You can create a scatter plot that consists of all of the B scree plots generated by the principal component analysis computed on the B resamples:

```
/* create union of scree plots for bootstrap resamples */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Work.BootEvals");
dobj.SetVarFormat("Proportion", "BEST5.");

declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "Number", "Proportion");
p.SetMarkerSize(3);
```

To that scatter plot, you can overlay the PVE for the original sample, in addition to confidence intervals for the underlying parameter in the population of movies. The following statements overlay the quantities; the result is shown in [Figure 14.14](#).

```
/* overlay statistics of the original data */
use EigenValues; /* 1. read PVE for data */
read all var {Number Proportion};
close EigenValues;
p.DrawUseDataCoordinates(); /* define coordinate system */
p.DrawMarker(Number, Proportion, MARKER_X, 7); /* plot proportions */

use BootEvals; /* 2. read stats for resamples */
read all var {Number Proportion};
close BootEvals;

NumPC = ncol(VarNames); /* number of principal components */
s = shape(Proportion, 0, NumPC); /* 3. reshape results */

mean = s[:,]; /* 4. compute mean of each column */
alpha = 0.05; /* significance level for C.I. */
prob = alpha/2 || 1-alpha/2; /* lower/upper values for quantiles */
call qntl(CI, s, prob); /* compute C.I. for each column */
print mean, CI[rowname={LCI UCI} label="Confidence Intervals"];

p.DrawSetRegion(PLOTBACKGROUND);
p.DrawSetBrushColor(0xC8C8C8); /* light gray */
dx = 0.1; /* half-width of rectangles */
do i = 1 to NumPC; /* 5. draw rectangles and mean line */
    p.DrawRectangle(i-dx, CI[1,i], i+dx, CI[2,i], true);
    p.DrawLine(i-dx, mean[i], i+dx, mean[i]);
end;
```

The program consists of the following main steps:

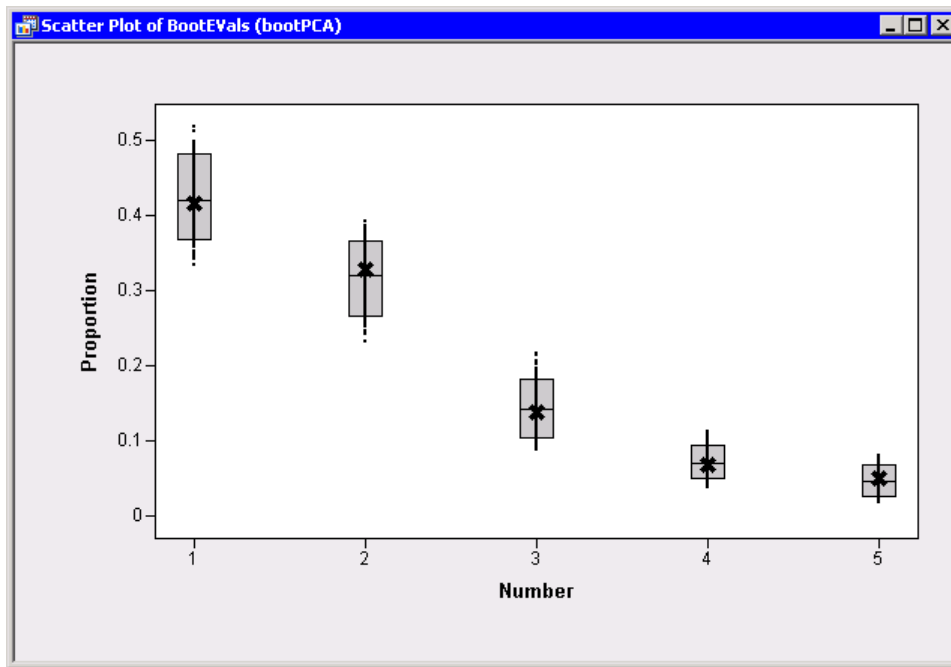
1. Read the statistics for the PVE of the original data. Those proportions are indicated on the scatter plot by large markers (x).
2. Read the `Proportion` variable that contains the PVE for each bootstrap resample. The vector `Proportion` has $5B$ elements.

3. Reshape the **Proportion** vector into a matrix with 5 columns so that the i th column represents the proportion of variance explained by the i th principal component.
4. Compute the mean and bootstrap percentile confidence interval for each column. The output is shown in Figure 14.13.
5. Plot rectangles that indicate the bootstrap percentile confidence intervals.

Figure 14.13 Principal Component Analysis

mean					
	0.4211707	0.3210511	0.1420953	0.0696367	0.0460463
Confidence Intervals					
LCI	0.3682345	0.2664716	0.1046701	0.049463	0.0264063
UCI	0.4837628	0.3663449	0.1829006	0.0947584	0.0686989

Figure 14.14 Scree Plot with Bootstrap Confidence Intervals



Notice that Figure 14.13 answers the question “how much uncertainty is in the PVE estimate for the sample data?” The answer is that there is quite a bit of uncertainty. The estimate for the proportion of variance explained by the first principal component is 0.416, but the confidence interval is fairly wide: [0.368, 0.484]. The half-width of the confidence interval is more than 10% of the size of the estimate. Similar results apply to proportion of variance explained by the other principal components.

14.5.2 Plotting the Bootstrap Distributions

If you want to view the bootstrap distributions of the PVE for each principal component, you can create a panel of histograms. The i th column of the **s** matrix (computed in the previous section) contains the data for the i th histogram. The **mean** and **CI** matrices contain the mean and 95% confidence intervals for each PVE. The following statements create a data object from the **s** matrix. For each column, the program creates a histogram and overlays the mean and bootstrap percentile confidence interval. The resulting panel of histograms is shown in [Figure 14.15](#).

```
/* create data object from bootstrap distribution;
   create histograms and overlay bootstrap mean and C.I. */
names = "Proportion1":"Proportion"+strip(char(NumPC));
declare DataObject dobj2;
dobj2 = DataObject.Create("PVE", names, s);

declare Histogram hist;
attrib = BLACK || DASHED || 1 || PLOTFOREGROUND;
do i = 1 to NumPC;
  hist = Histogram.Create(dobj2, names[i]);
  /* use a module distributed with SAS/IML Studio to compute
     the position of i_th plot in an array of 6 plots */
  run CalcPlotPosition(i, 6, left, top, width, height);
  hist.SetWindowPosition(left, top, width, height);

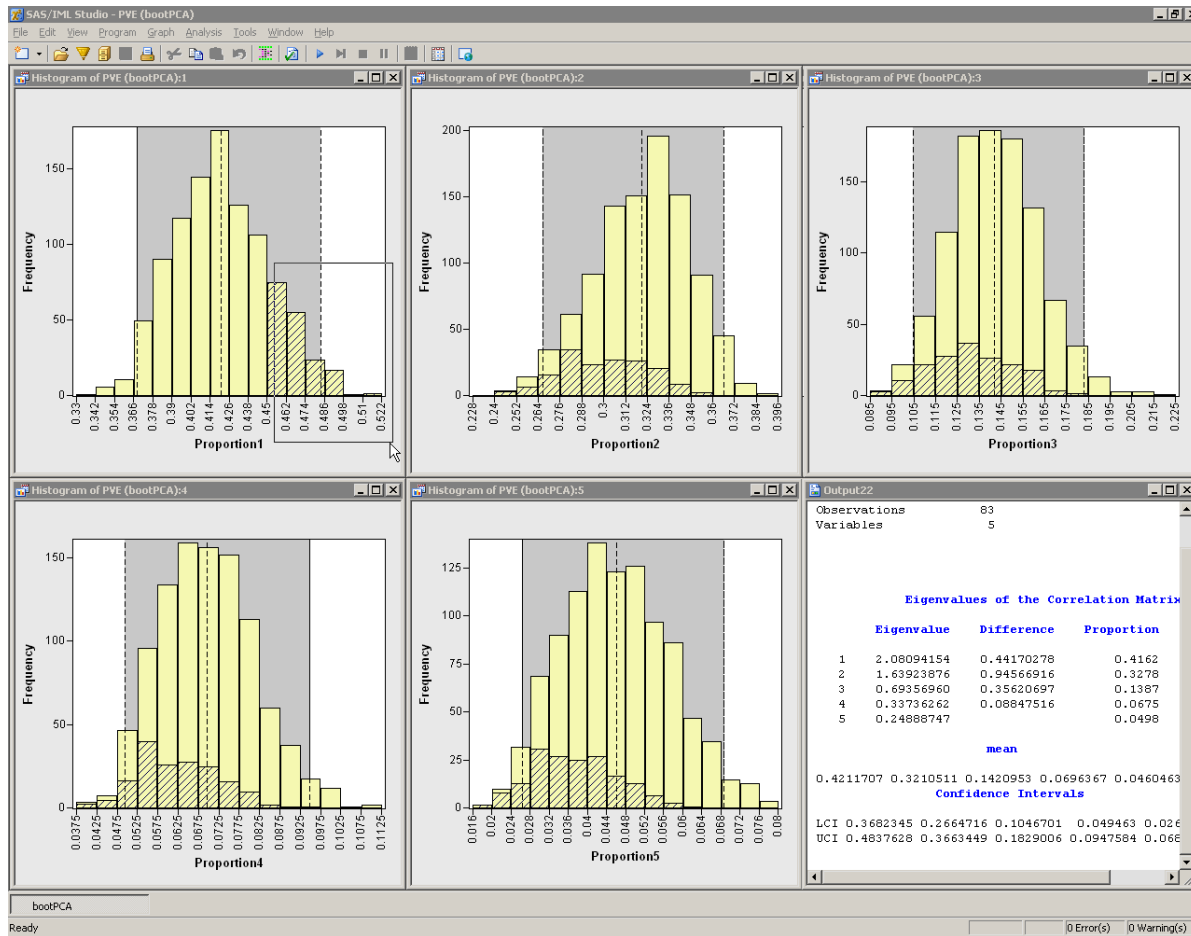
  /* plot mean and C.I. */
  run abline(hist, mean[i]//CI[i], {.,.,.}, attrib);

  /* draw rectangle in background to indicate C.I. */
  hist.DrawUseDataCoordinates();
  hist.GetAxisViewRange(YAXIS, YMin, YMax);
  hist.DrawSetRegion(PLOTBACKGROUND);
  hist.DrawSetBrushColor(0xC8C8C8);
  hist.DrawRectangle(CI[1,i], 0, CI[2,i], YMax+10, true);
end;
```

[Figure 14.15](#) shows the bootstrap distributions for the proportion of variance explained by each principal component. The distributions are slightly skewed; this is especially noticeable for Proportion2.

The figure also shows how the dynamically linked graphics in SAS/IML Studio can give additional insight into the fact that these five statistics are not independent, since the sum of the five proportions must equal unity. In the figure, large values of the Proportion1 variable are selected. The selected observations correspond to bootstrap resamples for which the value of the Proportion1 variable is large. For these bootstrap resamples, the Proportion2 variable tends to be smaller than the mean taken over all bootstrap resamples. Similar relationships hold for the other variables, and also for the bootstrap resamples for which the Proportion1 variable is small.

Figure 14.15 Panel of Histograms of Bootstrap Distributions



14.6 References

Cassell, D. L. (2007), "Don't Be Loopy: Re-Sampling and Simulation the SAS Way," in *Proceedings of the SAS Global Forum 2007 Conference*, Cary, NC: SAS Institute Inc., available at <http://www2.sas.com/proceedings/forum2007/183-2007.pdf>.

Cassell, D. L. (2010), "BootstrapMania!: Re-Sampling the SAS Way," in *Proceedings of the SAS Global Forum 2010 Conference*, Cary, NC: SAS Institute Inc., available at <http://support.sas.com/resources/papers/proceedings10/268-2010.pdf>.

Davison, A. C. and Hinkley, D. V. (1997), *Bootstrap Methods and Their Application*, Cambridge, UK: Cambridge University Press.

Efron, B. and Tibshirani, R. (1993), *An Introduction to the Bootstrap*, New York: Chapman & Hall.

Wicklin, R. (2008), "SAS Stat Studio: A Programming Environment for High-End Data Analysts," in *Proceedings of the SAS Global Forum 2008 Conference*, Cary, NC: SAS Institute Inc., available at <http://www2.sas.com/proceedings/forum2008/362-2008.pdf>.

Chapter 15

Timing Computations and the Performance of Algorithms

Contents

15.1	Overview of Timing Computations	371
15.2	Timing a Computation	372
15.3	Comparing the Performance of Algorithms	375
15.3.1	Two Algorithms That Delete Missing Values	375
15.3.2	Performance as the Size of the Data Varies	377
15.3.3	Performance as Characteristics of the Data Vary	377
15.4	Replicating Timings: Measuring Mean Performance	379
15.5	Timing Algorithms in PROC IML	383
15.6	Tips for Timing Algorithms	384
15.7	References	384

15.1 Overview of Timing Computations

In a complex language such the SAS/IML language, there is often more than one way to accomplish a given task. Furthermore, in the statistical literature there are often competing algorithms for computing the same statistic. In choosing which algorithm to implement, you can consider various characteristics:

- the time and effort required to implement each algorithm
- the memory requirements for each algorithm
- the performance of the algorithm

In this chapter, the *performance* of an algorithm means the time required for it to run on typical data, and also how that time changes with the size or characteristics of the data. For many cases, the memory requirements are either unimportant or are equivalent. Therefore, given two or more algorithms implemented in the SAS/IML language that perform the same task, it is useful to know how to measure the performance of the algorithms.

15.2 Timing a Computation

If a program is taking more time to run than you think it should, you can measure the time required by various portions of the program. This is sometimes called *profiling* the program. Suppose you have written a program that generates a random 1000×1000 matrix, inverts it, and uses the inverse to solve a linear system, as shown in the following statements:

```
/* use the INV function to solve a linear system; not efficient */
n = 1000;                      /* size of matrix          */
x = rannor(j(n, n, 1));        /* n x n matrix       */
y = rannor(j(n, 1, 1));        /* n x 1 vector       */

xInv = inv(x);                 /* compute inverse of x */
b = xInv*y;                    /* solve linear equation x*b=y */
```

It takes several seconds to run this program, so you might wonder where the bulk of the time is being spent: allocating the matrices with random numbers or solving the linear system. The key to timing a computation is to use the TIME function in Base SAS software. The TIME function returns the time of day as a SAS time value, which is the number of seconds since midnight. Therefore, if you call the TIME function immediately before and after a computation, the difference between those two times is the elapsed time (in seconds) required for the computation. The following statements employ this technique to time each portion of the computation:

```
/* time portions of the program to identify where time is spent */
n = 1000;                      /* size of matrix          */

/* measure time spent generating matrices */
t0 = time();                   /* begin timing RANNOR     */
x = rannor(j(n, n, 1));        /* n x n matrix           */
y = rannor(j(n, 1, 1));        /* n x 1 vector           */
t1 = time() - t0;              /* end timing              */
print "Elapsed time RANNOR (n=1000):" t1;

/* measure time spent solving linear system with the INV function */
t0 = time();                   /* begin timing INV        */
xInv = inv(x);                 /* compute inverse of x    */
b = xInv*y;                    /* solve linear equation x*b=y */
t2 = time() - t0;              /* end timing              */
print "Elapsed time INV (n=1000):" t2;
```

Figure 15.1 Times Required by Two Parts of a Program

	t1
Elapsed time RANNOR (n=1000):	0.282
	t2
Elapsed time INV (n=1000):	3.75

Figure 15.1 shows that the time required to allocate the matrices and fill them with random numbers is small compared with the time required to solve an $n \times n$ linear system for $n = 1000$. Therefore, if you want to improve the performance of this program, you should focus your attention on the statements that solve the linear system.

Programming Tip: To improve the performance of a program, use the TIME function to profile the program. After you identify the blocks of code that use the most time, you can try to improve the performance of those blocks.

After you identify the portion of the program that takes the most time, you can focus on optimizing that portion. Sometimes you can improve performance by rewriting the way that you implement an algorithm; other times you might decide to change algorithms. In the previous program, you can replace the INV function with a call to the SOLVE function to dramatically decrease the running time, as shown in Figure 15.2:

```
/* measure time spent solving linear system with the SOLVE function */
t0 = time();                      /* begin timing SOLVE          */
b = solve(x, y);                  /* solve linear equation directly */
t3 = time() - t0;                 /* end timing                  */
print "Elapsed time SOLVE (n=1000):" t3;
```

Figure 15.2 Time Required to Solve a Linear System with the SOLVE Function

	t3
Elapsed time SOLVE (n=1000):	1.312

Figure 15.2 shows that the SOLVE function solves the linear system in about 40% of the time required for the INV function.

Programming Technique: The following statements show how you can use the TIME function to determine the length of time for a computation:

```
t0 = time();                      /* begin timing */
/* --- put the computation here --- */
ElapsedTime = time() - t0;        /* end timing   */
```

You can use this technique to determine the length of time for most computations. Under the Windows operating system, this technique is limited to timing computations that take more than 15 milliseconds. This is the frequency at which the Windows operating system updates a certain time stamp (see Nilsson (2004) and references therein). Consequently, on Windows operating systems the TIME function returns the time of day to the nearest 15 milliseconds.

For example, you can try to time how long it takes to solve a linear system with 50 rows and columns by setting $n=50$ in the previous program. The resulting output is shown in Figure 15.3:

Figure 15.3 Time to Solve a 50×50 Linear System

	t3
Elapsed time SOLVE (n=50):	0

Because the TIME function measures the time of day to the nearest 15 milliseconds, if a computation finishes too quickly, the interval between calls to the TIME function is too short to measure, as shown in Figure 15.2. A solution to this problem is to repeat the computation many times (say, 1000 times) so that the total elapsed time is large enough to measure, as shown in the following statements:

```

/* measure time spent solving linear system 1000 times */
n = 50;                               /* size of matrix */
x = rannor(j(n, n, 1));                /* n x n matrix */
y = rannor(j(n, 1, 1));                /* n x 1 vector */

t0 = time();                           /* begin timing */
do i = 1 to 1000;                      /* repeat computations many times */
    b = solve(x, y);                   /* solve linear equation directly */
end;
t3 = time() - t0;                       /* end timing */
print "Elapsed time 1000 SOLVE (n=50):" t3;
print "Average time for SOLVE (n=50):" (t3/1000);

```

Figure 15.4 Time Required to Solve a 50×50 Linear System 1000 Times

	t3
Elapsed time 1000 SOLVE (n=50):	0.141
Average time for SOLVE (n=50):	0.000141

The elapsed time for solving a 50×50 linear system 1000 times is shown in Figure 15.4. If necessary, you can determine the average time required for each solution by dividing the cumulative times by 1000.

Programming Tip: When timing a computation that does not take very long (for example, less than 0.1 seconds), repeat the computation many times and measure the total elapsed time. The average time more accurately represents the true time required for each computation.

15.3 Comparing the Performance of Algorithms

Some algorithms are more efficient than others. For example, the previous section showed that less time is required to solve a linear equation by using the SOLVE function than by using the INV function. You can use this same timing technique to compare the performance among different algorithms. For example, Chapter 13, “[Sampling and Simulation](#),” describes several algorithms for sampling and simulation, and mentions that some are faster than others.

This section describes how to compare two or more algorithms and how to investigate how the algorithms perform as you modify characteristics of the data in a systematic manner. However, you should keep in mind that sometimes there is a trade-off between speed and accuracy or between speed and ease-of-programming. How fast an algorithm runs is only one criterion in determining the best way to write a statistical program.

15.3.1 Two Algorithms That Delete Missing Values

Section 3.3.4 describes how to remove missing values from a SAS/IML vector. The SAS/IML program that creates [Figure 3.11](#) consists of two main steps: find observations in a vector that are not missing, and use the subscript operator to extract those values into a smaller vector. However, there is an alternative way to form a vector of the nonmissing values: find the observations that *are* missing, and use the REMOVE function to create the vector that results from deleting them.

The following statements define two SAS/IML modules that perform the same task: given a vector \mathbf{x} of length n that contains k missing values, return the vector, \mathbf{y} , of length $n - k$ that results from deleting the missing values from \mathbf{x} . These modules will be used to illustrate how to time algorithms and profile performance.

```
/* Remove missing values from a vector. */
/* Investigate the relative speeds of each algorithm. */
/* Algorithm 1: use subscripts to remove missing values */
start DeleteMissBySubscript(x);
  do i = 1 to 1000;          /* repeat computation 1000 times */
    idx = loc(x^=.);        /* find elements NOT missing */
    if ncol(idx)>0 then
      y = x[idx];           /* extract them */
  end;
  return ( y );
finish;

/* Algorithm 2: use the REMOVE function to remove missing values */
start DeleteMissByRemove(x);
  do i = 1 to 1000;          /* repeat computation 1000 times */
    idx = loc(x=.);         /* find elements that ARE missing */
    if ncol(idx)>0 then
      y = remove(x, idx);   /* remove them; assign what is left */
  end;
  return ( y );
finish;
```

Because the subscript operator and the REMOVE function are so fast, each module performs the computation 1000 times before returning. Consequently, if you pass in a sufficiently large data vector to these modules, the time required for each module call should be greater than 0.1 seconds. (If not, increase the number of iterations in the modules.) Although these modules are not very sophisticated, they are useful in describing how to compare the performance of algorithms. The algorithms are simple, easy-to-understand, and do not take very long to run, yet analyzing them illustrates basic ideas that can be applied to analyzing more complicated statistical algorithms.

