

**MathSoft**

---

# **S-PLUS 2000 Programmer's Guide**

May 1999

Data Analysis Products Division

MathSoft, Inc.

Seattle, Washington

---

---

## Proprietary Notice

MathSoft, Inc. owns both this software program and its documentation. Both the program and documentation are copyrighted with all rights reserved by MathSoft.

The correct bibliographical reference for this document is as follows:

*S-PLUS 2000 Programmer's Guide*, Data Analysis Products Division, MathSoft, Seattle, WA.

Printed in the United States.

## Copyright Notice

Copyright © 1992-1999 MathSoft, Inc. All Rights Reserved.

## Acknowledgments

S-PLUS would not exist without the pioneering research of the Bell Labs S team at AT&T (now Lucent Technologies): Richard A. Becker, John M. Chambers, Allan R. Wilks, William S. Cleveland, and colleagues.

This release of S-PLUS includes specific work from a number of scientists:

The cluster library was written by Mia Hubert, Peter Rousseeuw and Anja Struyf (University of Antwerp).

Updates to functions provided to this and earlier releases of S-PLUS were provided by Brian Ripley (University of Oxford) and Terry Therneau (Mayo Clinic, Rochester).

# CONTENTS OVERVIEW

## Introduction

Chapter 1 The S-PLUS Language	21
-------------------------------	----

## Data Structures

Chapter 2 Data Objects	41
Chapter 3 Data Frames	69
Chapter 4 Import and Export	93

## Functions

Chapter 5 Writing Functions in S-PLUS	115
Chapter 6 Debugging Your Functions	177

## Graphics

Chapter 7 Editable Graphics Commands	207
Chapter 8 Traditional Graphics	237
Chapter 9 Traditional Trellis Graphics	331

## Object-Oriented Programming

Chapter 10 Object-Oriented Programming in S-PLUS	403
Chapter 11 Programming the User Interface Using S-PLUS	423
Chapter 12 Customized Analytics: A Detailed Example	457

**Connectivity**

<b>Chapter 13 Automation</b>	<b>495</b>
<b>Chapter 14 Calling S-PLUS Using DDE</b>	<b>531</b>

**Advanced Topics**

<b>Chapter 15 The Windows and DOS Interfaces</b>	<b>547</b>
<b>Chapter 16 Interfacing With C and Fortran Code</b>	<b>559</b>
<b>Chapter 17 Extending the User Interface</b>	<b>623</b>
<b>Chapter 18 Libraries</b>	<b>731</b>
<b>Chapter 19 Command Line Options</b>	<b>741</b>
<b>Chapter 20 Computing on the Language</b>	<b>763</b>
<b>Chapter 21 Data Management</b>	<b>787</b>
<b>Chapter 22 Using Less Time and Memory</b>	<b>809</b>
<b>Chapter 23 Simulations in S-PLUS</b>	<b>825</b>
<b>Chapter 24 Evaluation of Expressions</b>	<b>835</b>
<b>Chapter 25 The Validation Suite</b>	<b>847</b>

<b>Index</b>	<b>855</b>
--------------	------------

# CONTENTS

<b>Chapter 1 The S-PLUS Language</b>	<b>21</b>
<b>Introduction to S-PLUS</b>	<b>22</b>
Interpreted vs. Compiled Languages	23
Object-Oriented Programming	23
Programming Tools in S-PLUS	24
<b>Syntax of S-PLUS Expressions</b>	<b>26</b>
Names and Assignment	27
Subscripting	28
<b>Data Classes, Modes, and Types</b>	<b>30</b>
<b>The S-PLUS Programming Environment</b>	<b>35</b>
Editing Objects	35
Functions and Scripts	35
Transferring Data Objects	36
<b>Graphics Paradigms</b>	<b>37</b>
Editable Graphics	37
Traditional Graphics	37
Traditional Trellis Graphics	37
Converting Non-editable Graphics to Editable Graphics	38
When to Use Each Graphics System	38
<b>Chapter 2 Data Objects</b>	<b>41</b>
<b>Basic Data Objects</b>	<b>42</b>
Data Values	43
Coercion of Values	43
Attributes	44
<b>Vectors</b>	<b>47</b>
Creating Vectors	47
Vector Attributes	49
<b>Matrices</b>	<b>51</b>
Creating Matrices	51
Matrix Attributes	53
<b>Arrays</b>	<b>55</b>
Creating Arrays	56
Array Attributes	57

<b>Lists</b>	<b>58</b>
Creating Lists	58
List Attributes	60
<b>Factors and Ordered Factors</b>	<b>62</b>
Creating Factors	64
Creating Ordered Factors	65
Creating Factors from Continuous Data	66
Factor Attributes	67
<b>Chapter 3 Data Frames</b>	<b>69</b>
<b>The Benefits of Data Frames</b>	<b>70</b>
<b>Creating Data Frames</b>	<b>71</b>
<b>Combining Data Frames</b>	<b>76</b>
Combining Data Frames by Column	76
Combining Data Frames by Row	78
Merging Data Frames	79
<b>Applying Functions to Subsets of a Data Frame</b>	<b>82</b>
<b>Adding New Classes of Variables to Data Frames</b>	<b>88</b>
<b>Data Frame Attributes</b>	<b>90</b>
<b>Chapter 4 Import and Export</b>	<b>93</b>
<b>Importing Data Files</b>	<b>94</b>
Data Import Filters	94
Importing a Data File	95
Arguments to importData	96
Setting the Import Filter	98
Notes on Importing Files	100
<b>Other Data Import Functions</b>	<b>103</b>
Reading Vector and Matrix Data with scan	103
Reading Data Frames	105
<b>Exporting Data Sets</b>	<b>107</b>
Exporting Data to S-PLUS	108
Other Export Functions	108
<b>Exporting Graphs</b>	<b>111</b>
<b>Creating HTML Output</b>	<b>112</b>
Tables	112
Text	113
Graphs	113

---

<b>Chapter 5 Writing Functions in S-PLUS</b>	<b>115</b>
<b>Programming in S-PLUS</b>	<b>117</b>
<b>The Structure of Functions</b>	<b>118</b>
Functions and Names	118
Arguments	120
The Function Body	120
Return Values and Side Effects	120
Complex Arithmetic	121
Elementary Functions	123
Summary Functions	125
Comparison and Logical Operators	126
Assignments	127
Testing and Coercing Data	128
<b>Operating on Subsets of Data</b>	<b>131</b>
Subscripting on Matrices and Arrays	133
Subscripting on Lists	135
Subsetting from Data Frames	137
<b>Organizing Computations</b>	<b>138</b>
Programming Style	138
Control Flow	140
The if Statement	141
Handling Multiple Cases	143
The ifelse Function	144
The repeat Statement	146
The return, break, and next Statements	147
The while Statement	148
The for Statement	149
<b>Specifying Argument Lists</b>	<b>150</b>
Formal and Actual Names	150
Specifying Default Arguments	150
Missing-Argument Handling	151
Lazy Evaluation	151
Variable Numbers of Arguments	152
Required and Optional Arguments	153
<b>Error Handling</b>	<b>154</b>
<b>Data Input</b>	<b>156</b>
Reading Vector and Matrix Data With scan	156
Reading Data Frames	159

<b>Data Output</b>	<b>161</b>
Formatting Output	161
Constructing Return Values	163
Side Effects	165
Writing to Files	165
Creating Temporary Files	167
<b>Wrap-Up Actions</b>	<b>168</b>
<b>Extraction and Replacement Functions</b>	<b>170</b>
<b>Operators</b>	<b>174</b>
 <b>Chapter 6 Debugging Your Functions</b>	 <b>177</b>
<b>Basic S-PLUS Debugging</b>	<b>178</b>
Printing Intermediate Results	179
<b>Interactive Debugging</b>	<b>182</b>
Starting the Inspector	183
Examining Variables	184
Controlling Evaluation	188
Entering, Marking, and Tracking Functions	190
Entering Functions	191
Marking Functions	191
Marking the Current Expression	192
Viewing and Removing Marks	193
Tracking Functions	194
Modifying the Evaluation Frame	196
Error Actions in the Inspector	198
<b>Other Debugging Tools</b>	<b>202</b>
Using the S-PLUS Browser Function	202
Using the S-PLUS Debugger	203
Tracing Function Evaluation	204
 <b>Chapter 7 Editable Graphics Commands</b>	 <b>207</b>
<b>Getting Started with Editable Graphics</b>	<b>209</b>
<b>Graphics Objects</b>	<b>211</b>
Graph Sheets	211
Graphs	211
Axes	212
Plots	212
Annotation Objects	212



---

<b>Graphics Commands</b>	<b>213</b>
Getting Started With Simple Plots	213
Specifying Data for Plots	214
Setting Colors and Other Display Properties	215
Listing the Argument Names for an Object	216
<b>Multiple Plots on a Page</b>	<b>217</b>
Object Path Names	217
Combining Plots on a Graph	218
Laying Out Multiple Graphs on a Page	218
Laying Out Graphs on Multiple Pages	219
Conditioned Trellis Graphs	219
<b>Titles and Annotations</b>	<b>220</b>
<b>Plot Types</b>	<b>223</b>
2D Plots	223
3D Plot Palette	229
<b>Formatting Axes and Layout</b>	<b>232</b>
2D Plots	232
3D Plots	232
Creating Specialized Graphs Using Your Own Computations	233
Locating Positions on Your Graph	234
 <b>Chapter 8 Traditional Graphics</b>	 <b>237</b>
<b>Introduction</b>	<b>240</b>
<b>Getting Started With Simple Plots</b>	<b>241</b>
Plotting a Vector Data Object	241
Plotting Mathematical Functions	242
Creating Scatter Plots	244
Plotting Time Series Data Objects	244
<b>Frequently Used Plotting Options</b>	<b>248</b>
Plot Shape	248
Multiple Plot Layout	248
Titles	249
Axis Labels	251
Axis Limits	251
Logarithmic Axes	252
Plot Types	252
Line Types	255
Plotting Characters	255

Plotting Characters and Line Types for Multiple Graphs	257
Controlling Plotting Colors	258
<b>Interactively Adding Information to Your Plot</b>	<b>259</b>
Identifying Plotted Points	259
Adding Straight Line Fits to a Current Scatter Plot	260
Adding New Data to a Current Plot	261
Adding Text to Your Plot	262
<b>Making Bar Plots, Dot Charts and Pie Charts</b>	<b>265</b>
Bar Plots	265
Dot Charts	267
Pie Charts	269
<b>Visualizing the Distribution of Your Data</b>	<b>271</b>
Box Plots	271
Histograms	272
Density Plots	273
Quantile-Quantile Plots	274
<b>Visualizing Higher Dimensional Data</b>	<b>278</b>
Multivariate Data Plots	278
Scatterplot Matrices	278
Plotting Matrix Data	279
Star Plots	280
Faces	280
<b>3-D Plots: Contour, Perspective, and Image Plots</b>	<b>282</b>
Contour Plots	282
Perspective Plots	284
Image Plots	285
<b>Customizing Your Graphics</b>	<b>287</b>
<b>Low-level Graphics Functions and Graphics Parameters</b>	<b>288</b>
<b>Setting and Viewing Graphics Parameters</b>	<b>290</b>
<b>Controlling Graphics Regions</b>	<b>293</b>
Controlling the Outer Margin	294
Controlling Figure Margins	295
Controlling the Plot Area	296
<b>Controlling Text in Graphics</b>	<b>297</b>
Controlling Text and Symbol Size	297
Controlling Text Placement	298
Controlling Text Orientation	299
Controlling Line Width	300

---

Plotting Symbols in Margin	300
<b>Text in Figure Margins</b>	<b>301</b>
<b>Controlling Axes</b>	<b>303</b>
Enabling and Disabling Axes	303
Controlling Tick Marks and Axis Labels	303
Controlling Axis Style	306
Controlling Axis Boxes	307
<b>Controlling Multiple Plots</b>	<b>308</b>
<b>Overlaying Figures</b>	<b>311</b>
High-level Functions That Can Act as Low-level Functions	311
Overlaying Figures by Setting new=TRUE	311
Overlay Figures by Using subplot	312
<b>Adding Special Symbols to Plots</b>	<b>315</b>
Arrows and Line Segments	315
Adding Stars and Other Symbols	316
Custom Symbols	318
<b>Writing Graphics Functions</b>	<b>320</b>
Modifying Existing Functions	320
Combining High-Level and Low-Level Functions	323
Using Graphical Parameters	325
<b>Traditional Graphics Summary</b>	<b>327</b>
References	329
 <b>Chapter 9 Traditional Trellis Graphics</b>	 <b>331</b>
<b>A Roadmap of Trellis Graphics</b>	<b>333</b>
<b>Giving Data to General Display Functions</b>	<b>335</b>
A Data Set: gas	335
formula Argument	335
subset Argument	337
Data Frames	338
<b>Aspect Ratio</b>	<b>339</b>
<b>General Display Functions</b>	<b>341</b>
A Data Set: fuel.frame	341
A Data Set: gauss	354
The Display Functions and Their Formulas	357
<b>Arranging Several Graphs on One Page</b>	<b>359</b>
<b>Multipanel Conditioning</b>	<b>361</b>
A Data Set: barley	361

---

About Multipanel Display	361
Columns, Rows, and Pages	361
Packet Order and Panel Order	362
layout Argument	364
Main-Effects Ordering	366
Summary: The Layout of a Multipanel Display	368
A Data Set: ethanol	368
Conditioning on Discrete Values of a Numeric Variable	368
Conditioning on Intervals of a Numeric Variable	370
<b>Scales and Labels</b>	<b>374</b>
3-D Display: aspect Argument	376
Changing the Text in Strip Labels	376
<b>Panel Functions</b>	<b>378</b>
How to Change the Rendering in the Data Region	378
Passing Arguments to a Default Panel Function	378
A Panel Function for a Multipanel Display	379
Special Panel Functions	380
Commonly-Used S-PLUS Graphics Functions and Parameters	380
<b>Panel Functions and the Trellis Settings</b>	<b>381</b>
<b>Superposing Two or More Groups of Values on a Panel</b>	<b>384</b>
<b>Data Structures</b>	<b>391</b>
<b>More on Aspect Ratio and Scales: Prepanel Functions</b>	<b>394</b>
More on Multipanel Conditioning	395
<b>Summary of Trellis Functions and Arguments</b>	<b>398</b>
 <b>Chapter 10 Object-Oriented Programming in S-PLUS</b>	 <b>403</b>
<b>Fundamentals of Object-Oriented Programming</b>	<b>405</b>
Generic Functions in S-PLUS	406
Classes and Methods in S-PLUS	407
Public and Private Views of Methods	410
<b>Defining New Classes in S-PLUS</b>	<b>411</b>
<b>Group Methods</b>	<b>415</b>
<b>Replacement Methods</b>	<b>422</b>
 <b>Chapter 11 Programming the User Interface Using S-PLUS</b>	 <b>423</b>
<b>The GUI Toolkit</b>	<b>425</b>
GUI Objects	427
GUI Toolkit Functions	427
<b>General Object Manipulation</b>	<b>428</b>

---

<b>Information On Classes</b>	<b>438</b>
guiPrintClass	438
<b>Information on Properties</b>	<b>441</b>
<b>Object Dialogs</b>	<b>444</b>
<b>Selections</b>	<b>448</b>
<b>Options</b>	<b>451</b>
guiSetOption	451
guiGetOption	451
<b>Graphics Functions</b>	<b>452</b>
guiPlot	452
<b>Utilities</b>	<b>455</b>
<b>Summary of GUI Toolkit Functions</b>	<b>456</b>
 <b>Chapter 12 Customized Analytics: A Detailed Example</b>	 <b>457</b>
<b>Overview of the Case Study</b>	<b>458</b>
<b>The Basic Function</b>	<b>460</b>
<b>Enhancing the Function</b>	<b>461</b>
Type Checking	461
Adding Information on an Object	462
<b>Constructing Methods</b>	<b>465</b>
Adding a Class	465
Print Method	466
Summary Method	467
Plot Method	470
<b>Customized Graphical User Interface</b>	<b>474</b>
Adding Menu Items	475
Creating Toolbars	476
Removing Menu Items and Toolbars	477
Customizing the Dialog	477
Creating a Sophisticated Tabbed Dialog	479
The Menu Function	479
The GUI Function	480
Customizing the Context Menu	483
<b>Writing Help Files</b>	<b>487</b>
Creating the Help File	487
<b>Distributing Functions</b>	<b>489</b>
Using Text Files	489
Using Libraries	490

---

<b>Chapter 13 Automation</b>	<b>495</b>
<b>Introduction</b>	<b>496</b>
<b>Using S-PLUS as an Automation Server</b>	<b>497</b>
A Simple Example	497
Exposing Objects to Other Applications	503
Exploring Properties and Methods	504
Programming With Object Methods	505
Programming With Object Properties	510
Passing Data to Functions	511
Automating Embedded S-PLUS Graphs	514
<b>Using S-PLUS as an Automation Client</b>	<b>515</b>
A Simple Example	515
High-Level Automation Functions	521
Reference Counting Issues	522
<b>Automation Examples</b>	<b>526</b>
Server Examples	526
Client Examples	528
 <b>Chapter 14 Calling S-PLUS Using DDE</b>	 <b>531</b>
<b>Introduction</b>	<b>532</b>
<b>About DDE</b>	<b>533</b>
Working With DDE in S-PLUS	535
Starting a DDE Conversation	537
Executing S-PLUS Commands: DDE Execute	538
Sending Data to S-PLUS: DDE Poke	539
Getting Data From S-PLUS: DDE Request	540
Using Application Names in Client Program Scripts	542
 <b>Chapter 15 The Windows and DOS Interfaces</b>	 <b>547</b>
<b>Using the Windows Interface</b>	<b>548</b>
<b>Using the DOS Interface</b>	<b>552</b>
<b>Programming With the DOS and Windows Interfaces</b>	<b>554</b>
Functions for Archiving and Restoring Data	554
A Matrix Editor Using Microsoft Excel	556
 <b>Chapter 16 Interfacing With C and Fortran Code</b>	 <b>559</b>
<b>Overview</b>	<b>561</b>
<b>A Simple Example: Filtering Data</b>	<b>563</b>

---

<b>Using the C and Fortran Interfaces</b>	<b>566</b>
When Should You Consider the C or Fortran Interface?	566
Reasons for Avoiding C or Fortran	566
<b>Calling C or Fortran Routines From S-PLUS</b>	<b>568</b>
Calling C	568
Calling Fortran	569
<b>Writing C and Fortran Routines Suitable for Use in S-PLUS</b>	<b>572</b>
Handling IEEE Special Values	572
Allocating Memory	575
I/O in C Functions	576
I/O in Fortran Subroutines	577
Reporting Errors and Warnings	577
Generating Random Numbers	582
Returning Variable-Length Output Vectors	583
Calling Fortran From C	585
Calling C From Fortran	587
Calling S-PLUS Functions From C Code	587
<b>Using Dynamic Link Libraries (DLLs)</b>	<b>594</b>
Creating a DLL From C Source Code	594
Loading and Running the Code in the DLL	598
Unloading the DLL	600
Calling Functions in the S-PLUS Engine DLL	600
<b>Compiling and Loading Watcom Object Code</b>	<b>603</b>
Setting Up the Loading Environment	603
Using COMPILE to Create Watcom Object Files	605
Loading Watcom Object Files Using dyn.load	607
Statically Loading Watcom Object Code Using LOAD	608
Solving Problems With Static or Dynamic Loading	612
<b>Debugging Loaded Code</b>	<b>616</b>
Debugging C Code	616
Source-Level Debugging	618
<b>A Note on StatLib</b>	<b>622</b>
 <b>Chapter 17 Extending the User Interface</b>	 <b>623</b>
<b>Overview</b>	<b>625</b>
Motivation	625
Approaches	625
Architecture	626

<b>Menus</b>	<b>627</b>
Creating Menu Items	627
Menu Item Properties	628
Modifying Menu Items	632
Displaying Menus	634
Saving and Opening Menus	634
<b>Toolbars and Palettes</b>	<b>636</b>
Creating Toolbars	636
Toolbar Object Properties	637
Modifying Toolbars	639
Creating Toolbar Buttons	640
ToolbarButton Object Properties	641
Modifying Toolbar Buttons	643
Displaying Toolbars	644
Saving and Opening Toolbars	646
<b>Dialogs</b>	<b>647</b>
Creating Dialogs	649
Creating Property Objects	650
Property Object Properties	650
Modifying Property Objects	652
Creating FunctionInfo Objects	653
FunctionInfo Object Properties	654
Modifying FunctionInfo Objects	655
Displaying Dialogs	656
Example: The Contingency Table Dialog	657
<b>Dialog Controls</b>	<b>660</b>
Control Types	660
Copying Properties	671
ActiveX Controls in S-Plus dialogs	674
<b>Callback Functions</b>	<b>694</b>
Interdialog Communication	696
Example: Callback Functions	696
<b>Class Information</b>	<b>699</b>
Creating ClassInfo Objects	699
ClassInfo Object Properties	700
Modifying ClassInfo Objects	701
Example: Customizing the Context Menu	702



---

<b>Style Guidelines</b>	<b>706</b>
Basic Issues	706
Basic Dialogs	707
Modeling Dialogs	715
Modeling Dialog Functions	721
Class Information	727
Dialog Help	729
<b>Chapter 18 Libraries</b>	<b>731</b>
<b>Introduction</b>	<b>732</b>
<b>Creating a Library</b>	<b>734</b>
Steps in Creating a Library	735
Creating Directories	735
Storing Functions	736
Storing Interface Objects	736
Copying Help Files	738
Storing Other Files	738
Start-up and Exit Actions	739
<b>Distributing the Library</b>	<b>740</b>
<b>Chapter 19 Command Line Options</b>	<b>741</b>
<b>Using the Command Line</b>	<b>742</b>
<b>Command Line Parsing</b>	<b>744</b>
Variables	745
Switches	754
<b>Working With Projects</b>	<b>757</b>
The Preferences Directory	757
The Data Directory	758
<b>Customizing Your Session at Start-up and Closing</b>	<b>759</b>
Setting S_FIRST	759
Customizing Your Session at Closing	760
<b>Enhancing S-PLUS</b>	<b>761</b>
Adding Functions and Data Sets to Your System	761
Adding Object Code	762
<b>Chapter 20 Computing on the Language</b>	<b>763</b>
<b>Introduction</b>	<b>764</b>
<b>Symbolic Computations</b>	<b>766</b>
<b>Making Labels From Your Expressions</b>	<b>768</b>

<b>Creating File Names and Object Names</b>	<b>770</b>
<b>Building Expressions and Function Calls</b>	<b>771</b>
Building Unevaluated Expressions	771
Manipulating Function Definitions	772
Building Function Calls	776
<b>Argument Matching and Recovering Actual Arguments</b>	<b>780</b>
Interpreting Complicated Argument Lists	782
<b>Chapter 21 Data Management</b>	<b>787</b>
<b>Frames, Names and Values</b>	<b>788</b>
Frames and Argument Evaluation	793
Quick Call Functions	793
Creating and Moving Frames	793
<b>Databases in S-PLUS</b>	<b>795</b>
Database Dictionaries	799
Directory Databases and Object Storage	800
Recursive Objects as Databases	802
User-Defined Database Classes	803
<b>Matching Names and Values</b>	<b>805</b>
<b>Commitment of Assignments</b>	<b>807</b>
<b>Chapter 22 Using Less Time and Memory</b>	<b>809</b>
<b>Time and Memory</b>	<b>810</b>
How S-PLUS Allocates Memory	811
Why and When S-PLUS Copies Data	812
<b>Writing Good Code</b>	<b>816</b>
Use Vectorized Arithmetic	816
Avoid for Loops	817
Avoid Growing Data Sets	819
Avoid Looping Over Named Objects	820
Keep It Simple!	820
Reuse Computations	821
Reuse Code	822
Avoid Recursion	822
<b>Chapter 23 Simulations in S-PLUS</b>	<b>825</b>
<b>Working With Many Datasets</b>	<b>826</b>
Many Iterations and the For Function	827
The Advantages of lapply	827

---

Using the For Function	827
<b>A Simple Bootstrap Function</b>	<b>829</b>
<b>Monitoring Progress</b>	<b>831</b>
Recovery After Errors	832
<b>Summary of Programming Tips</b>	<b>833</b>
<b>Chapter 24 Evaluation of Expressions</b>	<b>835</b>
<b>S-PLUS Syntax and Grammar</b>	<b>836</b>
Literals	837
Calls	839
S-PLUS Evaluation	839
Internal Function Calls	840
Generic Dispatch	842
Assignments	843
Conditionals	844
Loops and Flow of Control	844
Grouping	846
<b>Chapter 25 The Validation Suite</b>	<b>847</b>
<b>Outline of the Validation Routines</b>	<b>848</b>
<b>Running the Tests</b>	<b>851</b>
<b>Creating Your Own Tests</b>	<b>853</b>
<b>Index</b>	<b>855</b>



# THE S-PLUS LANGUAGE

# 1

---

<b>Introduction to S-PLUS</b>	<b>22</b>
Interpreted vs. Compiled Languages	23
Object-Oriented Programming	23
Programming Tools in S-PLUS	24
<b>Syntax of S-PLUS Expressions</b>	<b>26</b>
Names and Assignment	27
Subscripting	28
<b>Data Classes, Modes, and Types</b>	<b>30</b>
<b>The S-PLUS Programming Environment</b>	<b>35</b>
Editing Objects	35
Functions and Scripts	35
Transferring Data Objects	36
<b>Graphics Paradigms</b>	<b>37</b>
Editable Graphics	37
Traditional Graphics	37
Traditional Trellis Graphics	37
Converting Non-editable Graphics to Editable Graphics	38
When to Use Each Graphics System	38

## INTRODUCTION TO S-PLUS

S-PLUS is a language specially created for exploratory data analysis and statistics. You can use S-PLUS productively and effectively without even writing a one-line program in the S-PLUS language. However, most users begin programming in S-PLUS almost subconsciously—defining functions to streamline repetitive computations, avoid typing mistakes in multi-line expressions, or simply to keep a record of a sequence of commands for future use. The next step is usually incorporating *flow-of-control* features to reduce repetition in these simple functions. From there it is a relatively short step to the creation of entirely new modules of S-PLUS functions, perhaps building on the object-oriented features that allow you to define new *classes* of objects and *methods* to handle them properly.

In this book, we concentrate on describing how to use the *language*.

As with any good book on programming, the goal of this book is to help you quickly produce useful S-PLUS functions, and then step back and delve more deeply into the internals of the S-PLUS language. Along the way, we will continually touch on those aspects of S-PLUS programming that are either particularly effective (such as vectorized arithmetic) or particularly troubling (memory use, for loops).

This chapter aims to familiarize you with the language, starting with a comparison of interpreted and compiled languages. We then briefly describe object-oriented programming as it relates to S-PLUS, although a full discussion is deferred until Chapter 10, Object-Oriented Programming in S-PLUS. We then describe the basic syntax and data types in S-PLUS. Programming in S-PLUS does not require, but greatly benefits from, programming tools such as text editors and source control. We touch on these tools briefly in section The S-PLUS Programming Environment (page 35). Finally, we introduce the various graphics paradigms, and discuss when each should be used.

**Note**

This book is intended for use with the S-PLUS Professional Edition. The full functionality of the S-PLUS language, described in these pages, is not available to Axum or S-PLUS Standard Edition users.

## Interpreted vs. Compiled Languages

Like Java, S-PLUS is an *interpreted* language, in which individual language expressions are read and then immediately executed. The S-PLUS interpreter, which carries out the actions specified by the S-PLUS expressions, is always interposed between your S-PLUS functions and the machine those functions are running on.

C and Fortran, by contrast, are *compiled* languages, in which complete programs in the language are translated by a compiler into the appropriate machine language. Once a program is compiled, it runs independently of the compiler. Interpreted programs, however, are useless without their associated interpreter. Thus, anyone who wants to use your S-PLUS programs needs to have a compatible version of S-PLUS.

The great advantage of interpreted languages is that they allow *incremental development*. You can write a function, run it, write another function, run that, then write a third function that calls the previous two. Incremental development is part of what makes S-PLUS an excellent *prototyping tool*. You can create an empty shell of a function, add features as desired, and relatively quickly create a working version of virtually any application. You can then evaluate your prototype to see if portions of the application might be more efficiently coded in C or Fortran, and if so, easily incorporate that compiled code into your finished S-PLUS application.

The disadvantage of interpreted languages is the overhead of the interpreter. Compiled code runs faster and requires less memory than interpreted code, in part because the compiler can look at the entire program and optimize the machine code to perform the required steps in the most efficient manner. Because there is no need for an interpreter, more computer resources can be devoted to the compiled program.

## Object- Oriented Programming

Traditional computer programming, as the very name implies, deals with *programs*, which are sequences of instructions that tell the computer what to *do*. In the sense that a computer language is a language, programs (in S-PLUS, *functions*) are *verbs*.

Object-oriented programming, by contrast, deals largely with *nouns*, namely, the data objects that traditional programs manipulate. In object-oriented programming, you start thinking about a type of object and try to imagine all the actions you might want to perform on objects of that type. You then define the actions specifically for that type of object. Typically, the first such action is to create *instances* of the type.

Suppose, for example, that you start thinking about some graphical objects, more specifically, circles on the computer screen. You want to be able to

create circles, but you also want to be able to draw them, redraw them, move them, and so on.

Using the object-oriented approach to programming, you would define a *class* of objects called `circle`, then define a function for generating circles. (Such functions are called *generator functions*.) What about drawing, redrawing, and moving? All of these are actions that may be performed on a wide variety of objects, but may well need to be implemented differently for each. An object-oriented approach, therefore, defines the actions *generically*, with generic functions called `draw`, `redraw`, `move`, and so on.

The actual implementation of the action for a specific class is called a *method*. For example, for our class `circle` we would define methods `draw.circle`, `redraw.circle`, and `move.circle`. The generic functions include a mechanism for determining the class of their arguments and calling the appropriate methods, so that, for example, if `orb` is an object of class `circle`, the call `draw(orb)` would automatically call the method `draw.circle` and `draw orb`.

We will take up object-oriented programming in detail in Chapter 10, Object-Oriented Programming in S-PLUS.

## Programming Tools in S-PLUS

There are two main tools for developing S-PLUS programs: the Commands window and Script windows. The Commands window will be familiar to all users of S-PLUS prior to version 4. Only one Commands window can be open, and the easiest way to do this is simply click on its Standard toolbar button.



Figure 1.1: The Commands window button on the Standard toolbar.

The `>` prompt in the Commands window indicates S-PLUS is ready for your input. You can now type expressions for S-PLUS to interpret. Throughout this book, we show typed commands preceded by the S-PLUS prompt, as in the following example, because this representation matches what you see in the Commands window:

```
> plot(corn.rain)
```

If you type in examples from the text, or cut and paste examples from the on-line manuals, be sure to *omit* the prompt character. To exit the Commands



window, simply use the close window tool on the top right of the window. The command

```
> q()
```

will close down S-PLUS altogether.

Script windows, on the other hand, do not execute each statement as it is typed in, nor is there a prompt character. They are for developing longer S-PLUS programs, and for building programs from a variety of sources, such as the History log.

For your first sessions programming in S-PLUS, we recommend you use the Commands window.

## SYNTAX OF S-PLUS EXPRESSIONS

You interact with S-PLUS by typing *expressions*, which the S-PLUS interpreter evaluates and executes. S-PLUS recognizes a wide variety of expressions, but in interactive use the most common are *names*, which return the current definition of the named data object, and *function calls*, which carry out a specified computation. Typing the name of a built-in S-PLUS function, for example, shows the current definition of the function:

```
> sqrt
function(x)
x ^ 0.5
```

A name is any combination of letters, numerals, and periods (.) that does not begin with a numeral.

### Note

This definition applies to *syntactic* names, that is, names recognized by the S-PLUS interpreter as names. S-PLUS provides a mechanism by which virtually any *character string*, including non-syntactic names, can be supplied as the name of the data object. This mechanism is described in Chapter 21, Data Management.