The statements below create a vector, \mathbf{x} , of length $n = 10,000$ and assigns missing values to the first 5% of the elements. This vector is passed into the two modules, and times are recorded for each module call. The times are shown in Figure 15.5.

```

/* compare time required by each algorithm (5% missing values) */
n   = 1e4;                      /* length of data vector      */
x = j(n, 1, 1);                 /* constant data vector      */
pct = 0.05;                     /* percentage of missing values */

NumMissing = ceil(n*pct);        /* number of missing values   */
x[1:NumMissing] = .;            /* assign missing values      */

print pct[r="PCT Missing:"];
t0 = time();                    /* begin timing               */
y = DeleteMissBySubscript(x);    /* run the first algorithm    */
t1 = time() - t0;               /* end timing                 */
print t1[r="Elapsed time SUBSCRIPT:"];

t0 = time();                    /* begin timing               */
y = DeleteMissByRemove(x);       /* run the second algorithm   */
t2 = time() - t0;               /* end timing                 */
print t2[r="Elapsed time REMOVE:"];

```

Figure 15.5 Times Required for Two Algorithms

pct	
PCT Missing:	0.05
t1	
Elapsed time SUBSCRIPT:	0.453
t2	
Elapsed time REMOVE:	0.25

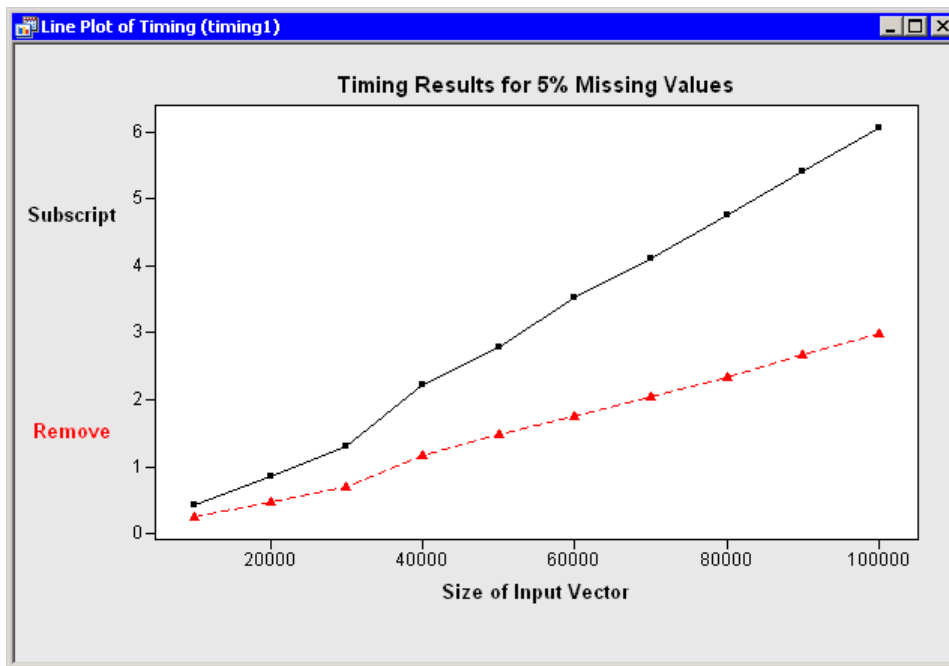
It appears from Figure 15.5 that using the REMOVE function is faster than using the subscript operator for these data.

15.3.2 Performance as the Size of the Data Varies

A common question in statistical programming is “How does the time required by an algorithm scale with the size of the input data?” To answer this question, you can systematically increase the size of the input data and graph the time required by an algorithm as a function of the data size. The graph will indicate whether the time required by your implementation is linear in the size of the data, is quadratic, or has some other relationship.

The time required to run the `DeleteMissBySubscript` and `DeleteMissByRemove` modules certainly depends on the length of the input vector `x`: more data means more time is required to search for and delete missing values. Figure 15.6 shows the relationship between the time required by these modules as the size of the input data is varied. The statements that create Figure 15.6 are not shown, but are a simple variation of the technique that are described in the next section. As expected, the performance for each module is roughly linear in the number of observations.

Figure 15.6 Time Versus the Size of the Input Vector (5% Missing Values)



15.3.3 Performance as Characteristics of the Data Vary

Figure 15.5 shows that the `DeleteMissByRemove` algorithm is faster than the `DeleteMissBySubscript` algorithm when about 5% of the data are missing. However, does the performance of the modules also depend on the percentage of missing values in the input vector? This section examines that question.

You can modify the program in Section 15.3.1 to answer this question. Instead of running the modules a single time for a single value of the `pct` variable, you can use the `DO` function to define a list of percentages. The following program loops over a list of percentages, creates an input vector that contains the specified percentage of missing values, and then calls the modules:

```

/* compare time spent by each algorithm as a parameter is varied */
n = 1e4;                      /* length of data vector      */
x0 = j(n, 1, 1);              /* constant data vector      */

/* define list of percentages */
pctList = 0.01 || 0.05 || do(0.1, 0.9, 0.1) || 0.95;

/* allocate 3 columns for results: "PctMissing" "Subscript" "Remove" */
results = j(ncol(pctList), 3);
do k = 1 to ncol(pctList);
    pct = pctList[k];          /* percentage of missing values */
    NumMissing = ceil(n*pct);  /* number of missing values     */
    x = x0;                    /* copy original data           */
    x[1:NumMissing] = .;       /* assign missing values        */

    t0 = time();
    y = DeleteMissBySubscript(x); /* run the first algorithm      */
    t1 = time() - t0;

    t0 = time();
    y = DeleteMissByRemove(x);   /* run the second algorithm     */
    t2 = time() - t0;

    results[k,] = pct || t1 || t2;
end;

```

The **results** matrix records the times that are associated with each value of the **pctList** vector. When the iterations complete, you can print or graph the results. The following IMLPlus statements create a line plot of the results by graphing each algorithm's time versus the percentage of missing values in the input vector. The graph is shown in [Figure 15.7](#).

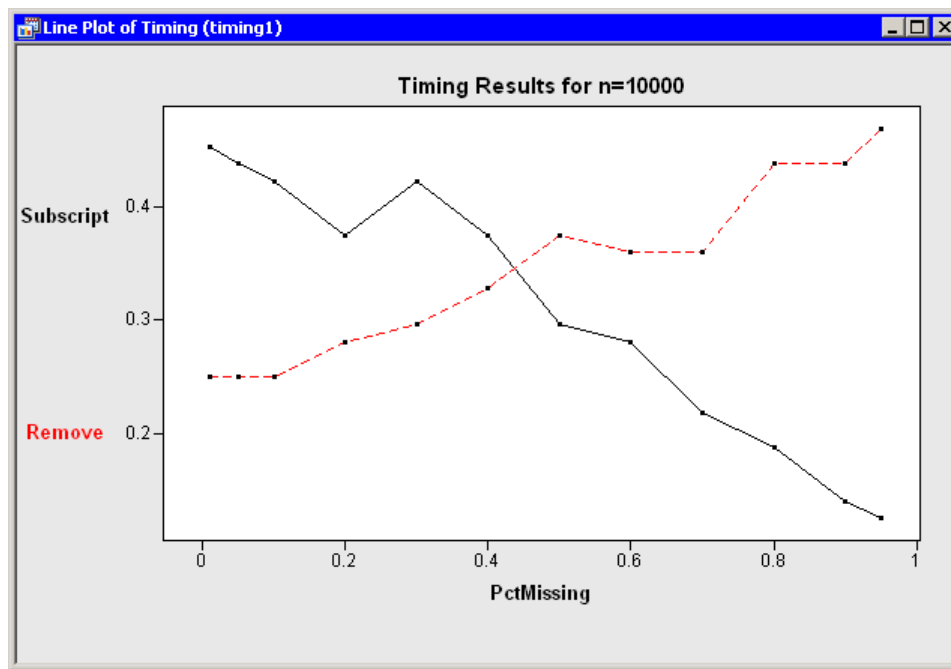
```

/* create a line plot of the timing results as a parameter is varied */
declare DataObject dobj;
dobj = DataObject.Create("Timing",
    {"PctMissing" "Subscript" "Remove"}, results);

declare LinePlot line;
line = LinePlot.Create(dobj, "PctMissing", {"Subscript" "Remove"});
line.SetTitleText("Timing Results for n="+strip(putn(n,"BEST.")), true);

```

The vertical axis on the figure is the number of seconds required by each module call. The figure shows that the performance of the two algorithms is very different as you vary the percentage of missing values in the input vector. The time required by the `DeleteMissBySubscript` is negatively correlated with the percentage of missing values, whereas the `DeleteMissByRemove` module is positively correlated. Both relationships can be modeled as linear, assuming that the bumps in [Figure 15.7](#) indicate the presence of noise in measuring the times. When the percentage of missing values is small, the `DeleteMissByRemove` module is faster than the `DeleteMissBySubscript` module; when the percentage is large, the opposite is true.

Figure 15.7 Time for Two Algorithms Versus the Percentage of Missing Values

Graphs such as [Figure 15.7](#) can help you determine which algorithm is best for a particular application. If you know *a priori* that your data contain a small percentage of missing values, then you might decide to choose the DeleteMissByRemove module. Alternatively, you might look at the scale of the vertical axis and decide that the total time required for either algorithm is so small that it does not matter which algorithm you choose.

Programming Tip: Use a line plot to compare the timing of each algorithm as a single parameter is varied.

15.4 Replicating Timings: Measuring Mean Performance

Every time you run the program that creates [Figure 15.7](#), you will get a slightly different graph. At first this might seem surprising. After all, there are no random numbers being used in the computation; the numerical computations are exactly the same every time you run the program. Nevertheless, if you run the program again on the same computer, you will observe different times.

The resolution to this paradox is to recall that the operating system of a modern computer enables many applications to run “simultaneously.” However, the applications are not really running at the same time. Instead the operating system dynamically switches between the applications, spending a fraction of a second processing the computational needs of one application before moving on to the next.

When you run a program such as the one that creates [Figure 15.7](#), the times that you see for each trial are only estimates of some theoretical “minimum time” that the program could achieve on a dedicated CPU. In reality, the CPU is multitasking while the program runs. The operating system might be checking to see if you have received new e-mail. Or it might be scanning for viruses or performing any one of a dozen system administration tasks that occur in the background while you are working. In short, the computer is not devoting all of its computational resources to running your SAS/IML program, and is, in fact, busy running other programs as well. This fact makes the precise timing of algorithms difficult.

This section describes one way to deal with the imprecise measurements: repeat the measurements several times so that you obtain multiple times associated with each trial. Then you can use statistical models to predict the mean time for each trial.

The basic idea is simple: in order to obtain multiple times for each trial, put a loop around the program in the previous section so that it runs several times. In practice, it is more efficient to put the new loop around the calls to the modules, since you can therefore reuse the input vector for each call. This idea is implemented by the following statements, which run each trial five times:

```
/* time each algorithm multiple times as a parameter is varied */
NReplicates = 5; /* run each trial several times */
n = 1e4; /* length of data vector */
x0 = j(n, 1, 1); /* constant data vector */

/* define list of percentages */
pctList = 0.01 || 0.05 || do(0.1, 0.9, 0.1) || 0.95;

NTrials = NReplicates * ncol(PctList);
/* allocate 3 columns for results: "PctMissing" "Subscript" "Remove" */
results = j(2*NTrials, 3);
cnt = 1; /* index to store results */
do k = 1 to ncol(pctList);
  pct = pctList[k]; /* percentage of missing values */
  NumMissing = ceil(n*pct); /* number of missing values */
  x = x0; /* copy original data */
  x[1:NumMissing] = .; /* assign missing values */

  do repl = 1 to NReplicates; /* repeat multiple times */
    t0 = time();
    y = DeleteMissBySubscript(x); /* run the first algorithm */
    t1 = time() - t0;
    results[cnt,] = pct || repl || t1;
    cnt = cnt+1;

    t0 = time();
    y = DeleteMissByRemove(x); /* run the second algorithm */
    t2 = time() - t0;
    results[cnt,] = pct || repl || t2;
    cnt = cnt+1;
  end;
end;
```

The **results** matrix contains the timings for both modules: the times for the DeleteMissBySubscript module are contained in the odd rows, whereas the times for the DeleteMissByRemove

module are contained in the even rows. The following statements create a data object from these data and add a variable that indicates the algorithm. The markers for the even rows are set to red triangles.

```
/* create scatter plot of Time versus PctMissing for each trial */
declare DataObject dobj;
dobj = DataObject.Create("Timing",
    {"PctMissing" "Replicate" "Time"}, results);
algorithm = {"Subscript", "Remove"}; /* odd rows are "Subscript" */
algVector = repeat(algorithm, NTrials); /* even rows are "Remove" */
dobj.AddVar("Algorithm", algVector); /* add this variable to data */

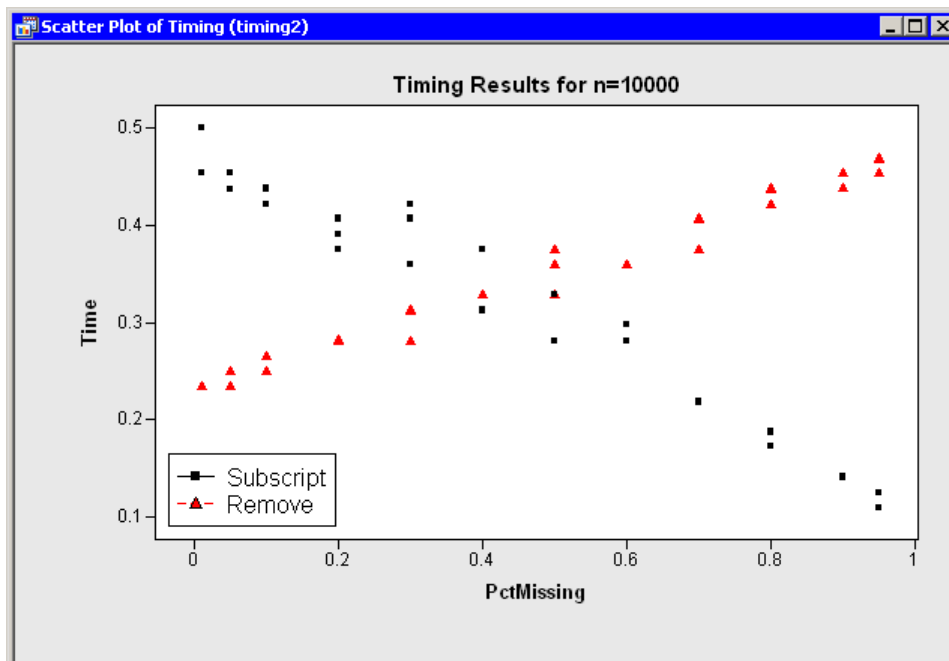
evenIdx = do(2, 2*NTrials,2); /* generate even row numbers */
dobj.SetMarkerColor(evenIdx, RED); /* color them red and... */
dobj.SetMarkerShape(evenIdx, MARKER_TRIANGLE); /* mark as triangles */
```

The data were purposely structured so that the timings for both modules can be displayed in a single scatter plot of the Time variable versus the PctMissing variable, as shown in Figure 15.8, which is created by the following statements:

```
declare ScatterPlot p;
p = ScatterPlot.Create(dobj, "PctMissing", "Time");
p.SetMarkerSize(5);
p.SetTitleText("Timing Results for n="+strip(putn(n, "BEST.")), true);

/* add a legend */
color = BLACK || RED;
style = SOLID || DASHED;
shape = MARKER_SQUARE || MARKER_TRIANGLE;
run DrawLegend(p, algorithm, 12, color, style, shape, -1, "ILB");
```

Figure 15.8 Timing Data with Replicates



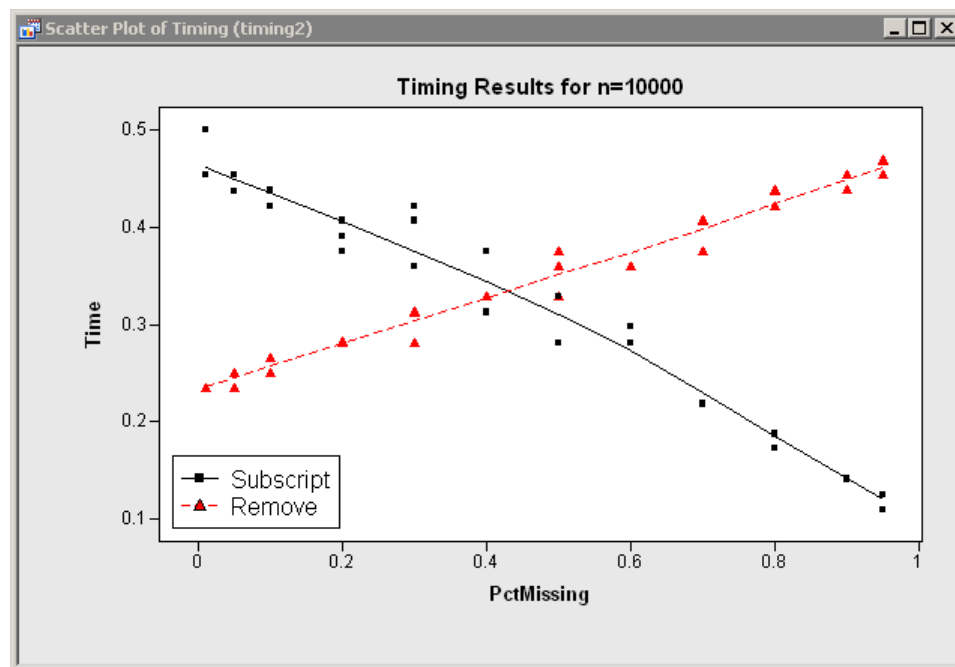
Notice that the replicated timings are consistent with the timings shown in Figure 15.7, but the individual timings vary by up to 0.05 seconds from one another.

If you want to model the mean time as a function of the parameter (the percentage of missing values), you can compute a scatter plot smoother for each algorithm. SAS/IML software provides several smoothers such as a cubic spline smoother (the SPLINE subroutine) and a thin-plate spline smoother (the TPSPLINE subroutine). In addition, you can call SAS/STAT procedures such as the LOESS and GAM procedures from IMLPlus. The following statements call the TPSPLINE subroutine to compute a smoother for the timings for each algorithm. The resulting curves are overlaid on the scatter plot and shown in Figure 15.9:

```
/* compute a scatter plot smoother: a thin-plate smoothing spline */
p.DrawUseDataCoordinates();
lambda = -2;                                /* parameter for TPSPLINE */
do i = 1 to nrow(algorithm);                 /* for each algorithm */
    idx = loc(algVector=algorithm[i]);
    x = results[idx, 1];
    y = results[idx, 3];                      /* extract the data */
    call tpspline(fit, coef, adia, gcv, x, y, lambda);

    p.DrawSetPenAttributes(color[i], style[i], 1);
    p.DrawLine(x, fit);                      /* draw the smoother */
end;
```

Figure 15.9 Smoothers Added to Timing Data



The TPSPLINE function computes the unique design points (that is, the unique values of PctMissing) and uses the mean value of each design point to fit the spline smoother. The figure shows that the performance of the DeleteMissBySubscript module is almost linear and decreases with the percentage of missing values. In contrast, the time used by the DeleteMissByRemove module tends to increase with the percentage of missing values in the input vector.

Programming Tip: Use a scatter plot to graph the timing information from multiple runs of each algorithm as you vary a single parameter. You can use a scatter plot smoother to better visualize the trend in the replicates.

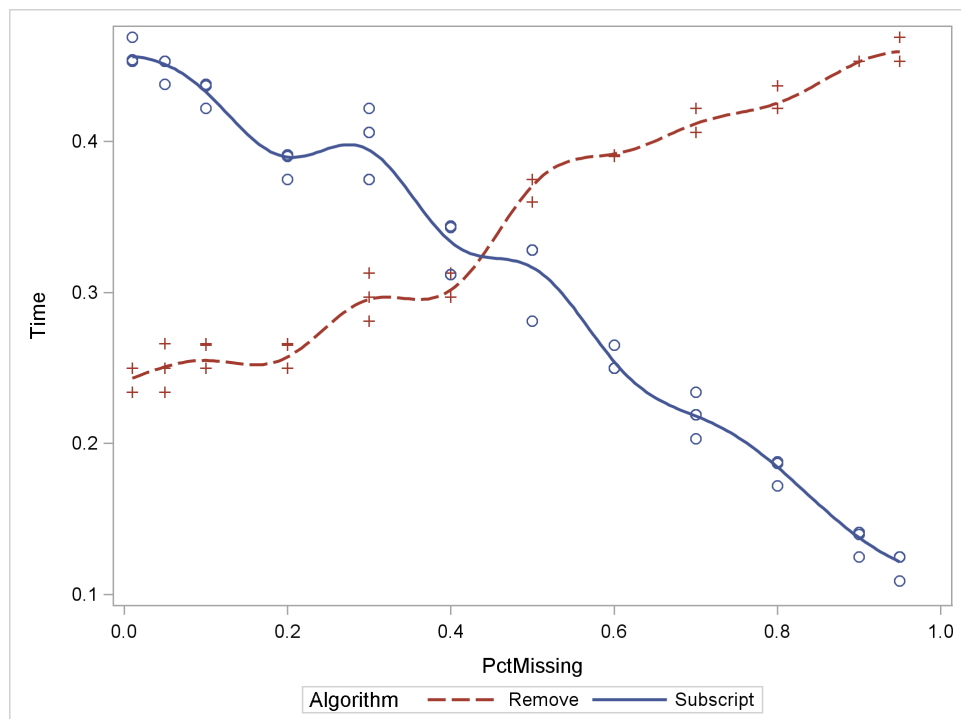
15.5 Timing Algorithms in PROC IML

All of the numerical techniques in this chapter are valid for measuring the performance of algorithms in PROC IML. The only difference is that the IMLPlus graphics shown in this chapter do not run in PROC IML. However, you can write the data to a SAS data set and use the SGPLOT procedure to visualize the results.

For example, to create Figure 15.9, you can use the CREATE and APPEND statements in SAS/IML to write the timing data to a data set named DeleteMissing. You can then call the SGPLOT procedure on these data. Although the SGPLOT procedure does not support thin-plate spline smoothers, you can use the PBSPLINE statement to fit a penalized B-spline through the data. The following statements demonstrate one way to visualize these data outside of SAS/IML Studio. Figure 15.10 shows the graph that is produced.

```
proc sgplot data=DeleteMissing;
  pbspline x=PctMissing y=Time / group=Algorithm;
run;
```

Figure 15.10 ODS Graphics Plot of Smoothers and Data



15.6 Tips for Timing Algorithms

The following tips can help you to make accurate assessments of the performance of algorithms:

- Encapsulate the algorithm into a module in order to make it easier to call and to change parameters in the algorithm.
- Use small quantities of data when you are developing and debugging your program.
- Maintain a quiet machine when timing the algorithm. Close down e-mail and your Web browser. Do not start or run other applications during the timing (unless you purposely want to examine how the algorithm timing varies under realistic conditions.)
- For long-running algorithms (say, greater than 15 seconds), it is not necessary to repeat each timing multiple times as described in [Section 15.4](#). The variation in times will usually be small compared with the time required by the algorithm.
- Notice that the nonparametric model used in [Section 15.4](#) is a thin-plate spline, and not a loess model. The loess model is not a good choice for modeling replicated measurements because of the way the loess algorithm chooses points to use for local estimation.
- You can wrap the TIME function around a SUBMIT statement to measure the time required to call a SAS procedure or to call R functions.
- The way that an algorithm varies according to the size of the data is not always easy to understand. The performance of an algorithm will vary from CPU to CPU, depending on the size of certain memory caches, buffers, and page sizes.

15.7 References

Nilsson, J. (2004), "Implement a Continuously Updating, High-Resolution Time Provider for Windows," *MSDN Magazine*.

URL <http://msdn.microsoft.com/en-us/magazine/cc163996.aspx>

Chapter 16

Interactive Techniques

Contents

16.1 Overview of Interactive Techniques	385
16.2 Pausing a Program to Enable Interaction	385
16.3 Attaching Menus to Graphs	386
16.4 Linking Related Data	390
16.5 Dialog Boxes in SAS/IML Studio	396
16.5.1 Displaying Simple Dialog Boxes	396
16.5.2 Displaying a List in a Dialog Box	399
16.6 Creating a Dialog Box with Java	402
16.7 Creating a Dialog Box with R	404
16.7.1 The Main Idea	404
16.7.2 A First Modal Dialog Box	405
16.7.3 A Modal Dialog Box with a Checkbox	406
16.7.4 Case Study: A Modal Dialog Box for a Correlation Analysis	408
16.8 References	411

16.1 Overview of Interactive Techniques

This chapter describes IMLPlus features that enable programmers to add interactive features to IMLPlus programs. This includes the following:

- pausing a program so that the person who is running the program can interact with the data
- attaching a menu to a graph that calls a module when the menu is selected
- creating dialog boxes that prompt the user for input

16.2 Pausing a Program to Enable Interaction

The PAUSE statement is described in [Section 5.9.4](#). By default, when the PAUSE statement is executed, the Auxiliary Input window appears. However, if the text string in the PAUSE statement

begins with the word “NoDialog:”, then the Auxiliary Input window is not displayed, but the program still pauses its execution. The remainder of the message is displayed in the SAS/IML Studio status bar, which is located in the lower left corner of the SAS/IML Studio application.

When a program is paused, you can still interact with the SAS/IML Studio GUI. In particular, you can interact with plots and data tables, which means that you can select observations. Consequently, when the program resumes, the program can get the selected observations and use them in some computation. Alternatively, the program can modify characteristics of the selected observations.

Programming Tip: You can select observations in plots and data tables when an IMLPlus program is paused.

For example, the following statements create a scatter plot. The program then pauses and prompts the user to select certain observations. When the program resumes, the selected observations are colored red.

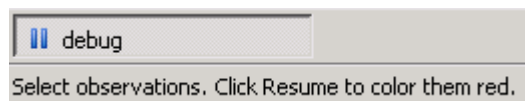
```
/* pause program to enable user to interact with the data */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Movies");

declare ScatterPlot plot;
plot = ScatterPlot.Create(dobj, "ReleaseDate", "US_Gross");

pause "NoDialog: Select observations. Click Resume to color them red.";
dobj.GetSelectedObsNumbers(SelObsIdx);
if ncol(SelObsIdx) > 0 then
    dobj.SetMarkerColor(SelObsIdx, RED);
```

The PAUSE message is displayed in the status bar, as shown in [Figure 16.1](#). When the user clicks the resume icon (▶) or presses ALT+F5, the program resumes and the selected observations are colored red.

Figure 16.1 A Message in the Status Bar



16.3 Attaching Menus to Graphs

In SAS/IML Studio, you can attach a menu to any data table or graph. You can associate IMLPlus statements with a menu item, which means that SAS/IML Studio executes those statements when you select the menu item. In this way, you can attach an analysis to a data table or a graph. The menu is called an *action menu* because it enables you to define some action that is applied to the data or to the graph.

Usually the “action” is to compute some statistic or to draw some feature on the graph. You could add a kernel density estimate to a histogram as described in [Section 4.5](#), add a loess smoother to a scatter plot as described in [Section 9.7](#), or add a rug plot to a histogram as described in [Section 9.5](#).

Programming Tip: When you create an action menu, encapsulate the “action” statements into an IMLPlus module.

This section describes how to create an action menu that is attached to a histogram. The action menu calls the RugPlot module that is described in [Section 9.5](#). This section assumes that the RugPlot module is stored in a directory on the SAS/IML Studio module search path, as described in [Section 5.7](#).

In the following example, the AppendActionItem method in the DataView class specifies the text and action for a menu item on a histogram:

```
/* add action menu to histogram */
declare Histogram hist;
hist = Histogram.Create(dobj, "Budget");
hist.AppendActionMenuItem("Draw a Rug Plot",          /* 1 */
                          "run OnRugPlot();",         /* 2 */
                          );

start OnRugPlot();                                     /* 3 */
  declare Histogram h;
  h = DataView.GetInitiator();                         /* 4 */
  run RugPlot(h);
finish;
```

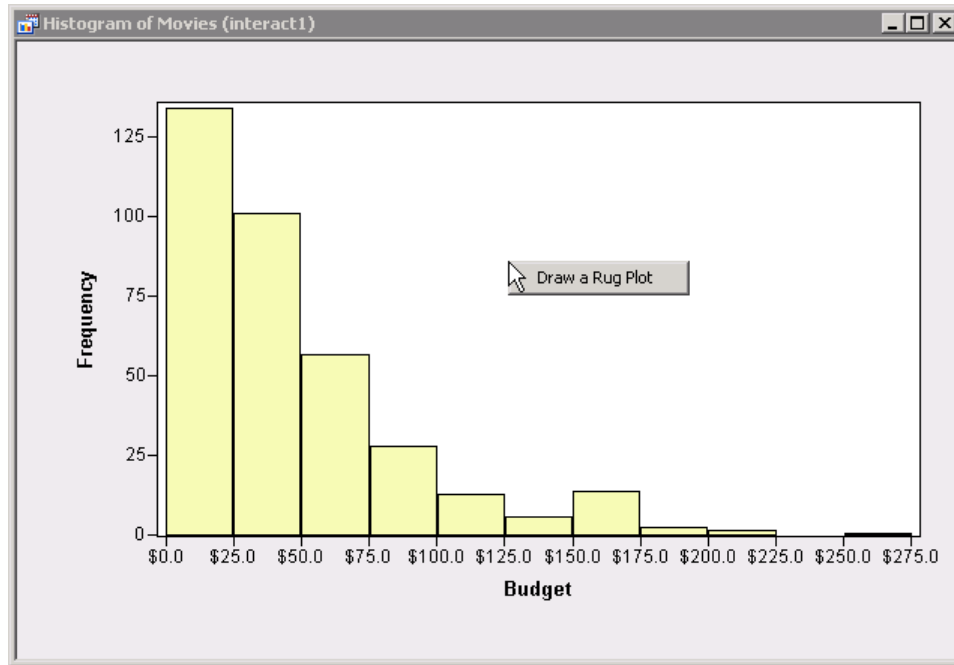
The action menu in this example consists of a single menu item. The program contains the following main steps:

1. The first string to the AppendActionMenuItem method specifies the text that appears on the action menu item, as shown in [Figure 16.2](#).
2. The second string specifies the statement that is executed when the menu item is selected. This is usually a module call. In this case, selecting the action menu item executes the statement `run OnRugPlot();`.
3. The OnRugPlot module is defined. The module does not take any arguments.
4. Inside the OnRugPlot module, the DataView.GetInitiator method is called. This method returns the graph or data table object from which the action menu item was selected. In this example, the method returns a Histogram object. That Histogram object is passed to the RugPlot module, which draws the rug plot on the histogram.

Programming Tip: You can use the DataView.GetInitiator method to obtain the graph or data table that displays an action menu item.

The action menu item is not visible until you press F11 in the window that contains the action menu. When you press F11, the action menu appears as shown in [Figure 16.2](#). When you select **Draw a Rug Plot**, the OnRugPlot module is called. This causes the rug plot to appear on the histogram, as shown in [Figure 9.18](#).

Figure 16.2 An Action Menu



Programming Tip: To display an action menu, press F11 when the window is active.

The next example extends the previous example by creating an action menu that defines two menu items. The first menu item calls a module that draws a rug plot. The second menu item calls a module that removes the rug plot from the histogram.

```
/* add action menu with two menu items to histogram */
declare Histogram hist2;
hist2 = Histogram.Create(dobj, "Budget");
hist2.AppendActionMenuItem("Draw a Rug Plot",          /* 1 */
                           "run OnRugPlot2();",
                           );
hist2.AppendActionMenuItem("Remove Rug Plot",          /* 2 */
                           "run OnRemoveRugPlot();",
                           );

start OnRugPlot2();                                     /* 3 */
  declare Histogram h;
  h = DataView.GetInitiator();
  h.DrawRemoveCommands("RugBlock");
  h.DrawBeginBlock("RugBlock");                          /* 4 */
  run RugPlot(h);
  h.DrawEndBlock();
finish;
```

```

start OnRemoveRugPlot();                               /* 5 */
    declare Histogram h;
    h = DataView.GetInitiator();
    h.DrawRemoveCommands("RugBlock");                   /* 6 */
finish;

```

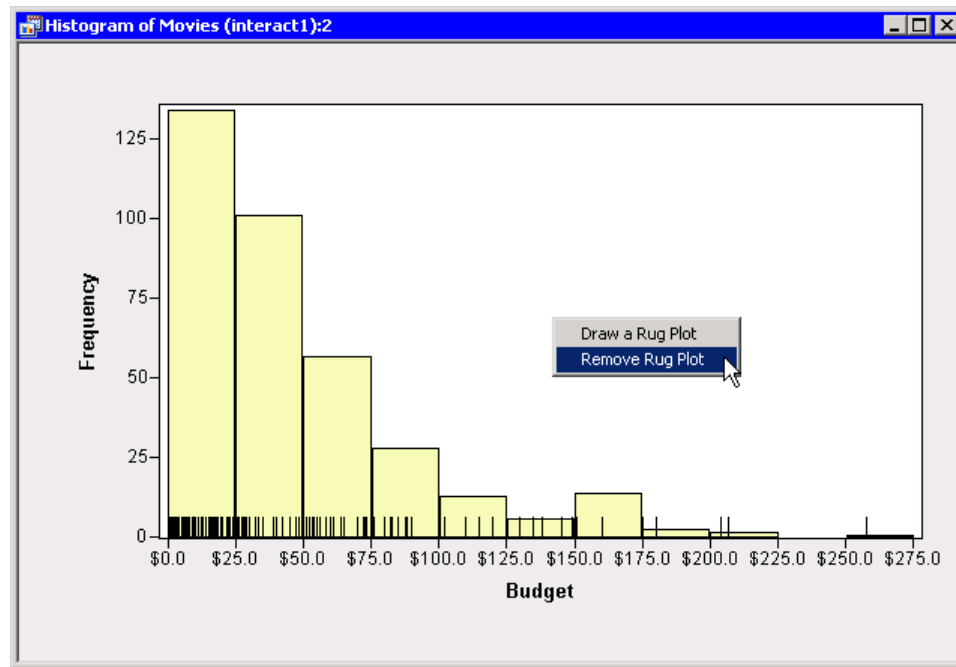
The program contains the following main steps:

1. The first call to the AppendActionMenuItem method specifies that the OnRugPlot2 module is called when the **Draw a Rug Plot** menu item is selected.
2. The second call to the AppendActionMenuItem method specifies that the OnRemoveRugPlot module is called when the **Remove Rug Plot** menu item is selected.
3. The OnRugPlot2 module is defined. The module uses the DataView.GetInitiator method to obtain the Histogram object from which the action menu item was selected.
4. Inside the module, there are three new method calls that define and name a block of drawing statements. All of the drawing methods in the RugPlot module are contained within this block. The name of the block is "RugBlock."
 - a) The DrawRemoveCommands method ensures that any previous block of statements with the same name is removed. You cannot have two blocks of statements with the same name.
 - b) The DrawBeginBlock method marks the beginning of the statement block and names the block.
 - c) The DrawEndBlock method marks the end of the statement block.

The purpose of creating this block is so that these drawing statements can be removed in the OnRemoveRugPlot module.

5. The OnRemoveRugPlot module is defined. This module also calls the DataView.GetInitiator method to obtain the Histogram object.
6. The OnRemoveRugPlot module contains a call to the DrawRemoveCommand method. This method removes all drawing associated with the "RugBlock" statements. The result is that the histogram no longer displays the tick marks that were drawn by the RugPlot module. After the call to DrawRemoveCommands, the Histogram is in the same state as before the RugPlot module was called.

You can now press F11 and select the first menu item to draw a rug plot. When you no longer want the rug plot to be displayed, you can press F11 and select the second menu item to erase the rug plot, as shown in [Figure 16.3](#).

Figure 16.3 An Action Menu That Removes the Rug Plot

You can implement many other interesting techniques with action menus. The SAS/IML Studio online Help contains several sections that describe action menus. To view the documentation, select **Help►Help Topics** from the SAS/IML Studio main menu. Action menus are described in two sections:

- Select the chapter titled “The Plots” and the section titled “The Action Menu.”
- Select the chapter titled “IMLPlus Class Reference” and the documentation for the action menu methods in the DataView class.

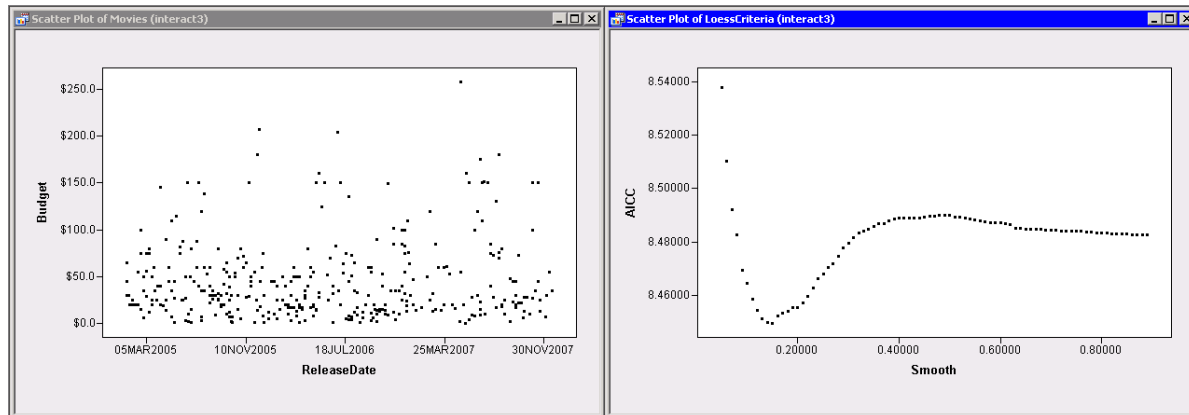
16.4 Linking Related Data

This book primarily focuses on analyzing and displaying one data set at a time. However, it is also possible to use the interactive graphics of SAS/IML Studio to analyze several related data sets in a single program. This section describes how to link graphics from two related data sets by using the action menus that are described in [Section 16.3](#).

The example in this section involves comparing different nonparametric smoothers for scatter plot data. When you fit a model to data, you often choose a single model from among a family of possible models. It is common to choose the model that optimizes some criterion that measures the agreement between the data and the model. For example, when choosing among several loess models, it is common to choose the model that minimizes the corrected Akaike’s information criterion (AICC).

Figure 16.4 shows two plots. The “observation plot” on the left shows observations from the Movies data set. The “criterion plot” on the right displays the AICC criterion versus the smoothing parameter for a series of loess models. Every point in the criterion plot represents a possible curve through the points in the observation plot. In that sense, these two plots are related to each other. However, the data for these plots reside in different data objects.

Figure 16.4 Observation Plot (left) and a Criterion Plot (right)



If you want to visualize and compare the smoothers that are associated with a particular set of smoothing parameters, you can use the following technique:

1. Create the two plots.
2. Attach an action menu to the criterion plot.
3. Select several parameter values in the criterion plot.
4. When the action menu is chosen, do the following:
 - a) Get the selected parameter values.
 - b) Graph the associated loess curves on the observation plot.

Section 16.3 shows that an action menu can be used to modify the plot that is displaying the action menu. The new idea in this section is that the criterion plot, which displays the action menu, can modify a *different* plot (the observation plot). In IMLPlus, you can *attach* the observation plot to the criterion plot by using the `AddAttachedObject` method in the `DataView` class.

Programming Technique: You can use the technique in this section to attach an object to a plot. In this way, any action menu that is associated with the plot has access to the attached object and its data.

The following statements begin this example by creating the observation plot in [Figure 16.4](#):

```
/* create a scatter plot of the observations */
DSName = "Sasuser.Movies";
XVarName = "ReleaseDate";
YVarName = "Budget";

declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet (DSName);
declare ScatterPlot p;
p = ScatterPlot.Create(dobj, XVarName, YVarName);
```

To create the criterion plot, you need to call the LOESS procedure and use the SMOOTH= option in the MODEL statement to fit models for a list of values of the smoothing parameter, as shown in the following module:

```
/* compute AICC criterion for a range of smoothing values */
start ComputeLinkedCriterionPlot(ScatterPlot CritPlot, ScatterPlot p);
  /* 1. write data to server */
  p.GetVars(ROLE_X, XVarName);
  p.GetVars(ROLE_Y, YVarName);
  declare DataObject dobj = p.GetDataObject();
  dobj.WriteVarsToServerDataSet (XVarName//YVarName, "Work", "LoessIn", true);

  /* 2. call PROC loess to compute data for criterion plot */
  submit XVarName YVarName;
  proc loess data=Work.LoessIn;
    model &YVarName = &XVarName / select = AICC
                                smooth = 0.05 to 0.9 by 0.01;
    ods output ModelSummary=LoessCriteria;
  run;
  endsubmit;

  /* 3. create data object from SmoothingCriterion table */
  declare DataObject CritDobj;
  CritDobj = DataObject.CreateFromServerDataSet("Work.LoessCriteria");
  CritPlot = ScatterPlot.Create(CritDobj, "Smooth", "AICC");

  /* 4. give the criterion plot a link to the original scatter plot */
  CritPlot.AddAttachedObject("ObsPlot", p);

  /* 5. create an action menu */
  CritPlot.AppendActionMenuItem("Draw Loess Curves for Selected Parameters",
                                "run OnDrawLoessParam();" );

finish;

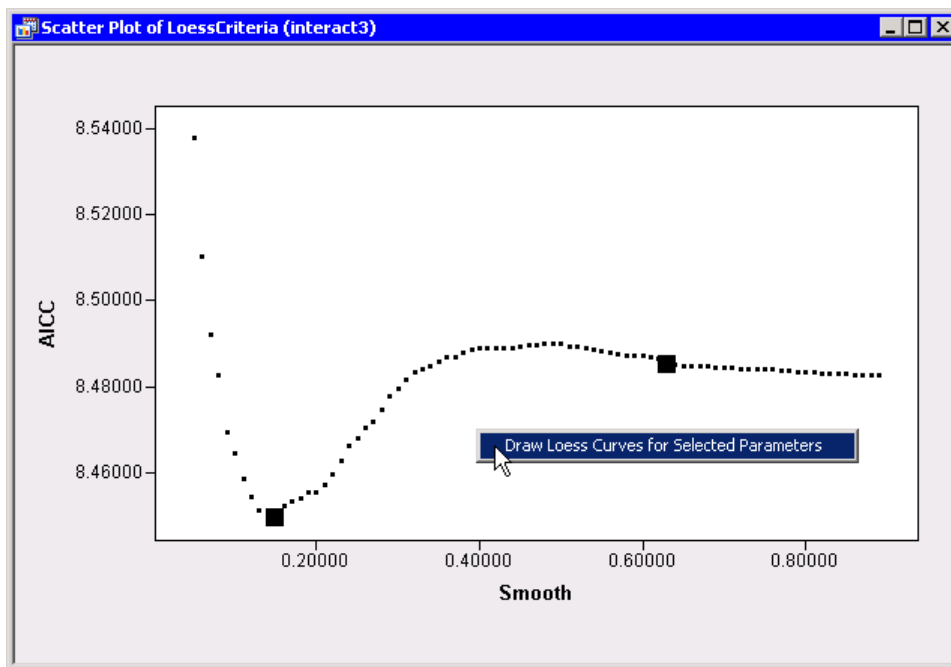
/* call the module to create the criterion plot */
declare ScatterPlot CritPlot;
run ComputeLinkedCriterionPlot(CritPlot, p);
```

The ComputeLinkedCriterionPlot module takes two arguments. The first argument is the criterion plot. The second argument is the observation plot. The criterion plot is created in the module and returned as an output parameter. The module has five main steps:

1. Get the data object that is associated with the observation plot. Write the X and Y variables to the LoessIn data set.
2. Call PROC LOESS to compute smoothers for a sequence of values of the smoothing parameter. The sequence is specified by using the SMOOTH= option in the MODEL statement. The ODS OUTPUT statement creates the LoessCriteria data set, which contains AICC values for each value of the smoothing parameter.
3. Create a data object from the LoessCriteria data set and create the criterion plot.
4. Attach the observation plot to the criterion plot. This is the new idea in this section. The AddAttachedObject method stores a reference to the observation plot object. The reference is associated with the name "ObsPlot." The plot reference can be obtained at a later time by calling the GetAttachedObject method, as is done later in Step 7.
5. Create an action menu item. When the item is selected, the OnDrawLoessParam module will be executed. (That module is not yet defined.)