S-PLUS is case sensitive, so that `x` and `X` are different names. A function call is usually typed as a function name followed by an argument list (which may be empty) enclosed in parentheses:

```
> graphsheet()
> plot(corn.rai n)
> mean(corn.rai n)
```

```
[1] 10.78421
```

All S-PLUS expressions return a value, which may be NULL. Normally, this return value is automatically printed. Some functions, however, such as `graphsheet`, `plot`, and `q` are called primarily for their side effects, such as starting or closing a graphics device, plotting points, or ending an S-PLUS session. Such functions frequently have the automatic printing of their values suppressed.

If you type an incomplete expression (for example, by omitting the closing parenthesis in a function call), S-PLUS provides a *continuation* prompt (+, by default) to indicate that more input is required to complete the expression.

*Infix operators* are functions with two arguments that have the special calling syntax *arg1 op arg2*. For example, consider the familiar mathematical operators:

```
> 2 + 7
[1] 9
> 12.4 / 3
[1] 4.133333
```

## Names and Assignment

One of the most frequently used infix operators is the *assignment* operator `<-` (and its equivalent, the underscore, `_`) used to associate names and values. For example, the expression

```
> aba <- 7
```

associates the value 7 with the name `aba`. The value of an assignment expression is the assigned value, that is, the value on the right side of the assignment arrow. Assignment suppresses automatic printing, but you can use the `print` function to force S-PLUS to print the expression's value as follows:

```
> print(aba <- 7)
[1] 7
```

If we now type the name `aba`, we see the stored value:

```
> aba
[1] 7
```

The value on the right of the assignment arrow can be any S-PLUS expression; the left side can be any syntactic name or character string. There are a few reserved names, such as `if` and `function`. A complete list can be found in Chapter 24, Evaluation of Expressions. Assignments typed at the S-PLUS prompt are *permanent*; objects created in this way endure from session to session, until removed. Assignments within functions, however, are local to the function; they endure only as long as the call to the function in which they occur. For a complete discussion of assignment, see Chapter 21, Data Management.

Object names must begin with a letter, and may include any combination of upper and lower case letters, numbers and periods ("."). For example,

`mydata`, `my.data` and `my.data.1` are all legal names. Note the use of the period to enhance readability.

## Subscripting

Another common operator is the *subscript* operator `[`, used to extract subsets of an S-PLUS data object. The syntax for subscripting is *object* [*subscript*]. Here *object* can be any S-PLUS object and *subscript* typically takes one of the following forms:

- Positive integers corresponding to the position in the data object of the desired subset. For example, the `letters` data set consists of the 26 lowercase letters. We can pick the third letter using a positive integer subscript as follows:

```
> letters[3]
```

```
[1] "c"
```

- Negative integers corresponding to the position in the data object of points to be *excluded*:

```
> letters[-3]
```

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

- Logical values; true values correspond to the points in the desired subset, false values correspond to excluded points:

```
> i <- 1:26
```

```
> i
```

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
[18] 18 19 20 21 22 23 24 25 26
```

```
> i < 13
```

```
[1] T T T T T T T T T T T F F F F F F F F F F F F
```

```
> letters[i < 13]
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

Subscripting is *extremely* important in making efficient use of S-PLUS because it emphasizes treating data objects as whole entities, rather than as collections of individual observations. This point of view is central to S-PLUS's utility as a data analysis computing environment. For a full discussion of subscripting, see the section Operating on Subsets of Data (page 131).

## DATA CLASSES, MODES, AND TYPES

Every S-PLUS expression is interpreted by the S-PLUS evaluator and returns a value. The return value is a *data object* that can be assigned a name. Data objects are composed of any number of *elements*, which in simple data objects correspond to individual data points, and in more complicated objects may consist of whole data objects. Simple elements are literal expressions of the following *modes*:

- **logical**: The values T (or TRUE) and F (or FALSE).
- **numeric**: Floating-point real numbers (double-precision by default). Numerical values can be written as integers (for example, 3, -4), decimal fractions (4.52, -6.003), or in scientific notation (6.02e23, 8e-47).
- **complex**: Complex numbers of the form  $a + bi$ , where  $a$  and  $b$  are numeric (for example,  $3 + 1.23i$ ).
- **character**: character strings enclosed by matching double quotes (") or apostrophes ('), for example, "Alabama", 'idea'.

The modes are listed in order from least informative (logical) to most informative (character); this order is important when considering data objects formed from elements of different modes.

The number of elements in a data object determines the object's *length*. The simplest data objects, *vectors*, are completely classified by their mode and length.

### Note

At a deep level, *all* S-PLUS data objects are vectors. We will see some instances later in this book where that representation is convenient. In general, however, the notion of a vector as a simple data object is the correct one.

A vector of length 1 can be created simply by typing a literal and pressing ENTER:

```
> 7.4
```

```
[1] 7.4
```

```
> "hel l o"  
[1] "hel l o"
```

To combine multiple elements into a vector, use the `c` function:

```
> c(T, F, T)  
[1] T F T  
  
> c(8.3, 9.2, 11)  
[1] 8.3 9.2 11.0
```

If you try to combine elements of different modes into a single vector, S-PLUS coerces all the elements to the most informative mode:

```
> c(T, 8.3, F)  
[1] 1.0 8.3 0.0  
  
> c(8.3, 9 + 6i)  
[1] 8.3+0i 9.0+6i  
  
> c(T, 8.3, "hel l o")  
[1] "TRUE" "8.3" "hel l o"
```

You can obtain the mode and length of any data object using the `mode` and `length` functions, respectively:

```
> mode(c(T, 8.3, F))  
[1] "numeri c"  
  
> length(c(T, 8.3, "hel l o"))  
[1] 3
```

The mode and length of a data object are *attributes* of the data object. These two attributes, shared by all data objects, are called *implicit* attributes. Vectors have only these attributes; other attributes specify the structure and other distinguishing aspects of other *types* of data object. For example, if we introduce a multi-dimensional structure to the elements in a vector by adding a "dim" attribute, we create a data object called an *array*. The `dim` attribute is a numeric vector that specifies how many elements are to be arranged in each of a number of dimensions. The length of the "dim" attribute specifies the number of dimensions. In the special case where "dim"

has length 2, the array is given the name *matrix*. In this case, the first element of the "dim" attribute specifies the number of rows, the second element specifies the number of columns.

The most important attribute for distinguishing types of data objects is the "class" attribute, which, if non-null, is a character vector of arbitrary length. The first element of the "class" attribute determines the *class* of the data object. Classes formalize the notion of a data type—in S-PLUS, all data classes are types, but only those types for which the "class" attribute is explicitly set are classes. Classes are important to the object-oriented programming aspect of the S-PLUS language; they are used to select appropriate *methods* for calls to *generic* functions. (See Chapter 10, Object-Oriented Programming in S-PLUS, for complete details.)

The following tables show the built-in data types and some of the new classes in S-PLUS. You can easily add new classes; see the section Defining New Classes in S-PLUS (page 411). The tables distinguish between two broad groups of objects: data objects such as vectors and matrices that are restricted to data of a single mode and are called *atomic* data objects, and data objects that may contain entire data objects as elements and are called *recursive* data objects.

Table 1.1: Atomic S-PLUS data types and classes.

Atomic Data Type	Defining Attribute	Class
Vector	see text	-
Matrix	"di m"	-
Array	"di m"	-
Time series	"tspar"	"rts", "cts", "its"
Factor	-	"factor"
Ordered factor	-	"ordered"



Table 1.2: Some of the recursive S-PLUS data types and classes.

Recursive Data Type	Defining Attribute	Class
List	see text	-
Data frame	-	"data.frame"
Model formula	-	"formula"
Model design	-	"design"
Model terms	-	"terms"
ANOVA model	-	"aov", "maov", "aovlist"
ANOVA table	-	"anova"
Linear model	-	"lm", "mlm"
Generalized linear model	-	"glm"
Generalized additive model	-	"gam"
Minimum-sum nonlinear model	-	"ms"
Least-squares nonlinear model	-	"nls"
Seasonal time-series decomposition	-	"stl"
Tree-based model	-	"tree"
Local regression model	-	"loess"

The most generally useful of the recursive data types is the `list` function, which can be used to combine arbitrary collections of S-PLUS data objects into a single object. For example, suppose you have a vector `x` of character data, a matrix `y` of logical data, and a time series `z` as shown below:

```
> x <- c("Tom", "Dick", "Harry")
> y <- matrix(c(T, F, T, F), ncol=2)
> z <- ts(sin(1:36), start=1989)
```

You can combine these into a single S-PLUS data object (of mode "list") using the `list` function:

```
> mylist <- list(x=x, y=y, z=z)

> mylist

$x:
[1] "Tom" "Dick" "Harry"

$y:
      [,1] [,2]
[1,]    T    T
[2,]    F    F

$z:
1989:  0.841470985  0.909297427  0.141120008 -0.756802495
1993: -0.958924275 -0.279415498  0.656986599  0.989358247
1997:  0.412118485 -0.544021111 -0.999990207 -0.536572918
2001:  0.420167037  0.990607356  0.650287840 -0.287903317
2005: -0.961397492 -0.750987247  0.149877210  0.912945251
2009:  0.836655639 -0.008851309 -0.846220404 -0.905578362
2013: -0.132351750  0.762558450  0.956375928  0.270905788
2017: -0.663633884 -0.988031624 -0.404037645  0.551426681
2021:  0.999911860  0.529082686 -0.428182669 -0.991778853
```

The `list` data type is an extremely powerful tool in S-PLUS, and we shall use it extensively throughout this book.

# THE S-PLUS PROGRAMMING ENVIRONMENT

S-PLUS uses tools available in the Windows environment. Some of these tools are built into S-PLUS as functions—for example, the `edit` function, which allows you to edit with the Windows Notepad editor. Windows software, including spreadsheets such as Microsoft Excel and word processors such as Microsoft Word, can be called from S-PLUS using the `dos` and `system` functions.

In this section, we give a brief introduction to the most common tools for writing, editing, and testing your S-PLUS functions, as well as tools for transferring data objects between computers with differing architectures.

**Editing Objects** You can edit S-PLUS data by using the command `Edit.data`. This will call up a Data window, described in the *User's Guide*. The function `Edit` should be used to edit S-PLUS functions.

For most routine editing purposes, you will want to use the `Edit` or `Edit.data` functions. If you find the `Edit` function too limiting, and want to call up a more powerful editor, then use the `fix` function.

```
> fix(x)
```

The `fix` function uses an editor you specify with the S-PLUS `editor` option. At the S-PLUS prompt, type the following:

```
> options(editor="editor ")
```

where *editor* is the binary executable (.exe) that runs your favorite text editor. To set this option for each S-PLUS session, add the expression to your `.First` function. This option defaults to Notepad in S-PLUS.

Once you've set up S-PLUS to work with your favorite editor, writing and testing new functions requires following the simple sequence of writing the function, running the function, editing the function, and so on.

## Functions and Scripts

Writing functions is the preferred way to incorporate new functionality into S-PLUS. Functions allow you to combine a series of S-PLUS expressions into a single executable call. Every function returns a single value, which for functions built from multiple expressions is the value of the last expression in the function's body. Sometimes, however, you may be interested in some or all of the intermediate results of the combined expressions. You can (as we

shall see in the section Data Output (page 161)) pull the intermediate results together into a return list. Sometimes, however, you may want those intermediate results to be stored as individual data objects. In such cases, it makes sense to program your task as an S-PLUS *script*, which is just a text file containing valid S-PLUS expressions. You can run S-PLUS scripts in any of the following ways:

1. Loading it into a Script window, highlighting the required code and clicking the Run button on the Script toolbar. See Chapter 8, Using Script and Report Windows, in the *User's Guide*.
2. The source function in the Commands window.
3. The S-PLUS BATCH utility.

The methods differ primarily in that S-PLUS BATCH runs as a background task and produces a file containing both the input and the output of the job (you can suppress the input). This is frequently useful if you have a complicated debugging task and need to recreate the output of a number of expressions.

## Transferring Data Objects

S-PLUS runs on a variety of hardware platforms with a variety of architectures. The binary representation of S-PLUS objects varies from platform to platform, so if you want to share your functions or data sets with users on other platforms, you need to first dump them to a portable ASCII format with one of several S-PLUS functions, transfer the ASCII file, then restore them using one of several S-PLUS functions.

The functions for dumping and restoring are roughly paired: `dump`, `source`, `data.dump` and `data.restore`. Objects dumped with `dump` must be restored with `source`—the ASCII form produced by `dump` is just an S-PLUS script, which you can read or edit just like any text file. Objects dumped by `data.dump` result in files that are *not* S-PLUS scripts; in fact, these files are in a special format that was not intended to be read by humans. Such objects can be restored only by using the `data.restore` function. The `data.dump` and `data.restore` functions are much faster than the `dump` and `source` functions, and should always be used when transferring large data sets, such as image data. The `dump` function should be used when you want to transfer an object, such as a function definition, that may need editing before being restored.

The functions `data.dump` and `data.restore` are used for importing and exporting files with the S-PLUS transport file format (see Chapter 4, Import and Export, for more details).

---

# GRAPHICS PARADIGMS

In S-PLUS there are three basic graphics paradigms, which we will refer to as Editable Graphics, Traditional Graphics, and Traditional Trellis Graphics.

## Editable Graphics

Editable object-oriented graphics objects represent complete plots, or elements added to plots such as lines, comments, and legends. The plots generated from the plot palettes are each a single graph type with sub-objects representing points, lines, axes, and more.

While most users will generate these graphs through menus and toolbars, commands are also available to generate the plots programmatically. This graphics system is new to S-PLUS version 4.

Chapter 7, Editable Graphics Commands, describes these graphics.

## Traditional Graphics

Traditional S-PLUS language functions are available to provide the wide variety of plot types available in versions of S-PLUS prior to version 4. These functions may be used to create plots identical to those in previous versions of S-PLUS. The emphasis in the design of S-PLUS 4 has been to provide complete backwards compatibility for existing S-PLUS graphics with a route to convert the traditional graphics into editable graphics.

Chapter 8, Traditional Graphics, describes these graphics.

## Traditional Trellis Graphics

The Trellis graphics paradigm introduced in S-PLUS 3.3 for Windows provides multipanel conditioning to effectively discover relationships present in data. These graphics were implemented using calls to the traditional graphics language functions. These are still available in S-PLUS 4.

The Trellis conditioning has also been re-implemented as an option available for all object-oriented plot types.

Chapter 9, Traditional Trellis Graphics, describes using conditioning with the object-oriented plots.

## Converting Non-editable Graphics to Editable Graphics

By default, traditional graphics commands will produce a single composite graphics object which renders quickly. This object may be annotated but its individual components are not available for editing. To edit individual components -- such as points and lines in the graph -- first convert the graph to individual graphics objects by right-clicking on the graph and selecting Convert to Objects from the context menu.

The conversion step may be avoided by creating editable graphics objects directly. To turn on this editable graphics mode, press the Editable Graphics button on the Commands window toolbar. Alternately, you may open a Graph sheet device in editable graphics mode using

```
> graphsheets(object.mode="object-oriented").
```

However, as editable graphics are slower to render than non-editable graphics we strongly recommend creating non-editable graphics and converting them to editable graphics, when needed, rather than using object-oriented mode.

Traditional Trellis graphs are created by changing the axis system for each panel, strip, and plot. This corresponds to a large number of plot and graph objects in the editable graphics system. Due to the complexity of the plots produced by traditional Trellis we strongly recommend that non-editable graphics mode be used when producing traditional Trellis plots.

## When to Use Each Graphics System

The existence of multiple interconnected graphics systems is largely due to the evolution of S-PLUS as graphics methodology and technology has evolved. Here we describe the genesis of each system and the resulting benefits which derive therefrom.

### Traditional Graphics

The traditional graphics system is based on the pioneering work by researchers at AT&T Bell Labs in graphical layout and perception. It is optimized to provide smart default formatting and layout, while providing programmatic specification of plot characteristics at a fine level of control. These graphics have become the standard in statistical publication-quality graphics due to their refined look and ease of use.

As they pre-dated modern object-oriented programming they are based on the rendering of low level graphics components such as points and lines rather than on higher level graphics objects. This provides quicker rendering than editable graphics but does not yield a high level graphics object which may be accessed for editing. To change a traditional graph the model is to regenerate the graph with new specifications rather than to modify a graph object, although the ability to convert to editable graphics does introduce the capability of editing low level graph components.

Traditional graphics are produced by the techniques in the statistics dialogs for speed of rendering and consistency with previous versions of S-PLUS. It is likely that users will want to use traditional graphics for similar reasons. Routines which use these graphics are widespread, and their usage is well documented in both these manuals and third party texts. Also, additional graphics methods are available through traditional graphics which have not been implemented as editable graphics.

### **Traditional Trellis Graphics**

Trellis graphics were introduced in S-PLUS 3.3 as a powerful new technique for exploring multivariate structure in data. They were implemented in traditional graphics for convenience and to make them available to S-PLUS users. This implementation is described in Chapter 9, Traditional Trellis Graphics.

Trellis conditioning has been incorporated directly into the editable graphics system, making the power of multipanel conditioning available in all editable graphs. Due to the complexity of Trellis plots, the point-and-click graph property specification is a much more convenient way to develop a Trellis graph.

Traditional Trellis graphics will be of interest to users wanting more control over the contents of each panel than is available in the editable graphics. Also, additional graphics methods are available through traditional Trellis graphics which have not been implemented as editable graphics.

### **Editable Graphics**

Editable graphics are new to S-PLUS version 4. They have been developed based on modern C++ object-oriented programming structures. As such they are based on a model of creating an object of a particular class with properties containing a description of the object. The user edits the object by modifying its properties. Multiple graphics objects form an object hierarchy of plots within graphs within Graph sheets which together represent a graphic.

Programmers used to using this type of object-oriented programming will prefer to program by creating and modifying editable graphics objects. Users of previous versions of S-PLUS may want to transition towards using editable graphics when doing so provides benefits not available with the traditional graphics, and continue to use traditional graphics when they can leverage their existing experience to get superior results.

As these graphics are new to S-PLUS 4 for Windows, they are not available to users running S-PLUS for Unix, Version 3.4 or earlier. If you will be making your functions available to users on both Windows and Unix platforms you will need to use traditional graphics and traditional Trellis graphics, rather than editable graphics.





# DATA OBJECTS

# 2

---

<b>Basic Data Objects</b>	<b>42</b>
Data Values	43
Coercion of Values	43
Attributes	44
<b>Vectors</b>	<b>47</b>
Creating Vectors	47
Vector Attributes	49
<b>Matrices</b>	<b>51</b>
Creating Matrices	51
Matrix Attributes	53
<b>Arrays</b>	<b>55</b>
Creating Arrays	56
Array Attributes	57
<b>Lists</b>	<b>58</b>
Creating Lists	58
List Attributes	60
<b>Factors and Ordered Factors</b>	<b>62</b>
Creating Factors	64
Creating Ordered Factors	65
Creating Factors from Continuous Data	66
Factor Attributes	67

## BASIC DATA OBJECTS

Before you can use S-PLUS to analyze any data, those data must be organized into coherent collections having specific structure. Such a collection is called a *data object*. There are seven basic types of data objects in S-PLUS:

1. vector
2. matrix
3. array
4. list
5. factor
6. time series
7. data frame

Data frames are complex and are discussed in full in the next chapter. Time series are discussed in several chapters in the *Guide to Statistics, Volume 2*. This chapter covers the first five basic types of data object.

All are created by generalizing from the simplest data object, the *vector*. As we have seen, a vector is simply a sequence of values. Furthermore, the values within a vector must all be of the same kind—logical, numeric, complex, or character strings. The number of values, and what kind they are, completely define the data object as a vector. Characteristics associated with each data object are called *attributes*. The number of values in a vector is called the `length` attribute of the vector, and the kind of observations is called the `mode` of the vector.

To create more complicated data objects, additional attributes are associated with the object. For example, we have seen how to create a matrix from a vector by specifying the number of rows or columns the vector is to fill. In so doing, we have specified an additional attribute, namely, the *dimension*, or `dim`, of the matrix.

There are functions for creating, testing for, and coercing to each of the basic data types. For example, there is a `matrix` function for creating matrices, an `is.matrix` function for testing whether an object *is* a matrix, and an `as.matrix` function for *coercing* objects to matrix form. These functions are explained in this chapter along with other functions that are useful for creating each of the data types discussed.

Data objects can contain not only logical, numeric, complex, and character values, but also functions, operators, function calls, and evaluations. All the different types (classes) of S-PLUS objects can be manipulated in the same

way: saved, assigned, edited, combined, or passed as arguments to functions. This general definition of data objects, coupled with class-specific methods, forms the backbone of *object-oriented programming*, and provides exceptional flexibility in extending the capabilities of S-PLUS.

In this chapter, we describe the various types of data objects, beginning with a discussion of the most common types of data values that can be assigned to those data objects. Then, for each data type, we present examples of the type, introduce functions for creating data objects of that type, and describe the data type in terms of its attributes.

## Data Values

In S-PLUS, a data object is a collection of *values*. The kinds (modes) of values most commonly used in data analysis are as follows.

Missing and non-existent data values are represented by such special symbols as NA.

Data objects can be classified into two categories—*atomic* and *non-atomic*. Atomic objects (vectors, matrices, arrays, time series, and categories) can contain values of only one kind, whereas non-atomic objects (lists and data frames) can contain values of all kinds.

The kind of values within an atomic object determine the object's *mode*. For example, a vector containing only logical values has mode "logical". You can see the mode of any S-PLUS object with the mode function:

```
> mode(car.gals)
[1] "numeric"
```

You can create vectors with logical, numeric, complex, or character values, but you can't create a vector that has both numeric and character values. If you try to create a vector with values of more than one mode, S-PLUS *coerces* all the values to a single mode. The rules for doing this are discussed in the section Coercion of Values (page 43).

Non-atomic objects (such as lists and data frames) allow you to combine objects of different modes into a single object. You can, for example, create a list with a numeric component and a "character" component. Lists can also contain lists as components. Many functions, especially statistical procedures, return lists containing objects resulting from the computations.

## Coercion of Values

When values of different modes are combined into a single atomic object, S-PLUS converts or *coerces* all values to a single mode in a way that preserves as much information as possible. The modes listed in the section Data Values

(page 43) are arranged in order of increasing information—"logical", "numeric", "complex", and "character". (Numeric data can be further subdivided by storage mode—here the order is "integer", "single", and "double".) Thus, mixed values are all converted to the mode of the value with the most informative mode. For example, suppose we combine a logical value, a numeric value, and a character value, as follows:

```
> c(T, 2, "seven")
[1] "TRUE" "2" "seven"
```

S-PLUS coerces all three values to mode "character", because this is the most informative mode represented. Similarly, in the following example all the values are coerced to mode "numeric":

```
> c(T, F, pi, 7)
[1] 1.000000 0.000000 3.141593 7.000000
```

When logical values are coerced to numeric, TRUE values become the number 1 and FALSE values become the number 0.

The same kind of coercion occurs when values of different modes are combined in computations. For example, "logical" values are coerced to zeros and ones in "numeric" computations.

For information on explicit coercion in S-PLUS, see the section Testing and Coercing Data (page 128).

## Attributes

*Attributes* specify the characteristics of any data object. Every data object receives two attributes simply by being defined—`length`, which specifies the number of "values" in the object, and `mode`, which specifies the mode of the object. Because these attributes are assigned implicitly, they are called *implicit* attributes.

The meaning of the `length` attribute of any object depends on the type of object. For atomic objects, the length corresponds to the total number of values. For lists, the length corresponds to the number of components, which is the same as the number of variables or columns for a data frame.

Other attributes define the various data types. For example, a matrix is distinguished from a vector because it has a "numeric" vector of length two as its `dim` attribute. An attribute that defines a new data object is called a *defining* attribute.

Some attributes may be associated with an object, but need not be. For example, the `names` attribute stores labels associated with each value for a vector or with each component for a list, and the `dimnames` attribute stores

row and column labels for a matrix, or more generally, dimension names for an array. Because these attributes are strictly optional, they are called *optional* attributes. For each attribute, there is a corresponding function for viewing that attribute. For example, you use the `dimnames` function to view the `dimnames` attribute:

```
> dimnames(iris)

[[1]]:
character(0)
[[2]]:
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."
[[3]]:
[1] "Setosa" "Versicolour" "Virginica"
```

You can also use these corresponding functions to define attributes. For example, suppose you create a vector `identity` containing the values 1 0 0 1. To turn `identity` into a  $2 \times 2$  matrix, you can use the `dim` function as follows:

```
> identity <- c(1, 0, 0, 1)
> dim(identity) <- c(2, 2)
> identity

      [,1] [,2]
[1,]     1     0
[2,]     0     1
```

You can view an object's defining and optional attributes with the `attributes` function:

```
> attributes(iris)

$dim:
[1] 50 4 3
$dimnames:
$dimnames[[1]]:
character(0)

$dimnames[[2]]:
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."

$dimnames[[3]]:
[1] "Setosa" "Versicolour" "Virginica"
```

The `attributes` function does not return any information on the implicit viewing attributes `mode` and `length`. To view these attributes, use the functions `mode` and `length`, respectively.

# VECTORS

The simplest type of data object in S-PLUS is a vector. A vector is simply an *ordered* set of values. The order of the values is emphasized because ordering provides a convenient way of extracting parts of a vector.

## Creating Vectors

If you want to create a vector of any of the modes discussed in the section Data Values (page 43), you can do so in a number of ways. You have seen that you can combine arbitrary values to create a vector with the `c` function, and type in data from the keyboard or a data file with the `scan` function. For more information, see the section Data Input (page 156) and the section Data Output (page 161).

Other functions are useful for repeating values or generating sequences of numeric values. The `rep` function repeats a value by specifying either a `times` argument or a `length` argument. If `times` is specified, the value is repeated the number of times specified (the value may be a vector):

```
> rep(NA, 5)

[1] NA NA NA NA NA

> rep(c(T, T, F), 2)

[1] T T F T T F
```

If `times` is a vector with the same length as the vector of values being repeated, each value is repeated the corresponding number of times.

```
> rep(c("yes", "no"), c(4, 2))

[1] "yes" "yes" "yes" "yes" "no" "no"
```

The sequence operator generates sequences of numeric values spaced one unit apart.

```
> 1:5

[1] 1 2 3 4 5

> 1.2:4

[1] 1.2 2.2 3.2
```

```
> 1: -1
```

```
[1] 1 0 -1
```

More generally, the `seq` function generates sequences of numeric values with an arbitrary increment. For example:

```
> seq(-pi, pi, .5)
```

```
[1] -3.1415927 -2.6415927 -2.1415927 -1.6415927 -1.1415927  
[6] -0.6415927 -0.1415927 0.3584073 0.8584073 1.3584073  
[11] 1.8584073 2.3584073 2.8584073
```

You can specify the length of the vector and `seq` computes the increment:

```
> seq(-pi, pi, length=10)
```

```
[1] -3.1415927 -2.4434610 -1.7453293 -1.0471976 -0.3490659  
[6] 0.3490659 1.0471976 1.7453293 2.4434610 3.1415927
```

Or you can specify the beginning, the increment, and the length with either the `length` argument or the `along` argument:

```
> seq(1, by=.05, length=10)
```

```
[1] 1.00 1.05 1.10 1.15 1.20 1.25 1.30 1.35 1.40 1.45
```

```
> seq(1, by=.05, along=1:5)
```

```
[1] 1.00 1.05 1.10 1.15 1.20
```

See the help file for `seq` for more information on the `length` and `along` arguments.

To “initialize” a vector of a certain mode and length before you know the actual values, use the `vector` function. This function takes two arguments: the first specifies the mode and the second specifies the length:

```
> vector("logical", 3)
```

```
[1] F F F
```



The functions `logical`, `numeric`, `complex` and `character` generate vectors of the named mode. Each of these functions takes a single argument which specifies the length of the vector. Thus, `logical(3)` generates the same initialized vector as above.

*Table 2.1: Useful functions for creating vectors.*

Function	Description	Examples
<code>scan</code>	read values any mode	<code>scan()</code> , <code>scan("data")</code>
<code>c</code>	combines values any mode	<code>c(1, 3, 2, 6)</code> , <code>c("yes", "no")</code>
<code>rep</code>	repeat values any mode	<code>rep(NA, 5)</code> , <code>rep(c(1, 2), 3)</code>
<code>:</code>	numeric sequences	<code>1:5</code> , <code>1:-1</code>
<code>seq</code>	numeric sequences	<code>seq(-pi, pi, .5)</code>
<code>vector</code>	initialize vectors	<code>vector('complex', 5)</code>
<code>logical</code>	initialize logical vectors	<code>logical(3)</code>
<code>numeric</code>	initialize numeric vectors	<code>numeric(4)</code>
<code>complex</code>	initialize complex vectors	<code>complex(5)</code>
<code>character</code>	initialize character vectors	<code>character(6)</code>

## Vector Attributes

Table 2.2 summarizes the attributes of a vector.

*Table 2.2: Attributes of a vector.*

Attribute	Description
<code>length</code>	number of values
<code>mode</code>	kind of values
<code>names</code>	value labels

The `names` attribute is an optional attribute used to associate specific information, such as case labels or value identifiers, with each value of the vector. To create a vector with named values, you assign the names with the `names` function:

```
> numbered.letters <- letters
> names(numbered.letters) <- paste("obs", 1:26, sep=" ")
> numbered.letters

obs1 obs2 obs3 obs4 obs5 obs6 obs7 obs8 obs9 obs10 obs11
"a"  "b"  "c"  "d"  "e"  "f"  "g"  "h"  "i"  "j"  "k"
obs12 obs13 obs14 obs15 obs16 obs17 obs18 obs19 obs20 obs21
"l"   "m"   "n"   "o"   "p"   "q"   "r"   "s"   "t"   "u"
obs22 obs23 obs24 obs25 obs26
"v"   "w"   "x"   "y"   "z"
```

In the above example, the first 26 integers are converted to character strings by the `paste` function and then attached to each value. The quotes around the numbers are suppressed in the printing. The actual values of the vector `numbered.letters` are character strings, each containing one letter.

If you specify too many or too few names for the values, S-PLUS gives an error message.

# MATRICES

Matrices are used to arrange values by rows and columns in a rectangular table. For data analysis, different variables are usually represented by different columns, and different cases or subjects are represented by different rows. Thus matrices are convenient for grouping together observations that have been measured on the same set of subjects and variables.

Matrices differ from vectors by having a `dim` attribute, which specifies the *dimension* of the matrix, that is, the number of rows and columns. Any vector can be turned into a matrix simply by specifying its `dim` attribute.

## Creating Matrices

To create a matrix from an existing vector, use the `dim` function to set the `dim` attribute. To use `dim`, you assign a vector of two integers specifying the number of rows and columns. For example:

```
> mat <- rep(1:4, rep(3, 4))
> mat

[1] 1 1 1 2 2 2 3 3 3 4 4 4

> dim(mat) <- c(3, 4)
> mat

      [, 1] [, 2] [, 3] [, 4]
[1,]     1     2     3     4
[2,]     1     2     3     4
[3,]     1     2     3     4
```

More often, you need to combine several vectors or matrices into a single matrix. To combine vectors (and matrices) into matrices, use the functions `cbind` and `rbind` functions. The `cbind` function combines vectors column by column, and `rbind` combines vectors row by row. You can easily combine counts for a 2×3 contingency table using `rbind`:

```
> rbind(c(200688, 24, 33), c(201083, 27, 115))

      [, 1] [, 2] [, 3]
[1,] 200688  24   33
[2,] 201083  27  115
```

Use the `cbind` function similarly for columns. When vectors of different lengths are combined using `cbind` or `rbind`, the shorter ones are replicated cyclically so that the matrix is “filled in.” If matrices are combined, they must have matching numbers of rows when using `cbind` and matching numbers of columns when using `rbind`. Otherwise, S-PLUS prints an error message and the objects are not combined.