A user can select parameter values in the criterion plot (by clicking on them) and then press F11 to display the action menu, as shown in Figure 16.5. When the user selects the action menu item, the action menu will call the OnDrawLoessParam module. This module needs to get the selected parameters and draw the associated loess curves on the observation plot.

Figure 16.5 Smoothing Parameters versus AICC Criterion



The OnDrawLoessParam module is defined by the following statements:

```

/* compute loess curves for selected smoothing parameters */
start OnDrawLoessParam();
  /* 6. get selected parameters */
  declare ScatterPlot CritPlot;
  CritPlot = DataView.GetInitiator();
  declare DataObject CritDobj;
  CritDobj = CritPlot.GetDataObject();
  CritDobj.GetVarSelectedData("Smooth", s);
  if ncol(s)=0 then return;

  /* 7. get linked scatter plot and names of variables */
  declare ScatterPlot p;
  p = CritPlot.GetAttachedObject("ObsPlot");
  p.GetVars(ROLE_X, XVarName);
  p.GetVars(ROLE_Y, YVarName);

  /* 8. call PROC loess to compute smoothers (assume LoessIn exists) */
  submit XVarName YVarName s;
  proc loess data=Work.LoessIn;
    model &YVarName = &XVarName / smooth = &s;
    score / ;
    ods output ScoreResults=LoessOut;
  run;

  proc sort data=LoessOut;
    by SmoothingParameter &XVarName;
  run;
  endsubmit;

  /* 9. read data for loess curves */
  PredVarName = "P_" + YVarName;
  use LoessOut;
  read all var "SmoothingParameter" into smooth;
  read all var XVarName into allX;
  read all var PredVarName into allPred;
  close LoessOut;

  /* 10. overlay loess smoothers on scatter plot */
  uSmooth = unique(smooth);
  p.DrawRemoveCommands("LoessCurves");
  p.DrawBeginBlock("LoessCurves");
  p.DrawUseDataCoordinates();
  do i = 1 to ncol(uSmooth);
    idx = loc(smooth=uSmooth[i]);
    x = allX[idx];
    pred = allPred[idx];
    p.DrawLine(x, pred);
  end;
  p.DrawEndBlock();
finish;

```

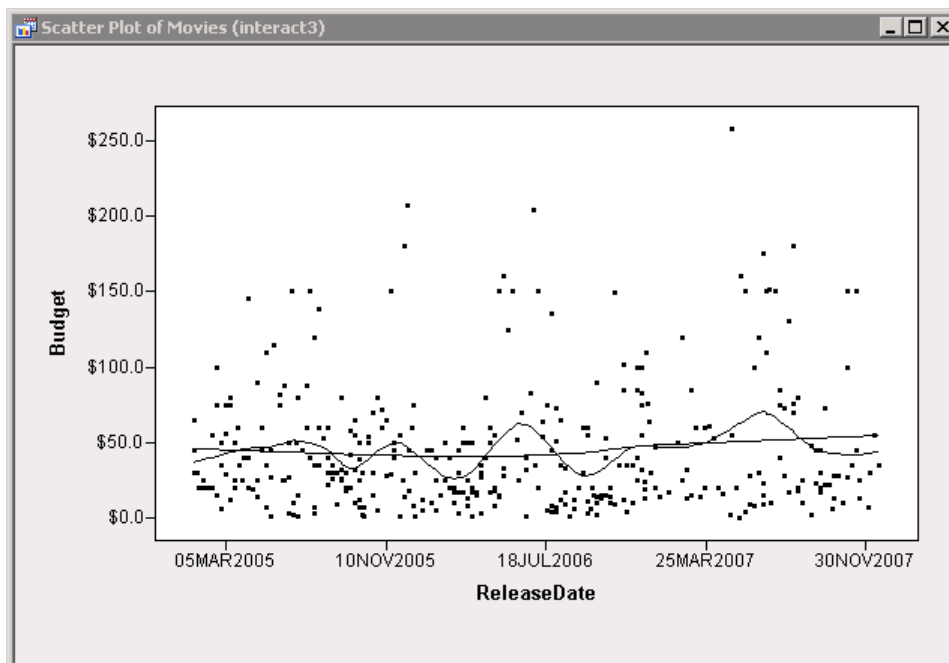
Because the module is called from an action menu, the module does not have any arguments; all

of the data that module needs is obtained from the criterion plot. The OnDrawLoessParam module consists of the following main steps:

6. The `GetInitiator` and `GetDataObject` methods enable you to get the data that underlie the criterion plot. The `GetVarSelectedData` method retrieves the smoothing parameter values that are currently selected. These values are stored in the vector `s`.
7. The `GetAttachedObject` method retrieves a reference to the observation plot. The `GetVars` method obtains the names of the X and Y variables.
8. The names of the variables and the selected smoothing values are passed as parameters to PROC LOESS. The LOESS procedure computes and scores the models for those parameters. The predicted values are saved to the `LoessOut` data set. For each value of the smoothing parameter, the SORT procedure sorts the output values by the X variable.
9. The data for the loess curves are read into SAS/IML vectors.
10. As described in [Section 16.3](#), the curves are drawn inside a named block of statements. (If the action menu is called a second time, the `DrawRemoveCommands` method removes all drawing that is associated with the first call to the action menu.) A DO loop is used to plot each loess curve on the observation plot. For each unique value of the smoothing parameter, the module extracts the relevant X and Y values and draws the curve.

Figure 16.6 shows the result of selecting the action menu for the highlighted parameters shown in Figure 16.5. The curve with many undulations corresponds to the smaller value of the smoothing parameter; the gently sloping curve corresponds to the larger value. You could improve the OnDrawLoessParam module by also drawing a legend on the observation plot. This is left as an exercise.

Figure 16.6 Loess Curves That Correspond to the Selected Parameters



This technique is useful for comparing multiple loess curves when the criterion plot has multiple local minima. It is also useful for visualizing statistics that are associated with maps. You can write a program that enables you to select a country or region and to choose an action menu that carries out an analysis on data that are associated with the selected region.

16.5 Dialog Boxes in SAS/IML Studio

SAS/IML Studio is distributed with modules that create and display several simple dialog boxes. All of these modules begin with the prefix “DoDialog” or “DoMessageBox.” You can use these dialog boxes to query for user input or to display information to the user. These dialog boxes are *modal*, meaning that the program halts execution until the dialog box is dismissed.

All of the modules (and others) in this section are documented in the SAS/IML Studio online Help, in the chapter titled “IMLPlus Module Reference.”

16.5.1 Displaying Simple Dialog Boxes

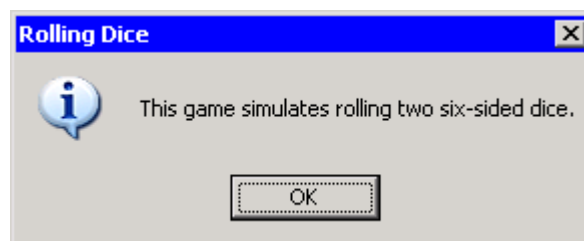
This section describes three simple dialog boxes that you can use to query for user input. The three examples in this section are part of a single program that simulates rolling two six-sided dice a user-determined number of times, and enables you to display a graph that shows the distribution of the values for the dice rolls.

The simplest dialog box is a message box. This dialog box presents the user with information and then waits until the user clicks **OK**. The following statements use the DoMessageBoxOK module to display a message box with an **OK** button:

```
title = "Rolling Dice";  
message = "This game simulates rolling two six-sided dice.";  
run DoMessageBoxOK(title, message);
```

The message box is displayed in [Figure 16.7](#). The DoMessageBoxOK has two arguments. The first is a string that appears in the title bar of the window. The second is the message that is displayed to the user.

Figure 16.7 A Message Box That Contains an **OK** Button

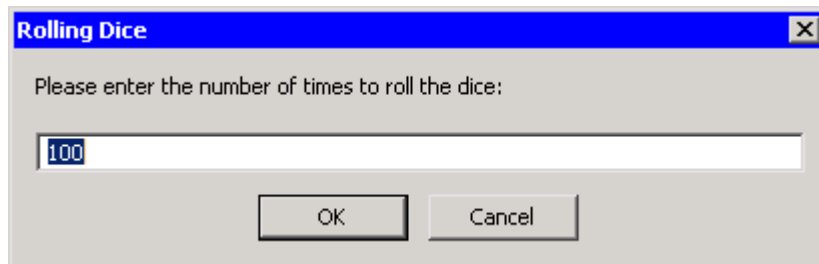


Another simple dialog box is one that contains a single edit field, as shown in Figure 16.8. The DoDialogModifyDouble module takes a default numerical value and enables the user to accept or modify that value, as shown in the following statements:

```
/* prompt user for a value to use in an algorithm */
message = "Please enter the number of times to roll the dice:";
N = 100;                                /* default number of rolls */
rc = DoDialogModifyDouble(N, title, message);
if rc & N>1 then do;                    /* valid specification? */
    N = int(N);                        /* truncate value to integer */
    d = j(N, 2);                      /* allocate Nx2 vector (2 dice) */
    call randgen(d, "uniform");       /* fill with values in (0,1) */
    d = ceil(6*d);                    /* convert to integers in 1-6 */
    sum = d[,+];                      /* values of the N rolls */
end;
else
    print "Good-bye";                  /* canceled or invalid value */
```

For this part of the program, the value 100 is the default number of dice rolls. The user can modify that value. When the user clicks **OK**, the value in the dialog box is copied into the scalar matrix **N**. The value returned by the module (called the *return code*) is stored in the **rc** matrix. The return code indicates whether the user entered a valid number and clicked **OK**. A value of 0 indicates that you should not use the value of **N**; a nonzero value indicates a successful return from the dialog box module.

Figure 16.8 A Dialog Box with a Numerical Input Field



The program uses the value from the dialog box to simulate rolling two dice **N** number of times. The RANDGEN function returns an $N \times 2$ matrix of values in the range (0, 1). If you call this matrix *t*, then 6*t* is a matrix with values in the range (0, 6). The CEIL function returns the closest integer greater than a given value, which results in uniformly distributed integers with values one through six. Finally, the subscript operator (+) sums the two columns of the matrix, as described in the section “Writing Efficient SAS/IML Programs” on page 79.

The vector **d** contains the values of the simulated dice rolls. The following statements use the DoMessageBoxYesNo module to ask the user whether the program should display a bar chart that shows the distribution of the values:

```
/* prompt user to determine the behavior of an algorithm */
message = "Would you like to see the distribution of the results?";
```

```
rc = DoMessageBoxYesNo(title, message);
if rc then do;                                     /* create the bar chart */
    declare BarChart bar;
    bar = BarChart.Create(title, sum);
    bar.SetAxisLabel(XAXIS, "Sum");
    bar.SetOtherThreshold(0);
end;
```

The dialog box is shown in Figure 16.9. If the user clicks **Yes**, the program creates and displays a bar chart similar to the one shown in Figure 16.10. As the number of simulated rolls gets larger, the distribution of the sample approaches a triangular distribution.

Figure 16.9 A Message Box That Contains **Yes** and **No** Buttons

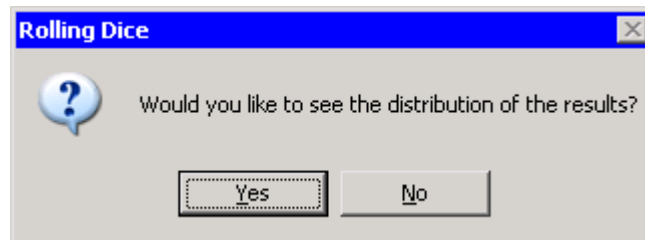
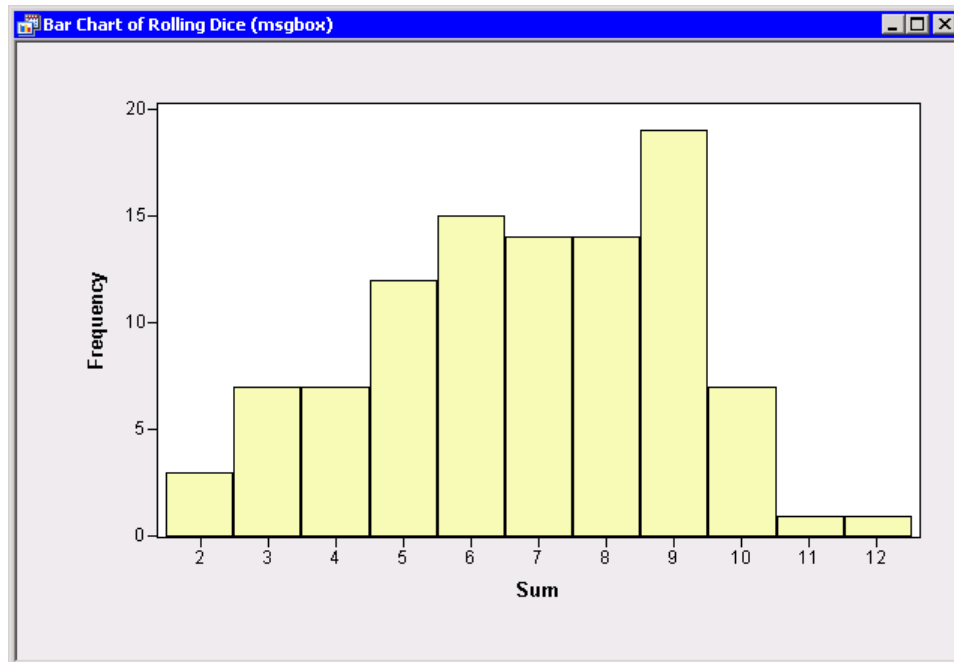


Figure 16.10 Possible Results from a Simulation of Rolling Dice



Programming Tip: SAS/IML Studio is distributed with modules that create simple dialog boxes. You can use these modules to query for input from the user.

16.5.2 Displaying a List in a Dialog Box

When you want to present the user with a list of possible choices, you can use the DoDialogGetListItem or DoDialogGetListItems modules. The DoDialogGetListItem module enables the user to choose a single item from the list, whereas the DoDialogGetListItems module enables the user to select one or more items from a list.

The two modules have a similar syntax. The first argument is a matrix that serves as an output argument. When the module returns, the matrix contains the indices of the items that the user selected. The second and third arguments are the title and message for the dialog box; these arguments are identical to the arguments for the DoMessageBoxOK dialog box. The fourth argument is a character matrix; each row contains the text for the list. The last argument is a character matrix that labels the columns in the list.

The following statements present the user with a list of all the numeric variables in a data set, and then run a correlation analysis on the variables that the user selects:

```
/* prompt user to choose variables for an analysis */
declare DataObject dobj;
dobj = DataObject.CreateFromServerDataSet("Sasuser.Vehicles");

/* get vector of numeric variables */
do i = 1 to dobj.GetNumVar();          /* 1 */
    name = dobj.GetVarName(i);        /* get the name */
    if dobj.IsNumeric(name) then      /* if variable is numeric */
        varNames = varNames // name; /* add it to the vector */
end;

rc = DoDialogGetListItems(selections, /* 2 */
    "Correlation Analysis",
    "Please select variables for correlation analysis.",
    varNames, "Name");

if rc then do;                        /* 3 */
    vars = varNames[selections];      /* 4 */
    dobj.WriteVarsToServerDataSet(vars, /* 5 */
        "Work", "In", true);

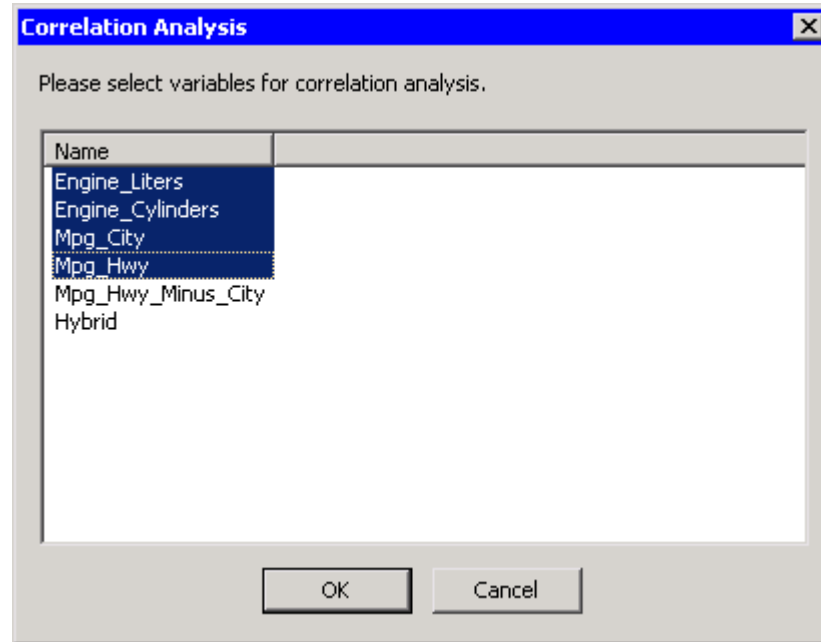
    submit vars;                      /* 6 */
    proc corr data=In nosimple noprob;
        var &vars;                    /* analyze selected vars */
    run;
endsubmit;
end;
```

This example uses a data object to read the Vehicles data. The main steps of the program are as follows:

1. For each variable in the data object, the program queries the data object to determine the variable's name and whether the variable is numeric. If so, the variable name is appended to a vector. (After the DO loop, a careful programmer will check that the **varNames** vector has at least two elements.)

- The vector is passed to the DoDialogGetListItems module, which displays a list of the variables and prompts the user to select variables for a correlation analysis. A possible choice is shown in Figure 16.11.

Figure 16.11 A Dialog Box That Contains a List



- The user can dismiss the Correlation Analysis dialog box by clicking **OK** or **Cancel**. The value of the matrix **rc** reflects that choice. If the user clicks **OK**, the matrix **rc** is nonzero.
- The numeric vector **selections** contains the list items that the user selected. For example, the vector returned from the selections shown in Figure 16.11 is {1, 2, 3, 4}. The program gets the names of the selected variables by extracting elements from the **varNames** vector into the vector **vars**.
- In order to call a procedure, you must make sure that the data are in a SAS data set in a libref. That is already the case for this example, but in general a data object can be created from many sources. Consequently, you should write the selected variables to a SAS data set such as Work.In.
- The names of the selected variables are passed to the SAS System by listing the **vars** vector in the SUBMIT statement. Inside the SUBMIT block, the expression **&vars** refers to the contents of this vector. The output from PROC CORR is shown in Figure 16.12.

Figure 16.12 Results of a Correlation Analysis on Selected Variables

The CORR Procedure				
4 Variables:	Engine_Liters	Engine_Cylinders	Mpg_City	Mpg_Hwy
Pearson Correlation Coefficients, N = 1187				
	Engine_Liters	Engine_Cylinders	Mpg_City	Mpg_Hwy
Engine_Liters Engine Displacement (liters)	1.00000	0.89673	-0.79525	-0.81224
Engine_Cylinders Number of Cylinders	0.89673	1.00000	-0.74190	-0.72572
Mpg_City MPG City	-0.79525	-0.74190	1.00000	0.91865
Mpg_Hwy MPG Highway	-0.81224	-0.72572	0.91865	1.00000

Programming Tip: You can use the DoDialogGetListItems module to create a dialog box that contains a list of items. You can use this module to prompt the person who is running the program to select one or more items from the list.

If you need more control over the number of items the user can select or if you want to supply a default set of values, use the DoDialogGetListItemsEx module.

The preceding example creates a vector by concatenating character strings within a loop. However, [Section 2.9](#) advises that you avoid doing this. In the example, the loop is over variables in a data set. When the number of variables is small (say, less than 100), the inefficiency does not affect the overall performance of the program. However, if your data contain thousands of variables, you should avoid concatenating character strings within a loop. The following code gives an alternative formulation:

```

/* get vector of numeric variables (more efficient) */
k = dobj.GetNumVar();           /* number of variables */
blank32 = subpad("", 1, 32);    /* string with 32 blanks */
varNames = j(k, 1, blank32);   /* allocate result vector */
count = 0;                     /* keep count of numeric vars */
do i = 1 to k;                 /* for each variable... */
    name = dobj.GetVarName(i);  /* get the name */
    if dobj.IsNumeric(name) then do; /* if the variable is numeric */
        count = count + 1;      /* increment the counter */
        varNames[count] = name; /* add name to the vector */
    end;
end;
varNames = strip(varNames[1:count]); /* shrink & strip blanks */

```

Analyzing the logic of the alternative formulation is left as an exercise for the reader.

16.6 Creating a Dialog Box with Java

This example shows how to call Java classes to create a modal dialog box. This feature might be useful to the statistical programmer who wants to create dialog boxes as part of an IMLPlus program. This section does not attempt to explain how to program in Java, but rather assumes that the reader is familiar with Java programming.

This book primarily describes the Java classes that are provided as part of SAS/IML Studio, such as those in `com.sas.imlplus.client.graphics`. However, you can call any public method in any Java class, regardless of its origin. The documentation of a class lists the public methods for the class.

For example, if you are interested in using Java Swing components to create a GUI for an IMLPlus program, you can run the following statements directly from a SAS/IML Studio program window:

```
/* create a modal dialog by using Java Swing classes */
import javax.swing.JOptionPane; /* import statements MUST be first */

/* Text for the buttons */
declare String[] distrib = {"Normal", "Uniform", "Exponential"};

response = JOptionPane.showOptionDialog(
    null,                                /* parent component */
    "Sample from what distribution?",    /* message */
    "Pseudorandom Sampling",           /* title of dialog box */
    JOptionPane.DEFAULT_OPTION,        /* option type */
    JOptionPane.PLAIN_MESSAGE,         /* messageType */
    null,                               /* no icon */
    distrib,                            /* text for buttons */
    distrib[0] );                      /* default selection */

msg = "Note: Modal dialog. Program is blocked until the dialog is closed.";
print msg;
print "The selected choice was " response " = " (distrib[response]);
call randseed(12345);                  /* set seed */
x = j(5, 1);                           /* allocate vector */
call randgen(x, distrib[response]); /* generate from chosen distrib */
print x;
```

The program uses the **import** statement to import the `JOptionPane` class in the `javax.swing` package. The **import** statement enables you to refer to Java classes without always specifying the package that contains the class. Like the **declare** keyword, the **import** keyword must be specified in lower-case characters. An **import** statement must precede other IMLPlus statements.

The Java `String` class is used to create an array of character strings. Because IMLPlus automatically imports the `java.lang` package (which contains the `String` class), you do not need to explicitly import the package that contains the `String` class.

The arguments to the Java `showOptionsDialog` method are explained in comments in the previous program. A simple internet search readily provides links to the documentation and examples for the `JOptionPane` class.

The program displays a dialog box as shown in Figure 16.13. If you click the “Normal” button, then the dialog box returns the value 0, which is stored in the `response` variable. Because the index for Java arrays is zero-based, the expression `distrib[response]` is a string that can be used as an argument to the RANDGEN subroutine. Figure 16.14 shows output from running the program.

Figure 16.13 Modal Dialog Box Created in Java

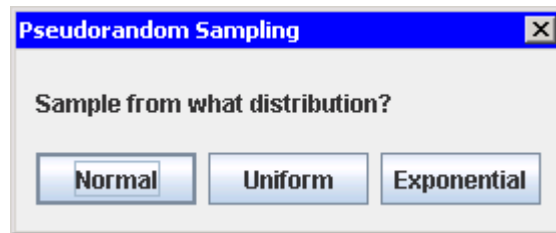


Figure 16.14 Result of Calling a Java Modal Dialog Box

```

                                msg

      Note: Modal dialog. Program is blocked until the dialog is closed.

                                response

      The selected choice was           0 = Normal

                                x

                                0.2642335
                                1.0747269
                                0.8179241
                                -0.552775
                                1.5401449

```

If you want to use classes defined by yourself or that you downloaded from another source, you can do that as well. The instructions are given in the online Help, in the chapter “Extending IMLPlus: Using Java.”

Programming Tip: The Java language contains an extensive set of classes (called the Java Swing components) that you can use to build dialog boxes and other GUI elements. If you are a Java programmer, you can use these components to create dialog boxes as part of a SAS/IML Studio program.

16.7 Creating a Dialog Box with R

As described earlier in this chapter, SAS/IML Studio provides several simple dialog boxes that you can use to write programs that present options and choices for users of the programs. However, SAS/IML Studio does not provide a toolkit that enables you to design and build arbitrary dialog boxes from within an IMLPlus program. This section describes an advanced topic: how to use R packages to build modal dialog boxes that can be incorporated into IMLPlus programs.

The examples in this section use the **tcltk** package (Dalgaard 2001), which is distributed with R. The Tcl/Tk toolkit used by the **tcltk** package is included with the Microsoft Windows versions of R by default. There are other packages that hide some of the low-level details of creating GUI components, but installing these packages can be a complicated undertaking. For example, some R users prefer the **gWidgets** package, which uses the GTK2 toolkit. However, it is not trivial to install the GTK2 toolkit, so this section uses the **tcltk** package.

If you are not familiar with Tcl and Tk, these tools are standards in building GUI components. The Tool Command Language (Tcl) is a scripting language often used for building dialog boxes and even complete user interfaces. The associated toolkit of graphical user interface components is known as Tk.

Programming Tip: The examples in this section use the **tcltk** package because it is distributed with R. There are other packages (such as the **gWidgets** package) that you might want to examine before deciding which package to use to create a dialog box.

16.7.1 The Main Idea

Suppose you want to enable the user to choose some option that is associated with an analysis. The default value for the option is off, which is encoded by 0, but the user can turn the option on, which is encoded by 1. You can create a dialog box in R so that the user can select a value for the option. The dialog box records the choice in an R variable, which can be transferred to a SAS/IML matrix. The following statements implement this idea without creating the dialog box:

```
/* send default value to R; retrieve new value */
x = 0;
run ExportMatrixToR(x, "Value");
submit / R;
    # The user's choice is recorded in the Value variable.
    Value <- 1
endsubmit;
run ImportMatrixFromR(x, "Value");
print x;
```

The main idea is to create one or more R variables that contain the chosen options. These variables are then imported into SAS/IML matrices, and the algorithm continues based on the chosen options.

16.7.2 A First Modal Dialog Box

Using the previous program as a template, the following program creates a button on a Tk window. The window is created as a *modal* dialog box, which means that the user must dismiss the window before the program continues. When the button on the window is pressed, the dialog box sets the **Value** variable to 1.

```
/* create modal dialog box in R */
x = 0;
run ExportMatrixToR(x, "Value");
submit / R;
  library(tcltk)
  win <- tkoplevel()           # 1. create a container window
  tktitle(win) <- "Main Window" #   set title for container

  OnButton <- function() {    # 2. define handler for button
    Value <- 1                #   set value at global scope
    tkdestroy(win)            #   close the parent window
  }

  set.but <- tkbutton(         # 3. create button
    win,                      #   specify parent window
    text = "Set Return Value", #   button text
    command = OnButton        #   func to call upon button press
  )

  tkgrid(set.but)             # 4. add button to window

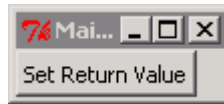
  tkfocus(win)               # 5. move the focus to window
  tkwait.window(win)          #   wait until window is closed
endsubmit;
run ImportMatrixFromR(x, "Value");
print x;
```

The window created by the program is shown in [Figure 16.15](#). The following list briefly describes the main R statements:

1. The **tkoplevel** function creates a window that contains the GUI elements. (For this program, the window contains a single button.) The variable **win** is an object that can be used to set properties of the window. The **tktitle** function adds the title “Main Window” to the window, although only the first few characters of the title are visible in [Figure 16.15](#).
2. The button needs to call a function when it is pressed. This function is called **OnButton** in the program. The function does two things:
 - a) It sets the **Value** variable to the value 1. Note that the **<-** operator ensures that the **Value** variable is set at the global scope of the program, rather than being local to the **OnButton** function.
 - b) It calls the **tkdestroy** function, which closes the main window.
3. The **tkbutton** function creates a button with the text “Set Return Value.” When the button is pressed, the **OnButton** function is called.
4. The **tkgrid** function arranges the button in the main window.

5. The `tkfocus` function gives the keyboard focus to the window.
6. The `tkwait.window` function tells the program to wait until the window is closed, which occurs in the `OnButton` function.

Figure 16.15 Modal Dialog Box Created in R



The flow of the program is as follows. The `ExportMatrixToR` module creates the `value` variable with the value 0. The R statements create the dialog box in [Figure 16.15](#) and wait until the window is dismissed. When you click **Set Return Value**, the `value` variable is set to the value 1 and the window is destroyed. This causes the program to continue and to execute the statement immediately following the `tkwait` function call. The `ImportMatrixToR` module reads the value of the `value` variable into the SAS/IML matrix `x`, and this value is printed.

16.7.3 A Modal Dialog Box with a Checkbox

The previous section showed how to create a modal dialog box by using the `tcltk` package. This section extends the example by creating a checkbox and adding **OK** and **Cancel** buttons to the dialog box.

Suppose you want to call the `CORR` procedure to examine the relationships between several variables. The `CORR` procedure has several options, one of which is the `COV` option that specifies whether to compute the variance-covariance matrix of the variables. You can write a program that displays a dialog box with a checkbox that gives the user the option of computing the covariance matrix. The dialog box might look like [Figure 16.16](#). When the user clicks **OK**, the state of the checkbox determines whether to specify the `COV` option in the `PROC CORR` statement.

Figure 16.16 Modal Dialog Box with Checkbox



The following statements use the `tcltk` package to create the dialog box in [Figure 16.16](#). The state of the checkbox is recorded in the `cov.option` variable in R:

```

/* create dialog box in R; retrieve options that the user selects */
covOption = 0;
run ExportMatrixToR(covOption, "cov.option");
submit / R;

require(tcltk)
OK.pressed <- FALSE
win <- tkoplevel()           # 1. create a container window
tktitle(win) <- "CORR Options" # set title for container
tcl.cov.option <- tclVar(cov.option) # 2. create Tcl var for widget

cb.cov.option <- tkcheckboxbutton(win) # 3. create checkbox for option
label.cov.option <- tklabel(win, text="Show covariances")
tkconfigure(cb.cov.option, variable=tcl.cov.option)

tkgrid(cb.cov.option,           # 4. arrange widgets
       label.cov.option, sticky="w")
onOK <- function() {           # 5. define handler for buttons
  cov.option <- as.numeric(tclvalue(tcl.cov.option))
  OK.pressed <- TRUE
  tkdestroy(win)
}
onCancel <- function() {
  tkdestroy(win)
}

buttonsFrame <- tkframe(win)    # 6. create and layout OK/Cancel
OK.but <- tkbutton(buttonsFrame, text="OK",
                   command=onOK, default="active")
cancel.but <- tkbutton(buttonsFrame, text="Cancel", command=onCancel)
tkgrid(OK.but, tklabel(buttonsFrame, text=" "), cancel.but)
tkgrid(buttonsFrame, columnspan=2)

tkfocus(win)                  # 7. move the focus to window
tkwait.window(win)             # wait until window is closed
endsubmit;

```

The program begins by defining a default value (0, which means “off”) for the covariance option. This value is exported to an R variable named `cov.option`. The following list briefly describes the main R statements:

1. The main window is created as described in the previous section.
2. The `tclVar` function creates a special object of class `tclVar` called `tcl.cov.option`. This variable is associated with the checkbox and is automatically updated whenever the state of the checkbox changes.
3. The program creates two widgets: a checkbox and a label. The `tkconfigure` function ensures that the value of the `tcl.cov.option` variable reflects the state of the checkbox.
4. The `tkgrid` function arranges the checkbox and label in the main window. Each widget is left-aligned by using the `sticky="w"` option.

5. The `onOK` and `onCancel` functions are defined. These functions are called when the **OK** or **Cancel** buttons are clicked. Note that the `onOK` function gets the value of the `tcl.cov.option` variable and copies it to the numeric variable `cov.option`.
6. The **OK** and **Cancel** buttons are packed into a frame that is then added to the bottom of the dialog box.
7. As explained previously, the dialog box is set to be modal.

After the dialog appears, the user must press either **OK** or **Cancel**. If **OK** is pressed, the `onOK` function is called, which sets the `cov.option` variable to reflect the state of the checkbox and then sets the `OK.pressed` variable to **TRUE**. If the window is closed by clicking **Cancel** or by clicking the close icon (X) in the window's title bar, then the value of the `OK.pressed` variable is **FALSE**.

The following statements continue the program by checking the value of the `OK.pressed` variable. If the variable indicates that **OK** was pressed, then the value of `cov.option` determines whether to set the COV option in the PROC CORR statement.

```
/* retrieve user's choices from R variables */
run ImportMatrixFromR(pressed, "OK.pressed");

if pressed then do;                                /* OK was pressed          */
  run ImportMatrixFromR(covOption, "cov.option");
  if covOption then
    covParam = "COV";                               /* checkbox was selected */
  else
    covParam="";                                    /* not selected          */
  submit covParam;                                  /* pass option to CORR   */
  proc corr data=Sasuser.Movies noprob &covParam;
    var Sex Violence Profanity;
  run;
  endsubmit;
end;
else do;
  /* Cancel button was pressed */
end;
```

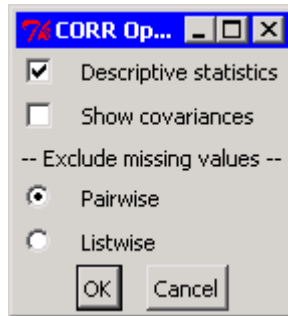
Programming Tip: Create a modal dialog box in R to prompt for the options used in an analysis. The dialog box should create variables that indicate the value of each option. You can read these values into SAS/IML matrices so that the analysis can use the selected options.

16.7.4 Case Study: A Modal Dialog Box for a Correlation Analysis

The previous section describes how to create a dialog box with a single checkbox that specifies whether to use the COV option in the PROC CORR statement. This section builds on the previous example by adding two more widgets to the dialog box. The first is another checkbox that determines whether to use the NOSIMPLE option. The second is a radio button group that specifies how the CORR procedure should handle missing values. By default, the CORR procedure computes the

correlation for each pair of variables and deletes missing values in a pairwise fashion. Alternatively, the NOMISS option specifies that an entire observation be deleted if any variable has a missing value for that observation. This is called listwise deletion of missing values. The dialog box is shown in Figure 16.17.

Figure 16.17 Modal Dialog Box for a Correlation Analysis



The following statements define the default values for the various options and transfer these values to R:

```
/* define and display modal dialog box in R */
descOption = 1;
covOption = 0;
missOption = "pair";
run ExportMatrixToR(descOption, "desc.option");
run ExportMatrixToR(covOption, "cov.option");
run ExportMatrixToR(missOption, "miss.option");
```

The next portion of the program is long, but only because the dialog box now contains five widgets: two checkboxes, a radio button group, and the **OK** and **Cancel** buttons. However, the main steps of the program are similar to the steps explained in the previous section.

```
submit / R;
require(tcltk)
OK.pressed <- FALSE

# 1. create Tcl variables for each widget
tcl.desc.option <- tclVar(desc.option)
tcl.cov.option <- tclVar(cov.option)
tcl.miss.option <- tclVar(miss.option)

# 2. create a container window; set window title
win <- tkoplevel()
tktitle(win) <- "CORR Options"

# 3. create descriptive statistics checkbox
cb.desc.option <- tkcheckboxbutton(win)
label.desc.option <- tklabel(win, text="Descriptive statistics")
tkconfigure(cb.desc.option, variable=tcl.desc.option)

# 4. create covariance checkbox
cb.cov.option <- tkcheckboxbutton(win)
```

```

label.cov.option <- tklabel(win, text="Show covariances")
tkconfigure(cb.cov.option, variable=tcl.cov.option)

# 5. create radio button for handling missing values
label.miss.option <- tklabel(win, text="-- Exclude missing values --")
rb.miss.pair <- tkradiobutton(win)
rb.miss.list <- tkradiobutton(win)
label.miss.pair <- tklabel(win, text="Pairwise")
label.miss.list <- tklabel(win, text="Listwise")
tkconfigure(rb.miss.pair, variable=tcl.miss.option, value="pair")
tkconfigure(rb.miss.list, variable=tcl.miss.option, value="list")

# 6. all widgets created. Arrange them on window
tkgrid(cb.desc.option, label.desc.option, sticky="w")
tkgrid(cb.cov.option, label.cov.option, sticky="w")
tkgrid(label.miss.option, columnspan=2)
tkgrid(rb.miss.pair, label.miss.pair, sticky="w")
tkgrid(rb.miss.list, label.miss.list, sticky="w")

# 7. define handler for buttons
onOK <- function() {
  desc.option <- as.numeric(tclvalue(tcl.desc.option))
  cov.option <- as.numeric(tclvalue(tcl.cov.option))
  miss.option <- as.character(tclvalue(tcl.miss.option))
  OK.pressed <- TRUE
  tkdestroy(win)
}
onCancel <- function() {
  tkdestroy(win)
}

# 8. create and layout OK/Cancel buttons
buttonsFrame <- tkframe(win)
OK.but <- tkbutton(buttonsFrame, text="OK",
  command=onOK, default="active")
cancel.but <- tkbutton(buttonsFrame, text="Cancel", command=onCancel)
tkgrid(OK.but, tklabel(buttonsFrame, text=" "), cancel.but)
tkgrid(buttonsFrame, columnspan=2)

# 9. move the focus to window and wait until window is closed
tkfocus(win)
tkwait.window(win)
endsubmit;

```

The only new step is the creation of the radio button group. Two radio buttons are created by using the `tkradiobutton` function. Both of them are configured to update the `tcl.miss.option` variable. The first radio button sets the value of that variable to the string “pair”; the other sets the value to “list.”

The remainder of the program reads the variables that are set by the dialog box. The program checks whether the **OK** button was clicked and, if so, reads the relevant R variables into SAS/IML matrices. Their values determine whether the NOSIMPLE, COV, and NOMISS options are specified in the PROC CORR statement.


```

/* retrieve user's choices from R variables; call SAS procedure */
run ImportMatrixFromR(pressed, "OK.pressed");
if pressed then do;
  run ImportMatrixFromR(descOption, "desc.option");
  run ImportMatrixFromR(covOption, "cov.option");
  run ImportMatrixFromR(missOption, "miss.option");
  descParam = choose(descOption, "", "NOSIMPLE");
  covParam = choose(covOption, "COV", "");
  missParam = choose(upcase(missOption)="PAIR", "", "NOMISS");

  submit descParam covParam missParam;
    proc corr data=Sasuser.Movies noprob
      &descParam &covParam &missParam;
      var Sex Violence Profanity;
    run;
  endsubmit;
end;
else do;
  /* Cancel button was pressed */
end;

```

The examples in this section have demonstrated that you can use R to construct a dialog box in an IMLPlus program. For more information on creating dialog boxes with the **tcltk** package, see the Web page of examples from Wettenhall (2004).

16.8 References

- Dalgaard, P. (2001), "A Primer on the R-Tcl/Tk Package," *R News*, 1(3), 27–31.
 URL <http://cran.r-project.org/doc/Rnews/>
- Wettenhall, J. (2004), "R TclTk Coding Examples," Last accessed July 12, 2010.
 URL http://www.sciviews.org/_rgui/tcltk/

Part IV

Appendixes

Appendix A

Description of Data Sets

Contents

A.1	Installing the Data	415
A.2	Vehicles Data	415
A.3	Movies Data	416
A.4	Birthdays2002 Data	417

A.1 Installing the Data

This book uses several data sets to illustrate the programming analyses and techniques. This appendix describes the data. The section “[Data and Programs Used in This Book](#)” on page 15 describes how to download and install the data sets used in this book.

A.2 Vehicles Data

The Vehicles data set contains information about 1,187 passenger vehicles produced during 2007 as compiled by the US Environmental Protection Agency (EPA) Fuel Economy Guide. Each observation contains information about a vehicle. Some vehicles in the data are flexible-fuel vehicles (FFVs), which are vehicles that can run on gasoline or E85 (a mixture of 85% ethanol and 15% gasoline). These vehicles appear in the data twice, once with the MPG estimates given for gasoline and once for the MPG estimates given for E85. FFVs operating on E85 usually experience a 20-30% drop in miles per gallon due to ethanol’s lower energy content.

The data set contains the following variables:

Category	a description of the vehicle type: compact car, large car, midsize car, and so forth
Make	the vehicle’s manufacturer
Model	the vehicle’s model designation
Engine_Liters	a measurement, in liters, of the size of the combustion chambers in the engine

Engine_Cylinders	the number of cylinders in the engine								
Mpg_City	the EPA miles-per-gallon (MPG) estimate for city driving								
Mpg_Hwy	the EPA MPG estimate for highway driving								
Mpg_Hwy_Minus_City	the difference between the MPG estimates for city and highway driving								
Hybrid	an indicator variable that contains the value 1 if the vehicle is a hybrid-electric vehicle								
Notes	a variable that indicates whether the vehicle is one of the following kinds of vehicles: <table data-bbox="584 562 1438 783"> <tr> <td>E85</td><td>the MPG estimates for this vehicle were measured for E85 fuel</td></tr> <tr> <td>Gas</td><td>the MPG estimates for this vehicle were measured for gasoline</td></tr> <tr> <td>HEV</td><td>the vehicle is a hybrid-electric vehicle</td></tr> <tr> <td>Tax</td><td>the vehicle is subject to a “gas guzzler” tax as required by the Energy Tax Act of 1978. The tax does not apply to light trucks.</td></tr> </table>	E85	the MPG estimates for this vehicle were measured for E85 fuel	Gas	the MPG estimates for this vehicle were measured for gasoline	HEV	the vehicle is a hybrid-electric vehicle	Tax	the vehicle is subject to a “gas guzzler” tax as required by the Energy Tax Act of 1978. The tax does not apply to light trucks.
E85	the MPG estimates for this vehicle were measured for E85 fuel								
Gas	the MPG estimates for this vehicle were measured for gasoline								
HEV	the vehicle is a hybrid-electric vehicle								
Tax	the vehicle is subject to a “gas guzzler” tax as required by the Energy Tax Act of 1978. The tax does not apply to light trucks.								