Use the function `matrix` to convert objects to matrices. Combine the values into a single vector using `c` and then group them by specifying the number of columns or rows. To create a matrix from two vectors, `grp` and `thw`, use `matrix` as follows:

```
> heart <- matrix(c(grp, thw), ncol=2)
```

If you provide fewer values as arguments to `matrix` than are required to complete the matrix, the values are replicated cyclically until the matrix is filled in. If you provide more data than necessary to complete the matrix, excess values are discarded.

If either `ncol` or `nrow` is provided, *but not both*, the missing argument is computed using the following relations:

- `nrow` = the smallest integer equal to or greater than the number of values divided by the number of columns.
- `ncol` = the smallest integer equal to or greater than the number of values divided by the number of rows.

Thus, `nrow` and `ncol` are computed to create the smallest matrix from all the values when `ncol` or `nrow` is given individually.

By default the values are placed in the matrix column by column. That is, all the rows of the first column are filled, then the rows of the second column are filled, etc. To fill the matrix row by row, set the `byrow` argument to `T`. For example:

```
> matrix(1:12, ncol=3, byrow=T)
```

```
      [,1] [,2] [,3]  
[1,]    1    2    3  
[2,]    4    5    6  
[3,]    7    8    9  
[4,]   10   11   12
```

The `byrow` argument is especially useful when reading in data from a text file that is arranged in a table. The data are read in (with `scan`) row by row in this case, so the `byrow` argument is used to place the values in a matrix correctly.

## Matrix Attributes

The attributes of a matrix are presented in Table 2.3.

*Table 2.3: Attributes of a matrix.*

Attribute	Description
<code>length</code>	number of values
<code>mode</code>	kind of values
<code>dim</code>	number of rows and columns
<code>dimnames</code>	row and column names

The `dim` attribute is the defining attribute for matrices. Any data object having a numeric vector of length 2 as its `dim` attribute is a matrix. The `dim` attribute specifies the number of rows and columns of the matrix. You check the dimensions or `dim` attribute of a matrix with the `dim` function:

```
> dim(mat)
```

```
[1] 3 4
```

For a vector you saw that you could assign names to each value with the `names` function. For matrices, you can assign names to the rows and columns with the `dimnames` function. To create a matrix with row and column names of your own, create a list with two components, one for rows and one for columns, and assign them using the `dimnames` function.

```
> dimnames(mat) <- list(paste("row", letters[1:3]),
+ paste("col ", LETTERS[1:4]))
> mat
```

```
      col A col B col C col D
row a    1    2    3    4
row b    1    2    3    4
row c    1    2    3    4
```

In the example above, `letters` and `LETTERS` are character vectors with values the letters of the alphabet in lower and upper case, respectively. The character strings "row" and "col " are replicated to match the length of vectors containing the letters for labeling. The `paste` function binds values into a single character string.

To suppress either row or column labels, use the `NULL` value for the corresponding component of the list. For example, to suppress the row labels and number the columns:

```
> di mnames(mat) <- l i s t(NULL, paste("col ", 1:4))
> mat
```

```
      col 1 col 2 col 3 col 4
[1,]      1      2      3      4
[2,]      1      2      3      4
[3,]      1      2      3      4
```

To specify the row and column labels when defining a matrix with `matrix`, use the optional argument `di mnames` as follows:

```
> mat2 <- matri x(1:12, ncol=4,
+ di mnames=l i s t(NULL, paste("col ", 1:4)))
```

A second set of functions for working with matrices is described in Chapter 14, The Object-Oriented Matrix Library, of the *Guide to Statistics, Volume 2*. The library includes constructor functions for a `Matrix` class and numerous subclasses, and methods for many matrix computations based on the LAPACK library of numerical Fortran routines.

# ARRAYS

Arrays generalize matrices by extending the `dim` attribute to more than two dimensions. If the rows and columns of a matrix are the length and width of a rectangular arrangement of equal-sized cubes, then length, width, and height represent the dimensions of a three-way array. You can visualize a series of equal-sized rectangles or cubes stacked one on top of the other to form a three-dimensional box. The box is composed of cells (the individual cubes) and each cell is specified by its position along the length, width, and height of the box. An example of a three-dimensional array is the `iris` data set in S-PLUS. The first two cases are presented here:

```
> iris[1:2,,]

, , Setosa
      Sepal L. Sepal W. Petal L. Petal W.
[1,]      5.1      3.5      1.4      0.2
[2,]      4.9      3.0      1.4      0.2
, , Versicolor
      Sepal L. Sepal W. Petal L. Petal W.
[1,]      7.0      3.2      4.7      1.4
[2,]      6.4      3.2      4.5      1.5
, , Virginica
      Sepal L. Sepal W. Petal L. Petal W.
[1,]      6.3      3.3      6.0      2.5
[2,]      5.8      2.7      5.1      1.9
```

The data present 50 observations of sepal length and width and petal length and width for each of three species of iris (Setosa, Versicolor, and Virginica). The `dim` attribute of `iris` represents the length, width, and height in the box analogy:

```
> dim(iris)

[1] 50 4 3
```

There is no limit to the number of dimensions of an array. Additional dimensions are represented in the `dim` attribute as additional values in the vector; the number of values is the number of dimensions. From this, we can think of a matrix as a two-dimensional array and a vector as a one-dimensional array.

## Creating Arrays

To create an array in S-PLUS, use the `array` function. The `array` function is analogous to `matrix`. It takes data and the appropriate dimensions as arguments, then produces the array. If no data is supplied, the array is filled with `NA`s.

When passing values to `array`, combine them in a vector so that the first dimension varies fastest, the second dimension the next fastest, and so on. The following example shows how this works:

```
> array(c(1: 8, 11: 18, 111: 118), dim=c(2, 4, 3))

, , 1
     [, 1] [, 2] [, 3] [, 4]
[1,]      1      3      5      7
[2,]      2      4      6      8
, , 2
     [, 1] [, 2] [, 3] [, 4]
[1,]     11     13     15     17
[2,]     12     14     16     18
, , 3
     [, 1] [, 2] [, 3] [, 4]
[1,]    111    113    115    117
[2,]    112    114    116    118
```

The first dimension (the rows) is incremented first. This is equivalent to placing the values column by column. The second dimension (the columns) is incremented second. The third dimension is incremented by filling a matrix for each level of the third dimension.

For creating arrays from existing vectors, the `dim` function works for arrays in the same way it works for matrices. The `dim` function lets you set the `dim` attribute as you can for a matrix. For example, if the data above were stored in the vector `vec`, you could create the above array by defining the `dim` attribute with the vector `c(2, 4, 3)`:

```
> vec

[1] 1 2 3 4 5 6 7 8 11 12 13
[12] 14 15 16 17 18 111 112 113 114 115 116
[23] 117 118

> dim(vec) <- c(2, 4, 3)
```

To name each level of each dimension, use the `dimnames` argument to `array`. This passes a *list* of names in the same way as is done for matrices. For more information on `dimnames`, see the section Matrix Attributes (page 53).



# Array Attributes

The attributes for an array are identical to the attributes for a matrix. For more information, including examples, see the section Matrix Attributes (page 53). Table 2.4 presents each attribute with a brief description.

*Table 2.4: Attributes of an array.*

Attribute	Description
<code>length</code>	number of values
<code>mode</code>	kind of values
<code>dim</code>	vector with size for each dimension
<code>dimnames</code>	list of names for each dimension

## LISTS

Up to this point, all the data types described have been *atomic*, meaning they contain data of only one mode. Often, however, you need to create objects that not only contain data of mixed modes but also preserve the mode of each value. For example, the attributes of an array may contain both the `dim` attribute (a numeric vector), and the `dimnames` attribute (a character vector), and it is important to preserve those modes:

```
> attributes(iris)

$dim:
[1] 50 4 3

$dimnames:
$dimnames[[1]]:
character(0)

$dimnames[[2]]:
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."

$dimnames[[3]]:
[1] "Setosa" "Versicolour" "Virginica"
```

The value returned by `attributes` is a simple example of an S-PLUS *list*. Lists are a very general data type. Lists are made up of *components*, where each component consists of one data object, of any type. That is, from component to component, the *mode* and *type* of the object can change.

For example, the attributes list for the `iris` data set consists of two components, a `dim` component and a `dimnames` component. The `dim` component is a numeric vector of length three. The `dimnames` component is another list with three components. The first component is an empty character vector (`character(0)`), the second component is a vector of four character strings indicating whether the measurement is sepal length or width or petal length or width, and the third component is a vector of three character strings specifying the species of iris.

### Creating Lists

To create a list, use the `list` function. Each argument to `list` defines a component of the list. Naming an argument, using the form *name=component*, creates a name for the corresponding component. For example, you can create a list from the two vectors `grp` and `thw` as follows:

---

```
> grp <- c(rep(1, 11), rep(2, 10))
> thw <- c(450, 760, 325, 495, 285, 450, 460, 375, 310, 615, 425, 245,
+ 350, 340, 300, 310, 270, 300, 360, 405, 290)
> heart.list <- list(group=grp, thw=thw,
+ descrip="heart data")
> heart.list
```

```
$group:
```

```
[1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

```
$thw:
```

```
[1] 450 760 325 495 285 450 460 375 310 615 425 245 350
```

```
[14] 340 300 310 270 300 360 405 290
```

```
$descrip:
```

```
[1] "heart data"
```

The first component of the list contains a numeric vector with grouping information for the data, so it is named `group`. The second component is the total heart weight (`thw`) in grams. The name of the component is the same as the name of the object stored in that component. The `thw` on the left of the equal sign is the component name and the `thw` on the right of the equal sign is the object stored there. The third component contains a character vector which briefly describes the data.

To access a list component, specify the name of the list and the name of the component, separated by a `$`. For example, to display the grouping data:

```
> heart.list$group
```

```
[1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

More generally, you can access list components by an index number enclosed in double brackets (`[[ ]]`). For example, the grouping information can also be accessed by:

```
> heart.list[[1]]
```

```
[1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

Once you've accessed a component, you can specify particular values of the component in the usual way, using the single bracket `[ ]` notation. For example, since the `group` component is a vector, you can obtain the 11th and 12th elements with:

```
> heart.list[[1]][11:12]
```

```
[1] 1 2
```

or

```
> heart.list$group[11:12]
```

```
[1] 1 2
```

If you define a list without naming the components, components can be accessed only using the double bracket notation. When the components are named you can use either the double bracket notation or the names convention with a `$` separating the list name and the component name.

## List Attributes

The usual attributes of `length` and `mode` are implicit for lists as for any object. The `length` attribute gives the number of components. This is equivalent to specifying the number of “values” of a list, where each value is an entire data object.

```
> length(heart.list)
```

```
[1] 3
```

The mode of a list is `"list"`. This provides a way to classify lists, even though their components may be mixed in mode.

```
> mode(heart.list)
```

```
[1] "list"
```

The `names` attribute specifies the names of the components of a list. Use the `names` function to see it:

```
> names(heart.list)
```

```
[1] "group" "thw" "descrip"
```

---

The names of a list's components can be changed by assigning them with the `names` function:

```
> names(heart.list) <- c("group", "total heart weight",  
+ "descri p")  
> names(heart.list)
```

```
[1] "group" "total heart weight" "descri p"
```

Table 2.5 summarizes the attributes of a list and gives a brief description of each.

*Table 2.5: Attributes of a list.*

Attribute	Description
<code>length</code>	number of values
<code>mode</code>	"list"
<code>names</code>	names for each component

## FACTORS AND ORDERED FACTORS

In data analysis, many kinds of data are qualitative rather than quantitative or numeric. If observations can be assigned only to a category, rather than given a specific numeric value, they are termed qualitative or categorical. The values assigned to these variables are typically short character descriptions of the category to which the observation belongs. The following lists some examples of categorical variables:

- *gender*, where the values are "male" and "female".
- *marital status*, where the values might be "single", "married", "separated", "divorced".
- *experimental status*, where the values might be "treatment" and "control".

Categorical data in S-PLUS is represented with a data type called a factors. The data frame `fuel.frame` has a variable named `Type` which classifies each automobile as either Small, Sporty, Compact, Medium, Large, or Van.

```
> fuel.frame$Type
```

```
[1] Small Small Small Small Small Small Small
[8] Small Small Small Small Small Small Sporty
[15] Sporty Sporty Sporty Sporty Sporty Sporty Sporty
[22] Sporty Compact Compact Compact Compact Compact Compact
[29] Compact Compact Compact Compact Compact Compact Compact
[36] Compact Compact Medium Medium Medium Medium Medium
[43] Medium Medium Medium Medium Medium Medium Medium
[50] Medium Large Large Large Van Van Van
[57] Van Van Van Van
```

When you print a factor, the values correspond to the *level* of the factor for each data point or observation. Internally, a factor keeps track of the levels or different categorical values contained in the data and indices which point to the appropriate level for each data point. The different levels of a factor are stored in an attribute called "levels".

Factor objects are a natural form for categorical data in an object-oriented programming environment, because they have a "class" attribute that allows specific method functions to be developed for them. For example, the

generic `print` function uses the `print.factor` method to print factors. If you override `print.factor` by calling `print.default`, you can see how a factor is stored internally.

```
> print.default(fuel.frame$Type)

[1] 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 1 1 1
[26] 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3
[51] 2 2 2 6 6 6 6 6 6 6 6
attr(,"levels"):
[1] "Compact" "Large" "Medium" "Small" "Sporty" "Van"
attr(,"class"):
[1] "factor"
```

The integers serve as indices to the values in the "levels" attribute. You can return the integer indices directly with the `codes` function.

```
> codes(fuel.frame$Type)

[1] 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 1 1 1
[26] 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3
[51] 2 2 2 6 6 6 6 6 6 6 6
```

Or, you can examine the "levels" of a factor with the `levels` function.

```
> levels(fuel.frame$Type)

[1] "Compact" "Large" "Medium" "Small" "Sporty" "Van"
```

The `print.factor` function is roughly equivalent to

```
> levels(fuel.frame$Type)[codes(fuel.frame$Type)]
```

except the quotes are dropped. To get the number of cases of each level in a factor, call `summary`:

```
> summary(fuel.frame$Type)

Compact Large Medium Small Sporty Van
      15      3      13      13      9      7
```

## Creating Factors

To create a factor, use the `factor` function. The `factor` function takes data with categorical values and creates a data object of class "factor". For example, you can categorize a group of 10 students by gender as follows:

```
> classl ist <- c("mal e", "femal e", "mal e", "mal e", "mal e",
+ "femal e", "femal e", "mal e", "femal e", "mal e")

> factor(classl ist)

[1] mal e femal e mal e mal e mal e femal e femal e mal e
[9] femal e mal e
```

S-PLUS creates two levels with labels "femal e", and "mal e", respectively.

*Table 2.6: Arguments to factor.*

Argument	Description
<code>x</code>	data, to be thought of as taking values on the finite set of levels.
<code>l evel s</code>	optional vector of <code>l evel s</code> for the factor. The default value of levels is the <i>sorted</i> list of distinct values of <code>x</code> .
<code>l abel s</code>	optional vector of values to use as labels for the <code>l evel s</code> of the factor. The default is <code>as.character(l evel s)</code> .
<code>excl ude</code>	a vector of values to be excluded from forming <code>l evel s</code> .

The `l evel s` argument allows you to specify the levels you want to use or to order them the way you want. For example, if you want to include certain categories in an analysis, you can specify them with the `l evel s` argument. Any values omitted from the `l evel s` argument are considered missing.

```
> intensi ty <- factor(c("Hi ", "Med", "Lo", "Hi ", "Hi ", "Lo"),
+ l evel s = c("Lo", "Hi "))
> intensi ty

[1] Hi  NA Lo Hi  Hi  Lo

> l evel s(intensi ty)

[1] "Lo" "Hi "
```

If you had left the `l evel s` argument off, the "l evel s" would have been ordered alphabetically as "Hi ", "Low", "Medi um". You use the `l abel s` argument if you want the levels to be something other than the original data.



```
> factor(c("Hi ", "Lo", "Med", "Hi ", "Hi ", "Lo"),
+ level s=c("Lo", "Hi "), label s = c("LowDose", "Hi ghDose"))
```

```
[1] Hi ghDose LowDose NA Hi ghDose Hi ghDose LowDose
```

### Warning: ordering of levels and labels arguments

If you provide the `level s` and `label s` arguments, then you must order them in the same way. If you don't provide the `level s` argument but do provide the `label s` argument, then you must order the labels the same way S-PLUS orders the levels of the factor, which is alphabetically for character strings and numerically for a numeric vector which is converted to a factor.

Use the `excl ude` argument to indicate which values to exclude from the levels of the resulting factor. Any value that appears in both `x` and `excl ude` will be `NA` in the result and will not appear in the "level s" attribute. The `intensi ty` factor could alternatively have been produced with:

```
> factor(c("Hi ", "Med", "Lo", "Hi ", "Hi ", "Lo"),
+ excl ude =c("Med"))
```

```
[1] Hi NA Lo Hi Hi Lo
```

## Creating Ordered Factors

If the *order* of the levels of a factor is important, you can represent the data as a special type of factor called an *ordered factor*. Use the `ordered` function to create ordered factors. The arguments to `ordered` are the same as those to `factor`. To create an ordered version of the `intensity` factor do:

```
> ordered(c("Hi ", "Med", "Lo", "Hi ", "Hi ", "Lo"),
+ level s=c("Lo", "Med", "Hi "))
```

```
[1] Hi Med Lo Hi Hi Lo
```

```
Lo < Med < Hi
```

The order relationship between the different levels is printed for an ordered factor along with the values. The order of the values used in the `level s` argument determines the order placed on the levels.

### Warning

If you don't provide a `level s` argument, an ordering will be placed on the levels corresponding to the default ordering of the levels by S-PLUS.

## Creating Factors from Continuous Data

To create categorical data out of numerical or continuous data, use the `cut` function. You provide either a vector of specific break points or an integer specifying how many groups to divide the numerical data into, then `cut` creates levels corresponding to the specified ranges. All the values falling in any particular range are assigned the same level. For example, the murder rates in the 50 states can be grouped into "Hi gh" and "Low" values using `cut`:

```
> cut(state.x77[, "Murder"], breaks=c(0, 8, 16))

[1] 2 2 1 2 2 1 1 1 2 2 1 1 2 1 1 1 2 2 1 2 1 2 1 2 2
[26] 1 1 2 1 1 2 2 2 1 1 1 1 1 1 2 1 2 2 1 1 2 1 1 1 1
attr(, "levels"):
[1] " 0+ thru 8" "8+ thru 16"
```

The breakpoints must completely enclose the values you want included in the factors. *Data less than or equal to the first breakpoint or greater than the last breakpoint are returned as NA.*

To create a specific number of groups, by partitioning the range of the data into equal-sized intervals, use an integer value for the `breaks` argument:

```
> cut(state.x77[, "Murder"], breaks=2)

[1] 2 2 1 2 2 1 1 1 2 2 1 1 2 1 1 1 2 2 1 2 1 2 1 2 2
[26] 1 1 2 1 1 2 2 2 1 1 1 1 1 1 2 1 2 2 1 1 2 1 1 1 1
attr(, "levels"):
[1] "1.263+ thru 8.250" "8.250+ thru 15.237"
```

By default, `cut` creates *labels* of the form *first breakpoint thru second breakpoint*, etc., using either the breakpoints you provide or the ones it creates. However, you can assign different labels to the levels with the `label s` argument.

```
> cut(state.x77[, "Murder"], c(0, 8, 16),
+ label s=c("Low", "Hi gh"))

[1] 2 2 1 2 2 1 1 1 2 2 1 1 2 1 1 1 2 2 1 2 1 2 1 2 2
[26] 1 1 2 1 1 2 2 2 1 1 1 1 1 1 2 1 2 2 1 1 2 1 1 1 1
attr(, "levels"):
[1] "Low" "Hi gh"
```

### Note

As you may notice from the style of printing in the above examples, `cut` does not produce factors directly. Rather, the value returned by `cut` is a *category* object.

To create a factor from the output of `cut`, just call `factor` with the call to `cut` as its only argument:

```
> factor(cut(state.x77[, "Murder"], c(0, 8, 16),
+ labels=c("Low", "High")))

[1] High High Low High High Low Low Low High High
[11] Low Low High Low Low Low High High Low High
[21] Low High Low High High Low Low High Low Low
[31] High High High Low Low Low Low Low Low High
[41] Low High High Low Low High Low Low Low Low
```

## Factor Attributes

Factor data objects are vectors, so in addition to the `length` and `mode` attributes they may have a `names` attribute. The `mode` attribute is always `"numeric"`, because, internally, the values of a factor are integers specifying which level each observation belongs to.

*Table 2.7: Attributes of a factor.*

Attribute	Description
<code>length</code>	number of values
<code>mode</code>	<code>"numeric"</code>
<code>names</code>	case or value labels
<code>levels</code>	values of the factor object
<code>class</code>	<code>"factor"</code>

The defining attribute of a factor is the `"class"` attribute which has the value `"factor"`. Table 2.7 lists the attributes of a factor object. For more information about the `length`, `mode` and `names` attributes, see the section [Vector Attributes](#) (page 49). Ordered factors have the same attributes as factors, except the value of the `"class"` attribute is `"ordered"` `"factor"`.



---

<b>The Benefits of Data Frames</b>	<b>70</b>
<b>Creating Data Frames</b>	<b>71</b>
<b>Combining Data Frames</b>	<b>76</b>
Combining Data Frames by Column	76
Combining Data Frames by Row	78
Merging Data Frames	79
<b>Applying Functions to Subsets of a Data Frame</b>	<b>82</b>
<b>Adding New Classes of Variables to Data Frames</b>	<b>88</b>
<b>Data Frame Attributes</b>	<b>90</b>

Data frames are data objects designed primarily for data analysis and modeling. You can think of them as *generalized* matrices—generalized in a way different from the way arrays generalize matrices. Arrays generalize the *dimensional* aspect of a matrix; data frames generalize the *mode* aspect of a matrix. Matrices can be of only one mode (for example, "l o g i c a l ", "n u m e r i c", "c o m p l e x", "c h a r a c t e r"). Data frames, however, allow you to mix modes from column to column. For example, you could have a column of "character" values, a column of "n u m e r i c" values, a column of categorical values, and a column of "l o g i c a l " values. Each column of a data frame corresponds to a particular variable; each row corresponds to a single “case” or set of observations.

## THE BENEFITS OF DATA FRAMES

The main benefit of a data frame is that it allows you to mix data of different types into a single object in preparation for analysis and modeling. The idea of a data frame is to group data by variables (columns) regardless of their type. Then all the observations on a particular set of variables can be grouped into a single data frame. This is particularly useful in data analysis where it is typical to have a "character" variable labeling each observation, one or more "numeric" variables of observations, and one or more categorical variables for grouping observations. An example is a built-in data set, `solder`, with information on a welding experiment conducted by AT&T at their Dallas factory.

```
> sampleruns <- sample(row.names(solder), 10)
> solder[ sampleruns, ]
```

	Opening	Solder	Mask	PadType	Panel	skips
380	L	Thick	A3	L7	2	0
545	L	Thick	B3	D4	2	0
462	L	Thin	A3	D6	3	3
809	S	Thick	B6	L9	2	7
609	S	Thick	B3	L4	3	19
492	M	Thin	A6	D6	3	8
525	S	Thin	A6	L6	3	18
313	M	Thin	A3	L6	1	1
408	M	Thick	A6	D7	3	11
540	S	Thin	A6	L9	3	22

A sample of 10 of the 900 observations is presented for all six variables. The variable `skips` is the outcome which measures the number of visible soldering skips on a particular run of the experiment. The other variables are categorical and describe the levels of various factors which define the run. The row names on the left are the run numbers for the experiment. Combined in `solder` are character data (the row names), categorical data (the factors), and numeric data (the outcome).

# CREATING DATA FRAMES

You can create data frames in several ways.

- `read.table` reads in data from an external file.
- `data.frame` binds together S-PLUS objects of various kinds.
- `as.data.frame` coerces objects of a particular type to objects of class `data.frame`.

You can also combine existing data frames in several ways, using the `cbind`, `rbind`, and `merge` functions.

The `read.table` function reads data stored in a text file in table format directly into S-PLUS. It is discussed in detail in the section Data Input (page 156). The `as.data.frame` function is primarily a support function for the top-level `data.frame` function—it provides a mechanism for defining how new variable classes should be included in newly-constructed data frames. This mechanism is discussed further in the section Adding New Classes of Variables to Data Frames (page 88).

For most purposes, when you want to create or modify data frames within S-PLUS, you use the `data.frame` function or one of the combining functions `cbind`, `rbind` or `merge`. This section focuses specifically on the `data.frame` function for combining S-PLUS objects into data frames. The following section discusses the functions for combining existing data frames.

The `data.frame` function is used for creating data frames from existing S-PLUS data objects rather than from data in an external text file. The only required argument to `data.frame` is one or more data objects. All of the objects must produce columns of the same length. Vectors must have the same number of observations as the number of rows of the data frame, matrices must have the same number of rows as the data frame, and lists must have components that match in lengths for vectors or rows for matrices. If the objects don't match appropriately, you get an error message saying the "arguments imply differing number of rows". For example, suppose we have vectors of various modes, each having length 20, along with a matrix with two columns and 20 rows, and a data frame with 20 observations for each of three variables. We can combine these into a data frame as follows.

```
> my.logical <- sample(c(T,F), size=20, replace=T)
> my.complex <- rnorm(20) + runif(20)*1i
> my.numeric <- rnorm(20)
```

```

> my.matrix <- matrix(rnorm(40), ncol=2)
> my.df <- kyphosis[1:20, 1:3]
> my.df2 <- data.frame(my.logical, my.complex, my.numeric,
+ my.matrix, my.df)
> my.df2

```

	my.logical		my.complex	my.numeric
1	FALSE	-1.8831606111+0.501943978i	1.09345678	
2	FALSE	0.3368386818+0.858758209i	0.09873739	
3	TRUE	-0.0003541437+0.381377962i	-0.91776485	
4	FALSE	1.2066770747+0.006793533i	-1.76152800	
5	FALSE	-0.0204049459+0.158040394i	0.30370197	
6	FALSE	-1.0119328923+0.860326129i	-0.52486689	
7	FALSE	0.9163081264+0.474985190i	1.46745534	
8	FALSE	-1.3829848791+0.932033515i	0.45363152	
9	FALSE	-0.4695526978+0.795743512i	0.40777969	
10	TRUE	-0.8035892599+0.256793795i	0.53622210	
11	TRUE	0.9026407992+0.637563583i	0.07595690	
12	TRUE	-1.1558698525+0.655271475i	0.32395563	
13	FALSE	0.1049802819+0.706128572i	-1.35316648	
14	TRUE	0.2302154933+0.373451429i	-2.42261503	
16	FALSE	2.3956811151+0.086245694i	0.34412995	
17	TRUE	0.0824999817+0.258623377i	2.46456956	
18	FALSE	-0.0248816697+0.417373099i	2.99062594	
19	TRUE	0.7525617816+0.636045368i	-1.55640891	
20	TRUE	-1.1078423455+0.011345901i	1.27173450	
21	TRUE	-2.2280610717+0.517812594i	1.54472022	

	X1	X2	Kyphosis	Age	Number
1	0.80316229	2.28681400	absent	71	3
2	-0.58580658	-0.06509133	absent	158	3
3	0.88756407	-0.89849793	present	128	4
4	-2.35672715	0.68797076	absent	2	5
5	1.26986158	-0.76204606	absent	1	4
6	-1.10805175	-1.02164143	absent	1	2
7	0.56273335	1.34946448	absent	61	2
8	0.24542337	1.35936982	absent	37	3
9	0.29190516	2.24852247	absent	113	2
10	0.98675866	-1.27076525	present	59	6
11	0.10125951	0.19835740	present	82	5
12	0.30351481	2.48467422	absent	148	3
13	0.04480753	-1.60470965	absent	18	5
14	1.43504492	1.35172992	absent	1	4



---

```

16 -2.45929501 -0.58286780 absent 168 3
17 0.90746053 -0.48598155 absent 1 3
18 0.50886476 0.96350421 absent 78 6
19 -1.11844146 -0.56341008 absent 175 5
20 0.51371598 1.32382209 absent 80 5
21 0.58229738 -0.87364793 absent 27 4

```

The names of the objects are used for the variable names in the data frame. Row names for the data frame are obtained from the first object with a `names`, `dimnames`, or `row.names` attribute having *unique* values. In the above example, the object was `my.df`:

```

> my.df
      Kyphosi s Age Number
1      absent  71      3
2      absent 158      3
3    present 128      4
4      absent   2      5
5      absent   1      4
6      absent   1      2
7      absent  61      2
8      absent  37      3
9      absent 113      2
10    present  59      6
11    present  82      5
12      absent 148      3
13      absent  18      5
14      absent   1      4
16      absent 168      3
17      absent   1      3
18      absent  78      6
19      absent 175      5
20      absent  80      5
21      absent  27      4

```

The row names are *not* just the row numbers—in our subset, the number 15 is missing. The fifteenth row of `kyphosi s`, and hence `my.df`, has the row name "16".

The attributes of special types of vectors (such as factors) are not lost when they are combined in a data frame. They can be retrieved by asking for the attributes of the particular variable of interest. More detail is given in the section *Data Frame Attributes* (page 90).

Each vector adds one variable to the data frame. Matrices and data frames provide as many variables to the new data frame as they have columns or variables, respectively. Lists, because they can be built from virtually any data object, are more complicated—they provide as many variables as all of their components taken together.

When combining objects of different types into a data frame, some objects may be altered somewhat to be more suitable for further analysis. For example, numeric vectors and factors remain unchanged in the data frame. Character and logical vectors, however, are converted to factors before being included in the data frame. The conversion is done because S-PLUS assumes that character and logical data will most commonly be taken to be a categorical variable in any modeling that is to follow. If you want to keep a character or logical vector “as is” in the data frame, pass the vector to `data.frame` wrapped in a call to the `I` function, which returns the vector unchanged but with the added class “AsIs”.

For example, consider the following logical vector, `my.logical`:

```
> my.logical  
[1] T T T T T F T T F T T F T F T T T T T
```

We can combine it as is with a numeric vector `rnorm(20)` in a data frame as follows:

```
> my.df <- data.frame(a=rnorm(20), b=I(my.logical))  
> my.df  
      a b  
1 -0.6960192 T  
2  0.4342069 T  
3  0.4512564 T  
4 -0.8785964 T  
5  0.8857739 T  
6 -0.2865727 F  
7 -1.0415919 T  
8 -2.2958470 T  
9  0.7277701 F  
10 -0.6382045 T  
11 -0.9127547 T  
12  0.1771526 F  
13  0.5361920 T  
14  0.3633339 F  
15  0.5164660 T
```

```

16  0.4362987 T
17 -1.2920592 T
18  0.8314435 T
19 -0.6188006 T
20  1.4910625 T

```

```
> mode(my.df$b)
```

```
[1] "logical"
```

You can provide a character vector as the `row.names` argument to `data.frame`. Just make sure it is the same length as the data objects you are combining into the data frame.