A.3 Movies Data

The Movies data set contains information about 359 movies released in the US in the time span 2005–2007. Each observation contains information about a movie. The data were obtained from several sources, including the Web sites kids-in-mind.com and the-numbers.com.

Title	the movie’s title										
MPAARating	the movie rating as determined by the Rating Board of the Motion Picture Association of America. The ratings are as follows: <table data-bbox="500 1297 1438 1785"> <tr> <td>G</td><td>the content of the movie is appropriate for general audiences. A G-rated movie does not contain graphic language, sexual content, or violence.</td></tr> <tr> <td>PG</td><td>the movie contains language, sexual content, or violence that might be inappropriate for young children. Parental guidance is suggested.</td></tr> <tr> <td>PG-13</td><td>the movie contains language, sexual content, or violence that might be inappropriate for children under the age of 13. Parental guidance is suggested.</td></tr> <tr> <td>R</td><td>the movie contains language, sexual content, or violence that most parents would consider inappropriate for children under the age of 17. The movie is “restricted”; children under the age of 17 are not admitted unless accompanied by a parent or guardian.</td></tr> <tr> <td>NR</td><td>the movie was not rated by the MPAA’s Rating Board.</td></tr> </table>	G	the content of the movie is appropriate for general audiences. A G-rated movie does not contain graphic language, sexual content, or violence.	PG	the movie contains language, sexual content, or violence that might be inappropriate for young children. Parental guidance is suggested.	PG-13	the movie contains language, sexual content, or violence that might be inappropriate for children under the age of 13. Parental guidance is suggested.	R	the movie contains language, sexual content, or violence that most parents would consider inappropriate for children under the age of 17. The movie is “restricted”; children under the age of 17 are not admitted unless accompanied by a parent or guardian.	NR	the movie was not rated by the MPAA’s Rating Board.
G	the content of the movie is appropriate for general audiences. A G-rated movie does not contain graphic language, sexual content, or violence.										
PG	the movie contains language, sexual content, or violence that might be inappropriate for young children. Parental guidance is suggested.										
PG-13	the movie contains language, sexual content, or violence that might be inappropriate for children under the age of 13. Parental guidance is suggested.										
R	the movie contains language, sexual content, or violence that most parents would consider inappropriate for children under the age of 17. The movie is “restricted”; children under the age of 17 are not admitted unless accompanied by a parent or guardian.										
NR	the movie was not rated by the MPAA’s Rating Board.										
ReleaseDate	the date the film was first released in the US										
Budget	the movie’s budget, in millions of dollars										

US_Gross	the movie's gross revenues in the US, in millions of dollars
World_Gross	the movie's gross revenues worldwide, in millions of dollars
Sex	a value 0–10 that indicates the amount of sexual references and nudity in a movie, as determined by the kids-in-mind.com Web site
Violence	a value 0–10 that indicates the amount of violence and gore in a movie, as determined by the kids-in-mind.com Web site
Profanity	a value 0–10 that indicates the quantity of profane language in a movie, as determined by the kids-in-mind.com Web site
NumAANomin	the number of Academy Awards nominations for a movie
NumAAWon	the number of Academy Awards won by a movie
AANomin	an indicator variable that contains the value 1 if the movie was nominated for at least one Academy Award
AAWon	an indicator variable that contains the value 1 if the movie won at least one Academy Award
Distributor	the company that distributed the movie
Year	the year that the movie was released

A.4 Birthdays2002 Data

The Birthdays2002 data set contains information about the number of live births in the US for each day in the year 2002. The data are available from the National Center for Health Statistics (NCHS) Web site: www.cdc.gov/nchs.

Date	a value that indicates the day of the year
Births	the number of live births in the US for the day
Month	a value 1–12 that indicates the month of the year
DaysInMonth	the number of days in the month
DOW	a value 1–7 that indicates the day of the week: Sunday (1) through Saturday (7).
Percentage	the number of live births relative to the mean number of daily births for the year. The NCHS calls this quantity the <i>index of occurrence</i> .

Appendix B

SAS/IML Operators, Functions, and Statements

Contents

B.1	Overview of the SAS/IML Language	419
B.2	A Summary of Frequently Used SAS/IML Operators	420
B.3	A Summary of Functions and Subroutines	420
B.3.1	Mathematical Functions	420
B.3.2	Probability Functions	421
B.3.3	Descriptive Statistical Functions	421
B.3.4	Matrix Query Functions	422
B.3.5	Matrix Reshaping Functions	422
B.3.6	Linear Algebra Functions	423
B.3.7	Set Functions	423
B.3.8	Formatting Functions	424
B.3.9	Module Statements	424
B.3.10	Control Statements	425
B.3.11	Statements for Reading and Writing SAS Data Sets	425
B.3.12	Options for Printing Matrices	425

B.1 Overview of the SAS/IML Language

The SAS/IML language contains over 300 built-in functions and subroutines. There are also hundreds of functions in Base SAS software that you can call from SAS/IML programs.

The best overview of these many functions is Chapter 23, “Language Reference” (*SAS/IML User’s Guide*). The section “Overview” lists the SAS/IML functions according to their functionality; the section “Base SAS Functions Accessible from SAS/IML Software” lists the Base SAS functions according to their functionality.

This appendix presents tables that list functions discussed in this book and also related functions that were not explicitly used in this book but that are used often in practice.

B.2 A Summary of Frequently Used SAS/IML Operators

The SAS/IML language provides standard elementwise operators such as addition, subtraction, multiplication, and division. See [Section 2.12](#) for details. The following table lists frequently used SAS/IML operators:

Table B.1 Frequently Used SAS/IML Operators

Statement	Description
&	Logical AND
	Logical OR
^	Logical negation
-	Elementwise unary negation
+	Elementwise addition
-	Elementwise subtraction
#	Elementwise multiplication
/	Elementwise division
##	Elementwise power
*	Matrix multiplication
**	Matrix power
`	Matrix transpose
	Horizontal concatenation of matrices
//	Vertical concatenation of matrices

A complete list of operators is available in the Chapter 23, “Language Reference” (*SAS/IML User’s Guide*).

The SAS/IML language does not have bitwise logical operators; use the BAND, BNOT, BOR, and BXOR functions for bitwise operations.

B.3 A Summary of Functions and Subroutines

This section lists many of the SAS/IML functions and subroutines that are used in statistical data analysis. For a complete list of functions, see the “Overview” section of the Chapter 23, “Language Reference” (*SAS/IML User’s Guide*).

B.3.1 Mathematical Functions

The SAS/IML language contains all of the standard mathematical functions that are available in the SAS DATA step. If you call one of these functions with a matrix argument, the function returns a

matrix of values. For example, the following statement applies the SQRT function to every element of a 1×4 matrix:

```
x = {1 4 9 16};
s = sqrt(x);
```

Table B.2 indicates a few of the many mathematical functions that are available in the SAS/IML language.

Table B.2 Mathematical Functions

Function	Description
ARCOS, ARSIN, ATAN, ...	Inverse trigonometric functions
COS, SIN, TAN, ...	Trigonometric functions
EXP, LOG, SQRT, ...	Basic mathematical functions
INT, FLOOR, CEIL, ...	Truncation functions

B.3.2 Probability Functions

Table B.3 indicates a few of the probability functions that are available in the SAS/IML language.

Table B.3 Probability Functions and Subroutines

Function	Description
CDF	Returns a value from a cumulative probability distribution
NORMAL	Generates pseudorandom numbers from a normal distribution
PDF	Returns a value from a probability density distribution
QUANTILE	Returns a quantile from a distribution
RANDGEN Call	Generates pseudorandom numbers from a distribution
RANDSEED Call	Specifies the sequence for the RANDGEN subroutine
UNIFORM	Generates pseudorandom numbers from a uniform distribution

You can use the NORMAL and UNIFORM functions for generating small samples. However, the RANDGEN subroutine implements a Mersenne-Twister random number generator that is preferred by experts in statistical computation. You should use the RANDGEN subroutine when you intend to generate millions of random numbers.

B.3.3 Descriptive Statistical Functions

Several functions for descriptive statistics were introduced in SAS/IML 9.22: COUNTMISS, COUNTN, MEAN, PROD, QNTL, and VAR. Table B.4 lists some of the SAS/IML functions and subroutines for computing descriptive statistics.

Table B.4 Descriptive Statistical Functions and Subroutines

Function	Description
COUNTMISS	Returns the number of missing elements
COUNTN	Returns the number of nonmissing elements
CUSUM	Returns the cumulative sum of the elements
MIN, MAX	Returns the minimum or maximum element
MEAN	Returns various means of the elements
PROD	Returns the product of the elements
QNTL Call	Returns quantiles of the elements
RANK	Returns the ranks of the elements
RANKTIE	Returns the tied ranks of the elements
SSQ	Returns the sum of the squares of elements
SUM	Returns the sum of the elements
VAR	Returns the variance of the elements

B.3.4 Matrix Query Functions

A matrix query function gives you information about data in a matrix. What are the dimensions of the matrix? Are the data numerical or character? Are any elements nonzero? What are the locations of the nonzero elements? [Table B.5](#) lists some of the matrix query functions.

Table B.5 Matrix Query Functions

Function	Description
ALL	Returns 1 if all elements are nonzero
ANY	Returns 1 if any element is nonzero
CHOOSE	Returns a value based on a criterion evaluated for each observation
LENGTH	Returns a size for each element in a character matrix. For English characters, this size corresponds to the number of characters in each element.
LOC	Returns the indices that satisfy a criterion
NCOL, NROW	Returns the number of columns or rows
NLENG	Returns a scalar number that indicates the size (in bytes) of the elements of a matrix
TYPE	Returns the type of matrix: 'C', 'N', 'U'

B.3.5 Matrix Reshaping Functions

[Table B.6](#) lists functions that create, resize, or reshape a matrix. See Chapter 2, “Getting Started with the SAS/IML Matrix Programming Language,” for details.

Table B.6 Functions That Create and Reshape Matrices

Function	Description
DIAG	Creates a diagonal matrix
DO	Creates an arithmetic sequence
I	Creates an identity matrix
INSERT	Inserts rows or columns into a matrix
J	Creates a matrix with identical values
REMOVE	Removes rows or columns from a matrix
REPEAT	Creates a matrix with repeated values
SHAPE	Reshapes a matrix
T	Transposes a matrix
VECDIAG	Returns a vector that contains the diagonal elements of a matrix

B.3.6 Linear Algebra Functions

Table B.7 lists a few functions that are useful for performing linear algebraic operations such as solving a system of linear equations. The SAS/IML language contains many other advanced functions for matrix computations.

Table B.7 Linear Algebra Functions and Subroutines

Function	Description
DET	Returns the determinant
EIGEN Call	Computes eigenvectors and eigenvalues
GINV	Returns a generalized inverse for a matrix
INV	Returns the inverse for a nonsingular matrix
QR Call	Computes the QR decomposition of a matrix
ROOT	Computes the Cholesky decomposition of a symmetric positive definite matrix
SOLVE	Returns the solution to a linear system of equations
SVD Call	Computes the singular value decomposition
TRACE	Returns the sum of the diagonal elements

B.3.7 Set Functions

Table B.8 lists functions that are useful for finding the union and intersection of sets of values. See Section 2.11 for details.

Table B.8 Functions for Set Operations

Function	Description
SETDIF	Returns the elements of a set that are not in another set
UNION	Returns the elements in the union of sets
UNIQUE	Returns a sorted list of unique values
XSECT	Returns the elements in the intersection of sets

B.3.8 Formatting Functions

Table B.9 lists functions that are useful for formatting data and for manipulating character strings. Many of these functions are documented in the *SAS Language Reference: Dictionary*.

Table B.9 Formatting Functions

Function	Description
CHAR	Returns the result of applying a <i>w.d</i> format to a numeric matrix
NUM	Returns a numeric matrix from a character string that contain numbers
PUTC	Returns the result of applying a format to a character matrix
PUTN	Returns the result of applying a format to a numeric matrix
STRIP	Removes leading and trailing blanks from a character string
TRIM	Removes trailing blanks from a character string
LEFT	Left-aligns a character string
RIGHT	Right-aligns a character string

B.3.9 Module Statements

A module is a user-defined function or subroutine. Table B.10 lists statements that are useful for defining, storing, and loading modules. The mechanism for storing and loading IMLPlus modules is different than the PROC IML mechanism, as explained in [Section 5.7](#).

Table B.10 Module Statements

Statement	Description
FINISH	Ends the definition of a module
LOAD	Loads the definition of a previously stored module
START	Begins the definition of a module
STORE	Stores a defined module

B.3.10 Control Statements

A control statement enables you to iterate and to conditionally execute certain statements. See [Section 2.8](#) for more information. [Table B.11](#) lists some of the important control statements.

Table B.11 Control Statements

Statement	Description
DO	Iterates over a series of statements by incrementing a counter
DO/UNTIL	Iterates over a series of statements until a condition is satisfied
DO/WHILE	Iterates over a series of statements while a condition is satisfied
IF-THEN/ELSE	Tests a condition and executes certain statements depending on whether the condition is true or false
SUBMIT-ENDSUBMIT	Send statements to the SAS System or to R

The SUBMIT and ENDSUBMIT statements were introduced in SAS/IML 9.22.

B.3.11 Statements for Reading and Writing SAS Data Sets

You can read data from a SAS data set into SAS/IML vectors. You can also read numerical (or character) data into a matrix. You can create SAS data sets from data that are in SAS/IML matrices. [Table B.12](#) lists some of the statements that are useful for reading and writing data sets. See [Chapter 3, “Programming Techniques for Data Analysis,”](#) for details.

Table B.12 Statements for Reading and Writing Data

Statement	Description
APPEND	Writes data from matrices
CLOSE	Closes a SAS data set after reading or writing
CREATE	Creates a SAS data set for writing
READ	Reads data into matrices
USE	Opens a SAS data set for reading

B.3.12 Options for Printing Matrices

You can print the values of one or more matrices by using the PRINT statement. You can list multiple matrices on a single print statement. If you separate the matrices with commas, each matrix is printed on a separate row. [Table B.13](#) lists optional arguments that you can use to print matrices with headers or labels, or to format the data. See [Section 2.2.1](#) for details.

Table B.13 Options to the PRINT Statement

Option	Description
COLNAME=	Specifies a character matrix that contains labels for each column
ROWNAME=	Specifies a character matrix that contains labels for each row
FORMAT=	Specifies a SAS format to use when displaying the data
LABEL=	Specifies a label for the matrix

Appendix C

IMLPlus Classes, Methods, and Statements

Contents

C.1	Overview of IMLPlus Classes, Methods, and Statements	427
C.2	The DataObject Class	428
C.3	The DataView Class	429
C.4	The Plot Class	429
C.5	Methods for Creating and Modifying Plots	431
C.5.1	Bar Chart Methods	432
C.5.2	Box Plot Methods	432
C.5.3	Histogram Methods	432
C.5.4	Line Plot Methods	433
C.5.5	Scatter Plot Methods	433
C.6	Calling SAS Procedures	433
C.7	Calling R Functions	434

C.1 Overview of IMLPlus Classes, Methods, and Statements

The programming language of SAS/IML Studio is called *IMLPlus*. IMLPlus is an extension of SAS/IML that contains additional programming features. IMLPlus combines the flexibility of programming in the SAS/IML language with the power to call SAS procedures and to create and modify dynamically linked statistical graphics. The IMLPlus language is described in [Chapter 5](#).

The IMLPlus language is object-oriented. For example, to create a scatter plot, you instantiate an object of the ScatterPlot class. To modify the scatter plot, you call methods that are in the ScatterPlot class or in any base class. Classes and methods are described in [Chapter 6](#); creating an IMLPlus graph is described in [Chapter 7](#).

This appendix presents tables that contain IMLPlus methods discussed in this book and also functions that were not explicitly used in this book but that are used often in practice. This appendix also contains a section that summarizes how to submit SAS statements or R statements from SAS/IML Studio.

C.2 The DataObject Class

The most important class in the IMLPlus language is the DataObject class. The DataObject class is used to manage an in-memory copy of data. The DataObject class contains methods that query, retrieve, and manipulate the data. The class also manages graphical information about observations such as the shape and color of markers, the selected state of observations, and whether observations are displayed in plots or are excluded.

Table C.1 lists frequently used methods in the DataObject class. You can view the complete set of DataView methods from the SAS/IML Studio online Help. Choose **Help ► Help Topics** and then expand the **IMLPlus Class Reference ► DataObject** section.

Table C.1 Frequently Used DataObject Class Methods

Method	Description
AddVar	Adds a new variable to the data object
AddVars	Adds new variables to the data object
Create	Instantiates a data object from SAS/IML matrices
CreateFromFile	Instantiates a data object from a SAS data set available from a drive on the local computer
CreateFromR	Instantiates a data object from an R data frame
CreateFromServerDataSet	Instantiates a data object from a SAS data set in a libref
GetNumObs	Returns the number of observations in the data object
GetNumVar	Returns the number of variables in the data object
GetSelectedObsNumbers	Gets the indices of selected observations
GetVarData	Gets the values for a variable
GetVarNames	Gets the names of variables
IncludeInAnalysis	Specifies whether an observation should be included in statistical analyses
IncludeInPlots	Specifies whether an observation should be included in IMLPlus graphs
IsNominal	Returns true if a variable is nominal and false otherwise
IsNumeric	Returns true if a variable is numeric and false otherwise
SelectObs	Selects observations
SelectObsWhere	Selects observations that satisfy a one-variable criterion
SetMarkerColor	Sets the color of markers that represent an observation in a plot
SetMarkerShape	Sets the shape of markers that represent an observation in a plot
SetNominal	Specifies that a variable is a classification variable
SetRoleVar	Specifies that a variable is used in a certain role, such as to label observations in a plot
SetVarFormat	Sets the SAS format associated with a variable
WriteToServerDataSet	Writes the data into a SAS data set in a libref
WriteVarsToServerDataSet	Writes the specified variables into a SAS data set in a libref

C.3 The DataView Class

The DataView class is the base class for the DataTable class and the Plot class. You cannot instantiate a DataView object, since there is not a “Create” method in the class, but you can call methods in this class from any data table and any plot.

The most frequently used methods in this class are related to positioning or showing a window. Less common methods include the methods related to *action menu*, which are discussed in Chapter 16, “Interactive Techniques.” Table C.2 lists frequently used methods.

Table C.2 Frequently Used DataView Class Methods

Method	Description
ActivateWindow	Displays a window in the foreground
AppendActionMenuItem	Attaches a menu item to a data view. The menu is displayed when you press the F11 key.
GetDataObject	Returns the data object associated with the data view
GetInitiator	Returns the data view whose action menu was selected
IsInstance	Determines whether a plot is an object of a certain class
SetWindowPosition	Positions a data view within the main SAS/IML Studio window
ShowWindow	Displays a window

You can view the complete set of DataView methods from the SAS/IML Studio online Help. Choose **Help ► Help Topics** and then expand the **IMLPlus Class Reference ► DataView** section.

C.4 The Plot Class

The Plot class is a base class for all plots. You cannot instantiate a Plot object, since there is not a “Create” method in the class, but you can call methods in this class from any plot.

The most frequently used methods in this class are related to modifying plot attributes or to drawing on a plot, as discussed in Chapter 9, “Drawing on Graphs.” You can view the complete set of DataView methods from the SAS/IML Studio online Help. Choose **Help ► Help Topics** and then expand the **IMLPlus Class Reference ► DataView** section.

Table C.3 lists methods that are associated with getting and setting properties of a graph’s axis. The first argument to each method is a keyword that specifies the axis. Valid values are XAXIS, YAXIS, or (in some cases) ZAXIS. Except for the SetAxisViewRange method (which is defined in the Plot2D class), these methods are all in the Plot class.

Table C.3 Methods Associated with Axis Properties

Method	Description
GetAxisTickAnchor	Returns the tick anchor for an axis
GetAxisTickUnit	Returns the tick unit, which is the increment between evenly spaced tick marks
GetAxisViewRange	Returns the minimum and maximum values of the axis range
SetAxisLabel	Specifies the label for an axis
SetAxisMinorTicks	Specifies the number of minor tick marks for an axis
SetAxisNumericTicks	Specifies the ticks for a numeric axis
SetAxisTicks	Specifies positions and labels for axis tick marks
SetAxisViewRange	Specifies the axis range

Table C.4 lists methods that draw lines, polygons, text, and other figures on a graph. The methods are provided by the Plot class. See Chapter 9, “Drawing on Graphs.”

Table C.4 Methods Associated with Drawing on Graphs

Category/Method	Description
DEFINE COORDINATES	
DrawSetRegion	Specifies the drawing regions: PLOTBACKGROUND, PLOT-FOREGROUND, or GRAPHFOREGROUND
DrawUseDataCoordinates	Specifies that objects are to be drawn in the coordinate system defined by the data
DrawUseNormalizedCoordinates	Specifies a coordinate system for a graph that is independent of the data
SET PROPERTIES	
DrawSetPenAttributes	Specifies the color, style, and width of a graphical pen in a single call
DrawSetPenColor	Specifies the color of the graphical pen. Valid values include predefined integers such as RED and BLACK, and hexadecimal values such as 04169E1x.
DrawSetPenStyle	Specifies the style of the graphical pen. Valid values are OFF, SOLID, DASHED, DOTTED, DASHDOT, and DASHDOTDOT.
DrawSetPenWidth	Specifies the width of the graphical pen. Typical values are 1–5.
DrawSetBrushColor	Specifies the color of the graphical brush. The brush is used to fill the interior of polygons, arcs, and markers.
DrawSetTextAlignment	Specifies the text alignment for subsequent DrawText commands. Valid values for horizontal alignment are ALIGN_LEFT, ALIGN_CENTER, and ALIGN_RIGHT. Valid values for vertical alignment are ALIGN_BOTTOM, ALIGN_BASELINE, ALIGN_CENTER, and ALIGN_TOP.

Table C.4 Methods Associated with Drawing on Graphs (*continued*)

Category/Method	Description
DRAW FIGURES	
DrawLine	Draws a line or polyline
DrawMarker	Draws markers
DrawPolygon	Draws a polygon
DrawRectangle	Draws a rectangle
DrawText	Draws text
ANIMATION	
DrawBeginBlock	Defines the beginning of a named set of drawing statements
DrawEnableAutoUpdate	Specifies whether each drawing command should cause the plot to redraw itself
DrawEndBlock	Defines the end of a named set of drawing statements
DrawRemoveCommands	Deletes a named set of drawing statements along with any figures that were drawn by those statements

Table C.5 lists other frequently used methods. Except for the SetGraphAreaMargins method (which is defined in the Plot2D class), these methods are all in the Plot class.

Table C.5 Frequently Used Methods in the Plot Class

Method	Description
CopyToOutputDocument	Copies a static image of a plot to the output window
GetVars	Gets the names of the variables in a graph that have a specified role
SetGraphAreaMargins	Specifies the amount of space between the plot area and the edge of the window
SetMarkerSize	Specifies the size of the marker that represents an observation in a graph
SetObsLabelVar	Specifies a variable that labels observations in a graph
SetPlotAreaMargins	Specifies the amount of space between the data and the edge of the plot area
SetTitleText	Specifies the title for a graph (often used with ShowTitle)
ShowObs	Specifies whether observations are always displayed or are displayed only when they are selected
ShowReferenceLines	Specifies whether a reference grid is displayed
ShowTitle	Specifies whether the title is visible

C.5 Methods for Creating and Modifying Plots

In addition to the methods in the DataView and Plot classes, each kind of graph has methods that create and modify the graph. This section presents frequently used methods for the plot types discussed in this book. You can view the complete set of methods for any IMLPlus class from the

SAS/IML Studio online Help. Choose **Help ► Help Topics** and then expand the **IMLPlus Class Reference** section.

C.5.1 Bar Chart Methods

Table C.6 summarizes frequently used methods in the BarChart class. See Section 7.3 for details.

Table C.6 Frequently Used Methods in the BarChart Class

Method	Description
BarChart.Create	Creates a bar chart
SetOtherThreshold	Specifies a cutoff percentage for determining which categories are combined into a generic category called “Others”
ShowBarLabels	Shows or hides labels of the count or percentage for each category
ShowPercentage	Specifies whether the graph’s vertical axis displays counts or percentages

C.5.2 Box Plot Methods

Table C.7 summarizes the frequently used methods in the BoxPlot class. See Section 7.7 for details.

Table C.7 Frequently Used Methods in the BoxPlot Class

Method	Description
BoxPlot.Create	Creates a bar chart
SetWhiskerLength	Specifies the length of the box plot whiskers as a multiple of the interquartile range
ShowMeanMarker	Specifies whether to overlay the mean and standard deviation on the box plot
ShowNotches	Specifies whether to display a notched box plot. This variation of the box plot indicates that medians of two box plots are significantly different at approximately the 0.05 significance level if the corresponding notches do not overlap.

C.5.3 Histogram Methods

Each graph type has methods that control the appearance of the graph. Table C.8 summarizes frequently used methods in the Histogram class. See Section 7.4 for details.

Table C.8 Frequently Used Methods in the Histogram Class

Method	Description
Histogram.Create	Creates a histogram
ReBin	Specifies the offset and width for the histogram bins
ShowBarLabels	Shows or hides labels of the count, percentage, or density for each category
ShowDensity	Specifies whether the graph's vertical axis displays counts or density
ShowPercentage	Specifies whether the graph's vertical axis displays counts or percentages

C.5.4 Line Plot Methods

Table C.9 summarizes frequently used methods in the LinePlot class. See [Section 7.6](#) for details.

Table C.9 Frequently Used Methods in the LinePlot Class

Method	Description
LinePlot.Create	Creates a line plot with one or more Y variables
LinePlot.CreateWithGroup	Creates a line plot in which each line is determined by categories of a grouping variable
AddVar	Adds a new Y variable to an existing line plot
ConnectPoints	Specifies whether to connect observations for each line
SetLineAttributes	Specifies the color, style, and width of a line
SetLineColor	Specifies the color of a line
SetLineStyle	Specifies the style of a line
SetLineWidth	Specifies the width of a line
SetLineMarkerShape	Specifies the marker shape for a line
ShowPoints	Specifies whether to show the markers for a line

C.5.5 Scatter Plot Methods

Except for ScatterPlot.Create method, the ScatterPlot class does not contain any frequently used methods.

C.6 Calling SAS Procedures

You can call any SAS procedure, DATA step, or macro from within an IMLPlus program. Simply enclose the SAS statements between the SUBMIT and ENDSUBMIT statements, as shown in the following example:

```

submit;                                /* begin SUBMIT block */
data a;                                /* run a DATA step   */
    do x = 1 to 10;
        y = x + rannor(1);
        output;
    end;
run;

proc reg data=a;                        /* run a procedure   */
    model y = x;
quit;
endsubmit;                             /* end SUBMIT block  */

```

You can pass parameters into SUBMIT blocks as described in [Section 4.4](#).

C.7 Calling R Functions

R is an open-source language and environment for statistical computing. You can call R functions from an IMLPlus program provided that you have installed a 32-bit Windows version of R on the same PC that runs SAS/IML Studio. Enclose the R statements between the SUBMIT and ENDSUBMIT statements. The SUBMIT statement must contain the R option, as shown in the following example:

```

submit / R;
lm(Height ~ Girth, data=trees)
endsubmit;

```

You can pass parameters into SUBMIT blocks as described in [Section 4.4](#). The mechanism is the same for calling R functions as it is for executing SAS statements.

Appendix D

Modules for Compatability with SAS/IML 9.22

Contents

D.1	Overview of SAS/IML 9.22 Modules	435
D.2	The Mean Module	435
D.3	The Var Module	436
D.4	The Qntl Module	436

D.1 Overview of SAS/IML 9.22 Modules

This book presents a few examples that use functions that were introduced in SAS/IML 9.22. If you have not yet upgraded to SAS/IML 9.22 or SAS 9.3, you might want to define and run the following modules, which implement portions of the functionality of the MEAN and VAR functions and the QNTL subroutine.

The function definitions will fail if you are running SAS/IML 9.22 or later because the SAS/IML language does not permit you to redefine built-in functions.

D.2 The Mean Module

```
/* Mean: return sample mean of each column of a data matrix */
/* For this module
   *   x       is a matrix that contains the data
   */
start Mean(x);
  mean = x[,];
  return ( mean );
finish;
```

D.3 The Var Module

```

/* Var: return sample variance of each column of a data matrix */
/* For this module
*   x       is a matrix that contains the data
*/
start Var(x);
  mean = x[,];
  countn = j(1, ncol(x));          /* allocate vector for counts */
  do i = 1 to ncol(x);             /* count nonmissing values */
    countn[i] = sum(x[,i]^=.);      /* in each column */
  end;
  var = (x-mean)[##,] / (countn-1);
  return ( var );
finish;

```

D.4 The Qntl Module

```

/* Qntl: compute quantiles (Defn. 5 from the UNIVARIATE doc) */
/* For this module
*   q       upon return, q contains the specified quantiles of
*           the data.
*   x       is a matrix. The module computes quantiles for each column.
*   p       specifies the quantiles. For example, 0.5 specifies the
*           median, whereas {0.25 0.75} specifies the first and
*           third quartiles.
* This module does not handle missing values in the data.
*/
start Qntl(q, x, p);                /* definition 5 from UNIVARIATE doc */
  n = nrow(x);                      /* assume nonmissing data */
  q = j(ncol(p), ncol(x));
  do j = 1 to ncol(x);
    y = x[,j];
    call sort(y,1);
    do i = 1 to ncol(p);
      k = n*p[i];                   /* position in ordered data */
      k1 = int(k);                   /* indices into ordered data */
      k2 = k1 + 1;
      g = k - k1;
      if g>0 then
        q[i,j] = y[k2];             /* return a data value */
      else
        q[i,j] = (y[k1]+y[k2])/2;   /* average adjacent data */
      end;
    end;
  end;
finish;

```

Appendix E

ODS Statements

Contents

E.1	Overview of ODS Statements	437
E.2	Finding the Names of ODS Tables	437
E.3	Selecting and Excluding ODS Tables	438
E.4	Creating Data Sets from ODS Tables	439
E.5	Creating ODS Statistical Graphics	440

E.1 Overview of ODS Statements

The SAS Output Delivery System (ODS) enables you to control the output from SAS programs. SAS procedures (including PROC IML) use ODS to display results. Most results are displayed as ODS tables; ODS graphics are briefly discussed in [Section E.5](#). The statements that are described in this chapter are presented for tables, but also apply to ODS graphical objects.

There are ODS statements that enable you to control the output that is displayed, the destination for the output (such as the LISTING or HTML destinations), and many other aspects of the output. This appendix describes a few elementary ODS statements that are used in this book. For a more complete description, see the *SAS Output Delivery System: User's Guide*.

E.2 Finding the Names of ODS Tables

Before you can include, exclude, or save tables, you need to know the names of the tables. You can determine the names of ODS tables by using the ODS TRACE statement. The most basic syntax is as follows:

```
ODS TRACE < ON / OFF > ;
```

When you turn on tracing, the SAS System displays the names of subsequent ODS tables that are produced. The names are usually printed to the SAS log. For example, the following statements display the names of tables that are created by the REG procedure:

```
ods trace on;
proc reg data=sashelp.class;
model Weight = Height;
run;
ods trace off;
```

The content of the SAS log is shown in [Figure E.1](#). The procedure creates four tables: the NObs table, the ANOVA table, the FitStatistics table, and the ParameterEstimates table.

Figure E.1 Names of ODS Tables

```
Output Added:
-----
Name:      NObs
Label:     Number of Observations
Template:  Stat.Reg.NObs
Path:     Reg.MODEL1.Fit.Weight.NObs
-----

Output Added:
-----
Name:      ANOVA
Label:     Analysis of Variance
Template:  Stat.REG.ANOVA
Path:     Reg.MODEL1.Fit.Weight.ANOVA
-----

Output Added:
-----
Name:      FitStatistics
Label:     Fit Statistics
Template:  Stat.REG.FitStatistics
Path:     Reg.MODEL1.Fit.Weight.FitStatistics
-----

Output Added:
-----
Name:      ParameterEstimates
Label:     Parameter Estimates
Template:  Stat.REG.ParameterEstimates
Path:     Reg.MODEL1.Fit.Weight.ParameterEstimates
-----
```

E.3 Selecting and Excluding ODS Tables

You can limit the output from a SAS procedure by using the ODS SELECT and ODS EXCLUDE statements. The ODS SELECT statement specifies the tables that you want displayed; the ODS EXCLUDE statement specifies the tables that you want to suppress. The basic syntax is as follows:

ODS SELECT *table-names* / ALL / NONE ;

ODS EXCLUDE *table-names* / ALL / NONE ;

For example, if you want PROC REG to display only the NObs and ParameterEstimates tables, you can specify either of the following statements:

```
ods select NObs ParameterEstimates;
ods exclude ANOVA FitStatistics;
```

E.4 Creating Data Sets from ODS Tables

You can use the ODS OUTPUT statement to create a SAS data set from an ODS table. For example, use the following statements to create a SAS data set named PE from the ParameterEstimates table that is produced by PROC REG:

```
ods output ParameterEstimates=PE;
proc reg data=sashelp.class;
model Weight = Height;
run;
```

You can then use the data set as input for another procedure such as PROC IML. Of course, you usually need to know the names of the variables in the data set in order to use the data. You can use PROC CONTENTS to display the variable names, or you can use the CONTENTS function in PROC IML, as shown in the following statements:

```
proc iml;
VarNames = contents("PE");
print VarNames;
```

Figure E.2 Variable Names

VarNames
Model
Dependent
Variable
DF
Estimate
StdErr
tValue
Probt

E.5 Creating ODS Statistical Graphics

Many SAS procedures support ODS statistical graphics. You can use the ODS GRAPHICS statement to initialize the ODS statistical graphics system. The basic syntax is as follows:

ODS GRAPHICS *ON* / *OFF* ;

[Section 12.10](#) contains an example that turns on ODS graphics for a regression analysis.

Index

Symbols

() (parentheses), 117
* (matrix multiplication), 49
+ (plus sign)
 as addition operator, 47
 as string concatenation operator, 24
 as subscript reduction operator, 66, 81
- (minus sign)
 as subtraction operator, 47
- (negation operator), 47
/ (slash)
 as division operator, 47
 in FREE statement, 52
// (vertical concatenation operator), 42
: (index creation operator), 25
: (subscript reduction operator), 67, 81
; (semicolon) in statements, 5
< (less than operator), 36
<= (less than or equals operator), 36
= (equal sign), 36
> (greater than operator), 36
>= (greater than or equals operator), 36
@ symbol, 100
(multiplication operator), 47
(exponentiation operator), 47
\$ (dollar sign), 254
& (ampersand)
 as AND operator, 43, 62
 in vector name, 93
 variables and, 103, 271
` (transpose operator), 50, 73
^ (NOT operator), 43
^= (not equals operator), 36
|| (horizontal concatenation operator), 41
| (OR operator), 43

A

abstract classes, 134
action menu
 attaching to graphs, 386
 methods supporting, 139, 429
addition operator (+), 47
AICC (Akaike's information criterion), 217, 267, 390
aicc function (R), 271
algorithm performance

 comparing, 375
 timing algorithms, 383
ALIAS statement, 118
aliases
 creating for modules, 117
 defined, 114
ALL function, 422
ALL option, READ statement, 56
ampersand (&)
 as AND operator, 43, 62
 in vector name, 93
 variables and, 103, 271
AND operator (&), 43, 62
ANY function, 422
APPEND statement
 description, 425
 timing algorithms, 383
 transferring data, 92
 writing data sets, 58
arguments
 evaluating, 75
 functions requiring, 103
 optional, 207
 passing by reference, 74
 to modules, 331
Auto Hide property, 294
Auxiliary Input window, 124

B

bar charts
 creating, 135
 creating from data objects, 146
 creating from vectors, 145
 defined, 145
 modifying graphs, 146
BarChart class
 creating graphs, 135, 145
 frequently used methods, 147, 432
 modifying graphs, 146
base classes, 134, 141
birthday matching problem
 case study, 338
 overview, 332
 simulating, 335
BlendColors module, 120, 236
bootstrap distributions
 computing estimates, 353

- creating, 351
 - defined, 349
 - for the mean, 350
 - plotting, 368
- bootstrap methods
 - case study, 363
 - comparing two groups, 356
 - defined, 349
 - overview, 349
- box plots
 - categorical data and, 193
 - creating, 162
 - creating from data objects, 163
 - creating from vectors, 163
 - creating grouped, 164
 - defined, 161
- BoxPlot class
 - creating graphs, 161
 - frequently used methods, 165, 432
- break points, 232
- BY group processing, 71
- BY statement, 362, 365

C

- C functions, 346
- C= option, HISTOGRAM statement (UNIVARIATE), 95
- case sensitivity, 5, 130, 254
- categorical variables
 - changing display order of, 236
 - displaying on graphs, 193
 - indicating values via marker shapes, 226
- CDF function, 421
- CEIL function, 312, 397
- central limit theorem, 355
- CHAR function, 71, 424
- character matrix
 - length of, 22
 - overview, 18, 21
- CHOOSE function, 65, 319, 422
- CLASS statement, 289
- classes, *see* IMLPlus classes
- classification variables
 - creating line plots with, 158
 - regression diagnostics and, 289
- client data sets, 174
- CLOSE statement, 425
- CMISS function, 77
- coin-tossing simulation, 312
- COLNAME= option
 - FROM clause, CREATE statement, 58
 - PRINT statement, 19, 36, 57, 71, 426
- colon operator (:), 25

- color interpolation, 234
- color ramp, 119, 232
- ColorCodeObs module, 120, 236
- ColorCodeObsByGroups module, 120, 236
- column vectors
 - extracting data from matrices, 32
 - overview, 18, 28
- columns, standardizing in matrix, 83
- COLVEC module, 331
- comparison operators, 36
- CONCAT function, 24, 97, 100
- concatenation operators
 - overview, 41, 97
 - padding strings and, 24
- concrete classes, 134
- conditioning plot, 274
- confidence intervals, plotting, 366
- constant matrices, 24
- continuous variables
 - coloring by values, 232
 - marker colors to indicate values, 229
- ContourPlot class, 168
- control statements, *see also* DO statement, *see also* IF-THEN/ELSE statement
 - overview, 38, 425
 - syntax for, 5
- Cook's *D* statistic, 284, 292
- coordinate systems
 - drawing on graphs, 191
 - practical differences, 202
- CopyServerDataToDataObject module, 119, 184, 299
- CORR procedure, 406, 408
- correlation analysis, 408
- COUNTMISS function, 422
- COUNTN function, 67, 422
- COV option, CORR procedure, 406, 408
- craps (game) simulation, 322
- CREATE statement
 - description, 425
 - timing algorithms, 383
 - transferring data, 92
 - writing data sets, 58, 113
- CUSUM function, 422
- cutoff values, 232

D

- data attributes
 - changing category display order, 236
 - changing marker properties, 226
 - getting/setting, 244
 - R functions, 256
 - selecting observations, 241

- data frames
 - creating, 256
 - defined, 254
 - passing data to R, 254
 - R objects and, 260
- data objects
 - adding variables to, 180
 - creating, 132, 174
 - creating bar charts from, 146
 - creating box plots from, 163
 - creating data sets from, 177
 - creating from data sets, 174
 - creating from matrices, 177
 - creating histograms from, 150
 - creating line plots from, 156, 160
 - creating linked graphs from, 175
 - creating matrices from, 179
 - creating scatter plots from, 154
 - defined, 144
- data sets
 - adding variables from, 184
 - Birthdays2002, 417
 - creating data objects from, 174
 - creating from data objects, 177
 - creating from matrices, 58
 - creating from tables, 439
 - creating matrices from, 56
 - linking related data, 390
 - Movies, 416
 - reading/writing, 56, 113, 425
 - server/client, 174
 - transferring data from R, 257
 - transferring to R functions, 254
 - Vehicles, 415
- DataObject class
 - adding variables to data objects, 180
 - categorical variable display order, 236
 - changing marker properties, 226
 - class hierarchy, 135
 - creating data objects, 174, 177
 - creating data sets, 177
 - creating graphs, 144
 - creating linked graphs, 175
 - creating matrices, 179
 - creating R data frames, 256
 - frequently used methods, 243, 259, 428
 - getting/setting attributes, 244
 - overview, 112, 131, 173, 185, 428
 - schematic description of roles, 132
 - selecting observations, 241
 - transferring data from R, 257
- DATASETS procedure, 102
- DataTable class, 138
- DataView class
 - class hierarchy, 134
 - creating graphs, 136
 - displaying action menu items, 387
 - frequently used methods, 429
 - linking related data, 391
 - overview, 139, 429
- DATE7. format, 156, 170
- DATETIMEw. format, 256
- DATEw. format, 256
- debugging programs
 - Auxiliary Input window and, 124
 - jumping to error location, 121
 - jumping to errors in modules, 123
 - PAUSE statement and, 125
- declare** keyword, 130, 144
- default module storage directory (DMSD), 114
- derived classes, defined, 134
- descriptive statistical functions, 421
- DET function, 423
- DIAG function, 33, 423
- diagonal of a matrix, 33
- dialog boxes
 - creating with Java, 402
 - creating with R, 404
 - displaying lists in, 399
 - displaying simple, 396
 - in SAS/IML Studio, 126, 396
 - modal, 405
- division operator (/), 47
- DMSD (default module storage directory), 114
- DO function, 25, 377, 423
- DO statement, 5, 38, 39, 425
- DO/UNTIL statement, 40, 425
- DO/WHILE statement, 40, 425
- DoDialogGetDouble module, 120
- DoDialogGetListItem module, 120, 399
- DoDialogGetListItems module, 120, 399, 401
- DoDialogGetString module, 120
- DoDialogModifyDouble module, 120, 397
- DoErrorMessageBoxOK module, 120
- dollar sign (\$), 254
- DoMessageBoxOK module, 120, 399
- DoMessageBoxYesNo module, 120, 397
- DrawContinuousLegend module, 120
- drawing regions
 - drawing on graphs, 191
 - foreground/background, 197
 - predication band case study, 198
- drawing subsystem, 187
- DrawInset module, 120, 206
- DrawLegend module
 - adjusting graph margins, 208
 - comparing regression models, 300
 - description, 120

- overview, 204
- DrawPolygonsByGroups module, 120
- dynamically linked graphs, 136

E

- ECDF (empirical cumulative distribution function), 287
- EIGEN subroutine, 423
- elementwise operators, 47
- empirical cumulative distribution function (ECDF), 287
- empty matrix, 64, 207
- ENDSUBMIT statement, *see* SUBMIT statement
- equal sign (=), 36
- error handling when calling procedures, 101
- error messages, interpreting, 31
- EXECUTE subroutine, 104
- ExecuteOSProgram module, 308
- exponentiation operator (##), 47
- ExportDataSetToR module, 254, 256
- ExportMatrixToR module, 255
- ExportToR module, 256