```

> data.frame(price, country, reliability, type,
+ row.names=c("Acura", "Audi", "BMW", "Chev", "Ford",
+ "Mazda", "MazdaMX", "Nissan", "Olds", "Toyota"))

```

	price	country	reliability	type
Acura	11950	Japan	5	Small
Audi	26900	Germany	NA	Medium
BMW	15900	Germany	NA	Small
Chev	13900	USA	1	Small
Ford	12900	USA	1	Small
Mazda	10900	Japan	2	Small
MazdaMX	11900	Japan	2	Small
Nissan	14900	Japan	2	Small
Olds	16900	USA	1	Small
Toyota	17900	Japan	2	Small

## COMBINING DATA FRAMES

We have already seen one way to combine data frames—since data frames are legal inputs to the `data.frame` function, you can use `data.frame` directly to combine one or more data frames. For certain specific combinations, other functions may be more appropriate. This section discusses three general cases:

1. Combining data frames *by column*. This case arises when you have new variables to add to an existing data frame, or have two or more data frames having observations of different variables for identical subjects. The principal tool in this case is the `cbind` function.
2. Combining data frames *by row*. This case arises when you have multiple studies providing observations of the same variables for different sets of subjects. For this task, use the `rbind` function.
3. Merging (or *joining*) data frames. This case arises when you have two data frames containing some information in common, and you want to get as much information as possible from both data frames about the overlapping cases. For this case, use the `merge` function.

All three of the functions mentioned above (`cbind`, `rbind`, and `merge`) have methods for data frames, but in the usual cases, you can simply call the generic function and obtain the correct result.

### Combining Data Frames by Column

Suppose you have a data frame consisting of factor variables defining an experimental design. When the experiment is complete, you can add the vector of observed responses as another variable in the data frame. In this case, you are simply adding another column to the existing data frame, and the natural tool for this in S-PLUS is the `cbind` function. For example, consider the simple built-in design matrix `oa.4.2p3`, representing a half-fraction of a  $2^4$  design.

```
> oa.4.2p3
```

```
   A  B  C
1 A1 B1 C1
2 A1 B2 C2
3 A2 B1 C2
4 A2 B2 C1
```

If we run an experiment with this design, we obtain a vector of length four, one observation for each row of the design data frame. We can combine the observations with the design using `cbind` as follows.

```
> run1 <- cbind(oa.4.2p3, resp=c(46, 34, 44, 30))
> run1
```

```
      A  B  C resp
1 A1 B1 C1 46
2 A1 B2 C2 34
3 A2 B1 C2 44
4 A2 B2 C1 30
```

Another use of `cbind` is to bind a constant vector to a data frame, as in the following example.

```
> fuel1 <- cbind(1, fuel.frame)
> fuel1
```

```
      1 Weight Displacement Mileage Fuel Type
Eagle Summit 4 1 2560 97 33 3.030303 Small
Ford Escort 4 1 2345 114 33 3.030303 Small
Ford Festiva 4 1 1845 81 37 2.702703 Small
Honda Civic 4 1 2260 91 32 3.125000 Small
Mazda Protege 4 1 2440 113 32 3.125000 Small
. . .
```

As a more substantial example, consider the built-in data sets `cu.summary`, `cu.specs`, and `cu.dimensions`. Each of these data sets contains observations about a number of car models, but the list of car models is slightly different in each. All, however, contain data for the cars listed in the data set `common.names`.

```
> common.names

[1] "Acura Integra" "Acura Legend"
[3] "Audi 100"      "Audi 80"
[5] "BMW 325i"      "BMW 535i"
[7] "Buick Century" "Buick Electra"
. . .
```

The data sets `match.summary`, `match.specs`, and `match.dims` contain the row subscripts to obtain observations about the models listed in `common.names` from, respectively, `cu.summary`, `cu.specs`, and `cu.dimensions`. We can use these data sets and the `cbind` function to compile a general car information data set.

```
> car.mi.ne <- cbind(cu.dimensions[match.dims, ],
+ cu.specs[match.specs, ], cu.summary[match.summary, ],
+ row.names=common.names)
```

Compare `car.mi.ne` to the built-in data set `car.all`, constructed in a similar fashion.

## Combining Data Frames by Row

Suppose you are pooling the data from several research studies. You have data frames with observations of equivalent, or roughly equivalent, variables for several sets of subjects. Renaming variables as necessary, you can subscript the data sets to obtain new data sets having a common set of variables. You can then use `rbind` to obtain a new data frame containing all the observations from the studies.

For example, consider the following data frames.

```
> rand.df1
```

	norm	uni f	bi nom
1	1.64542042	0.45375156	41
2	1.64542042	0.83783769	44
3	-0.13593118	0.31408490	53
4	0.26271524	0.57312325	34
5	-0.01900051	0.25753044	47
6	0.14986005	0.35389326	41
7	0.07429523	0.53649764	43
8	-0.80310861	0.06334192	38
9	0.47110022	0.24843933	44
10	-1.70465453	0.78770638	45

```
> rand.df2
```

	norm	bi nom	chi sq
1	0.3485193	50	19.359238
2	1.6454204	41	13.547288
3	1.4330907	53	4.968438
4	-0.8531461	55	4.458559
5	0.8741626	47	2.589351

These data frames have the common variables `norm` and `bi nom`; we subscript and combine the resulting data frames as follows.

```
> rbind(rand.df1[, c("norm", "bi nom")],
+ rand.df2[, c("norm", "bi nom")])
```

```

      norm  bi nom
1    1. 64542042    41
2    1. 64542042    44
3   -0. 13593118    53
4    0. 26271524    34
5   -0. 01900051    47
6    0. 14986005    41
7    0. 07429523    43
8   -0. 80310861    38
9    0. 47110022    44
10  -1. 70465453    45
11   0. 34851926    50
12   1. 64542042    41
13   1. 43309068    53
14  -0. 85314606    55
15   0. 87416262    47

```

**Warning**

Use `rbind` (and, in particular, `rbind.data.frame`) only when you have complete data frames, as in the above example. Do not use it in a loop to add one row at a time to an existing data frame—this is very inefficient. To build a data frame, write all the observations to a data file and use `read.table` to read it in.

**Merging Data Frames**

In many situations, you may have data from multiple sources with some duplicated data. To get the cleanest possible data set for analysis, you want to *merge* or *join* the data before proceeding with the analysis. For example, player statistics extracted from *Total Baseball* overlap somewhat with player statistics extracted from *The Baseball Encyclopedia*. You can use the `merge` function to join two data frames by their common data. For example, consider the following made-up data sets.

```
> baseball.off
```

```

      player years.ML  BA HR
1 Whitehead      4 0.308 10
2     Jones      3 0.235 11
3     Smith      5 0.207  4
4   Russell     NA 0.270 19
5     Ayer      7 0.283  5

```

```
> baseball.def
```

	player	years	ML	A	FA
1	Smith	5	300	0.974	
2	Jones	3	7	0.990	
3	Whitehead	4	9	0.980	
4	Russell	NA	55	0.963	
5	Ayer	7	532	0.955	

These can be merged by the two columns they have in common using `merge`:

```
> merge(baseball.off, baseball.def)
```

	player	years	ML	BA	HR	A	FA
1	Ayer	7	0.283	5	532	0.955	
2	Jones	3	0.235	11	7	0.990	
3	Russell	NA	0.270	19	55	0.963	
4	Smith	5	0.207	4	300	0.974	
5	Whitehead	4	0.308	10	9	0.980	

By default, `merge` joins by the columns having common names in the two data frames. You can specify different combinations using the `by`, `by.x`, and `by.y` arguments. For example, consider the data sets `authors` and `books`.

```
> authors
```

	FirstName	LastName	Age	Income	Home
1	Lorne	Green	82	1200000	California
2	Loren	Bl ye	40	40000	Washington
3	Robin	Green	45	25000	Washington
4	Robin	Howe	2	0	Alberta
5	Billy	Jaye	40	27500	Washington

```
> books
```

	AuthorFirstName	AuthorLastName	Book
1	Lorne	Green	Bonanza
2	Loren	Bl ye	Mi dwi fery
3	Loren	Bl ye	Gardeni ng
4	Loren	Bl ye	Perenni al s
5	Robin	Green	Who_dun_i t?
6	Rich	Cal away	Spl us

The data sets have different variable names, but overlapping information. Using the `by.x` and `by.y` arguments to `merge`, we can join the data sets by the first and last names:



---

```
> merge(authors, books, by.x=c("FirstName", "LastName"),
+ by.y=c("AuthorFirstName", "AuthorLastName"))
```

	FirstName	LastName	Age	Income	Home	Book
1	Loren	Bl ye	40	40000	Washing ton	Mi dwi fery
2	Loren	Bl ye	40	40000	Washing ton	Gardeni ng
3	Loren	Bl ye	40	40000	Washing ton	Perenni al s
4	Lorne	Green	82	1200000	Cal i forni a	Bonanza
5	Robi n	Green	45	25000	Washi ngton	Who_dun_i t?

Because the desired “by” columns are in the same position in both books and authors, we can accomplish the same result more simply as follows.

```
> merge(authors, books, by=1:2)
```

More examples can be found in the `merge` help file.

## APPLYING FUNCTIONS TO SUBSETS OF A DATA FRAME

A common operation on data with factor variables is to repeat an analysis for each level of a single factor, or for all combinations of levels of several factors. SAS users are familiar with this operation as the BY statement. In S-PLUS, you can perform these operations using the `by` or `aggregate` function. Use `aggregate` when you want numeric summaries of each variable computed for each level; use `by` when you want to use all the data to construct a model for each level.

The `aggregate` function allows you to partition a data frame or a matrix by one or more grouping vectors, and then apply a function to the resulting columns. The function must be one that returns a single value, such as `mean` or `sum`. You can also use `aggregate` to partition a time series (univariate or multivariate) by frequency and apply a summary function to the resulting time series.

For data frames, `aggregate` returns a data frame with a factor variable column for each group or level in the index vector, and a column of numeric values resulting from applying the specified function to the subgroups for each variable in the original data frame.

```
> aggregate(state.x77[, c("Population", "Area")],
+           by=state.division, FUN = sum)
```

	Group	Population	Area
1	New England	12187	62951
2	Middle Atlantic	37269	100318
3	South Atlantic	32946	266909
4	East South Central	13516	178982
5	West South Central	20868	427791
6	East North Central	40945	244101
7	West North Central	16691	507723
8	Mountain	9625	856047
9	Pacific	28274	891972

### Warning

For most numeric summaries, *all* variables in the data frame must be numeric. Thus, if we attempt to repeat the above example with the kyphosis data, using kyphosis as the by variable, we get an error:

```
> aggregate(kyphosis, by=kyphosis$Kyphosis, FUN=sum)
```

```
Error in Summary.factor(structure(.Data = c(1, 1, ...:
  A factor is not a numeric object
Dumped
```

For time series, `aggregate` returns a new, shorter time series that summarizes the values in the time interval given by a new frequency. For instance you can quickly extract the yearly maximum, minimum, and average from the monthly housing start data in the time series `hstart`:

```
> aggregate(hstart, nf = 1, fun=max)
```

```
1966: 143.0 137.0 164.9 159.9 143.8 205.9 231.0 234.2 160.9
start del tat frequency
1966      1          1
```

```
> aggregate(hstart, nf = 1, fun=min)
```

```
1966: 62.3 61.7 82.7 85.3 69.2 104.6 150.9 90.6 54.9
start del tat frequency
1966      1          1
```

```
> aggregate(hstart, nf = 1, fun=mean)
```

```
1966: 99.6 110.2 128.8 125.0 122.4 173.7 198.2 171.5 112.6
start del tat frequency
1966      1          1
```

The `by` function allows you to partition a data frame according to one or more categorical indices (conditioning variables) and then apply a function to the resulting subsets of the data frame. Each subset is considered a separate data frame, hence, unlike the `FUN` argument to `aggregate`, the function passed to `by` does *not* need to have a numeric result. Thus, `by` is useful for functions that work on data frames by fitting models, for example.

```
> by(kyphosi s, I NDI CES=kyphosi s$Kyphosi s, FUN=summary)
```

```
kyphosi s$Kyphosi s: absent
```

Kyphosi s	Age	Number	Start
absent : 64	Min. : 1.00	Min. : 2.00	Min. : 1.00
present: 0	1st Qu.: 18.00	1st Qu.: 3.00	1st Qu.: 11.00
	Medi an : 79.00	Medi an : 4.00	Medi an : 14.00
	Mean : 79.89	Mean : 3.75	Mean : 12.61
	3rd Qu.: 131.00	3rd Qu.: 5.00	3rd Qu.: 16.00
	Max. : 206.00	Max. : 9.00	Max. : 18.00

```
.....
```

```
kyphosi s$Kyphosi s: present
```

Kyphosi s	Age	Number	Start
absent : 0	Min. : 15.00	Min. : 3.000	Min. : 1.000
present: 17	1st Qu.: 73.00	1st Qu.: 4.000	1st Qu.: 5.000
	Medi an : 105.00	Medi an : 5.000	Medi an : 6.000
	Mean : 97.82	Mean : 5.176	Mean : 7.294
	3rd Qu.: 128.00	3rd Qu.: 6.000	3rd Qu.: 12.000
	Max. : 157.00	Max. : 10.000	Max. : 14.000

The applied function supplied as the FUN argument must accept a data frame as its first argument; if you want to apply a function that does not naturally accept a data frame as its first argument, you must define a function that does so on the fly. For example, one common application of the by function is to repeat model fitting for each level or combination of levels; the modeling functions, however, generally have a *formula* as their first argument. The following call to by shows how to define the FUN argument to fit a linear model to each level:

```
> by(kyphosi s, I i st(Kyphosi s=kyphosi s$Kyphosi s,
+   OI der=kyphosi s$Age>105),
+   functi on(data)l m(Number~Start, data=data))
```

```
Kyphosi s: absent
```

```
OI der: FALSE
```

```
Call :
```

```
l m(formula = Number~Start, data = data)
```

```
Coeffi ci ents:
```

```
(Intercept)      Start
  4.885736  -0.08764492
```

```
Degrees of freedom: 39 total ; 37 residual
```

```
Resi dual standard error: 1.261852
```

```
Kyphosis: present
Order: FALSE
Call:
lm(formula = Number~Start, data = data)
```

```
Coefficients:
(Intercept)      Start
  6.371257 -0.1191617
Degrees of freedom: 9 total; 7 residual
Residual standard error: 1.170313
```

```
Kyphosis: absent
Order: TRUE
```

```
... .
```

As in the above example, you should define your FUN argument simply. If you need additional parameters for the modeling function, specify them fully in the call to the modeling function, rather than attempting to pass them in through a "... " argument.

### Warning

Again, as with aggregate, you need to be careful that the function you are applying by to works with data frames, and often you need to be careful that it works with factors as well. For example, consider the following two examples.

```
> by(kyphosis, kyphosis$Kyphosis, function(data)
+ apply(data, 2, mean))
```

```
kyphosis$Kyphosis: absent
Kyphosis Age Number      Start
   NA   NA   3.75  12.60938
```

```
kyphosis$Kyphosis: present
Kyphosis      Age      Number      Start
   NA 97.82353  5.176471  7.294118
```

```
Warning messages:
```

```
1: 64 missing values generated coercing from character to
   numeric in: as.double(x)
2: 17 missing values generated coercing from character to
   numeric in: as.double(x)
```

```
> by(kyphosis, kyphosis$Kyphosis, function(data)
+   apply(data, 2, max))
```

```
Error in FUN(x): Numeric summary undefined for mode
"character"
```

```
Dumped
```

The functions `mean` and `max` are not very different, conceptually. Both return a single number summary of their input, both are only meaningful for numeric data. Because of implementation differences, however, the first example returns appropriate values and the second example dumps. However, when all the variables in your data frame are numeric, or when you want to use `by` with a matrix, you should encounter few difficulties.

```
> dimnames(state.x77)[[2]][4] <- "Li fe. Exp"
> by(state.x77[, c("Murder", "Popul ati on", "Li fe. Exp")],
+   state.region, summary)
```

```
INDICES: Northeast
```

	Murder	Popul ati on	Li fe. Exp
Min. :	2.400	Min. : 472	Min. : 70.39
1st Qu. :	3.100	1st Qu. : 931	1st Qu. : 70.55
Medi an :	3.300	Medi an : 3100	Medi an : 71.23
Mean :	4.722	Mean : 5495	Mean : 71.26
3rd Qu. :	5.500	3rd Qu. : 7333	3rd Qu. : 71.83
Max. :	10.900	Max. : 18080	Max. : 72.48

```
INDICES: South
```

	Murder	Popul ati on	Li fe. Exp
Min. :	6.20	Min. : 579	Min. : 67.96
1st Qu. :	9.25	1st Qu. : 2622	1st Qu. : 68.98
Medi an :	10.85	Medi an : 3710	Medi an : 70.07
Mean :	10.58	Mean : 4208	Mean : 69.71
3rd Qu. :	12.27	3rd Qu. : 4944	3rd Qu. : 70.33
Max. :	15.10	Max. : 12240	Max. : 71.42

```
...

```

Closely related to the `by` and aggregate functions is the `tapply` function, which allows you to partition a *vector* according to one or more categorical indices. Each index is a vector of logical or factor values the same length as the data vector; to use more than one index create a list of index vectors.

For example, suppose you want to compute a mean murder rate by region. You can use `tapply` as follows.

```
> tapply(state.x77[, "Murder"], state.region, mean)
```

```
Northeast      South North Central      West
  4.722222 10.58125          5.275 7.215385
```

To compute the mean murder rate by region *and* income, use `tapply` as follows.

```
> income.lev <- cut(state.x77[, "Income"],
+ summary(state.x77[, "Income"])[-4])
> income.lev
```

```
[1] 1 4 3 1 4 4 4 3 4 2 4 2 4 2 3 3 1
[18] 1 1 4 3 3 3 NA 2 2 2 4 2 4 1 4 1 4
[35] 3 1 3 2 3 1 2 1 2 2 1 3 4 1 2 3
attr(,"levels"):
[1] "3098+ thru 3993" "3993+ thru 4519"
[3] "4519+ thru 4814" "4814+ thru 6315"
```

```
> tapply(state.x77[, "Murder"], list(state.region,
income.lev), mean)
```

```
              3098+ thru 3993 3993+ thru 4519
Northeast              4.10000              4.700000
South                  10.64444              13.050000
North Central              NA              4.800000
West                    9.70000              4.933333
              4519+ thru 4814 4814+ thru 6315
Northeast              2.85              6.40
South                  7.85              9.60
North Central          5.52              5.85
West                   6.30              8.40
```





```
> as.data.frame.character
function(x, row.names = NULL, optional = F,
        na.strings = "NA", ...)
  as.data.frame.vector(factor(x, exclude = na.strings),
                        row.names, optional)
```

This method converts its input to a factor, then calls the function `as.data.frame.vector`.

You can create new methods from scratch, provided they have the same arguments as `as.data.frame`.

```
> as.data.frame
function(x, row.names = NULL, optional = F, ...)
  UseMethod("as.data.frame")
```

The argument `"..."` allows the generic function to pass any method-specific arguments to the appropriate method.

If you've already built a function to construct data frames from a certain class of data, you can use it in defining your `as.data.frame` method. Your method just needs to account for all the formal arguments of `as.data.frame`. For example, suppose you have a class `loops` and a function `make.df.loops` for creating data frames from objects of that class. You can define a method `as.data.frame.loops` as follows.

```
> as.data.frame.loops
function(x, row.names = NULL, optional = F, ...)
{
  x <- make.df.loops(x, ...)
  if(!is.null(row.names))
  {
    row.names <- as.character(row.names)
    if(length(row.names) != nrow(x))
      stop(paste("Provided", length(row.names),
                "names for", nrow(x), "rows"))
    attr(x, "row.names") <- row.names
  }
  x
}
```

This method takes account of user-supplied row names, but ignores the argument `optional`, a flag that is TRUE when the method is not expected to generate non-trivial row names or variable names for a calling function.

## DATA FRAME ATTRIBUTES

Data frames, like all data objects, have the implicit attributes "length" and "mode". Because data frames are represented internally as lists, they have mode "list" and length equal to their number of variables, which is the number of components of their list representation.

Additional attributes of a data frame can be examined by calling the `attributes` function:

```
> attributes(auto)

$names:
[1] "Pri ce"  "Country" "Rel i ab" "Mi l eage" "Type"

$row.names:
[1] "Acura Integra4"      "Audi 1005"      "BMW325i 6"
[4] "ChevLumi na4"      "FordFesti va4"  "Mazda929V6"
[7] "MazdaMX-5Mi ata"    "Ni ssan300ZXV6" "Ol dsCal ai s4"
[10] "ToyotaCressi da6"

$class:
[1] "data.frame"
```

The variable names are stored in the `names` attribute and the row names are stored in the `rownames` attribute. There is also a `class` attribute with value `data.frame`. All data frames have `class` attribute `data.frame`.

Data frames preserve most attributes of special types of vectors, and these attributes may be accessed after the original objects have been combined into data frames. For example, categorical data have `class` and `levels` attributes preserved in data frames. You can access the defining attributes of a particular variable by specifying the variable in the data frame and passing it to the `attributes` function. Many of the variables in the `cu.summary` data frame are categorical—for example, the country of manufacture.

```
> attributes(cu.summary[, "Country"])

$levels:
[1] "Brazi l"  "Engl and"  "France"  "Germany"
[5] "Japan"    "Japan/USA" "Korea"   "Mexi co"
[9] "Sweden"   "USA"

$class:
[1] "factor"
```

The `levels` attribute is as you would expect for a categorical variable. Additionally, there is a `class` attribute with a value of `factor`. Objects of class `factor` are discussed in the section [Factors and Ordered Factors](#) (page 62). One attribute that is *not* preserved is the `names` attribute; the names for each variable are taken to be the row names of the data frame.

The attributes of a data frame are summarized in the table below. For attributes associated with a particular variable in a data frame, see the attribute section for the corresponding object type.

**Table 3.2:** *Attributes of Data Frames.*

Attribute	Description
<code>"length"</code>	The number of variables in the data frame.
<code>"mode"</code>	All data frames are of mode <code>"list"</code>
<code>"names"</code>	The names of the variables (columns) in the data frame.
<code>"row.names"</code>	The names of the rows in the data frame.
<code>"class"</code>	All data frames are of class <code>"data.frame"</code> .



# IMPORT AND EXPORT

# 4

---

<b>Importing Data Files</b>	<b>94</b>
Data Import Filters	94
Importing a Data File	95
Arguments to importData	96
Setting the Import Filter	98
Notes on Importing Files	100
<b>Other Data Import Functions</b>	<b>103</b>
Reading Vector and Matrix Data with scan	103
Reading Data Frames	105
<b>Exporting Data Sets</b>	<b>107</b>
Exporting Data to S-PLUS	108
Other Export Functions	108
<b>Exporting Graphs</b>	<b>111</b>
<b>Creating HTML Output</b>	<b>112</b>
Tables	112
Text	113
Graphs	113

## IMPORTING DATA FILES

One easy method of getting data into S-PLUS for plotting and analysis is to import the data file. The principal tool for importing data is the `import.data` function.

### Data Import Filters

Using `import.data`, you can select from the following file types to import into S-PLUS:

Format	Type	Default Extensions	Notes
ASCII	"ASCII"	<b>.txt, .csv</b>	
Formatted ASCII	"FASCII"	<b>.fix</b>	
Microsoft Excel	"EXCEL"	<b>.xls</b>	
Quattro Pro	"QUATTRO"	<b>.wq*, .wb*</b>	
Lotus	"LOTUS"	<b>.wk*, .wrk</b>	
dBase	"DBASE"	<b>.dbf</b>	II, II+, III, IV files
FoxPro			use same import filter as dBase files above
Systat	"SYSTAT"	<b>.sys</b>	double or single precision .sys files
SPSS	"SPSS"	<b>.sav</b>	
SPSS Export	"SPSSP"	<b>.por</b>	
SAS files	"SAS"	<b>.sd2</b>	Files from Windows
SAS Transport	"SAS_TPT"	<b>.tpt, .xpt</b>	version 6.x. Some special export options may need to be specified in your SAS program. We suggest using the SAS Xport engine (not PROC CPORT) to read and write these files.

Format	Type	Default Extensions	Notes
Matlab	"MATLAB"	<b>.mat</b>	must contain a single matrix in file
STATA	"STATA"	<b>.dta</b>	Versions 2.0 and higher
Gauss	"GAUSS" or "GAUSS96"	<b>.dat</b>	automatically reads the related DHT file.

Importing a Data File

In most cases, all you need to do to import a data file is to call `import.data` with the name of the file as a character string argument, and the name under which to save the new data frame. As long as the specified file has one of the default extensions shown in the above table, you need not specify a type, nor in most cases, any other information. For example, suppose you have a SAS data set **rain.sd2** in your startup directory. You can read this into S-PLUS using `importData` as follows:

```
import.data("rain.sd2", DataFrame="sas.rain.data")
```

Note

If a file extension is inappropriate, an error may appear indicating an unrecognized format or the data file may be converted incorrectly.

If you have trouble reading the data, most likely you just need to supply additional arguments to `import.data` to specify extra information required by the data importer to read the data correctly.

# Arguments to importData

The import.data function has the arguments shown in Table 4.1.

Table 4.1:Arguments to import.data.

Argument	Required	Description
FileName	Required	A character string giving the name of the file and directory path.
DataFrame	Required	A character string giving the name of the data frame to be created.
FileType	Optional	See the Type column in the previous table.
Col Names	Optional	A character vector of column names for the data columns to import, (separated by any of the delimiters specified in the Delimiters field). Specify one column name for each imported column (for example, Apples, Oranges, Pears). You can use an asterisk (*) to denote a missing name (for example, Apples, *, Pears).
NameRow	Optional	An integer denoting which column is to be used as the row names for the resulting data frame. If specified, the column of row names is dropped from the resulting data frame.
Format	Optional	A single character string specifying the format for formatted ASCII text files (type "FASCII"). See notes on Importing ASCII Files.
Filter	Optional	See the section Setting the Import Filter.
TargetStartCol	Optional	Starting column in source (from 1 to n). For example, if you specify 5, S-PLUS reads the columns beginning with column 5 and places them in the new data frame beginning at the Target Start Column. Spreadsheet-style letters (for example, A, AB) can be used to specify the start and end columns to import.
EndCol	Optional	End column in source. The default (-1) means to read to the last column.
StartRow	Optional	Starting row from range in source. (Spreadsheets only.) For example, if you specify row 10, S-PLUS reads the rows beginning with row 10 and places them in the new data frame beginning at row 1.



Table 4.1: Arguments to `import.data`.

Argument	Required	Description
<code>EndRow</code>	Optional	End row from range in source. (Spreadsheets only). The default (-1) is to read to the last row in the spreadsheet.
<code>Col Names</code>	Optional	The row containing the column names. If the file you are importing contains names for the columns of data, S-PLUS can use these names as column names. In the <code>colNameRow</code> argument, specify which row number (in the file being imported) contains the column names. If you do not specify a named row, S-PLUS attempts to locate column names in the first row of the file. Specify Row 0 to have S-PLUS not search for a name row. In a delimited ASCII file, the name row must come before the first data rows to be read in (the start row).
<code>Delimiters</code>	Optional	Character string giving range of characters that might be used as delimiters. Commas, spaces, and tabs (denoted by <code>\t</code> ) are the default delimiters. If you replace the default delimiters with another delimiter, any column names or format strings you specify must be separated by the specified delimiter (see the section <b>Notes on Importing ASCII (Delimited ASCII) Files</b> (page 100)). Carriage returns or line feeds are not allowed because they must terminate each row.
<code>SeparateDelimiters</code>	Optional	When set to the default of <code>False</code> , it indicates that multiple concurrent delimiters are treated as one single delimiter. This is useful if a delimiter is also used as a filler (typically a space). When set to <code>True</code> every delimiter that is not preceded by a value indicates that a missing value (NA) should be inserted in the resulting cell in the data frame being created.
<code>Preview</code>	Optional	Imports all columns as character columns.
<code>OdbcConnection</code>	Optional	Required if <code>FileType == "ODBC"</code> . Encrypted character string containing ODBC connection string.
<code>OdbcSqlQuery</code>	Optional	Only meaningful if <code>FileType == "ODBC"</code> . It contains an optional SQL query. If no query is specified the first table of the data source is used.

Setting the Import Filter

The `filter` argument to `import.data` allows you to subset the data you import. By specifying a query, or *filter*, you gain additional functionality, such as taking a random sampling of the data. Use the following examples and explanation of the filter syntax to create your statement. A blank filter is the default and results in all data being imported.

Note

The `filter` argument is ignored if the `type` argument (or, equivalently, file extension specified in the `file` argument) is set to `"ASCII"` or `"FASCII"`.

Case Selection

You select cases by using a case-selection statement in the `filter` argument. The case-selection or `where` statement has the following form:

*"variable expression relational operator condition "*

Warning

The syntax used in the `filter` argument to `importData` and `exportData` is not standard S-PLUS syntax; and the expressions described are not standard S-PLUS expressions. Do not use the syntax described in this section *for any purpose* other than passing a `filter` argument to `import.data` or `export.data`.

Variable Expressions

You can specify a single variable or an expression involving several variables. All of the usual arithmetic operators ( `+` `-` `/` `*` `()` ) are available for use in variable expressions.

Relational Operators

The following relational operators are available:

Operator	
<code>=</code>	equals
<code>!=</code>	not equal
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal
<code>&gt;=</code>	greater than or equal
<code>&amp;</code>	and

Operator	
	or
!	not

## Examples

Examples of selection conditions given by filter expressions are:

```
"sex = 1 & age < 50"
"(income + benefits) / famsize < 4500"
"income1 >=20000 or income2 >= 20000"
"income1 >=20000 & income2 >= 20000"
"dept = 'auto loan' "
```

Note that strings used in case-selection expressions must be enclosed in single quotes if they contain embedded blanks.

Wildcards \* or ? are available to select subgroups of string variables. For example:

```
"account = ?????22"
"id = 3*"
```

The first statement will select any accounts that have 2's as the 5th and 6th characters in the string, while the second statement will select strings of any length that begin with 3.

The comma operator is used to list different values of the same variable name that will be used as selection criteria. It allows you to bypass lengthy OR expressions when giving lists of conditional values, for example:

```
"state = CA, WA, OR, AZ, NV"
"caseid != 22*, 30??. 4?00"
```

## Missing Variables

You can test to see that any variable is missing by comparing it to the special internal variable, NA. For example:

```
"income != NA & age != NA"
```

## Notes on Importing Files

### Notes on Importing ASCII (Delimited ASCII) Files

When importing ASCII files you have the option of specifying column names and data types for imported columns. This can be useful if you want to name columns or if you wish to skip over one or more columns when importing.

#### Format String

Use the `format` argument to `import.data` to specify the data types of the imported columns. For each column you need to specify a % sign and then the data type. Dates may automatically be imported as numbers. After importing, you can change the column format type to a dates format. One of the delimiters specified in the Delimiters field must separate each specification in the string. Here is an example ASCII format string:

```
%S, %F, %*, %F
```

The "s" denotes a string data type, "f" denotes a float data type, and the asterisk (\*) denotes a "skipped" column.

If you do not specify the data type of each column, S-PLUS looks at the first row of data to be read and uses the contents of this row to determine the data type of each column. A row of data must always end with a new line.

Note that field width specifications are irrelevant for ASCII files and are ignored.

Multiple delimiter characters are not grouped and treated the same as a single delimiter. For example, if the comma is a delimiter, two commas are interpreted as a missing field.

Double quotes (") are treated specially. They are always treated as an "enclosure" marker, and must always come in pairs. Any data contained between double quotes are read as a single unit of character data. Thus, spaces and commas can be used as delimiters, and spaces and commas can still be used within a character field as long as that field is enclosed within double quotes. Double quotes cannot be used as standard delimiters.

If a variable is specified to be numeric, and if the value of any cell cannot be interpreted as a number, that cell is filled in with a missing value. Incomplete rows are also filled in with missing values.

## Notes on Importing FASCII (Formatted ASCII) Files

You can use FASCII import to specify how each character in your imported file should be treated. For example, you must use FASCII for fixed width columns not separated by delimiters, if the rows in your file are not separated by line feeds or if your file splits each row of data into two or more lines.

For FASCII import, you need to specify the file name and the file type. In addition, because FASCII files are assumed to be non-delimited (for example, there are no commas or spaces separating fields), you also need to specify each column's field width and data type in the Format String. This tells S-PLUS where to separate the columns. Each column must be listed along with its data type: character or numeric and its field width. If you want to name the columns, specify a list of names in the `colNames` argument. (Column names cannot be read from the FASCII data file).

When importing FASCII files you need to specify the following arguments to `importData`.

### ColNames

Enter a character vector of column names for the imported data columns (separated by spaces or commas). Specify one column name for each imported column (for example, Apple, Oranges, Pears). You can use an asterisk (\*) to denote a missing name (for example, Apples, \*, Pears).

### Format

Specify the data types and field widths of the imported columns. For each column you need to specify a % sign, then the field width, and then the data type. Commas or spaces must separate each specification in the string. The format string is necessary because formatted ASCII files do not have delimiters (such as commas or spaces) separating each column of data. Here is an example format string:

```
%10s, %12f, %5*, %10f
```

The numbers denote the column widths, "s" denotes a string data type, "f" denotes a float data type, and the asterisk (\*) denotes a "skip". You may need to skip characters when you want to avoid importing some characters in the file. For example, you may want to skip blank characters or even certain parts of the data.

If you wish to import only some of the rows, specify a starting and ending row.

If each row ends with a new line, S-PLUS will treat the newline character as a single character-wide variable that is to be skipped.

**Notes on  
Importing Excel  
Files**

If your Excel worksheet contains only numeric data in a rectangular block, starting in the first row and column of the worksheet, then all you need to specify is the file name and file type. If a row contains names, specify the number of that row at the Name Row prompt (it does not have to be the first row). You can select a rectangular subset of your worksheet by specifying starting and ending columns and rows. Excel-style column names (for example, A, AB) can be used to specify the starting and ending columns.

**Notes on  
Importing Lotus  
Files**

If your Lotus-type worksheet contains numeric data only in a rectangular block, starting in the first row and column of the worksheet, then all you need to specify is the file name and file type. If a row contains names, specify the number of that row in the `col NameRow` argument (it does not have to be the first row). You can select a rectangular subset of your worksheet by specifying starting and ending columns and rows. Lotus-style column names (for example, A, AB) can be used to specify the starting and ending columns.

The row specified as the starting row is always read first to find out the data types of the columns. Therefore, there cannot be any blank cells in this row. In other rows, blank cells are filled in with missing values.

**Notes on  
Importing dBase  
Files**

S-PLUS imports dBase and dBase-compatible files. The file name and file type are often the only things you need specify for dBase-type files. Column names and data types are obtained from the dBase file. However, you can select a rectangular subset of your data by specifying starting and ending columns and rows.

## OTHER DATA IMPORT FUNCTIONS

While `import.data` is the recommended method for reading data files into S-PLUS, there are several other functions that you can use to read ASCII data into S-PLUS. These functions are commonly used by other functions in S-PLUS, so it is a good idea to familiarize yourself with them. The two functions discussed in this section are `scan` and `read.table`.

### Reading Vector and Matrix Data with `scan`

The `scan` function, which can read from either standard input or from a file, is commonly used to read data from keyboard input. By default, `scan` expects numeric data separated by white space, although there are options that let you specify the type of data being read and the separator. When using `scan` to read data files, it is helpful to think of each line of the data file as a *record*, or *case*, with individual observations as *fields*. For example, the following expression creates a matrix named `x` from a data file specified by the user:

```
x <- matrix(scan("filename"), ncol = 10, byrow = T)
```

Here the data file is assumed to have 10 columns of numeric data; the matrix contains a number of observations for each of these ten variables. To read in a file of character data, use `scan` with the `what` argument:

```
x <- matrix(scan("filename", what = ""), ncol=10, byrow=T)
```

Any character vector can be used in place of `""`. For most efficient memory allocation, `what` should be the same size as the object to be read in. For example, to read in a character vector of length 1000, use

```
> scan(what=character(1000))
```

The `what` argument to `scan` can also be used to read in data files of mixed type, for example, a file containing both numeric and character data, as in the following sample file, `table.dat`:

```
Tom 93 37
Joe 47 42
Dave 18 43
```

In this case, you provide a list as the value for `what`, with each list component corresponding to a particular field:

```
> z <- scan("table.dat", what=list("", 0, 0))
> z
```

```
[[1]]:  
[1] "Tom" "Joe" "Dave"
```

```
[[2]]:  
[1] 93 47 18
```

```
[[3]]:  
[1] 37 42 43
```

S-PLUS creates a list with separate components for each field specified in the `what` list. You can turn this into a matrix, with the subject names as column names, as follows:

```
> matz <- rbind(z[[2]], z[[3]])  
> dimnames(matz) <- list(NULL, z[[1]])  
> matz
```

```
      Tom Joe Dave  
[1,]  93  47  18  
[2,]  37  42  43
```

You can scan files containing multiple line records by using the argument `multi.line=T`. For example, suppose you have a file `heart.all` containing information in the following form:

```
johns 1  
450 54.6  
marks 1 760 73.5  
.  
.  
.
```

You can read it in with `scan` as follows:

```
> scan('heart.all', what=list("", 0, 0, 0), multi.line=T)  
[[1]]:  
[1] "johns" "marks" "avery" "able" "simpson"  
.  
.  
.  
[[4]]:  
[1] 54.6 73.5 50.3 44.6 58.1 61.3 75.3 41.1 51.5 41.7 59.7  
[12] 40.8 67.4 53.3 62.2 65.5 47.5 51.2 74.9 59.0 40.5
```



If your data is in *fixed format*, with fixed-width fields, you can use `scan` to read it in using the `widths` argument. For example, suppose you have a data file `dfile` with the following contents:

```
01giraffe.9346H01-04
88donkey .1220M00-15
77ant          L04-04
20gerbil .1220L01-12
22swallow.2333L01-03
12lemming      L01-23
```

You identify the fields as numeric data of width 2, character data of width 7, numeric data of width 5, character data of width 1, numeric data of width 2, a hyphen or minus sign that you don't want to read into S-PLUS, and numeric data of width 2. You specify these types using the `what` argument to `scan`. To simplify the call to `scan`, you define the list of `what` arguments separately:

```
> dfile.what <- list(code=0, name="", x=0, s="", n1=0,
+   NULL, n2=0)
```

(NULL indicates suppress scanning of the specified field.) You specify the widths as the `widths` argument to `scan`. Again, it simplifies the call to `scan` to define the `widths` vector separately:

```
> dfile.widths <- c(2, 7, 5, 1, 2, 1, 2)
```

You can now read the data in `dfile` into S-PLUS calling `scan` as follows:

```
> dfile <- scan("dfile", what=dfile.what,
+   widths=dfile.widths)
```

If some of your fixed-format character fields contain leading or trailing white space, you can use the `strip.white` argument to strip it away. (The `scan` function always strips white space from numeric fields.) See the `scan` help file for more details.

## Reading Data Frames

Data frames in S-PLUS were designed to resemble tables. They must have a rectangular arrangement of values and typically have row and column labels. Data frames arise frequently in designed experiments and other situations. If you have a text file with data arranged in the form of a table, you can read it into S-PLUS using the `read.table` function. For example, consider the data file `auto.dat`:

---

Model	Price	Country	Reliab	Mileage	Type
Acura Integra4	11950	Japan	5	NA	Small
Audi 1005	26900	Germany	NA	NA	Medium
BMW325i 6	24650	Germany	94	NA	Compact
ChevLumi na4	12140	USA	NA	NA	Medium
FordFestiva4	6319	Korea	4	37	Small
Mazda929V6	23300	Japan	5	21	Medium
MazdaMX-5Miata	13800	Japan	NA	NA	Sporty
Nissan300ZXV6	27900	Japan	NA	NA	Sporty
OldsCutlass4	9995	USA	2	23	Compact
ToyotaCressida6	21498	Japan	3	23	Medium

All fields are separated by spaces and the first line is a header line. To create a data frame from this data file, use `read.table` as follows:

```
> auto <- read.table('auto.dat', header=T)
> auto
```

	Price	Country	Reliab	Mileage	Type
Acura Integra4	11950	Japan	5	NA	Small
Audi 1005	26900	Germany	NA	NA	Medium
BMW325i 6	24650	Germany	94	NA	Compact
ChevLumi na4	12140	USA	NA	NA	Medium
FordFestiva4	6319	Korea	4	37	Small
Mazda929V6	23300	Japan	5	21	Medium
MazdaMX-5Miata	13800	Japan	NA	NA	Sporty
Nissan300ZXV6	27900	Japan	NA	NA	Sporty
OldsCutlass4	9995	USA	2	23	Compact
ToyotaCressida6	21498	Japan	3	23	Medium

As with `scan`, you can use `read.table` within functions to hide the mechanics of S-PLUS from the users of your functions.

## EXPORTING DATA SETS

You use the `export.data` function to export S-PLUS data objects to formats for applications other than S-PLUS. To export data for use by S-PLUS, use the `data.dump` function. When you are exporting to most file types with `export.data`, you typically need to specify only the data set, file name, and (depending on the file name you specified) the file type, and the data will be exported into a new data file using default settings. You can specify your own settings using additional arguments to `export.data`. All formats that can be imported from can be exported to.

The arguments to `export.data` are shown in Table 4.2.

*Table 4.2: Arguments to `exportData`.*

Argument	Required	Description
<code>DataSet</code>	Required	Data frame to be exported.
<code>FileName</code>	Required	A character string containing the name of the file to be created/updated.
<code>FileType</code>	Optional	One of: "ASCII", "DBASE", "EXCEL", "FASCII", "GAUSS", "GAUSS96", "HTML", "LOTUS", "MATLAB", "ODBC", "QUATTRO", "SAS", "SAS1", "SAS4", "SAS_TPT", "SPSS", "SPSSP", "STATA", "SYSTAT".
<code>Columns</code>	Optional	Character string specifying a subset of the columns to be exported.
<code>Rows</code>	Optional	Character string specifying a subset of the rows to be exported.
<code>Delimiter</code>	Optional	Character to be used as delimiter. (Used only with type "ASCII".)
<code>ColumnNames</code>	Optional	Logical flag specifying whether to output column names: TRUE or FALSE. Default is TRUE.
<code>Quotes</code>	Optional	Logical flag specifying whether to put quotes around character strings: TRUE or FALSE. Default is TRUE.
<code>LineLength</code>	Optional	Integer specifying maximum length, in characters, of one line.

Table 4.2: Arguments to `exportData`.

Argument	Required	Description
<code>OdbcConnecti on</code>	Optional	Required and meaningful only if <code>FileType</code> = "ODBC". This is an encrypted character string containing the ODBC connection string.
<code>OdbcTabl e</code>	Optional	Required name of table to be created if <code>FileType</code> = "ODBC".

## Exporting Data to S-PLUS

When you want to export data to share with another S-PLUS user, use the `data.dump` function:

```
> data.dump("matz")
```

By default, the data object `matz` is exported to the file **dumpdata** in your S-PLUS startup directory. You can specify a different output file with the `connecti on` argument to `data.dump`:

```
> data.dump("matz", connecti on="matz.dmp")
```

(The `connecti on` argument needn't specify a file; it can specify any valid S-PLUS connection object. See *Programming with Data* for more details on connections.)

If the data object you want to share is not on the working data, you must specify the object's location in the search path with the `where` argument:

```
> data.dump("hal i but", where="data")
```

## Other Export Functions

The inverse operation to the `scan` function is provided by the `cat` and `wri te` functions. Similarly, the inverse operation to `read.table` is provided by `wri te.table`. The result of either `wri te` or `cat` is just an ASCII file with data in it. There is no S-PLUS structure written in.

Of the two commands, `wri te` has an argument for specifying the number of columns and thus is more useful for retaining the format of a matrix.

By default, `wri te` writes matrices column by column, five values per line. If you want the matrix represented in the ASCII file in the same form it is represented in S-PLUS, transform the matrix first with the `t` function and specify the number of columns in your original matrix:

```
> mat

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

> write(t(mat), "mat", ncol = 4)
```

You can view the resulting file with a text editor or pager; it contains the following three lines:

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

The `cat` function is a general-purpose writing tool in S-PLUS, used for writing to the screen as well as writing to files. It can be useful in creating free-format data files for use with other software, particularly when used with the `format` function:

```
> cat(format(runif(100)), fill=T)

0.261401257 0.556708986 0.184055283 0.760029093 . . . .
```

The argument `fill=T` limits line length in the output file to the width specified in your options object. To use `cat` to write to a file, simply specify a file name with the `file` argument:

```
> x <- 1:1000
> cat(x, file="mydata", fill=T)
```

The files written by `cat` and `write` do not contain S-PLUS structure information; to read them back into S-PLUS you must reconstruct this information.

The `write.table` function can be used to export a data frame into an ASCII text file:

```
> write.table(fuel.frame, "fuel.txt")
> !vi fuel.txt
row.names, Weight, Displ., Mileage, Fuel, Type
Eagle Summit 4, 2560, 97, 33, 3.030303, Small
Ford Escort 4, 2345, 114, 33, 3.030303, Small
Ford Festiva 4, 1845, 81, 37, 2.702703, Small
Honda Civic 4, 2260, 91, 32, 3.125000, Small
Mazda Protege 4, 2440, 113, 32, 3.125000, Small
Mercury Tracer 4, 2285, 97, 26, 3.846154, Small
Nissan Sentra 4, 2275, 97, 33, 3.030303, Small
Pontiac LeMans 4, 2350, 98, 28, 3.571429, Small
. . .
```

# EXPORTING GRAPHS

The `export.graph` command is used to export a graph named `Name` to the file `FileName`, using file format specified by `ExportType`.

*Table 4.3: Arguments to `export.graph`.*

Argument	Required	Description
<code>Name</code>	Required	A character string specifying the object path name for a graphsheet.
<code>FileName</code>	Required	A character string giving the name of the file including the file's title plus the entire directory path.
<code>ExportType</code>	Optional	<p>"BMP", "CGM", "DXF", "EPS TIFF" (EPS w/TIFF), "EPS", "EPS PRINT" (EPS using PostScript printer driver), "GIF", "HGL", "IMG", "JPG", "MET", "PCL", "X", "PICT", "SDW" (AmiPro), "TIFF", "TGA", "WMF", "WPB" (WordPerfect Bitmap), "WPV" (WordPerfect Vector).</p> <p>If the optional argument <code>ExportType</code> is not specified, the file type can be inferred from the file extension used in the <code>FileName</code> argument.</p>

## Example

```
export.graph(
  Name = "GS1",
  Graph = "",
  Rotate = "",
  ExportType = "GIF",
  FileName = "F:\\GUI\\test2.GIF");
```

This will export the graph called "GS1" in GIF format to file "F:\\GUI\\test2.GIF".

## CREATING HTML OUTPUT

S-PLUS provides a variety of tools for generating HTML output. This chapter discusses how to generate HTML tables, save preformatted text output, and save graphs with HTML references.

### Tables

The `html.table` function may be used to generate a vector of character strings representing a vector, matrix, or data frame as an HTML table. The vector will contain one string for each line of HTML. This may be written to a file by specifying the `file` argument, or may be manipulated and later written to a file using the `write` function.

For example, we can create a file `catalyst.htm` containing the `catalyst` data frame using:

```
> html.table(catalyst, file="catalyst.htm")
```

In addition to accepting a vector, matrix, or data frame, the `html.table` function will accept a simple list with such structures as components of the list. It will then produce a sequence of tables with the list component names encoded as table captions. For example:

```
> my.results<-list("Regression Coefficients" =
+   coef(lm(Mileage~Weight,
+   fuel.frame))), "Correlations"=cor(fuel.frame[, 1:3]))
> html.table(my.results, file="my.htm")
```

The `html.table` function accepts any of the arguments to `format`, allowing specification of formatting details such as the number of digits displayed. In addition, `append` controls whether output is appended to the specified `file` or the file is overwritten. The `append` argument is also available in the `write` function, which is useful for interspersing `html.table` output and descriptive text:

```
> write("<H3> S-PLUS Code for the above </H3>
Continue string: <P> Put code here </P>",
+   file="my.htm", append=T)
```

Additional arguments to `html.table` are described in the function's help file.



Note that `html.table` is designed to work with the previously mentioned data structures. For other structures such as functions, calls, and objects with specific `print` methods, the results of `html.table` may not be satisfactory. Instead, the object may be printed as preformatted text and embedded in the HTML page.

## Text

The `sink` function may be used to direct S-PLUS text output to an HTML file. The preformatted output may be interspersed with the HTML markup tag `<PRE>` to denote that it is preformatted output. Additional textual description and HTML markup tags may be interspersed with the S-PLUS output using `cat`.

```
> sink("my.htm")
> cat("<H3> Linear Model Results </H3> \n")
> cat("<PRE>")
> summary(lm(Mileage~Weight, fuel.frame))
> cat("</PRE>")
> sink()
```

The `paste` and `deparse` functions are useful for constructing strings to display with `cat`. See their help files for details.

## Graphs

The two steps involved in embedding an S-PLUS graph in an HTML page are exporting the graph in a format such as GIF or JPG which is viewable with a web browser, and placing an `<IMG>` tag in the HTML file describing the location of the image.

Use the `export.graph` command to export a graph to a specific file:

```
> graphsheet(Name="MyGraph")
> xyplot(Mileage~Weight, fuel.frame)
> export.graph(Name="MyGraph", FileName="my.gif")
```

Use `sink` and `cat` to place an `<IMG>` tag in an HTML file. Note the use of `\` to include quotation marks in the text:

```
> sink("my.htm", append=T)
> cat("<IMG SRC=\"my.gif\">")
> sink()
```



# WRITING FUNCTIONS IN S-PLUS

# 5

---

<b>Programming in S-PLUS</b>	<b>117</b>
<b>The Structure of Functions</b>	<b>118</b>
Functions and Names	118
Arguments	120
The Function Body	120
Return Values and Side Effects	120
Complex Arithmetic	121
Elementary Functions	123
Summary Functions	125
Comparison and Logical Operators	126
Assignments	127
Testing and Coercing Data	128
<b>Operating on Subsets of Data</b>	<b>131</b>
Subscripting on Matrices and Arrays	133
Subscripting on Lists	135
Subsetting from Data Frames	137
<b>Organizing Computations</b>	<b>138</b>
Programming Style	138
Control Flow	140
The if Statement	141
Handling Multiple Cases	143
The ifelse Function	144
The repeat Statement	146
The return, break,	
and next Statements	147
The while Statement	148
The for Statement	149
<b>Specifying Argument Lists</b>	<b>150</b>
Formal and Actual Names	150
Specifying Default Arguments	150
Missing-Argument Handling	151

Lazy Evaluation	151
Variable Numbers of Arguments	152
Required and Optional Arguments	153
<b>Error Handling</b>	<b>154</b>
<b>Data Input</b>	<b>156</b>
Reading Vector and Matrix Data With scan	156
Reading Data Frames	159
<b>Data Output</b>	<b>161</b>
Formatting Output	161
Constructing Return Values	163
Side Effects	165
Writing to Files	165
Creating Temporary Files	167
<b>Wrap-Up Actions</b>	<b>168</b>
<b>Extraction and Replacement Functions</b>	<b>170</b>
<b>Operators</b>	<b>174</b>

---

## PROGRAMMING IN S-PLUS

Programming in S-PLUS consists largely of writing functions. Functions do most of the work in S-PLUS. The simplest functions arise naturally as shorthand for frequently-used combinations of S-PLUS expressions. For example, S-PLUS has no built-in function for calculating the standard deviation of a data set. It does, however, have a function for calculating the variance and another for calculating square roots. The standard deviation is simply the square root of the variance, so a standard deviation function can be created as follows:

```
stdev <- function(x) { sqrt(var(x)) }
```

You can build more complicated functions either by adding new features incrementally onto simpler functions, or by designing whole programs from scratch. As your functions grow more complex, proper use of programming features such as conditionals and error handling becomes more important.

This chapter describes the basic techniques of writing functions in S-PLUS. First it outlines the structure underlying all S-PLUS functions, then it describes some of the most useful functions for manipulating data within functions. From these few simple tools, you can build many useful functions. A section on organizing computations gives many useful tips on designing functions that take advantage of the strengths of S-PLUS. Later sections introduce techniques for argument handling, error handling, input and output, and wrap-up actions. With just the techniques in this chapter, you can create functions for virtually any computational task.

## THE STRUCTURE OF FUNCTIONS

All S-PLUS functions have the same structure: they consist of the reserved word `function`, an *argument list* (which may be empty), and a *body*. The `Edit` function, when called with a name that does not correspond to an existing S-PLUS object, creates a *function template* with the above structure. Thus, to create a new function `newfunc`, call `Edit` as follows:

```
> Edit(newfunc)
```

Edit the template as desired. Select `Script/Run` from the menu, or use the `Run` button on the `Script` toolbar, to source in the modified function. Use `File/Save` if you wish to save the function in an external script file. To edit the function using an alternate editor you may use `fix` (refer to the section `Editing Objects` (page 35)).

### Functions and Names

Most functions, when defined, are associated with *names*. The form of the name conveys some important information about the nature of the function. Most functions have simple, relatively short, alphanumeric names, such as `plot`, `na.omit`, or `add1`. These functions are always used in the form `function(arglist)`.

*Operators* are functions for performing mathematical or logical operations on one or two arguments, and are often most conveniently used in *infix* form, that is, between the two arguments. Familiar examples are `+`, `-`, and `*`. The names of such functions consist of the symbol used to represent them, within quotes, for example, `"+"`. You *can* use these names to call operators as functions in the ordinary way, as for example in `"+"(2, 3)`, which returns the number 5. A complete list of built-in operators is provided in Table 5.1.

Operators listed higher in the table have higher precedence than those listed below. Operators on the same line have equal precedence, and evaluation proceeds from left to right within an expression. Thus, for example,

```
> 7 + 5 - 8^2 / 19 * 2
```

```
[1] 5.263158
```

Here the exponentiation is done first,  $8^2 = 64$ . Division has the same precedence as multiplication, but appears to the left of the multiplication, so it is performed first:  $64/19 = 3.368421$ . Next comes the multiplication:  $3.368421 * 2 = 6.736842$ . Finally, S-PLUS performs the addition and subtraction:  $7 + 5 - 6.736842 = 5.263158$ .

You can override the normal precedence of operators by *grouping* with parentheses or braces:

```
> (7 + 5 - 8^2) / (19 * 2)
```

```
[1] -1.368421
```

*Table 5.1: Precedence of operators.*

Operator	Use
\$	component selection
[ [ [	subscripts, elements
^	exponentiation
-	unary minus
:	sequence operator
%% %/% %*%	modulus, integer divide, matrix multiply
* /	multiply, divide
+ -	add, subtract
<> <= >= == !=	comparison
!	not
&   &&	and, or
~	formulas
<<- -> <- _	assignments

#### Note

When using the ^ operator, if the base is a negative number, the exponent must be an integer.

The integer divide operator %/% produces an integral quotient, that is, the  $q$  in the Euclidean algorithm's  $a = bq + r$  for  $a \div b$ ; the modulus operator %% produces the remainder  $r$ .

In addition to providing these predefined operators, S-PLUS allows you to define your own infix operators; see the online help for details.

A third type of function, the *replacement* or *left-side* function, has the appearance of an ordinary function on the left-hand side of an assignment, e.g., `dim(x) <- c(3, 4)`. S-PLUS interprets this expression as the ordinary assignment `x <- "dim<-"(x, c(3, 4))`. The function `"dim<-"` is the replacement function corresponding to the ordinary function `dim`.

You can define replacement functions for any *extraction* function, that is, any function that returns some specific portion or attribute of a data object. Common extraction functions are the subscript operator `[ ]` and the `dim` and `names` functions. See the online help for complete details.

## Arguments

Arguments to a function specify the data the function is to operate on, and pass processing parameters to the function. Not all functions take arguments—for example, the `date` function. Functions without arguments are, of necessity, rigid and single-purpose. Their behavior can be modified only by editing the function. Arguments let you build multi-purpose functions with behavior that can be easily modified whenever the function is called.

For a complete discussion of allowable argument lists, see the section *Specifying Argument Lists* (page 150)

## The Function Body

The *body* of a function is the part of the function that actually does the work. The body consists of one or more S-PLUS statements or expressions. If there is more than one expression, the entire body must be enclosed in braces. Whether braces should always be included is a matter of programming style. MathSoft recommends including them, because it makes maintenance less accident-prone. By adding braces when you define the function, you ensure that they won't be forgotten when you add functionality to your "one-liner."

Most of this chapter, and in fact most of this book, is devoted to showing you how to write the most effective function body you can. This involves organizing the computations effectively and naturally, expressing them with suitable S-PLUS expressions, and returning the appropriate information.

## Return Values and Side Effects

Functions are defined to *do* something, and if everything goes as it's supposed to, a function will do something every time it is called. Most functions do one thing: return a *value*. A value can be any valid S-PLUS expression,



although usually it is a transformed version of the input data. Values returned from functions are not, in general, automatically saved. Most calls to functions, therefore, also involve an assignment:

```
> y <- f(x)
```

This ensures that the value is preserved for further analysis.

Sometimes, however, you want a function to do something besides return an S-PLUS expression. You may want to print something, or plot a graphics figure, or change some S-PLUS session options. Because the main goal of functions is to return values, these other actions are collectively called *side effects*. Sometimes the combination of a function's side effects and its value can be used to good advantage. For example, the `options` function has the side effect of changing the current S-PLUS session options. It returns a value, however, that consists of the set of options in effect before the current call. Thus, you can use options within a function both to *change* the options in effect and *save* the old options for restoration when the function exits:

```
options.old <- options(width=55)
on.exit(options(options.old))
```

By assigning the value of `options` to `options.old`, we save the old settings. The side effect, changing `options` to use a width of 55 characters, takes place whether or not we assign the value.

## Complex Arithmetic

Arithmetic operations on complex numbers work, for the most part, as you expect. You represent complex numbers in S-PLUS as a sum of the form  $a + bi$ , where  $a$  and  $b$  are real numbers. *You must not leave any space between the real number  $b$  and the symbol  $i$ .* If you do, you get the following syntax error:

Syntax error: name ("i") used illegally at this point:

Because the `+` operator has lower precedence than the `*`, `/`, and `^` operators, you must use parentheses to group complex arguments to these operators:

```
> (2-3i) * (4+6i)
[1] 26+0i
> (2+3i)^(3+2i)
[1] 4.714144-4.569828i
```

By default, S-PLUS performs real arithmetic if all arguments are real, and complex arithmetic if any arguments are complex. In particular, if you pass a real argument to a function such as `sqrt`, S-PLUS attempts to return a real value. If it can't, it returns `NA` and issues a domain error message. For example, here is the result when we pass the real number `-1` to `sqrt`:

```
> sqrt(-1)
```

```
[1] NA
```

To force S-PLUS to consider potential complex solutions, you must coerce the arguments to mode `"complex"`, typically by using the function `as.complex`:

```
> sqrt(as.complex(-1))
```

```
[1] 6.123032e-17+1i
```

In addition to the ordinary operators S-PLUS provides five special operators for manipulating complex numbers. `Re` and `Im` functions are used to extract the real part and imaginary parts, respectively, from a complex number:

```
> Re(as.complex(-3)^(1/3))
```

```
[1] 0.7211248
```

```
> Im(as.complex(-3)^(1/3))
```

```
[1] 1.249025
```

`Mod` and `Arg` functions return the *modulus* and *argument* for the polar representation of the complex number:

```
> Mod(2 + 2i)
```

```
[1] 2.828427
```

```
> Arg(2 + 2i)
```

```
[1] 0.7853982
```

`Conj` returns the conjugate of the complex number:

```
> Conj(as.complex(-3)^(1/3))
```

```
[1] 0.7211248-1.249025i
```

## Elementary Functions

In addition to the infix operators, S-PLUS includes a variety of elementary mathematical functions that also act in a vectorized way on numeric data sets. These include the familiar trigonometric and exponential functions, as well as several functions for computing elementary numerical results. The functions listed in Table 5.2 are the vectorized math functions implemented internally as part of the S-PLUS language. S-PLUS has many other built-in mathematical functions, some written wholly in the S-PLUS language, some written to take advantage of existing algorithms in Fortran or C. See the sections on Mathematical Computing in S-PLUS in the *Guide to Statistics, Volume 2* for more information.

*Table 5.2: Elementary Functions.*

Name	Operation
<code>sqrt</code>	square root
<code>abs</code>	absolute value
<code>sin</code> , <code>cos</code> , <code>tan</code>	trigonometric functions
<code>asin</code> , <code>acos</code> , <code>atan</code>	inverse trigonometric functions
<code>sinh</code> , <code>cosh</code> , <code>tanh</code>	hyperbolic trigonometric functions
<code>asinh</code> , <code>acosh</code> , <code>atanh</code>	inverse hyperbolic trigonometric functions
<code>exp</code> , <code>log</code>	exponential and natural logarithms
<code>log10</code>	common logarithm
<code>gamma</code> , <code>lgamma</code>	gamma function and its natural logarithm
<code>ceiling</code>	closest integer not less than element
<code>floor</code>	closest integer not greater than element
<code>trunc</code>	closest integer between element and zero
<code>round</code>	closest integer to element
<code>signif</code>	round to specified number of significant digits
<code>cumsum</code> , <code>cumprod</code>	cumulative sum and product

Each function acts *element-by-element* on its argument:

```
> J

      [,1] [,2] [,3] [,4]
[1,]   12   15    6   10
[2,]    2    9    2    7
[3,]   19   14   11   19
> sqrt(J)

      [,1]      [,2]      [,3]      [,4]
[1,]  3.464102  3.872983  2.449490  3.162278
[2,]  1.414214  3.000000  1.414214  2.645751
[3,]  4.358899  3.741657  3.316625  4.358899

> tan(J)

      [,1]      [,2]      [,3]      [,4]
[1,] -0.6358599 -0.8559934 -0.2910062  0.6483608
[2,] -2.1850399 -0.4523157 -2.1850399  0.8714480
[3,]  0.1515895  7.2446066 -225.9508465  0.1515895
```

The `trunc` function acts like `floor` for elements greater than 0 and like `ceiling` for elements less than 0:

```
> y

 [1] -2.6  1.5  9.7 -1.0 25.7 -4.6 -7.5 -2.7 -0.6
[10] -0.3  2.8  2.8

> trunc(y)

 [1] -2  1  9 -1 25 -4 -7 -2  0  0  2  2

> ceiling(y)

 [1] -2  2 10 -1 26 -4 -7 -2  0  0  3  3

> floor(y)

 [1] -3  1  9 -1 25 -5 -8 -3 -1 -1  2  2
```

The `round` function takes an optional second argument that allows you to specify how many digits to include after the decimal point:

```
> round(sqrt(J), digits=3)

      [, 1]  [, 2]  [, 3]  [, 4]
[1, ] 3.464 3.873 2.449 3.162
[2, ] 1.414 3.000 1.414 2.646
[3, ] 4.359 3.742 3.317 4.359
```

## Summary Functions

The mathematical operators and functions introduced so far have acted *element-by-element*, generally returning a value of the same length and mode as the input vector or matrix. S-PLUS also includes a number of functions for *summarizing* data. These functions take an input vector or matrix and return a single value. For example, the `sum` function and `prod` functions return the sum and product, respectively, of their arguments. Other useful summary functions include the following:

Table 5.3: Summary Functions.

Name	Operation
<code>min</code> , <code>max</code>	Return, respectively, the smallest and largest values in their arguments.
<code>range</code>	Returns a vector of length two containing the minimum and maximum of all the elements in all its arguments.
<code>mean</code> , <code>median</code>	Return, respectively, the arithmetic mean and median of their arguments. An optional argument to <code>mean</code> , <code>trim</code> , allows you to discard a specified fraction of the largest and smallest values.
<code>quantile</code>	Returns user-requested sample quantiles for a given data set. For example, <pre>&gt; quantile(corn.rain, c(.25, .75)) 25% 75% 9.425 12.075</pre>
<code>var</code>	Returns the variance of a vector, the variance-covariance of a data matrix, or covariances between matrices or vectors.
<code>cor</code>	Returns correlation matrix of a data matrix or correlations between matrices or vectors.

# Comparison and Logical Operators

Table 5.4 lists the S-PLUS operators for comparison and logic. Comparisons and logical operations are frequently convenient for extracting subsets of data, and conditionals using logical comparisons play an important role in flow of control in functions.

S-PLUS has two types of logical operators for And and Or operations. Table 5.4 refers to the two types as “Vectorized” and “Control.” The vectorized operators, as their name implies, evaluate the And or Or expressions element by element, returning a vector of T’s and F’s as appropriate. For example,