F

- FINISH statement, 72, 424
- fitted function (R), 262
- FLOOR function, 312
- FORMAT procedure, 247
- FORMAT= option, PRINT statement, 19, 426
- formatting functions, 424
- FREE statement, 42, 52
- Freedman-Diaconis bandwidth, 151
- FREQ procedure, 71
- FROM clause, APPEND statement, 58
- FROM clause, CREATE statement, 58
- function modules, 72, 270
- functions
 - as methods, 130, 144
 - built-in, 435
 - calling in R packages, 264
 - creating matrices, 24
 - descriptive statistical, 421
 - finding minimum of, 84
 - formatting, 424
 - linear algebra, 423
 - mathematical, 420
 - matrix query, 422
 - matrix reshaping, 422
 - probability, 421
 - requiring list arguments, 103
 - set, 46, 423
 - unimodal, 84

G

- GAM procedure, 382
- GCV (generalized cross validation), 217
- generic functions, 260
- GENMOD procedure, 358
- GetPersonalFilesDirectory module, 119
- GINV function, 423
- GLM procedure
 - comparing models, 297
 - MODEL statement, 98, 290
 - naming output variables, 96
 - OUTPUT statement, 96, 189, 232
 - regression curves on scatter plot, 188
 - running regression analysis, 280
 - variables for predicted/residual values, 182
- GLOBAL clause, START statement, 74, 117
- global symbols, 74
- golden section search, 84
- graphs, *see* IMLPlus graphs, *see* statistical graphs
- greater than operator (>), 36
- greater than or equals operator (>=), 36
- grouped box plots, 164
- GTK2 toolkit, 404
- gWidgets package (R), 404

H

- HARMEAN function, 103
- high-leverage observations, 286
- hist function (R), 273
- Histogram class
 - creating graphs, 149
 - frequently used methods, 150, 432
- HISTOGRAM statement, UNIVARIATE procedure, 95
- histograms
 - adding rug plots, 212
 - creating from data objects, 150
 - creating from vectors, 149
 - defined, 149
 - KDE case study, 215
- horizontal concatenation operator (||), 41

I

- I function, 34, 423
- identity matrix, 34
- IF-THEN/ELSE statement
 - description, 38, 425
 - handling errors, 102
 - logical operators and, 45
 - syntax for, 5
- IML procedure

- calling R functions, 252
- drawing comparison, 224
- global scope and, 74
- overview, 6
- RUN statement and, 6
- running, 7
- timing algorithms, 383
- IMLMLIB module library, 73, 78, 119
- IMLPlus
 - calling R functions, 112, 252, 434
 - calling SAS procedures, 90, 111, 433
 - color representation in, 229
 - debugging programs, 121
 - drawing comparison, 224
 - managing data, 112, 173
 - object-oriented programming, 130
 - overview, 109, 427
 - processing requirements, 8
 - SAS/IML comparison, 126
- IMLPlus classes
 - base classes, 134
 - creating graphs, 135
 - DataObject class, 131, 428
 - DataView class, 429
 - defined, 144
 - derived classes, 134
 - in modules, 141
 - overview, 129
 - Plot class, 429
 - viewing documentation, 133
- IMLPlus graphs
 - adding lines to, 210
 - adjusting margins, 208
 - attaching menus to, 386
 - changing axis tick positions, 220
 - characteristics, 112
 - coordinate systems, 191
 - creating, 135
 - creating dynamically linked, 136
 - drawing figures/diagrams, 222
 - drawing in background, 197
 - drawing in foreground, 197
 - drawing legends, 204
 - drawing regions, 191
 - drawing rug plots, 212
 - drawing subsystem, 187
 - KDE case study, 214
 - methods for creating/modifying, 431
 - methods for drawing, 430
 - plotting loess curve, 216
 - R functions and, 273
 - scatter plot case study, 198
- IMLPlus modules
 - base classes in, 141
 - debugging, 123
 - frequently used, 119
 - local variables in, 117
 - overview, 114
 - passing objects to, 139
 - storing/loading, 114
- import** keyword, 402
- ImportDataSetFromR module, 258
- ImportMatrixFromR module, 258
- index creation operator (:), 26
- index of occurrence, 338
- INFLUENCE option, MODEL statement (REG), 284
- influential observations, 284
- INSERT function, 43, 423
- insets, drawing on graphs, 203
- interactive techniques
 - attaching menus to graphs, 386
 - dialog boxes and, 396
 - linking related data, 390
 - pausing programs, 385
- INTERP= option, MODEL statement (LOESS), 268
- INTO clause, READ statement, 57
- IntToRGB module, 120, 230, 236
- INV function, 50, 372, 423
- IRR function, 104
- J**
 - J function, 24, 70, 423
 - Java language, 130, 402
 - javax.swing classes, 402
 - JOptionPane class (Java), 402
- K**
 - kernel density estimate case study, 94, 214
 - KERNEL option, HISTOGRAM statement (UNIVARIATE), 95
 - KernSmooth package (R), 264
- L**
 - LABEL= option, PRINT statement, 19, 36, 426
 - LCLM= option, OUTPUT statement (GLM), 96
 - least squares regression model, 296
 - LEFT function, 424
 - legends, drawing on graphs, 203
 - LENGTH function, 22, 422
 - less than operator (<), 36
 - less than or equals operator (<=), 36
 - LIBNAME statement, 59
 - library function (R), 264

line plots
 creating for several variables, 156
 creating for single variable, 155
 creating from data objects, 156, 160
 creating from vectors, 155
 creating with classification variables, 158
 defined, 155
 line segments, drawing, 191
 linear algebra functions, 423
 LinePlot class, 155, 433
 frequently used methods, 161
 lines function (R), 273
 linked graphs, 136, 175
 lists, displaying in dialog boxes, 399
 lm function (R), 259
 LOAD statement
 description, 424
 loading matrices, 53
 loading modules, 114
 storing modules, 76
 LOC function
 analyzing observations by categories, 68
 assigning values to observations, 65
 description, 422
 efficient programs and, 79
 handling missing values, 67
 locating observations, 60
 local variables, 73, 117
 loess curve case study, 216
 loess function (R), 267
 LOESS procedure
 calling, 382
 linking related data, 392
 MODEL statement, 268, 392
 plotting loess curve, 216
 SCORE statement, 217
 SELECT= option, 267
 logical operators, 43
 LOGISTIC procedure, 304
 logistic regression diagnostics, 303
 ls function (R), 255

M

macro variables, 99
 MAD (median absolute deviation), 4
 MAD function, 4
 margins, graph, 208
 marker properties, changing, 226
 mathematical functions, 420
 matrix multiplication operator (*), 49
 matrix operators, 47
 matrix query functions, 422
 matrix reshaping functions, 422

matrix transpose operator (`), 50, 73
 matrix/matrices
 changing shape of, 29
 combining, 47
 computing, 49
 creating, 18
 creating data objects from, 177
 creating data sets from, 58
 creating from data objects, 179
 creating from data sets, 56
 creating macro variables from, 99
 defined, 4
 diagonals of, 33
 dimensions of, 20
 empty, 64, 207
 extracting data from, 30
 length of, 22
 options for printing, 425
 passing data to R functions, 254
 printing, 19
 standardizing columns in, 83
 transferring data, 91
 transposing, 28
 types of, 21
 using functions to create, 24
 MATTRIB statement, 57
 MAX function, 422
 MEAN function
 description, 84, 422
 subscript reduction operators and, 67, 82
 mean function (R), 261
 Mean module, 435
 MEANS procedure, 361, 363
 median absolute deviation (MAD), 4
 Median module, 4
 menus, attaching to graphs, 386
 Mersenne-Twister random number generator, 27
 methods
 action menu and, 139, 429
 calling syntax, 130
 defined, 130, 144
 for creating/modifying graphs, 431
 for drawing graphs, 430
 related to action menu, 429
 MI procedure, 66
 MIANALYZE procedure, 66
 MIN function, 422
 minimal evaluation, 45
 minus sign (-)
 as subtraction operator, 47
 missing values
 algorithms that delete, 375
 comparison operators and, 37
 handling, 65

- passing, 207
 - R functions and, 261
 - standardizing data with, 84
- modal dialog boxes
 - defined, 405
 - for correlation analysis, 408
 - with checkboxes, 406
- MODEL statement
 - GLM procedure, 98, 290
 - LOESS procedure, 268, 392
 - REG procedure, 284
- modules, *see also* IMLPlus modules, *see also*
 - specific modules
 - aliases for, 114, 117
 - arguments to, 331
 - defined, 424
 - dialog boxes and, 126
 - drawing rug plots on graphs, 212
 - encapsulating R statements, 270
 - evaluating arguments, 75
 - finding minimum of functions, 84
 - function, 72
 - global symbols, 74
 - IMLMLIB library, 73, 78
 - local variables, 73
 - overview, 435
 - passing arguments by reference, 74
 - storing, 76
 - subroutine, 72
- MONNAMEw. format, 236
- MosaicPlot class, 168
- multiplication operator (#), 47

N

- names function (R), 254, 263
- NCOL function
 - analyzing observations by categories, 68
 - AND operator and, 62
 - description, 422
 - determining matrix dimensions, 20
 - empty matrix and, 64
 - operations on sets, 47
- negation operator (-), 47
- NLENG function, 22, 422
- NOMISS option, CORR procedure, 409
- NORMAL function, 27, 421
- NOSIMPLE option, CORR procedure, 408, 410
- not equals operator (\neq), 36
- NOT operator (\wedge), 43
- NROW function, 20, 64, 422
- NUM function, 424
- numeric matrix, 21

O

- object-oriented programming, 130
- objects
 - defined, 144
 - passing to modules, 139
- observations
 - analyzing by categories, 68
 - assigning values to, 65
 - attributes of, 246
 - identifying high-leverage, 286
 - identifying influential, 284
 - locating, 60
 - selecting, 241
- ODS EXCLUDE statement, 438
- ODS GRAPHICS statement, 440
- ODS HTML statement, 307
- ODS OUTPUT statement, 92, 439
- ODS SELECT statement, 438
- ODS statements
 - creating statistical graphics, 440
 - finding table names, 437
 - overview, 437
 - selecting/excluding tables, 438
- ODS statistical graphics, 307, 440
- ODS tables, *see* tables
- ODS TRACE statement, 437
- OK= option, SUBMIT statement, 101, 111
- onCancel function (R), 408
- onOK function (R), 408
- operators
 - comparison, 36
 - concatenation, 24, 41, 97
 - elementwise, 47
 - frequently used, 420
 - logical, 43
 - matrix, 47
 - subscript reduction, 66, 81
- Options dialog box, 114
- options function (R), 263
- OR operator (\vee), 43
- OUTHITS option, SURVEYSELECT procedure, 360, 362
- outliers
 - defined, 231
 - identifying, 286
 - influential observations and, 284
 - LOC function and, 63
 - marking via color, 231
- OUTPUT statement
 - GLM procedure, 96, 189, 232
 - UNIVARIATE procedure, 94, 101
- OutputDocument class, 277, 295

P

packages (R)
 calling functions in, 264
 defined, 252
 installing, 264

parameters
 optimizing smoothing, 267
 passing to procedures, 93, 111

parentheses (), 117

passing arguments by reference, 74

PAUSE statement, 125, 385

PBSPLINE statement, SGPLOT procedure, 383

PCTLPTS= option, OUTPUT statement
 (Univariate), 94, 101

PDF function, 157, 421

Pearson correlation coefficient, 72

PERCENT6.1 format, 245

performance considerations
 avoiding program loops, 79
 comparing algorithm performance, 375
 subscript reduction operators, 80

Plot class
 adjusting graph margins, 208
 creating graphs, 135
 drawing legends, 204
 drawing line segments, 191
 drawing subsystem, 187
 frequently used methods, 430
 graph coordinate systems, 191
 overview, 138, 429

plot function (R), 273

Plot2D class, 135, 431

plus sign (+)
 as addition operator, 47
 as string concatenation operator, 24
 as subscript reduction operator, 66, 81

points function (R), 273

PolygonPlot class, 168

power operator (##), 47

prediction bands, scatter plots, 198

PRINCOMP procedure, 363

PRINT statement
 COLNAME= option, 19, 36, 57, 71, 426
 FORMAT= option, 19, 230, 426
 LABEL= option, 19, 36, 426
 ROWNAME= option, 19, 71, 426
 syntax, 19

printing
 expressions, 35
 matrices, 19, 425
 submatrices, 35

PRINTNOW statement, 297

probability functions, 421

procedures

 bootstrap computations, 358
 calling from IMLPlus, 90, 111, 433
 checking return code, 111
 creating macro variables from matrices, 99
 functions requiring list arguments, 103
 handling errors, 101
 kernel density estimate case study, 94
 passing parameters, 93, 111
 transferring data, 91

PROD function, 82, 422

profiling programs, 372

programming techniques
 analyzing observations by, 68
 applying variable transformation, 59
 assigning values to observations, 65
 avoiding loops, 79
 finding minimum of functions, 84
 handling missing values, 65
 locating observations that satisfy criteria, 60
 reading/writing data, 56
 standardizing columns in matrix, 83

properties, variable, 244

pseudorandom matrices, 27

%PUT statement, 99

PUTC function, 23, 424

PUTN function, 71, 221, 424

Q

Q-Q plots, 287

Qntl module, 436

QNTL subroutine, 354, 422

QR subroutine, 423

QUANTILE function, 421

QUARTILE module, 133, 152

QUIT statement, 102

R

R language

 calling functions, 112, 252, 434
 calling R packages, 264
 creating dialog boxes, 404
 creating graphics, 273
 data attributes, 256
 encapsulating in modules, 270
 handling missing values, 261
 importing R objects, 259
 optimizing smoothing parameter, 267
 overview, 252
 passing data, 254

R, supported versions, 264

RANDGEN subroutine

- arguments to, 403
- coin-tossing simulation, 313
- description, 421
- dialog boxes and, 397
- Mersenne-Twister random number generator, 28
- outcomes with specified probabilities, 322, 327
- rolling-dice simulation, 321
- random number generation, 27, 319
- random sampling
 - bootstrap methods and, 350
 - defined, 312
 - with unequal probability, 327
- RANDSEED subroutine, 28, 58, 312, 321, 421
- RANK function, 239, 422
- RANKTIE function, 422
- READ statement, 56, 425
- reading data sets, 56, 113, 425
- reference lines, adding, 210
- REG procedure, 102, 284
- regression curves, overlaying, 188
- regression diagnostics
 - case study, 296
 - classification variables and, 289
 - comparing models, 292
 - displaying lines, 210
 - examining residuals distribution, 287
 - fitting regression models, 280
 - identifying high-leverage observations, 286
 - identifying influential observations, 284
 - identifying outliers, 286
 - logistic, 303
 - overview, 279
- REMOVE function, 43, 375, 423
- REMOVE statement, 53
- REPEAT function, 25, 423
- resampling, 359
- RESET STORAGE statement, 53, 77
- return code, 111, 397
- RETURN statement, 72
- RGB coordinate system, 199, 229
- RGBToInt module, 120, 230, 236
- RIGHT function, 424
- robust regression model, 296
- ROBUSTREG procedure, 63, 297
- rolling-dice simulation, 321
- ROOT function, 423
- RotatingPlot class, 168
- row index, 32
- row vectors
 - extracting data from matrices, 32
 - overview, 18, 28
 - standardizing columns in matrix, 83
- ROWNAME= option, PRINT statement, 19, 71, 426
- ROWVEC module, 330
- RSTUDENT= option, OUTPUT statement (GLM), 232
- rug plots, 212
- RUN statement, 6
- S
- sampling
 - overview, 311
 - with replacement, 329
 - with unequal probability, 327
- SAS Foundation, 9
- SAS Metadata Server Connection Wizard, 10
- SAS procedures, *see* procedures
- SAS/IML language, 3, *see also* IMLPlus, 18, 419
- SAS/IML modules, *see* modules
- SAS/IML software, *see* IML procedure, *see* SAS/IML Studio
- SAS/IML Studio, *see also* IMLPlus
 - calling C functions, 346
 - client-server architecture, 9
 - dialog boxes in, 126, 396
 - exploratory data analysis, 12
 - installing, 10
 - invoking, 10
 - online help, 432
 - overview, 8
 - pasting graphs, 277
 - running programs in, 11
 - workspace example, 8
- scalars, 18, 47
- scatter plots
 - adding rug plots, 212
 - categorical data and, 193
 - comparing analyses with, 295
 - creating, 427
 - creating from data objects, 154
 - creating from vectors, 154
 - defined, 153
 - marker shapes and, 226
 - modifying, 427
 - overlaying regression curves, 188
 - prediction band case study, 198
- ScatterPlot class, 153, 427, 433
- scree plot, 364, 366
- seed value, 27
- SELECT= option, LOESS procedure, 267
- SEM (standard error of the mean), 353
- semicolon (;), 5
- server data sets, 174
- set functions, 46, 423

- SETDIF function, 32, 46, 424
 - SGPLOT procedure, 383
 - SHAPE function, 29, 423
 - short-circuit evaluation, 45
 - SHOW NAMES statement, 51, 73
 - SHOW statement, 51
 - signature, defined, 130
 - simulation
 - birthday matching problem, 335, 341
 - coin tossing, 312
 - defined, 312
 - efficiency considerations, 319
 - game of craps, 322
 - rolling dice, 321
 - slash (/)
 - as division operator, 47
 - in FREE statement, 52
 - SMOOTH= option, MODEL statement (LOESS), 392
 - smoothing parameter, optimizing, 267
 - SOLUTION option, MODEL statement (GLM), 290
 - SOLVE function, 373, 423
 - SORT procedure, 71
 - SPLINE subroutine, 382
 - SQRT function, 45
 - SSQ function, 422
 - standard error of the mean (SEM), 353
 - START statement
 - description, 72, 424
 - GLOBAL clause, 74, 117
 - parentheses on, 117
 - statements, *see also* ODS statements
 - @ symbol and, 100
 - control, 425
 - module, 424
 - reading/writing data sets, 425
 - semicolon in, 5
 - statistical graphs, *see also* bar charts, *see also* box plots, *see also* histograms, *see also* IMLPlus graphs, *see also* line plots, *see also* scatter plots
 - changing graph axis format, 169
 - data sources for, 144
 - displaying underlying data, 168
 - drawbacks of creating from vectors, 172
 - ODS, 307, 440
 - overlying reference lines, 210
 - summary of graph types, 167
 - STORE statement
 - description, 424
 - saving matrices, 53
 - storing modules, 76, 115
 - storing modules, 76, 114
 - str function (R), 260
 - string concatenation operator (+), 24
 - strings, removing blanks in, 98
 - STRIP function
 - description, 424
 - removing blanks, 24, 98, 100
 - submatrices, printing, 35
 - SUBMIT statement
 - calling procedures, 90, 111, 434
 - calling R functions, 252, 266
 - creating macro variables from matrices, 99
 - description, 425
 - OK= option, 101, 111
 - passing procedure parameters, 93
 - R option, 253
 - SUBPAD function, 98
 - subroutine modules, 72
 - subscript reduction operators, 66, 81
 - subtraction operator (-), 47
 - SUM function, 66, 422
 - summary function (R), 254
 - SURVEYSELECT procedure, 358
 - SVD subroutine, 423
 - SYMGET function, 102
 - SYMGETN function, 102
 - SYMPUT subroutine, 99
 - SYSERR macro variable, 102
 - SYSERRORTEXT macro variable, 102
 - SYSWARNINGTEXT macro variable, 102
- T**
- T function, 28, 423
 - tables
 - creating data sets from, 439
 - finding names of, 437
 - listing destination, 307
 - regression diagnostics and, 291
 - selecting/excluding, 438
 - Tcl (Tool Command Language), 404
 - tlcltk package (R), 404, 406
 - tlclVar function (R), 407
 - tick marks, changing for date axis, 220
 - TIME function, 372
 - TIMEw. format, 256
 - timing computations
 - overview, 371
 - replicating timings, 379
 - timing algorithms, 383
 - tips for, 384
 - tkbutton function (R), 405
 - tkconfigure function (R), 407
 - tkfocus function (R), 406
 - tkgrid function (R), 405, 407

tktitle function (R), 405
 tktoplevel function (R), 405
 tkwait function (R), 406
 Tool Command Language (Tcl), 404
 TPSPLINE subroutine, 382
 TRACE function, 423
 transforming variables, 59, 181, 289
 transpose operator ('), 50, 73
 TRIM function, 24, 424
 TYPE function, 21, 47, 64, 422

U

UCLM= option, OUTPUT statement (GLM), 96
 UNIFORM function, 27, 421
 unimodal functions, 84
 UNION function, 46, 424
 UNIQUE function, 68, 79, 424
 UNIVARIATE procedure

- calling, 90
- comparing quantiles of data, 287
- handling errors, 101
- KDE case study, 95
- transferring data, 93

 USE statement, 56, 113, 425

V

VAR function, 84, 354, 422
 Var module, 354, 436
 variables, *see also* specific types of variables

- adding for predicted/residual values, 182
- adding from data sets, 184
- adding to data objects, 180
- categorical, 193
- creating from matrices, 58
- modules and, 73
- output, 96
- properties of, 244
- reading from data sets, 56
- transforming, 59, 181, 289

 VECDIAG function, 33, 423
 vectors

- column, 18
- combining, 47
- creating, 18
- creating bar charts from, 145
- creating box plots from, 163
- creating histograms from, 149
- creating line plots from, 155
- creating scatter plots from, 154
- module arguments as, 331
- row, 18

 vertical concatenation operator (//), 42

W

WHERE clause, 113
 Windows clipboard, 277
 Work library, 8, 52
 workspace bar, 9
 workspaces

- comparing analyses in, 293
- defined, 8
- example, 8
- managing, 51
- server connections, 10
- Work library and, 8, 52

 writing data sets, 56, 113, 425

X

XSECT function, 46, 424