```
> x
[1] 1.9 3.0 4.1 2.6 3.6 2.3 2.8 3.2 6.6 7.6 7.4 1.0

> x < 2 | x > 4
[1] T F T F F F F T T T T

> x > 2 & x < 4
[1] F T F T T T T F F F F
```

The control operators are used in constructing conditional statements with `if` or `else`. The expressions used in constructing such statements are expected to have only a single logical value, rather than a vector of such values as for the vectorized operations. The control operators have the further property that they are only evaluated as far as necessary to return a correct value. Thus, for example, in the expression

```
any(x > 1) && all(y < 0)
```

S-PLUS initially evaluates only the first condition, `any(x > 1)`, and after determining that, for some value in `x`, `x>1`, proceeds to evaluate the second half of the expression. Similarly, in the expression

```
all(x >= 1) || 2 > 7
```

S-PLUS stops evaluation with `all(x >= 1)`; because this condition is TRUE, so is the entire expression.

Table 5.4: Logical and Comparison Operators.

Operator	Explanation	Operator	Explanation
==	equal to	!=	not equal to
>	greater than	<	less than

Table 5.4: Logical and Comparison Operators.

Operator	Explanation	Operator	Explanation
>=	greater than or equal to	<=	less than or equal to
&	vectorized And		vectorized Or
&&	control And		control OR
!	not		

Comparisons involving the symbolic constants NA and NULL always return NA. To test whether a value is missing or null, use the testing functions `is.na` and `is.null`, respectively:

```
> is.na(c(3, NA, 4))
[1] F T F
> is.null(names(kyphosis))
[1] F
> is.null(names(letters))
[1] T
```

## Assignments

We have seen how data objects are created in S-PLUS by assigning *values* to *names*. We saw in the section Syntax of S-PLUS Expressions (page 26) that legal names consist of letters, numbers, and periods, and cannot begin with a number. The most common form of assignment in S-PLUS uses the *left assignment* operator, `<-`, which may also be written as a single underscore, `_`, to save typing. The standard syntax is: *name* `<-` *expr* or *name* `_` *expr*.

S-PLUS interprets the expression on the right hand side, returning a value. That value is then assigned to the *name*. Assignments from the S-PLUS prompt are performed in the current working directory. Assignments within functions are local assignments performed in the frame in which the function is evaluated. This means that you can freely assign variable names within functions, using any appropriate names, knowing that you will not overwrite any existing objects that might share that name. Frames are discussed in full in the section Frames and Argument Evaluation (page 793).

Equivalent to left assignment is *right assignment* (`expr -> name`), which is convenient when you type a complicated expression and then realize you've forgotten to assign a name to the returned value. (S-PLUS also protects you from your forgetfulness by storing the latest returned value in the object `.Last.value` in the working data.) For purposes of consistency, we recommend that you always use left assignment within functions. If you use right assignment in a function definition, then look at the function definition later; you see that S-PLUS has automatically reformatted the function to use left assignment. The `<<-` operator is like `<-`, except that it *always* writes to the working directory. Thus, you can make permanent assignments from within functions. Remember, however, that permanent assignment within a function produces a *side effect*, in that the assigned object is overwritten if it exists. This can lead to lost data. We *strongly* discourage the use of `<<-` within functions.

A more general form of assignment uses the `assign` function. The `assign` function lets you choose where the assignment takes place. You can assign either to a position in the search list, or to a particular frame. For example, to save some data on the session frame, you might use the following expression:

```
> assign("boo", 3, frame=0)
```

```
NULL
```

#### Hint: avoiding side effects of the `assign` function

The `assign` function can be used to write to permanent directories. As with `<<-`, we discourage such use within functions, because such assignments have potentially dangerous side effects.

## Testing and Coercing Data

Most functions expect input data of a particular type. Mathematical functions, for example, expect numeric input, while text processing functions expect character input. Other functions, designed to work with a wide variety of input data, have internal branches that use the data type of the input to determine what to do.

Unexpected data can often cause a function to stop with an error message. To protect against this behavior, many functions include expressions to test whether the input data is of the right type, and if necessary, coerce the data to be of that type. For example, functions for mathematical manipulation generally require that input data have numeric mode. Such functions frequently have conditionals of the following form:

```
if (!is.numeric(x)) x <- as.numeric(x)
```



Most testing in S-PLUS is done with functions having names beginning `i s.`, such as `i s.vector` and `i s.matrix`. Functions also exist to test for special values such as `NULL` and `NA` because comparisons involving these values always evaluate to `NA`. See the section *Comparison and Logical Operators* (page 126), for more information. Coercion is usually performed using functions with names of the form `as.type`, such as `as.vector` and `as.matrix`. Coercion using `as.type` functions is very strong, however, and can lead to loss of information. If all you need is to ensure that atomic data is of the proper *mode*, you can do this explicitly as follows:

```
mode(x) <- "type"
```

Objects with a `class` attribute can be tested in a more general way using the `inherits` function. For example, if you have a class `myclass`, you can test an object `x` for membership in the class using `inherits` as follows:

```
inherits(x, "myclass")
```

The table below lists the most common testing and coercing functions in S-PLUS.

The three functions `as.single`, `as.double`, and `as.integer` are used to modify the *storage mode* of numeric data. This is important if you need to pass code to C or Fortran routines, but can safely be ignored by everyone else.

*Table 5.5: Functions for testing and coercing data objects*

Type	Testing	Coercing
array	<code>i s.array</code>	<code>as.array</code>
character	<code>i s.character</code>	<code>as.character</code>
complex	<code>i s.complex</code>	<code>as.complex</code>
data.frame	<code>i s.data.frame</code>	<code>as.data.frame</code>
double	<code>i s.double</code>	<code>as.double</code>
factor	<code>i s.factor</code>	<code>as.factor</code>
integer	<code>i s.integer</code>	<code>as.integer</code>
list	<code>i s.list</code>	<code>as.list</code>
logical	<code>i s.logical</code>	<code>as.logical</code>
matrix	<code>i s.matrix</code>	<code>as.matrix</code>

*Table 5.5: Functions for testing and coercing data objects*

Type	Testing	Coercing
NA	i s. na	-
null	i s. null	as. null
numeric	i s. numeric	as. numeric
ts	i s. ts	as. ts
vector	i s. vector	as. vector
any class	i nheri ts	-
Note that ts stands for time series.		

## OPERATING ON SUBSETS OF DATA

Often, we want to perform calculations on only a selected subset of a data set. The most useful method for acting on a subset of a data set is to use *subscripting*. This is good S-PLUS programming because it treats the data object as a whole, rather than as a collection of elements.

*Subscripts* are used in mathematics and computer science to indicate position in an array. A vector is an ordered set of values which can be thought of as a one-dimensional array. A vector subscript corresponds to the element's position, or *index*, in the vector. In S-PLUS, the appropriate indices are constructed automatically from information supplied in one of the four following forms:

1. A vector of positive integers

In this case, the indices are simply the supplied integers, as in the following examples:

```
> x
[1] 1.9 3.0 4.1 2.6 3.6 2.3 2.8 3.2 6.6 7.6 7.4 1.0

> x[3]
[1] 4.1

> x[c(3, 5, 9)]
[1] 4.1 3.6 6.6
```

Similarly, `x[4]` would yield the fourth element and `x[7]` would yield the seventh element of `x`. To determine the total number of subscripts, use the function `length`, which returns the number of elements in atomic objects such as vectors and matrices, and the number of components in recursive objects such as lists. If a requested subscript is not in the range `1:length(x)`, S-PLUS returns `NA` to indicate a missing value.

2. A vector of negative integers

In this case, the indices are those of all *but* the indicated elements:

```
> x[-(3:5)]
[1] 1.9 3.0 2.3 2.8 3.2 6.6 7.6 7.4 1.0
```

### 3. A vector of logical values

The indices are those corresponding to positions for which the logical vector is TRUE. For example:

```
> x[x > 2]
[1] 3.0 4.1 2.6 3.6 2.3 2.8 3.2 6.6 7.6 7.4
> x[x > 2 & x < 4]
[1] 3.0 2.6 3.6 2.3 2.8 3.2
```

You can use a shorter logical vector to select, for example, every third element (S-PLUS automatically expands the subscripting vector to have the same length as the subscripted vector):

```
> x[c(F, F, T)]
[1] 4.1 2.3 6.6 1.0
```

### 4. A vector of character values

Here the indices are those corresponding to the positions of the character values in the subscripted vector's `names` attribute. This requires that the subscripted vector have a non-null `names` attribute. For example:

```
> state.abb["Alaska"]
Alaska
""
> names(state.abb) <- state.name
> state.abb["Alaska"]
Alaska
"AK"
```

The `""` after the first call of `state.abb["Alaska"]` is the character equivalent of an NA.

Any S-PLUS expression that evaluates to an appropriate subscript value can be included in the square brackets. This flexibility makes subscripting an even more powerful tool:

```
> state.name[grep("So*", state.name)]
[1] "South Carolina" "South Dakota"
```

## Subscripting on Matrices and Arrays

Subscripting data sets that are matrices or more general arrays is much like subscripting vectors. In fact, you can subscript them *exactly* like vectors if you keep in mind that arrays are stored in column-major order. For example, suppose we have the following matrix `J`:

```
> J

      [, 1] [, 2] [, 3] [, 4]
[1, ]    12    15     6    10
[2, ]     2     9     2     7
[3, ]    19    14    11    19
```

*Column-major order* simply specifies that the first three values fill the first column, the second three values fill the second column, and so on. So we could extract the ninth element of `J` as follows:

```
> J[9]

[1] 11
```

However, S-PLUS lets you use the structure of the array to your advantage, by allowing you to specify one subscript for each dimension of the array. Thus you can specify the desired elements of a matrix, for example, by specifying the rows and columns we want to extract:

```
> J[c(1, 3), c(2, 4)]

      [, 1] [, 2]
[1, ]    15    10
[2, ]    14    19
```

If the subscript for a given dimension is omitted, all subscripts are assumed. Thus, we have:

```
> J[, c(2, 4)]

      [, 1] [, 2]
[1, ]    15    10
[2, ]     9     7
[3, ]    14    19
```

As with vectors, the specified subscripts can be negative numbers, logical vectors, or character vectors, if appropriate.

```
> J[-1, c(2, 4)]

      [, 1] [, 2]
[1, ]     9     7
[2, ]    14    19

> illit <- state.x77[1:50, 3]
> state.x77[illit > 2, 3:5]

      Illiteracy Life Exp Murder
Alabama      2.1    69.05    15.1
Louisiana    2.8    68.76    13.2
Mississippi  2.4    68.09    12.5
New Mexico   2.2    70.32     9.7
South Carolina 2.3    67.96    11.6
Texas        2.2    70.90    12.2

> state.x77["Arizona", "Area"]

[1] 113417
```

The `drop=F` argument can be used within a matrix or array subscript to force S-PLUS not to drop dimensions. For example,

```
> J[1, 3, drop=F]

      [, 1]
[1, ]     6
```

As with vectors, matrix subscripts can be more complicated expressions to further increase the power of subscripts:

```
> stack.x[c(F, F, T), grep("Wa*", dimnames(stack.x)[[2]]),
+ drop=F]

      Water Temp
[1, ]         25
[2, ]         23
[3, ]         23
[4, ]         17
[5, ]         18
[6, ]         19
[7, ]         20
```

In the above example, the vector `c(F, F, T)` is replicated cyclically until its length matches the number of rows of `stack.x`. It thus selects every third row.

In general, operating on arrays of data is more complicated than operating on simple vectors. One problem is that subscripting can sometimes collapse your data. Another is that subscripting in the form `array[, , ]` yields only rectangular data sets. Often you need a way to extract more irregular subsets of arrays. You can do this by subscripting with a *matrix* representing the positions of the individual elements. For example, suppose we want to extract the elements of `J` in row 1, column 2 and row 3, column 3. We can do this by subscripting with the following matrix:

```
> subscr.mat <- matrix(c(1, 2, 3, 3), ncol=2, byrow=T)
> subscr.mat

      [, 1] [, 2]
[1, ]    1    2
[2, ]    3    3

> J[subscr.mat]

[1] 15 11
```

## Subscripting on Lists

Lists are vectors of mode `list` that can hold arbitrary S-PLUS objects as individual elements. Subscripting any vector returns a vector of the same mode, so that subscripting a list yields another object of mode `list`, as we can see by subscripting our sample list `mylist`:

```
> mylist[1]

$x:
[1] "Tom" "Dick" "Harry"

> mode(mylist[1])

[1] "list"
```

Yet the element `x` that we used in building `mylist` was of mode `character`. To get back that structure, we need to use the *list subscript* operator, `[[ ]]`:

```
> mylist[[1]]

[1] "Tom" "Dick" "Harry"
```

```
> mode(mylist[[1]])
```

```
[1] "character"
```

Using a vector of integers to subscript a list has a different effect from that for vectors or arrays. Subscripting a list always returns exactly one element; if the subscript is a vector or list, it is used *recursively*. That is, the first element of the subscript is used to extract an element from the top level list, the second element is used to extract an element from that element, and so on. For example, consider the list created as follows:

```
> biglist <- list(lista = list(listb = list(x = 1:10,  
+ y=10:20), cside=letters),  
+ listd = list("a", "r", "e"))
```

The following subscript expression returns the second element of the first element of the first element of `biglist`:

```
> biglist[[c(1, 1, 2)]]
```

```
[1] 10 11 12 13 14 15 16 17 18 19 20
```

The notation is shorthand for the following:

```
> biglist[[1]][[1]][[2]]
```

```
[1] 10 11 12 13 14 15 16 17 18 19 20
```

If the elements of a list are named, as in `mylist`, the named elements are called *components*, and can be extracted either by the list subscript operator or the component operator `$`:

```
> mylist$x
```

```
[1] "Tom" "Dick" "Harry"
```

Components of embedded lists can be extracted by nested use of the component operator:

```
> biglist$list$cside
```

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

You can also give a component name (as a character string) to the list subscript operator:



```
> biglist[[c("lista", "cside")]]
```

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

The component operator and the list subscript operator are equivalent; you can always use either one:

```
> biglist[["lista"]][listb]
```

```
$x:
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
$y:
```

```
[1] 10 11 12 13 14 15 16 17 18 19 20
```

## Subsetting from Data Frames

Data frames share characteristics of both matrices and lists, and subsetting from data frames shares characteristics of subsetting from matrices and subsetting from lists. For example, we can form a data frame from the numerous built-in data sets with information on the various states as follows:

```
> state.df <- data.frame(state.abb, state.center,
+   state.region, state.division, state.x77,
+   row.names=state.name)
```

The data frame `state.df` is a matrix with 50 rows and 14 columns, but it is also a list. For most subscripting purposes, it is most useful to treat the data frame as a matrix, and extract those rows and columns of most interest:

```
> state.df[5:7, 6:8]
```

	Population	Income	Illiteracy
California	21198	5114	1.1
Colorado	2541	4884	0.7
Connecticut	3100	5348	1.1

However, like lists, data frames have components; components are the columns of the data frame, and can be accessed just like list components:

```
> state.df$Population
```

```
[1] 3615 365 2212 2110 21198 2541 3100 579
[9] 8277 4931 868 813 11197 5313 2861 2280
[17] 3387 3806 1058 4122 5814 9111 3921 2341
[25] 4767 746 1544 590 812 7333 1144 18076
[33] 5441 637 10735 2715 2284 11860 931 2816
[41] 681 4173 12237 1203 472 4981 3559 1799
[49] 4589 376
```

## ORGANIZING COMPUTATIONS

As with any programming task, the key to successful function writing in S-PLUS is to organize your computations before you start. Break the problem down into its component pieces, and use the appropriate tools to complete each piece. Be sure to take advantage of existing functions, rather than writing new code to perform routine tasks.

The one major bit of wisdom added in the S-PLUS programming philosophy is this: *Treat every object as a whole*. Treating objects as whole entities is the basis for vectorized computation. You need not, and should not, operate on individual observations. Such computations in S-PLUS carry a high premium in both memory use and processing time.

Operating on whole objects is made simpler by a very flexible subscripting capability. In most cases where `for` loops or other looping constructions seem the most natural programming form, you will gain significantly in both performance and understandable code by using some form of subscripting.

### Programming Style

Most programmers have been exposed to at least some general rules of programming style. “Avoid Goto’s,” “Use top-down design,” and “Keep it modular” are some catch-phrases that embody style guidelines. S-PLUS has no Goto’s, but otherwise most of the style guidelines you’ve come to swear by are applicable to S-PLUS:

- *Modularize your code.* If you are designing a large function, see if you can use smaller functions to do most of the work. This reduces the size of the larger function, making it easier to follow, and makes each of the smaller functions reusable for other purposes.
- *Comment your code.* Comments are useful guides to the design of a function, particularly when you use an unusual or unfamiliar construction. Without comments, you may not be able to decipher your own code in six months, and may be completely opaque to anyone else who tries to read it. Comments within S-PLUS functions are sometimes roughly handled by the interpreter; it is safest to place a *blank* comment at the end of the line just before the place you want your substantive comment to appear:

```

primes <-
function(n = 100)
{
  n <- as.integer(abs(n))
  if(n < 2) return(integer(0))
  p <- 2:n
  smallp <- integer(0) #
# the sieve
  repeat {
    i <- p[1]
    smallp <- c(smallp, i)
    p <- p[p %% i != 0]
    if(i > sqrt(n)) break
  }
  c(smallp, p)
}

```

- *Document your code.* If you make your function available to others, include a help file describing its use, including complete descriptions of arguments and return values. The `prompt` function can be used to create a skeletal help file for any S-PLUS object.
- *Use existing functions.* If you already have a function to compute a least-squares fit, use it, don't rewrite it. S-PLUS has over two thousand functions built in, and most of these can be used to good effect in your own functions. Refer to the online manual, *Language Reference*, for full details.
- *Use parentheses to make groupings explicit.* If you're a sophisticated user, you can use the precedence of operations to your advantage in writing quick and dirty functions. But if you plan to maintain such functions, it is better to have the precedence explicit.
- *Avoid unnecessary looping.* As we've mentioned, the key point in S-PLUS programming is to treat data objects as whole objects. Whenever you see yourself using a `for` loop, ask yourself if the loop can be eliminated in favor of a single expression operating on the whole object. For atomic objects, such as vectors and matrices, this is almost always possible. Lists, however, do sometimes require `for` loops to get at the individual elements of the list. The `lapply` function performs this looping for you; you should use it in place of

explicitly constructing your own loops. Chapter 22, Using Less Time and Memory, gives several techniques for avoiding loops in S-PLUS code.

Many other useful programming rules can be found in Kernighan and Plauger (1974).

## Control Flow

S-PLUS expressions are normally evaluated sequentially. Groups of expressions can be collected within curly braces `{}`; such groups are treated as a single S-PLUS expression, but within that expression evaluation again proceeds sequentially. You can override the normal flow of control with the following looping constructions:

*Table 5.6: S-PLUS constructions.*

Construction	Description
<code>if (cond) expression</code>	Evaluates <i>cond</i> ; if true, evaluates <i>expression</i> .
<code>if (cond) expr else condexpr</code>	Evaluates <i>cond</i> ; if true, evaluates <i>expr</i> . If false, evaluates <i>condexpr</i> .
<code>ifelse(cond, expr1, expr2)</code>	The <code>ifelse</code> function is a <i>vectorized</i> version of the <code>if</code> statement. It evaluates the condition and returns elements of <i>expr1</i> for TRUE elements and elements of <i>expr2</i> for FALSE elements.
<code>switch(expr, ...)</code>	Evaluates <i>expr</i> , which must evaluate to either character or numeric. The value of <i>expr</i> is compared to the remaining arguments. If it matches one of these arguments exactly, the value of the evaluated argument is returned as the value of <code>switch</code> .
<code>break</code>	Terminates current loop and passes control out of the loop.
<code>next</code>	Terminates current iteration and immediately starts next iteration of the loop.
<code>return(expr)</code>	Terminates current function and immediately returns the value of <i>expr</i> .
<code>stop(message)</code>	Signals an error condition by terminating evaluation of the current function, printing the character string " <i>message</i> " as an error message, and returning to the S-PLUS prompt.

Table 5.6: S-PLUS constructions.

Construction	Description
<code>while (cond) expr</code>	Evaluates <i>cond</i> ; if true, evaluates <i>expr</i> ; then goes back to the top of the loop, evaluating <i>cond</i> again.
<code>repeat expr</code>	A simpler version of <code>while</code> , <code>repeat</code> performs no tests. It simply repeats <i>expr</i> indefinitely. Because they have no natural termination, repeat loops should have some breaks built in.
<code>for (name in expr1) expr2</code>	Evaluates <i>expr2</i> once for each <i>name</i> in <i>expr1</i> . Although <code>for</code> loops are widely used in most programming languages, they are generally less efficient in S-PLUS than calculations done by means of vectorized arithmetic.

In this section we give some examples showing each of these control constructs. We stress again that `for` loops are not efficient, although they are sometimes necessary. We will give several examples of code in which `for` loops can be successfully replaced by vectorized arithmetic.

## The if Statement

The `if` statement is the most common branching construction in S-PLUS. The syntax is simple:

```
if ( condition ) { expression }
```

where *condition* is any S-PLUS expression that evaluates to a logical value and *expression* is the expression to be evaluated if the *condition* is true. As with function bodies, the *expression* following the *condition* need only be braced if it has multiple statements. We suggest, however, that braces be included *at all times* for consistency and maintainability.

You use `if` statements to screen input data for suitability:

```
if (!is.numeric(x)) {
  stop("Data must be of mode numeric") }
```

The `stop` function stops evaluation of the calling function at the point where the stop occurs. The `stop` function takes a single argument, *message*, which should evaluate to a single character string. If *message* is supplied, the string is printed to the screen as the text of an error message. For example, under normal error handling, the above example would yield the following if `x` were not of numeric mode:

```
Error: Data must be of mode numeric  
Dumped
```

A common use of `if` statements is in missing-argument handling, using the `missing` function:

```
if (missing(y)) { y <- sqrt(x) }
```

When constructing conditions within `if` statements, you may want to test multiple conditions at once. Two operators, `&&` and `||` provide control AND and OR, respectively. The syntax is:

```
if (cond1 && cond2 ) expression  
if (cond1 || cond2 ) expression
```

These multiple condition operators evaluate only as far as necessary to return a value. For example, the `&&` operator first evaluates *cond1*. If *cond1* is true, then *cond2* is evaluated, and the result of that evaluation is the value of the whole condition statement. If *cond1* is false, however, `&&` returns false immediately, without evaluating *cond2*. Similarly, `||` evaluates only until it encounters a true statement, then returns true. It returns false only if every condition is false.

---

**Hint: using logical values**

---

There are actually three possible logical values: TRUE, FALSE, and NA. If an `if` statement encounters an NA, the calling function terminates, returning a message of the following form:

```
Missing value where logical needed
```

Do not confuse the vectorized AND and OR operators (`&` and `|`) with the conditional AND and OR. The vectorized operators return vectors of logical values, while conditionals return a single logical value.

---

## Handling Multiple Cases

One of the most common uses of the `if` statement is to provide branching for multiple cases. S-PLUS has no case statement, so (following Kernighan and Plauger) you will often implement cases using `if ... else if ... else ...` constructions. The idea is that you identify each case and have each case correspond to exactly one `if` or `else`. The general form is:

```
if ( case1 ) { expr1 }
else if ( case2 ) { expr2 }
else if ( case3 ) { expr3 }
else lastexpr
```

Such a construction makes it easy to follow the cases from one to the next, and serves as a check that all cases are covered. For example, here is a function to generate a specified number of random numbers from one of three distributions:

```
my.ran <-
function(n, distribution, shape)
{
# a function to generate n random numbers
  if(distribution == "gamma") rgamma(n, shape) else
  if(distribution == "exp") rexp(n) else
  if(distribution == "norm") rnorm(n) else
    stop("Unknown distribution")
}
```

S-PLUS provides a function, `switch`, that handles multiple cases in a slightly different way. The `switch` function takes as its first argument an S-PLUS expression, which is evaluated as the first step in the evaluation of `switch`. The value of the first argument, which should be a character string is then compared to the remaining arguments, and if it matches, the action specified by that argument is taken. The value can also be an integer in the range 1 to `nargs - 1`. In this case the integer `i` corresponds to the  $i+1$ th argument in the list. Such a structure tends to hide the nature of the individual cases; character values require that you label each individual case.

For example, we can rewrite `my.ran` as follows:

```
my.ran2 <-  
function(n, distribution, shape)  
{  
  # a function to generate n random numbers  
  switch(distribution, gamma = rgamma(n, shape),  
         exp = rexp(n), norm = rnorm(n),  
         stop("Unknown distribution"))  
}
```

In evaluating this expression, the interpreter evaluates the `distribution` argument to the `my.ran2` function. If the `distribution` argument is one of the three character strings "gamma", "exp", or "norm", the corresponding expression is evaluated. Otherwise, the `stop` expression is evaluated.

## The ifelse Function

The condition in an `if` statement must evaluate to a single logical value, either `TRUE` or `FALSE`. Thus, to carry out operations that involve multiple comparisons, the `if` statement would need to take place inside a loop. For example, here's an implementation of the *signum* function that takes a numeric object and puts a "1" wherever there is a positive value and a "-1" wherever there's a negative value:

```
sgn2 <-  
function(x)  
{  
  for(i in 1:length(x)) {  
    if(x[i] > 0)  
      x[i] <- 1  
    else if(x[i] < 0)  
      x[i] <- -1  
  }  
  x  
}
```

The `ifelse` function, however, provides a method for evaluating a condition for a whole vector or array of values, performing one action if the condition is true for a given element, and another action if the condition is false for the element. For example, here is a rewritten *signum* function, using `ifelse` twice:



```
sgn <- function(x)
{
  ifelse(x > 0, 1, ifelse(x < 0, -1, 0))
}
```

Not only is the version using `ifelse` much quicker, but it handles missing values:

```
> sgn2(c(1, 3, NA, -2))

Error in sgn2: Missing value where logical needed:
if(x[i] > 0) x[i] <- 1 else
if(x[i] < 0) x[i] <- -1
. . .
Dumped

> sgn(c(1, 3, NA, -2))

[1] 1 1 NA -1
```

The `ifelse` function essentially uses subscripting with some extra steps to behave correctly with NA values:

```
> ifelse
function(test, yes, no)
{
  answer <- test
  test <- as.logical(test)
  na <- is.na(test)
  n <- length(answer)
  test[na] <- F
  answer[test] <- rep(yes, length = n)[test]
  test[na] <- T
  answer[!test] <- rep(no, length = n)[!test]
  answer
}
```

The idea is to perform a test on an object, and replace those elements for which the test is true with one value, then replace those elements for which the test is false with another value. The `ifelse` function sets the subscripts corresponding to missing values to `F` before replacing the true elements (thus avoiding the error about missing values that `sgn2` reported), then resets those subscripts to `T` before replacing the false elements. The net result is that missing values remain missing.

However, if our original data have no missing values, we can improve things still further, using the original for loop as a hint. The telltale `x[i] <-` construction indicates we can try subscripting directly:

```
sgn3 <- function(x)
{
  x[x > 0] <- 1
  x[x < 0] <- -1
  x
}
```

## The repeat Statement

The `repeat` statement is the most simple-minded looping construction in S-PLUS. It performs no tests, but simply repeats a given expression indefinitely. The repeated expression should include some way out. The syntax for `repeat` is simple:

```
repeat expr
```

For example, here is a function that uses Newton's method to find real  $j$ th roots. Note the test for convergence inside the loop, and the `break` statement:

```
newton <-
function(n, j = 2, x = 1)
{
  # Use Newton's method to find jth root of n, starting
  # at old.x == x
  # Default is to find square root of n from old.x == 1
  old.x <- x
  repeat
  {
    new.x <- old.x - ((old.x^j - n) / (j * old.x^(j-1)))
    conv <- abs(new.x - old.x)
    if(conv/abs(old.x) < 1e-10)
      break
    old.x <- new.x
  }
  old.x
}
```

The `repeat` loop has no return value; to return the value of the last expression in the last completed loop, we must assign the value within the loop (to an object that exists outside the loop!), and then return that object. It is usually simpler to use a `return` statement inside the loop, as described in the next section.

## The return, break, and next Statements

It is often either necessary or prudent to leave a loop in the middle, before it reaches its natural end. In the case of `repeat`, which has no natural end, this is imperative. Similarly, a `while` loop may specify an action for one particular value that may be encountered anywhere in the loop. Once the action is performed, there is no point in continuing the loop. You exit loops from the middle using one of the three statements `break`, `next`, or `return`. Of these, `return` exits not only from the current loop, but from the current function, as well. In our `newton` example, that would serve us well—if we replace the `break` with a `return` we make it clearer what the returned value is:

```
repeat
{
  new.x <- old.x - ((old.x^j - n)/(j * old.x (j-1)))
  conv <- abs(new.x - old.x)
  if(conv/abs(old.x) < 1e-10)
    return(old.x)
  old.x <- new.x
}
```

However, such an abrupt departure from the function would not do if there remain additional calculations after the loop.

The `break` and `next` statements let you exit from the innermost current loop. The `break` statement tells S-PLUS to exit from the current loop and continue processing with the first expression following the loop. The `next` statement lets you exit from the current iteration of the current loop, and immediately start the next iteration. For example, here is a simple function that simulates drawing a card from a standard deck of 52 cards. If the card is not an ace, it is replaced, and another card is drawn. If the card is an ace, its suit is noted, it is replaced, and another card is drawn. The process continues until all four aces are drawn, at which time the function returns a statement of how many draws it took to draw all the aces:

```
draw.aces <-  
function()  
{  
  draws <- 0  
  aces.drawn <- rep(F, 4)  
  repeat {  
    draw <- sample(1:52, 1, replace = T)  
    draws <- draws + 1  
    if(draw %% 13 != 1)  
      next  
    aces.drawn[draw %% 13 + 1] <- T  
    if(all(aces.drawn))  
      break  
  }  
  cat("It took", draws,  
      "draws to draw all four of the aces!\n ")  
}
```

## The while Statement

You use the `while` statement to loop over an expression until a condition which is currently true becomes false or vice versa. For example, here is a function that returns a vector corresponding to the binary representation of an integer:

```
bitstring <-  
function(n)  
{  
  string <- numeric(32)  
  i <- 0  
  while(n > 0) {  
    string[32 - i] <- n %% 2  
    n <- n %% 2  
    i <- i + 1  
  }  
  firstone <- match(1, string)  
  string[firstone:32]  
}
```

Here is the result of calling the function on a given integer:

```
> bitstring(13)
[1] 1 1 0 1
```

Here `n` is made smaller on each iteration, eventually becoming zero. However, we have no way of knowing beforehand exactly how many times we need to execute the loop, so we use `while`. The syntax is simple:

```
while ( cond ) expr
```

Like the `for` statement described in the next subsection, the `while` statement is familiar to most programmers from other programming languages. And, like the `for` statement, it can often be avoided in S-PLUS programming. We have saved formal discussion of these two statements for last to emphasize that they are the operations of last resort in S-PLUS. You *may* need to use them, but you should always try a vectorized approach first.

## The for Statement

Using `for` loops is a traditional programming practice, and is fully supported in S-PLUS. You can, therefore, directly translate most Fortran-like `DO` loops into S-PLUS `for` loops and expect them to work. However, as we have already stated, `for` loops are not generally good S-PLUS programming, because they do not treat whole data objects. Instead, `for` loops attack the individual elements of data objects, and this is usually much less efficient in S-PLUS than techniques that operate on whole objects. *Always* be suspicious of any line in an S-PLUS function of the form

```
x[i] <- . . . .
```

There are, however, certain situations in which `for` loops may be necessary in S-PLUS:

- When the calculation on the  $i+1$ st element in a vector or array depends upon the result of the same calculation on the  $i$ th element.
- In some operations on lists (although the `lapply` and `sapply` functions perform some looping implicitly).

## SPECIFYING ARGUMENT LISTS

A well-chosen argument list can add considerable flexibility to most functions. At the same time, you may wish to maintain ease-of-use by specifying default values for as many arguments as possible. Some languages, notably C, make a distinction between a function's parameter list and a function *calls* argument list. S-PLUS maintains this distinction by talking of an argument's *formal name* (corresponding to the name specified in a parameter list) and its *actual name*, used when actually calling the function.

In this section we offer many examples of argument lists in S-PLUS functions, showing the wide variety available.

### Formal and Actual Names

When you define a function, you specify the arguments the function will accept by means of *formal names*. Formal names must be syntactically valid names (combinations of letters, numerals, and periods that do not begin with a numeral). The formal name `...` is used to pass arbitrary arguments.

When you *call* a function, you specify *actual names* for each argument. Unlike formal names, actual names can be any valid S-PLUS expression. (Of course, it must make sense to the function!) You can thus provide a function call, such as `length(x)`, as an argument.

### Specifying Default Arguments

You can specify the default value for any argument by explicitly providing the value when defining the formal argument, using the form *formal=value*. For example, consider the arguments to the `hist` function:

```
hist(x, nclass, breaks, plot=T,  
     probability=F, ..., xlab= deparse(substitute(x)))
```

Default arguments are supplied for the `plot`, `probability`, and `xlab` arguments. You can also specify defaults by providing code that allows for *missing* arguments. This technique is described in the next section.

## Missing-Argument Handling

You can test to see whether a given argument was supplied in the current function call by using an `if(missing(formal))` construction in the body of the function. For example, the following code sample shows how the `hist` function handles a missing `nclass` argument:

```
if(missing(nclass)) nclass <- log(length(x), base = 2) + 1
```

Thus, if `nclass` is missing (and no breaks are specified), `hist` determines a suitable value from the length of the data.

## Lazy Evaluation

Many programmers who come to S-PLUS from other programming languages make too much use of missing-argument handling. For example, they might write the following simple plotting function:

```
plotsqrt <-
function(x, y)
{
  z1 <- seq(1, x)
  if(missing(y)) plot(z1, sqrt(z1))
  else plot(z1, y)
}
```

Here the missing-argument construction simply supplies the default value `sqrt(z1)`. Because the supplied default depends upon the value `z1`, which is unknown until the completion of the first line of the body of the function, these programmers avoid using the default in the argument list. S-PLUS, however, uses *lazy evaluation*, which means that arguments are evaluated only as needed. Thus, we can write the above function more simply as follows:

```
plotsqrt2 <-
function(x, y=sqrt(z1))
{
  z1 <- seq(1, x)
  plot(z1, y)
}
```

S-PLUS doesn't need the value for `y` until the final expression, and at that time it can be successfully evaluated. In most programming languages, such a function definition would cause an error such as `Undefined variable sqrt(z1)`. But in S-PLUS, the argument isn't evaluated until the function body actually calls for it.

## Variable Numbers of Arguments

When you build a function for custom graphics or statistical procedures, you are often building on functions with a large number of arguments. Frequently, you need only a few of those arguments for your particular purposes. You can define just the arguments you need, but that reduces flexibility by limiting the number of ways you can modify the underlying command. Or, you can specify defaults in your new function that cover every argument of the underlying function, but this is a burden during programming. There is, however, a third option, that lets you specify the arguments you want to and still pass any other arguments to the underlying graphical and statistical functions.

The special formal name `...` is used in argument lists to specify that an arbitrary number of arguments are to be passed to a function within the body of the function. We have already seen one example:

```
> hi.st
function(x, nclass, breaks, plot = TRUE,
         probability = FALSE, ..., xlab =
         deparse(substitute(x)))
```

The `hi.st` function is a special purpose variant of the more general function `barplot`, and `barplot` can take a large number of arguments. Rather than duplicate all those arguments, `hi.st` uses `...` to pass any that the user specifies to `barplot`. Within the body of a function, the only valid use of `...` is as an argument inside a function call, as in this fragment from the `hi.st` function:

```
if(plot)
  invisible(barplot(counts, width = breaks,
                    histo = T, ..., xlab = xlab))
else list(breaks = breaks, counts = counts)
```

Here the `...` passes any unmatched arguments in the original call to `hi.st` to the `barplot` function. However, the arbitrary arguments can be passed to *any* function, so you can, for example, create a function to take the mean of an arbitrary number of data sets using the `mean` and `c` functions:

```
my.mean <- function(...) mean(c(...))
```

You could use the `list` function to loop over the various arguments in turn, for example to obtain the individual means of an arbitrary set of data sets:

```
all.means <-
function(...) {
  dsets <- list(...)
  n <- length(dsets)
  means <- numeric(n)
```



```

    for(i in 1:n)
    {      means[i] <- mean(dsets[[i]])
    }
    means
}

```

Arguments may follow ... in function definitions. Arguments so defined must be supplied by *name* when included in a function call, and they may not be abbreviated. For example, if we want to include the `trim` argument to `mean` in our `my.mean` function, we can do that as follows:

```

> my.mean <- function(... , trim=0.0)
{ mean(c(...), trim=trim) }

```

When calling `my.mean`, we can obtain the `trim` argument only by explicitly naming it:

```

> my.mean(corn.rai n, corn.yi el d, tri m=0.5)

[1] 17.95

```

## Required and Optional Arguments

Required arguments are those arguments for which the function definition provides neither a default value nor instructions on what to do if the argument is missing. All other arguments are optional. Most argument names can be abbreviated; exceptions are arguments passed through ... or arguments specified after a .... For example, consider the arguments to `hist`:

```

hist(x, ncl ass, breaks, plot=T,
     probabi lity=F, ..., xlab= deparse(substi tute(x)))

```

Here `x` is a required argument; `ncl ass` and `breaks` are optional arguments with missing-argument handling; `plot`, `probability`, and `xlab` are optional arguments with defaults; and ... allows you to pass other graphics parameters as arguments. The optional arguments listed *before* the ... can be abbreviated, as in `hist(course.grades, ncl =3)`, but `xlab` cannot be abbreviated. It must be written out in full, or it will not be matched. When an argument list includes ..., actual arguments that cannot be matched to a formal argument are simply ignored. If the argument list does not include ..., unmatched arguments generate an error of the form:

```

Error in call to function: argument argument not matched

```

## ERROR HANDLING

A too-often neglected aspect of function writing is error-handling; specifying what to do when something goes wrong. When writing “quick-and-dirty” functions for your own use, it doesn’t make sense to invest much time in “bullet-proofing” your functions—that is, testing the data for suitability at each stage of the calculation, and providing an informative error message and a graceful exit from the function if the data proves unsuitable. However, when you broaden the intended audience of your function, good error handling becomes crucial.

We have already seen one mechanism for a graceful exit from functions, the `stop` function. The `stop` function, when encountered, immediately stops evaluation of the current function, issues an error message, then dumps debugging information to a data object named `last.dump`. The `last.dump` object is a list, which can either be printed directly, or reformatted in a more useful form by the `traceback` function:

```
> traceback()

Message: Number of columns of x should be the same as
        number of rows of y
3: stop("Number of columns of x should be the same as
        number of rows of y"
2: 1:6 %*% rep(1/6, 1000)
1:
```

The amount of information in `last.dump` is controlled by the `error` argument to the `options` function. The default value is `dump.calls`, as shown here—the `last.dump` object gets a list of function calls, starting with the top-level call and including all calls within the function up to and including the call indicating the error. The `dump.frames` option provides more information, because it includes the complete set of frames created during the current evaluation. However, `dump.frames` can generate a very large `last.dump` object, so it should be used only for debugging purposes, not general error handling. Other options for the `error` argument to `options` are discussed in Chapter 6, *Debugging Your Functions*. The `warning` function is similar to `stop`, but issues a warning message when triggered by a potentially hazardous condition, such as data coercion:

```
if (!is.numeric(x))
{
  warning("Coercing to mode numeric")
  x <- as.numeric(x)
}
```

---

It is good programming practice to place `stop` statements within functions to mark the limits of the function's capability, as in our `newton` function:

```
if(n < 0 && j %% 2 == 0) stop("No real roots")
```

As with most matters of programming style, the degree to which you incorporate stops and warnings will depend upon the level of finish you are intending for your functions. Functions for distribution to others should be held to a higher standard than functions for your own use.

## DATA INPUT

Most data input to S-PLUS functions is in the form of named objects passed as required arguments to the functions. Thus, for example, we have

```
> mean(corn.rain)
```

```
[1] 10.78421
```

Data can also be generated “on-the-fly” by passing S-PLUS expressions, such as calls to the `c` function, as arguments:

```
> mean(c(5, 9, 23, 42))
```

```
[1] 19.75
```

However, if you are building turnkey systems or other applications in which you want to hide as much of the S-PLUS machinery as possible, you may want to build functions that read data from an existing file, create an S-PLUS object on the fly, then perform some analysis and return a value. Such functions are helpful in that they conceal many of the details of the structure of S-PLUS objects from users who may not know or care to know such details. For reading in data from files, the principal tools are `scan`, used to read ordinary sequential files, and `read.table`, used to read tabular data into S-PLUS data frames.

### Reading Vector and Matrix Data With `scan`

The standard mechanism for reading ASCII data into S-PLUS is to use the `scan` function, which can read from either standard input or from a file. By default, `scan` expects numeric data separated by white space, although there are options that let you specify the type of data being read and the separator. When using `scan` to read data files, it is helpful to think of each line of the data file as a *record*, with individual observations as *fields*. For example, the following expression creates a matrix named `x` from a data file specified by the user:

```
x <- matrix(scan(datafile), ncol = 10, byrow = T)
```

Here the data file is assumed to have 10 columns of numeric data; the matrix contains a number of observations for each of these ten variables. To read in a file of character data, use `scan` with the `what` argument:

```
x <- matrix(scan(datafile, what = ""), ncol = 10, byrow = T)
```

Any character vector can be used in place of `""`. For most efficient memory allocation, what should be the same size as the object to be read in. For example, to read in a character vector of length 1000, use

```
> scan(what=character(1000))
```

The `what` argument to `scan` can also be used to read in data files of mixed type, for example, a file containing both numeric and character data, as in the following sample file, `table.dat`:

```
Tom 93 37
Joe 47 42
Dave 18 43
```

In this case, you provide a list as the value for `what`, with each list component corresponding to a particular field:

```
> z <- scan("table.dat", what=list("", 0, 0))
> z
```

```
[[1]]:
[1] "Tom" "Joe" "Dave"
```

```
[[2]]:
[1] 93 47 18
```

```
[[3]]:
[1] 37 42 43
```

S-PLUS creates a list with separate components for each field specified in the `what` list. You can turn this into a matrix, with the subject names as column names, as follows:

```
> matz <- rbind(z[[2]], z[[3]])
> dimnames(matz) <- list(NULL, z[[1]])
> matz
```

```
      Tom Joe Dave
[1,]  93  47  18
[2,]  37  42  43
```

You can scan files containing multiple line records by using the argument `multi.line=T`. For example, suppose you have a file `heart.all` containing information in the following form:

```
j ohns 1
450 54.6
marks 1 760 73.5
. . .
```

You can read it in with `scan` as follows:

```
> scan('heart.all', what=list("", 0, 0, 0), multi.line=T)

[[1]]:
[1] "j ohns" "marks" "avery" "abl e" "si mpson"
. . .
[[4]]:
[1] 54.6 73.5 50.3 44.6 58.1 61.3 75.3 41.1 51.5 41.7 59.7
[12] 40.8 67.4 53.3 62.2 65.5 47.5 51.2 74.9 59.0 40.5
```

If your data is in *fixed format*, with fixed-width fields, you can use `scan` to read it in using the `widths` argument. For example, suppose you have a data file `dfi.le` with the following contents:

```
01gi raffe. 9346H01-04
88donkey . 1220M00-15
77ant          L04-04
20gerbi l . 1220L01-12
22swal low. 2333L01-03
12l emmi ng    L01-23
```

You identify the fields as numeric data of width 2, character data of width 7, numeric data of width 5, character data of width 1, numeric data of width 2, a hyphen or minus sign that you don't want to read into S-PLUS, and numeric data of width 2. You specify these types using the `what` argument to `scan`. To simplify the call to `scan`, you define the list of `what` arguments separately:

```
> dfi.le.what <- list(code=0, name="", x=0, s="", n1=0,
                     NULL, n2=0)
```

(NULL indicates suppress scanning of the specified field.) You specify the widths as the `widths` argument to `scan`. Again, it simplifies the call to `scan` to define the `widths` vector separately:

```
> dfi.le.widths <- c(2, 7, 5, 1, 2, 1, 2)
```

You can now read the data in `dfi le` into S-PLUS calling `scan` as follows:

```
> dfi le <- scan("dfi le", what=dfi le.what,
  wi dths=dfi le.wi dths)
```

If some of your fixed-format character fields contain leading or trailing white space, you can use the `stri p. whi te` argument to strip it away. (The `scan` function always strips white space from numeric fields.) See the `scan` help file for more details.

## Reading Data Frames

Data frames in S-PLUS were designed to resemble tables. They must have a rectangular arrangement of values and typically have row and column labels. Data frames arise frequently in designed experiments and other situations. If you have a text file with data arranged in the form of a table, you can read it into S-PLUS using the `read. table` function. For example, consider the data file `auto. dat`:

Model	Price	Country	Reliab	Mi leage	Type
Acura Integra4	11950	Japan	5	NA	Small
Audi 1005	26900	Germany	NA	NA	Medium
BMW325i 6	24650	Germany	94	NA	Compact
ChevLumi na4	12140	USA	NA	NA	Medium
FordFesti va4	6319	Korea	4	37	Small
Mazda929V6	23300	Japan	5	21	Medium
MazdaMX-5Mi ata	13800	Japan	NA	NA	Sporty
Nissan300ZXV6	27900	Japan	NA	NA	Sporty
OldsCal ai s4	9995	USA	2	23	Compact
ToyotaCressi da6	21498	Japan	3	23	Medium

All fields are separated by spaces and the first line is a header line. To create a data frame from this data file, use `read. table` as follows:

```
> auto <- read. table(' auto. dat', header=T)
> auto
```

	Price	Country	Reliab	Mi leage	Type
Acura Integra4	11950	Japan	5	NA	Small
Audi 1005	26900	Germany	NA	NA	Medium
BMW325i 6	24650	Germany	94	NA	Compact
ChevLumi na4	12140	USA	NA	NA	Medium
FordFesti va4	6319	Korea	4	37	Small
Mazda929V6	23300	Japan	5	21	Medium
MazdaMX-5Mi ata	13800	Japan	NA	NA	Sporty

Ni ssan300ZXV6	27900	Japan	NA	NA	Sporty
OI dsCal ai s4	9995	USA	2	23	Compact
ToyotaCressi da6	21498	Japan	3	23	Medi um

As with `scan`, you can use `read.table` within functions to hide the mechanics of S-PLUS from the users of your functions.



## DATA OUTPUT

S-PLUS is an interactive system, so that virtually anything you type prompts a response from S-PLUS. In general, this response is the value of the evaluated expression, which S-PLUS prints automatically upon completion of the evaluation. If the value is assigned, automatic printing is not performed:

```
> 7 + 3

[1] 10

> a <- 7 + 3
```

Other responses range from error messages to interactive prompts from within a function call. We discussed error messages in the section Error Handling (page 154). This section discusses the various “direct” forms of creating output—as returned values, as side effects, or as temporary or permanent data files.

### Formatting Output

The format of the automatically printed S-PLUS return value is determined in part by the mode of the returned object and in part by various session options. The `width` argument option, 80 by default, specifies how many characters fit on a line. The `length` argument option, 48 by default, specifies how many lines fit on a page, and also tells S-PLUS where to reprint matrix dimensions (they are printed once per page). If you are using S-PLUS in a standard  $80 \times 24$  terminal, the following call to the `options` function sets up your session so that long data sets can be conveniently viewed with the `page` function:

```
> options(length=23)
```

The `digits` option, 7 by default, specifies the number of significant digits to print. To see full double precision output, set `digits` to 17 as follows:

```
> options(digits=17)
> pi

[1] 3.1415926535897931
```

You can also call `print` explicitly with the `digits` argument to override the value of `options("digits")`:

```
> options(digits=7)
> print(pi, digits=17)
```

```
[1] 3.1415926535897931
```

To print numeric data as a formatted character string, use the `format` function, which returns a character vector the same length as its input in which all values have the same length:

```
> format(sqrt(1:10))
```

```
[1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068"
[6] "2.449490" "2.645751" "2.828427" "3.000000" "3.162278"
```

The `format` function respects the `digits` option:

```
> options(digits=3)
> format(sqrt(1:10))
```

```
[1] "1.00" "1.41" "1.73" "2.00" "2.24" "2.45" "2.65"
[8] "2.83" "3.00" "3.16"
```

You can also specify the `digits` as an argument to `format`. Like the `signif` function, `format` interprets the `digits` argument as the number of significant digits retained. However, it replaces trailing zeros with blanks to make all the strings the same length:

```
> format(sqrt(1:10), digits=3)
```

```
[1] "1" "1.41" "1.73" "2" "2.24" "2.45" "2.65"
[8] "2.83" "3" "3.16"
```

You can use `format` together with the `round` or `signif` to further control the action of the `digits` argument:

```
> format(round(sqrt(1:10), digits=5))
```

```
[1] "1.00000" "1.41421" "1.73205" "2.00000" "2.23607"
[6] "2.44949" "2.64575" "2.82843" "3.00000" "3.16228"
```

```
> format(signif(sqrt(1:10), digits=5))
```

```
[1] "1.0000" "1.4142" "1.7321" "2.0000" "2.2361" "2.4495"
[7] "2.6458" "2.8284" "3.0000" "3.1623"
```

The `round` function uses `digits` to specify the number of decimal places, while `sigfig` uses it to specify the number of significant digits retained.

## Constructing Return Values

When the body of a function is a braced expression, the value of the function is the value of the last expression within the braces. This fits well with the usual “top-down” design paradigm, where the goal is to start with some input or starting point, proceed through a set of operations, and return the finished output. For most simple functions, that is precisely what is going on, and the only thing you have to be sure of is that the final value is what you actually wanted returned. Thus, if the body of the function carries out a series of replacements, the final line might be the name of the object in which the replacements were done, as in:

```
bi gger <-
function(x, y)
{
  y.i s.bi gger <- y > x
  x[y.i s.bi gger] <- y[y.i s.bi gger]
  x
}
```

Often, however, particularly in functions doing more complicated analyses, you will want to return values generated throughout the function. The way to do this is to assign the intermediate calculations to temporary objects within the function, then gather the objects together into a return list.

For example, suppose you have a data file containing daily sales for each of ten department stores over a span of one month. Each month, you want to compute a summary of the months sales, using the daily sales information as the input data. Here is a function, `monthly.summary`, that reads in such a data file and creates a matrix of the input data, then performs the desired analysis:

```
monthly.summary <-
function(datafile)
{
  x <- matrix(scan(datafile), nrow = 10, byrow = T)
  store.totals <- as.vector(x %*% rep(1, ncol(x)))
  mean.sales <- mean(store.totals)
  attr(mean.sales, "dev") <- stdev(store.totals)
  best.performer <- sort.list(store.totals)[nrow(x)]
  list("Total Sales" = store.totals,
       "Average Sales" = mean.sales,
```

```
      Star = best.performer)  
}
```

There are several interesting features in this function. First, notice that the store totals are computed with matrix multiplication rather than with the `apply` function; for a chain with 100 stores, the performance improvement would be striking. Second, notice that the function has no side effects; all the calculations are assigned to objects in the frame of the function, then those objects are combined into a list which is the function's return value. This is the preferred S-PLUS method for returning a number of different results.

There are many reasons to avoid creating permanent objects from within functions, but the most compelling one is that if a function creates a data object, then it will overwrite any existing object of that name. Thus, if our `monthly.summary` function had created objects named `store.totals`, `mean.sales`, and `best.performer`, we would lose those objects every time we ran the function, unless we consciously copied them to different objects before rerunning `monthly.summary`. With the list paradigm, it is easy to create a data object for each month, then compare them element by element:

```
> Apr92 <- monthly.summary("april.sales")  
> May92 <- monthly.summary("may.sales")  
> Apr92
```

```
"Total Sales":  
[1] 55 59 91 87 101 183 116 119 78 166
```

```
"Average Sales":  
[1] 105.5  
attr("Average Sales", "dev"):  
[1] 42.16436
```

```
$Star:  
[1] 6
```

```
> May92
```

```
"Total Sales":  
[1] 65 49 71 91 105 163 126 129 81 116
```

```
"Average Sales":  
[1] 99.6  
attr("Average Sales", "dev"):  
[1] 34.76013
```

```
$Star:
[1] 6
```

## Side Effects

A *side effect* is any result of a function that is not a returned value. Examples include graphics plots, printed values, or newly created or permanently modified data objects. Not all side effects are bad—graphics functions are written explicitly to produce side effects, while their return values are usually of no interest. In such cases, you can suppress automatic printing by returning the value *invisibly*, using the `invisible` function. Most of the printing functions, such as `print.atomic`, do exactly this:

```
print.atomic <-
function(x, quote = T, ...)
{
  if(length(x) == 0)
    cat(mode(x), "(0)\n ", sep = "")
  else .C("pratom",
          list(x), list(character(0)), as.logical(quote))
  invisible(x)
}
```

You should watch out for and try to avoid hidden side effects, because they can wreak havoc with your data. For example, we created our `monthly.summary` function without side effects. Many S-PLUS programmers are tempted to use permanent assignment (the cause of most bad side effects) because it lets expressions within functions work exactly the same as they work at the S-PLUS prompt. The difference is that if you say

```
a <- expression
```

at the S-PLUS prompt, you are likely to be aware that `a`, if it exists, is about to be overwritten. On the other hand, if you call a function, `deadly`, that contains the same expression, you may have no idea that `a` is about to be destroyed.

## Writing to Files

Writing data to files from within functions is, in general, as dangerous a practice as permanent assignment. It is better to have special functions to create output data files. Such functions should include arguments for specifying the output file name and the format of the data to be included.

The actual writing can then be done by any of a number of S-PLUS functions. The simplest of these are `cat` and `write`. The result of either `write` or `cat` is just an ASCII file with data in it. There is no S-PLUS structure written in.

Of the two commands, `write` has an argument for specifying the number of columns and thus is more useful for retaining the format of a matrix.

By default, `write` writes matrices column by column, five values per line. If you want the matrix represented in the ASCII file in the same form it is represented in S-PLUS, transform the matrix first with the `t` function and specify the number of columns in your original matrix:

```
> mat

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

> write(t(mat), "mat", ncol=4)
```

You can view the resulting file with a text editor or pager; it contains the following three lines:

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

The `cat` function is a general-purpose writing tool in S-PLUS, used for writing to the screen as well as writing to files. It can be useful in creating free-format data files for use with other software, particularly when used with the `format` function:

```
> cat(format(runif(100)), fill=T)

0.261401257 0.556708986 0.184055283 0.760029093 . . . .
```

The argument `fill=T` limits line length in the output file to the width specified in your options object. To use `cat` to write to a file, simply specify a file name with the `file` argument:

```
> x <- 1:1000
> cat(x, file="mydata", fill=T)
```

The files written by `cat` and `write` do not contain S-PLUS structure information; to read them back into S-PLUS you must reconstruct this information. To write ASCII versions of actual S-PLUS objects, complete with structural information, use the `dump` and `dput` functions.

The `dump` function is useful primarily as a programmer's tool. It lets you dump editable, sourceable versions of S-PLUS objects. You can use `dump`, for example, to collect a module of functions and distribute it via electronic mail.

The `dput` function can be thought of as a companion to `assign`. Where `assign` creates S-PLUS objects, in binary form, `dput` creates ASCII versions of the data. You use `dput` as follows:

```
> dput(x, file="mydata")
```

The output of `dput` is in the form of a call to the `structure` function; this output can be read back into S-PLUS using the `dget` function.

## Creating Temporary Files

You can use `cat`, `write`, and `dput` together with the `tempfile` function to create uniquely named temporary files. Such files are convenient for use with text processing tools and a variety of other purposes. For example, the `ed` function creates a temporary file to hold the object being edited:

```
> ed
function(data, file = tempfile("ed."), editor = "ed")
{
  drop <- missing(file)
  if(missing(data)) {. . .}
  else if(mode(data) == "character" &&
           length(attributes(data)) == 0)
    cat(data, file = file, sep = "\n ")
  else if(is.atomic(data) &&
           length(attributes(data)) == 0)
    cat(data, file = file, fill = T)
  else dput(data, file = file)
  . . .
}
```

The `tempfile` function creates a unique name, in this case composed of the string `ed.` and a unique ID number. The `tempfile` function does *not* create the file! You must use `cat`, `write`, or `dput` to actually write the file.

The “temporary” files created as above are ordinary files, written to the directory specified by the `S_TMP` environment variable on your Windows system. This is customarily a temporary storage location, frequently wiped clean. However, to prevent overloading this directory, it is best if you incorporate file cleanup into your function. This is discussed in the next section.

## WRAP-UP ACTIONS

The more complicated your function, the more likely it is that it will complete with some loose ends dangling. For example, the function may create some temporary files, or alter some S-PLUS options or graphics parameters. It is good programming style to write functions that run cleanly, without permanently changing the environment. Wrap-up actions allow you to clean up the loose ends.

The most important wrap-up action is to ensure that the function returns the appropriate value or generates the desired side effect. Thus, the final line of a function is often simply the name of the object to be returned, or an expression for constructing that object, as in our `monthly.summary` example.

To restore changed parameters or specify arbitrary wrap-up actions, use the `on.exit` function. With `on.exit`, you ensure that the desired actions are carried out whether the function completes successfully or not. For example, highly recursive functions often overrun the default limit for nested expressions. Here is a version of the factorial function that raises the limit from 256 to 1024, then cleans up:

```
fac4 <-  
function(n)  
{  
  old <- options(expressions = 1024)  
  on.exit(options(old))  
  if(n <= 1) 1  
  else n * Recall(n - 1)  
}
```

Compare this with a function that uses the default expression nesting limit:

```
fac2 <-  
function(n)  
{  
  if(n <= 1) 1  
  else n * Recall(n - 1)  
}
```

Here is the response from S-PLUS when each function is tried on  $n=80$ :

```
> fac4(80)  
[1] 7.156946e+118
```



```
> fac2(80)
```

```
Error: Expressions nested beyond limit (256) - increase  
      limit with options(expressions=...)  
      only 28 of 86 frames dumped  
Dumped
```

The error message from `fac2` indicates that the 256 limit is back in effect, as we hoped it would be. To remove temporary files, use `on.exit` together with the `unlink` function:

```
fcn.A <- function(data, file=tempfile("fcn"))  
{  
  on.exit(unlink(file))  
  dput(data, file=file)  
  ...  
}
```

## EXTRACTION AND REPLACEMENT FUNCTIONS

Assignments in which the left hand side is a function call are handled differently from those in which the left hand side is a name. An expression of the form `f(x) <- value` is evaluated as the following assignment:

```
x <- "f<-"(x, value)
```

This evaluation requires, of course, a function "f<-" (read "f gets") corresponding to the function `f`. Functions appearing on the right hand side of assignments are *extraction* functions, that is, they take a data object and return either a portion of the data or some attribute of the data. For example, the `dim` function returns the `dim` attribute of a matrix:

```
> dim(state.x77)
```

```
[1] 50 8
```

The corresponding function "dim<-" *replaces* the returned attribute with a user-specified value. For this reason, "dim<-" and other "gets" functions are called *replacement functions*, or simply *replacements*. S-PLUS includes many replacements, most notably those associated with subscripting ("`<[-`" and "`<[[`" and those associated with attribute extraction ("`dim<-`", "`names<-`", "`class<-`"). Because the names of replacement functions (such as "dim<-" ) are not syntactic names, you must use the `get` function to examine the definitions of such functions:

```
> get("dim<-")
function(x, value)
{
  .Internal(dim(x) <- value, "Sreplace", T, 10)
}
```

Replacement functions should generally be defined whenever you define new extraction functions. New extraction functions, in turn, are generally associated with newly created attributes. A simple example is defining an attribute, `doc`, to hold a brief description of a data object. You can always assign any attribute to any object using the "`attr<-`" function. Functions that extract or replace attributes should usually start there:

```
"doc<-" <-
function(x, value)
{
  attr(x, "doc") <- value
  x
}
```

Two things are worth noting about the definition of "doc<-": first, it returns the complete, modified object, not just the modified attribute, and second, it does no assignment. The S-PLUS evaluator performs the actual assignment.

The corresponding extraction function, doc, starts naturally enough with the attr function. A simple version can be a "one-liner":

```
doc <- function(x) { attr(x, "doc") }
```

However, depending on the form that the doc attribute takes, you may want to modify the output somewhat. For example, suppose we want to use simple character strings, perhaps with newlines, as the values for the doc attribute:

```
> doc(geyser) <- "Waiting time between eruptions and the
Continue string: duration of the eruption for the Old
Continue string: Faithful geyser in Yellowstone."
```

Our simple doc function gives the following response:

```
> doc(geyser)

[1] "Waiting time between eruptions and the\nduration of
the eruption for the Old\nFaithful geyser in Yellowstone."
```

This is not the most readable form. But if we modify doc slightly, we get a much more readable output:

```
> doc <- function(x) { cat(attr(x, "doc"), "\n ") }
> doc(geyser)

Waiting time between eruptions and the
duration of the eruption for the Old
Faithful geyser in Yellowstone.
```

You can build extraction functions to extract almost any piece of data you are interested in. Such functions typically use other extraction functions as their starting point, as in the following functions for finding the even elements and odd elements, respectively, of an input vector:

```
evens <- function(x)
{
  indices <- seq(along = x)
  x[indices %% 2 == 0]
}
odds <- function(x)
{
  indices <- seq(along = x)
  x[indices %% 2 == 1]
}
```

The following simple examples illustrate the use of these functions:

```
> evens(1:10)

[1] 2 4 6 8 10

> odds(1:10)

[1] 1 3 5 7 9
```

Here we build on the subscripting function "[" to extract a certain subset of the data. The subscripting replacement function "[<-" is the logical place to begin writing the corresponding replacement functions "evens<-" and "odds<-":

```
"evens<-" <-
function(x, value)
{
  indices <- seq(along = x)
  x[indices %% 2 == 0] <- value
  x
}

"odds<-" <-
function(x, value)
{
  indices <- seq(along = x)
  x[indices %% 2 == 1] <- value
  x
}
```

The following simple examples illustrate replacement using these functions:

```
> xx <- 1:10
> odds(xx) <- c(10, 20, 30, 40, 50)
> evens(xx) <- c(5, 10, 15, 20, 25)
> xx

[1] 10 5 20 10 30 15 40 20 50 25
```

As a final example of extraction and replacement, consider the problem of extracting and replacing matrix column and row names. These names are usually extracted using the `dimnames` function and replaced using `"dimnames<-"`. It would be convenient, however, to be able to simply type `rownames(x)` and see the current row names. Here is a simple function that allows just such usage:

```
rownames <-  
function(x)  
{  
  if(!is.null(dimnames(x)[[1]]))  
  {  
    dimnames(x)[[1]]  
  } else  
  {  
    character(dim(x)[1])  
  }  
}
```

If there are currently *no* row names, `rownames` returns a vector of empty character strings. The return vector has length equal to the number of rows.

The corresponding replacement function saves the current column names, if any, and inserts the new row names:

```
"rownames<-" <-  
function(x, value)  
{  
  if(!is.null(dimnames(x)[[2]]))  
  {  
    colnames <- dimnames(x)[[2]]  
  } else  
  {  
    colnames <- NULL  
  }  
  dimnames(x) <- list(value, colnames)  
  x  
}
```

## OPERATORS

In addition to the built-in operators discussed previously, S-PLUS allows you to define your own infix operators. Such operators must have names of the form "*%anything%*", like the built-in operator "*%\*%*". These operators are ordinary functions, but because the string "*%anything%*" is not syntactically a name, you must refer to them in S-PLUS using the `get` function:

```
> get("%*%")
function(x, y)
{
  ldx <- length(dim(x))
  ldy <- length(dim(y))
  if(ldx != 2 && ldy != 2)
  {
    dim(x) <- c(1, length(x))
    dim(y) <- c(length(y), 1)
  }
  else ...
}
```

Here is an operator for raising a matrix to a power:

```
"%^%" <-
function(matrix, power)
{
  matrix <- as.matrix(matrix)
  if(ncol(matrix) != nrow(matrix))
    stop("matrix must be square")
  if(length(power) != 1)
    stop("power must be one number")
  if(all(t(matrix) == matrix))
  {
    # this is a symmetric matrix
    e <- eigen(matrix)
    m <- e$vectors %*% diag(e$values^power)
    %*% t(e$vectors)
  }
  else
  {
    # this is an asymmetric matrix
    if(trunc(power) != power)
      stop("integer power required for matrix")
    m <- diag(ncol(matrix))
    if(power != 0)
      for(i in 1:abs(power))
        m <- m %*% matrix
    if(power < 0)

```

---

```

        m <- solve(m)
    }
    m
}

```

Once defined, this operator can be used exactly as any other infix operator:

```

> x <- matrix(c(2, 1, 1, 1), ncol=2)
> x %^% 3

```

```

      [, 1] [, 2]
[1, ]   13    8
[2, ]    8    5

```

User-defined operators have precedence equivalent to the built-in operators `%%`, `%/%`, and `%*%`, that is, below `:` and above `*` and `/`.





# DEBUGGING YOUR FUNCTIONS

# 6

---

<b>Basic S-PLUS Debugging</b>	<b>178</b>
Printing Intermediate Results	179
<b>Interactive Debugging</b>	<b>182</b>
Starting the Inspector	183
Examining Variables	184
Controlling Evaluation	188
Entering, Marking, and Tracking Functions	190
Entering Functions	191
Marking Functions	191
Marking the Current Expression	192
Viewing and Removing Marks	193
Tracking Functions	194
Modifying the Evaluation Frame	196
Error Actions in the Inspector	198
<b>Other Debugging Tools</b>	<b>202</b>
Using the S-PLUS Browser Function	202
Using the S-PLUS Debugger	203
Tracing Function Evaluation	204

Debugging your functions generally takes much longer than writing them, because relatively few functions work exactly as you want them to the first time you use them. You can (and should) design large functions before writing a line of code, but because of the interactive nature of S-PLUS, it is often more efficient to simply type in a smaller function, then test it and see what improvements it might need.

S-PLUS has several tools for debugging your functions. In general, these tools make use of the techniques of the previous chapter to provide you with as much information as possible about the state of the evaluation.

This chapter describes several techniques for debugging S-PLUS functions, including using these built-in tools and using the techniques of Chapter 20, Computing on the Language, to extend these tools even further. Refer also to Chapter 21, Data Management, for a detailed discussion of frames.

## BASIC S-PLUS DEBUGGING

When an error occurs in an S-PLUS expression, S-PLUS generally returns an error message and the word `Dumped`:

```
> acf(corn.rai n, type="normal ")
```

```
Error in switch(i type + 1, : desi red type of ACF i s unknown
Dumped
```

With existing functions such as `acf`, most errors occur because of incorrectly specified arguments, such as nonexistent (or currently unattached) data objects, invalid choices of values (as in our choice of "normal " in the call to `acf`), or omitted required arguments. When you encounter a problem with a built-in function, then, your first debugging tool is probably the function's help file. Use the help file to be sure you have the correct calling syntax and have supplied the correct arguments.

Similarly, when you encounter a problem in a function you have newly written, the first debugging tool is the function's definition. Looking at the definition carefully can often reveal a variety of problems:

- *Misused functions.* If your function definition includes calls to unfamiliar functions, check the help files to be sure you are using those functions correctly.
- *Uninitialized variables* (often the culprit in messages such as `Cannot find object "object"`). Look for these particularly in looping constructions, because loops frequently contain assignments such as `a[i] <- value`. If `a` is initially empty you may well have forgotten to create it.
- *Inadequate input filtering.* You may have intended to allow vectors, matrices, and lists as input, but neglected to put in the code required to differentiate among the various cases. Similarly, you may have neglected to include `if` and `stop` statements to explicitly exclude certain cases.
- *Environmental dependencies.* Many functions implicitly use various settings of the S-PLUS environment. For example, graphics functions require active graphics devices and recursive functions often require deeper nesting than the default value of `options("expression")`.

A useful aid in examining your function is the `traceback` function, which lists the nested function calls currently being evaluated, starting with the function from which the error was returned and working outward to the original calling function. For the example above, `traceback` gives the following information:

```
> traceback()

Message: desired type of ACF is unknown
3: stop("desired type of ACF is unknown")
2: acf(corn.rain, type = "normal")
1:
```

Using `traceback` is a good way to focus your initial examination. You should get in the habit of typing `traceback()` whenever a function call returns an error and the Dumped message.

## Printing Intermediate Results

One of the oldest techniques for debugging, and still widely used, is to print intermediate results of computations directly to the screen. By examining intermediate results in this way, you can see if correct values are used as arguments to functions called within the top-level function.

This can be particularly useful when, for example, you are using `paste` to construct a set of elements. Suppose that you have written a function to make some data sets, with names of the form `datan`, where each data set contains some random numbers:

```
make.data.sets <-
function(n) {
  names <- paste("data", 1:n)
  for (i in 1:n)
  {
    assign(names[i], runif(100), where = 1)
  }
}
```

After writing this function, you try it:

```
> make.data.sets(5)
```

S-PLUS reports no errors, so you look for your newly created data set, `data4`:

```
> data4

Error: Object "data4" not found
```

To find out what names the function actually was creating, put a `cat` statement into `make.data.sets` after assigning names:

```
> make.data.sets
function(n)
{
  names <- paste("data", 1:n)
  cat(names, "\n ")
  for(i in 1:n)
  {
    assign(names[i], runif(100), where = 1)
  }
}
> make.data.sets(5)
```

```
data 1 data 2 data 3 data 4 data 5
```

The `cat` function prints the output in the simplest form possible; you can get more usual-looking S-PLUS output by using `print` instead:

```
> make.data.sets
function(n)
{
  names <- paste("data", 1:n)
  print(names)
  for(i in 1:n)
  {
    assign(names[i], runif(100), where = 1)
  }
}
> make.data.sets(5)
```

```
[1] "data 1" "data 2" "data 3" "data 4" "data 5"
```

The form of these names is not quite what we wanted, so we look at the `paste` help file, and discover that we need to specify the `sep` argument as `""`. We fix `make.data.sets`, but retain the call to `print` as a check:

```
> make.data.sets
function(n)
{
  names <- paste("data", 1:n, sep = "")
  print(names)
  for(i in 1:n)
  {
    assign(names[i], runif(100), where = 1)
  }
}
```

```
> make.data.sets(5)

"data1" "data2" "data3" "data4" "data5"

> data4

[1] 0.784289481 0.138882026 0.656852996 0.443559750
[5] 0.651548887 . . .
```

Now that `make.data.sets` works as we'd hoped it would, we can remove the `print` statement. (Of course, if you'd always like to see the exact names of the data sets created, you might want to leave it in.)

## INTERACTIVE DEBUGGING

Although `print` and `cat` statements can help you find many bugs, they aren't a particularly efficient way to debug functions, because you need to make your modifications in a text editor, run the function, examine the output, then return to the text editor to make further modifications. If you are examining a large number of assignments, the simple act of adding the `print` statements can become wearisome.

With the interactive debugging function `inspect` you can follow the evaluation of your function as closely as you want, from stepping through the evaluation expression-by-expression to running the function to completion, and almost any level of detail in between. While *inspecting* you can do any of the following tasks:

- *examine variables* in the function's evaluation frame. Thus, `print` and `cat` statements are unnecessary. You can also look at function definitions.
- *track* functions called by the current function. You can request that a message be printed on entry or exit, and that your own expressions be installed at those locations.
- *mark* the current expression. If the marked expression occurs again during the inspection session, evaluation halts at that point. Functions can be marked as well; evaluation will halt at the top of a marked function whenever it is called. *Marking* an expression or function corresponds to *setting a breakpoint*.
- *enter* a function; this allows you to step through a single function call, without stopping in subsequent calls to the same function.
- *examine* the current expression, together with the current calling stack. The calling stack lets you know how deeply nested the current expression is, and how you got there.
- *step* through *n* expressions or subexpressions. By default, the inspector automatically stops before each new expression or function call. You can also *do* groups of expressions, such as a braced set of expressions, or a complete conditional expression.

- *evaluate* arbitrary S-PLUS expressions. These expressions are evaluated in the local evaluation frame, so, for example, you can assign new values to objects in the local frame. In many cases, this lets you experiment with fixes to your code during the evaluation.
- *keep track* of expressions and functions that are marked or tracked, as well as expressions scheduled for evaluation on exit. You can also monitor the current function's return value.
- *complete evaluation* of the current loop or function, or resume evaluation, stopping only for marked functions or expressions.
- *look at objects* and evaluate expressions in any frame.

The following subsections describe these tasks in detail, and show how to perform them within `i nspect`.

## Starting the Inspector

To start a session with the inspector, call `i nspect` with a specific function call as an argument. For example, the call to `make.data.sets` with `n=5` resulted in a problem, so we can try to track it down by starting `i nspect` as follows:

```
> i nspect(make.data.sets(5))

entering function make.data.sets
stopped in make.data.sets (frame 3), at:
  names <- paste("data", 1:n)

d>
```

For simplicity, we call the function appearing in the argument to `i nspect` as the *function being inspected*. The `d>` prompt indicates that you are in the inspector environment. The inspector environment has a limited instruction set; the instructions are shown in Table 6.1. If you type anything at the inspector prompt other than those instructions, you get a syntax error message.

Inspector instructions are not S-PLUS function calls; do not use parentheses when issuing them. Use the `hel p` instruction to see a list of instructions; type `help instruction` for `hel p` on a particular instruction.

To leave the inspector and return to the S-PLUS prompt, use the instruction `qui t`.

## Examining Variables

You can obtain a listing of objects in the current evaluation frame with the `inspect` instruction `objects`. For example, in our call to `make.data.frames`, we obtain the following listing from `objects`:

```
d> objects

[1] ".Auto.print" ".entered." ".name." "n"
```

To examine the contents of these objects, use the `inspect` instruction `eval` followed by the object's name:

```
d> eval n

[1] 5
```

To examine a function definition, rather than a data variable, use the instruction `fundef`:

```
d> fundef make.data.sets

make.data.sets
function(n)
{
  names <- paste("data", 1:n)
  {
    for(i in 1:n)
    {
      assign(names[i], runif(100), where = 1 )
    }
  }
}
```

When you use `eval` or `fundef` to look at S-PLUS objects, you can in general just type the name of the object after the instruction, as in the examples above. Names in S-PLUS that correspond to the `inspect` function's keywords must be quoted when used as names. Thus, if you want to look at the definition of the `objects` function, you must quote the name `"objects"`, because `objects` is an `inspect` keyword. For a complete description of the quoting rules, type `help name` within an inspection session. For a complete list of the keywords, type `help keywords`.



One important question that arises in the search for bugs is “Which version of that variable is being used here?” You can answer that question using the `find` instruction. For example, consider again the examples `fcn.C` and `fcn.D` given in the section Matching Names and Values (page 805). We can use `find` inside the inspector to demonstrate that the value of `x` used by `fcn.D` is *not* the value defined in `fcn.C`:

```
> inspect(fcn.C())

entering function fcn.C
stopped in fcn.C (frame.3), at:
      x <- 3

d> track fcn.D

entry and exit tracking enabled for fcn.D

d> mark fcn.D

entry mark set for fcn.D
exit mark(s) set for fcn.D ( some or all were already set )

d> resume

entering function fcn.D
call was: fcn.D() from fcn.C (frame 3)
stopped in fcn.D (frame 4), at:
      return(x^2)

d> objects

[1] ".Auto.print" ".entered." ".name."

d> find x

.Data
```

See the section Entering, Marking, and Tracking Functions (page 190), for complete details on using the `track` and `mark` instructions.

You can inspect the value of variables in different frames by using the `up` or `down` instructions to change the frame in which `objects` looks for objects and `eval` evaluates them. For example, we could find the value 3 in `fcn.C`'s frame while in `fcn.D` as follows:

```

. . .
stopped in fcn.D , at:
    return(x^2)

d> objects

[1] ".Auto.print" ".entered." ".name."

d> up

fcn.C (frame 3)

d> objects

[1] ".Auto.print" ".entered." ".name." "x"

d> eval x

[1]
```

Table 6.1: Instructions for the interactive inspector.

Keyword	Help given
<code>help [ <i>instruction</i>   names   keywords ]</code>	Provides help on <i>instruction</i> , names, or keywords. With no arguments, <code>help</code> gives a summary of the available instructions.
<code>complete [loop   function]</code>	Evaluates to the end of the next <code>for/while/repeat</code> loop, or to the point of function return.
<code>debug.options [echo = T F] [marks = hard soft]</code>	With <code>echo=T</code> , expressions are printed before they are evaluated. With <code>marks=hard</code> , evaluation always halts at a marked expression. With <code>marks=soft</code> it halts only during a resume. Setting <code>marks=soft</code> is a way of temporarily hiding marks for <code>do</code> , <code>complete</code> , etc. The defaults are: <code>echo=F</code> , <code>marks=hard</code> . With no arguments, <code>debug.options</code> displays the current settings.
<code>do [n]</code>	Evaluates the next <i>n</i> expressions which are at the same level as the current one. The default is 1. Thus if evaluation is stopped directly ahead of a braced group, <code>do</code> does the entire group.

Table 6.1: Instructions for the interactive inspector.

Keyword	Help given
<code>down [n]</code>	Changes the local frame for instructions such as <code>objects</code> and <code>eval</code> to be <i>n</i> frames deeper than the current one. The default is 1. After any movement of the evaluator ( <code>step</code> , <code>resume</code> , etc.), the local frame at the next stop is that of the function stopped in.
<code>enter</code>	Enters the function called in the next expression.
<code>eval expr</code>	Evaluates the S-PLUS expression <i>expr</i> .
<code>find name</code>	Reports where <i>name</i> would be found by the evaluator.
<code>fundef [name]</code>	Prints the original function definition for <i>name</i> . Default is the current function. Tracked and marked functions will have modified function definitions temporarily installed; <code>fundef</code> is used to view the original. The modified and original versions will behave the same; the modified copy just incorporates tracing code.
<code>mark</code>	Remembers the current expression; evaluation will halt here from now on.
<code>mark name1 [name2 ...] [at entry exit]</code>	Arranges to stop in the named functions. The default is to stop at both entry and exit.
<code>objects</code>	Names of objects in this function's frame.
<code>on.exit</code>	Displays the current on-exit expressions for this function.
<code>quit</code>	Abandons evaluation, return to top-level prompt.
<code>resume</code>	Resumes evaluation.
<code>return.value</code>	Displays the return value, if known.
<code>show [tracks   marks   all]</code>	Displays installed tracks and marks. Default all.
<code>step [n]</code>	Evaluates the next <i>n</i> expressions. Default 1.
<code>track name1/ [name2/ ... ] [at entry exit] [print = T F] [with expr]</code>	Enables or modifies entry and/or exit tracking for the named functions. The default for print is T. You can use any S-PLUS expression as <i>expr</i> .
<code>unmark name1/ [name2 ... ] [at entry exit]</code>	Deletes mark points at the named locations in the named functions.

*Table 6.1: Instructions for the interactive inspector.*

Keyword	Help given
<code>unmark n1 [n2 ...]</code>	Deletes mark points <i>n1</i> , <i>n2</i> , .... See mark and show.
<code>unmark all</code>	Deletes all mark points.
<code>untrack name1/ [name2/ ...]</code>	Disables tracking for the named functions.
<code>up [n]</code>	Changes the local frame for instructions such as <code>objects</code> and <code>eval</code> to be <i>n</i> frames higher than the current one. The default is 1. After any movement of the evaluator ( <code>step</code> , <code>resume</code> , etc.), the local frame at the next stop is that of the function stopped in.
<code>where</code>	Displays stack of function calls, and current expression in current function.

# Controlling Evaluation

Within the inspector, you can control the granularity at which expressions are evaluated. For the finest control, use the `step` instruction, which by default, evaluates the next expression or subexpression. The inspector automatically determines stopping points before each expression. Issuing the `step` instruction once takes you to the next stopping point. To clarify these concepts, consider again our call to `make.data.sets`. You can see the current position using the `where` instruction:

```
d> where

Frame numbers and calls:

4: debug.tracer(what = TR.GENERIC, index = c(2, 1)) from 3
3: make.data.sets(5) from 1
2: inspect(make.data.sets(5)) from 1
1: from 1
-----
stopped in make.data.sets (frame 3), at:
      names <- paste("data", 1:n
```

The numbered lines in the output from `where` represent the call stack; they outline the frame hierarchy. The position is shown by the lines

```
stopped in make.data.sets (frame 3), at:
names <- paste("data", 1:n
```

If we issue the `step` instruction, we move to the next stopping point, which is right before the function call to `paste`:

```
d> step

stopped in make.data.sets (frame 3) , at:
  paste("data", 1:n)
```

Another `step` instruction completes the evaluation of the call to `paste`, and takes us to the beginning of the next expression:

```
d> step

stopped in make.data.sets (frame 3), at:
  return(for(i in 1:n)
    {      assign(names[i], runif(100), where = 1 )
    }
  ...
```

You can step over several stopping points by typing an integer after the `step` instruction. For example, you could step over the complete expression `names <- paste("data", 1:n)` with the instruction `step 2`.

You should distinguish between these automatically determined stopping points and *breakpoints*, which you insert using the `mark` instruction. Breakpoints allow you to stop evaluation at particular expressions or functions, and either step through from that point or resume evaluation until the next breakpoint is encountered. Breakpoints and marks are discussed in detail in the section *Entering, Marking, and Tracking Functions* (page 190). Another way to execute a complete expression is to use the `do` instruction. The `do` instruction has the advantage that you do not need to know how many stopping points the expression contains; `do` evaluates the entire current expression. For example, you can do the following complete expression with a single `do` instruction:

```
do {
  return(for(i in 1:n)
    {      assign(names[i], runif(100), where = 1 )
    }
  ...
}
```

The `do` instruction is particularly helpful when, as in this example, the current expression includes a loop or conditional expression. Using `step` causes the loop or conditional to be entered, and each subexpression evaluated in turn. Using `do` evaluates the entire expression atomically.

To evaluate larger pieces of the function, use the `complete` and `resume` instructions. Use `complete` to complete the current loop, if within a loop, or, if not, complete the current function. You can specify `complete loop` or `complete function` to override the default behavior. Thus, if you are within a `for` loop and type `complete function`, evaluation proceeds to the end of the current function. The inspector stops at the point after the function's last expression, before the on-exit expressions are executed. You can look at the return value and the on-exit expressions before exiting. Use the instruction `return.value` to see the return value; use the instruction `on.exit` to see the on-exit expressions.

Use `resume` to resume evaluation and proceed to the next breakpoint. If there are no further breakpoints, `resume` completes the call given to the inspector. Evaluation always stops at a breakpoint, unless you use the `debug.options` instruction to set `marks=soft`. If you specify the marks as "soft," the `do`, `step` and `complete` instructions ignore breakpoints, while `resume` stops at them as usual.

Entering,  
Marking, and  
Tracking  
Functions

By default, `inspect` lets you step through the expressions in the function being inspected. Function calls within the function being debugged are evaluated atomically. However, you can extend the step-through capability to such functions using the `enter` and `mark` instructions. You can also monitor calls to a function, without stepping through them, with the `track` instruction.

**Limitations on marking and tracking**

You cannot enter, mark, or track functions that are defined completely by a call to `.Internal`. Also, for technical reasons, you cannot enter, mark, or track any of the seven functions listed below:

`assign` `invisible` `assign.default` `on.exit` `exists` `remove` `exists.default`

## Entering Functions

If you want to step through a function in the current expression, and don't plan to step through it if it is called again, use the `enter` instruction. For example, while inspecting the call `lm(stack.loss ~ stack.x)`, you might want to step through the function `model.extract`. After stepping to the call to `model.extract`, you issue the `enter` instruction:

```
d> step

stopped in lm (frame 3), at:
  model.extract(m, weights)

d> enter

entering function model.extract
stopped in model.extract (frame 4), at:
what <- substitute(component)
```

## Marking Functions

To stop in a function each time it is called, use the `mark` instruction. For example, the `ar.burg` function makes several calls to `array`. If we want to stop in `array` while inspecting `ar.burg`, we issue the `mark` instruction and type the name of the function to be marked. By default, a breakpoint is inserted at the beginning and end of the function:

```
d> mark array

entry mark set for array exit mark(s) set for array
```

By default, each time the evaluator encounters a marked function, it stops once just after entering the function, and once just before exiting. If you want to stop only at entry or only at exit, you can use the optional `at` parameter to specify entry or exit as the desired breakpoint. For example, to stop each time `array` is entered, use `mark` as follows:

```
d> mark array at entry

entry mark set for array
```

To stop at the end of function evaluation for a function marked at entry, use `complete function` to complete the function evaluation:

```
. . .

d> where

Frame numbers and calls:
```

```
5: debug.tracer(what = TR.GENERIC, index = c(4, 1)) from 4
4: array(0, dim = c(nser, nser, order.max + 1)) from 3
3: ar.burg(lynx) from 1
2: inspect(ar.burg(lynx)) from 1
1: from 1
-----
stopped in array (frame 4), at:
      data <- as.vector(data)

d> complete function stopped in array (frame 4), at end;

      return value from: return(data) d>
```

To continue evaluation of the function being inspected, use `resume`:

```
d> resume

entering function array
stopped in array (frame 4), at:
      data <- as.vector(data)
```

## Marking the Current Expression

You can mark the current expression by giving the `mark` instruction with no arguments. This sets a breakpoint at the current expression. This can be useful, for example, if you are inspecting a function with an extensive loop inside it. If you want to stop at some expression in the loop each time the loop is evaluated, you can mark the expression. For example, consider again the `bitstring` function, defined in Chapter 5, *Writing Functions in S-PLUS*. To check the value of `n` in each iteration, you could use `mark` and `eval` together as follows. First, start the inspection by calling `bitstring`, then step to the first occurrence of the expression `i <- i + 1`. Issue the `mark` instruction, use `eval` to look at `n`, then use `resume` to resume evaluation of the loop. Each time the breakpoint is reached, evaluation stops. You can then use `eval` to check `n` again:

```
> inspect(bitstring(107))

entering function bitstring
stopped in bitstring (frame 3), at:
      string <- numeric(32)

d>

. . .
```



```
d> step

stopped in bitstring (frame 3), at:
      i <- i + 1

d> mark
d> eval n

[1] 53

d> resume

stopped in bitstring (frame 3), at:
      i <- i + 1
```

## Viewing and Removing Marks

Once you mark an expression, evaluation always stops at that expression, until you unmark it. The inspector maintains a list of marks, which you can view with the `show` instruction:

```
d> show marks
Marks: 1
: in array:
      data <- as.vector(data)
2 : in aperm:
      return(UseMethod("aperm"))
```

You can remove items from the list using the `unmark` instruction. With no arguments, `unmark` unmarks the current expression. If the current expression is not marked, you get a warning message. With one or more integer arguments, `unmark` unmarks the expressions associated with the given numbers:

```
d> show marks

Marks: 1
: in array:
      data <- as.vector(data)
2 : in aperm:
      return(UseMethod("aperm"))

d> unmark 2
```

With one or more name arguments, `unmark` unmarks the named functions:

```
d> unmark array

entry mark unset for array
```

The instruction `unmark all` unmarks all expressions.

## Tracking Functions

If you want to monitor the evaluation of a certain function, without stopping inside the function, use the `track` instruction to *track* the function. By default, a tracked function prints a message when it starts and just before it completes. As with marked functions, however, you can use the `at` parameter to specify entry or exit. You can perform more sophisticated tracking by specifying an arbitrary S-PLUS expression using the `with` parameter. For example, suppose you simply want to monitor calls to `array` inside `ar.burg`, and view the value returned by each call to `array`. You could do this by calling `track` as follows:

```
> inspect(ar.burg(lynx))

entering function ar.burg stopped
in ar.burg (frame 3), at:
      if(is.factor(x) || (is.data.frame(x) && any(
        supply(x, "is.factor"))))
        stop("cannot calculate the acf of factors")
      ...

d> track array at exit with cat("array returning",
  .ret.val., "\n ")
d> exit tracking enabled for array
d> resume

array returning 269 321 585 . . .
leaving function array
array returning 0 0 0 . . .
leaving function array
array returning 0 0 0 . . .
leaving function array
array returning 0
leaving function array
array returning 1.0877 -0.597623 0.251923
. . .
leaving function array
```

```

array returning 0 0 0 . . .
leaving function array
leaving function ar.burg . . .

```

The value `.ret.val.` is one of a number of values stored internally by `inspect`; these are named with leading periods (most have trailing periods, as well) to avoid conflicts with your own objects and standard S-PLUS objects. You can track a function giving different actions for entry and exit; this can be useful, for example, if you want to calculate the elapsed time of evaluation. To do so, you could define a function `func.entry.time` as follows:

```

func.entry.time <-
function(fun)
{
  assign("StartTime", proc.time(), frame=1)
  cat(deparse(substitute(fun)), "entered at time",
      get("StartTime", frame=1), "\n ")
}

```

Then define the exit function, `func.exit.time` as follows:

```

func.exit.time <-
function(fun)
{
  assign("StopTime", proc.time(), frame=1)
  assign("El Time", get("StopTime", frame=1) -
      get("StartTime", frame=1), frame=1)
  cat(deparse(substitute(fun)), "took time",
      get("El Time", frame=1), "\n ")
}

```

You can then track a function at entry with `func.entry.time` and track at exit with `func.exit.time`:

```

> inspect(ar.burg(lynx))

entering function ar.burg
stopped in ar.burg (frame 3), at:
  if(is.factor(x) || (is.data.frame(x) && any(sapply(x,
"is.factor"))))
    stop("cannot calculate the acf of factors")
...

d> track array at entry with func.entry.time(array)

entry tracking enabled for array

```

```
d> track array at exit with func.exit.time(array)
```

```
exit expression for array changed to:  
func.exit.time(array)
```

```
d> resume
```

```
entering function array  
array entered at time 58.5 26.85 8303 2.64 13.14  
array took time 0.5 0 1 0 0  
entering function array  
array entered at time 60.59 26.86 8306 2.64 13.14  
array took time 0.599998 0.0100002 1 0 0  
entering function array  
... ..
```

You can suppress the automatic messages entering function *fun* and leaving function *fun* by issuing the track instruction with the flag `print=F`. For example, in our previous example, our initial call to track specified tracking on entry, so only the entry message was printed. To suppress that message, simply add the flag `print=F` after the specification of entry or exit:

```
d> track array at entry print=F with func.entry.time(array)
```

## Modifying the Evaluation Frame

We have already seen one use of the `eval` instruction, to examine the objects in the current evaluation frame. More generally, you can use `eval` to evaluate *any* S-PLUS expression. In particular, you can modify values in the current evaluation frame, with those values then being used in the subsequent evaluation of the function being debugged. Thus, if you discover where your error occurs, you can modify the offending expression, evaluate it, and assign the appropriate value in the current frame. If the fix works, the complete evaluation should give the correct results. Of course, you still need to make the change (with the `fix` function) in the actual function. But using `eval` provides a useful testing tool inside the inspector. For example, once we have identified the problem in `make.data.sets` as occurring in the call to `paste`, we can go to the point at which the faulty names have been created:

```
> inspect(make.data.sets(5))
```

```
entering function make.data.sets  
stopped in make.data.sets (frame 3), at:  
  names <- paste("data", 1:n)
```

```
d> step 2
```

```
stopped in make.data.sets (frame 3), at:
  return(for(i in 1:n)
    {      assign(names[i], runif(100), where = 1 )
    }
  )
...

```

```
d> objects
```

```
[1] ". Auto. print" ". entered." ". name." "n"
[5] "names"
```

```
d> eval names
```

```
[1] "data 1" "data 2" "data 3" "data 4" "data 5"
```

Here we see that the names are not what we wanted. To test our assumption that we need the `sep=""` argument, use `eval` as follows:

```
d> eval names <- paste("data", 1:n, sep="")
d> eval names
```

```
[1] "data1" "data2" "data3" "data4" "data5"
```

Our change has given the correct names; now resume evaluation and see if the data sets are actually created:

```
d> resume

leaving function make.data.sets

> data1

[1] 0.94305062 0.61680487 0.15296083 0.25405207
[5] 0.81061184 . . .
```

## Error Actions in the Inspector

When an error occurs in the function being inspected, `inspect` calls the current `error.action`. By default, this action has three parts, as follows:

1. Produce a traceback of the sequence of function calls at the time of the error.
2. Dump the frames existing at the time of the error.
3. Start a restricted version of `inspect` that allows you to examine frames and evaluate expressions, but not proceed with further evaluation of the function being inspected.

Thus, you can examine the evaluation frame and the objects within it at the point the error occurred. You can use the `up` and `down` instructions to change frames, and the `objects`, `find`, `on.exit`, and `return.value` instructions to examine the contents of the frames. The instructions `eval`, `fundef`, `help`, and `quit` are also available in the restricted version of `inspect`. For example, consider the `primes` function described in Chapter 5, Writing Functions in S-PLUS. We can introduce an error by commenting out the line that defines the variable `smallp`:

```
primes <-
function(n = 100)
{
  n <- as.integer(abs(n))
  if(n < 2)
    return(integer(0))
  p <- 2:n
  # smallp <- integer(0)
  #
  # the sieve
  repeat
```

```

        {      i <- p[1]
              smallp <- c(smallp, i)
              p <- p[p %% i != 0]
              if(i > sqrt(n))
                break
        }
      c(smallp, p)
    }
  }
}

```

Now call `inspect` with a call to `primes`:

```

> inspect(primes())

entering function primes
stopped in primes (frame 3), at:
  n <- as.integer(abs(n))

d> do 2

stopped in primes (frame 3), ahead of loop:
  repeat
  {      i <- p[1]
        smallp <- c(smallp, i)
        ...

d> do

Error in primes(): Object "smallp" not found
Calls at time of error:

4: error = function() from 3
3: primes() from 1
2: inspect(primes()) from 1
1: from 1

Dumping frames ...
Dumped

Local frame (frame of error) is primes (frame 3)

```

A quick glance at the frame of `primes()` with `objects` shows that `smallp` is indeed not defined. Use the `quit` instruction to end the `inspect` session, then start it again. You can then use the `eval` instruction to specify an initial value for `smallp`, and watch the function complete successfully:

```

d> qui t
> i nspect(primes())

entering function primes
stopped in primes (frame 3), at:
      n <- as.integer(abs(n))

d> do 2

stopped in primes (frame 3), ahead of loop:
      repeat {
          i <- p[1]
          small p <- c(small p, i)
          ...

d> eval small p <- numeric(0)
d> resume

leaving function primes
[1]  2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
[18] 61 67 71 73 79 83 89 97

```

You can then edit the `primes` function to fix the error.

#### Limitations of `inspect`

- Functions defined within other functions or in function calls or argument default expressions cannot be tracked. They should work, though. Also, avoid assigning functions on frame 1, especially if you want to track them.
- Complex expressions inside `if`, `while`, and other *conditions* are never tracked. If you want to track them, assign them to a name outside the test condition and use the name inside the condition.
- Do not use `trace` if you plan to use `inspect`. The `trace` function creates a modified version of the function being traced, as does `inspect`. The modifications may not be completely compatible.
- Do not try to edit functions (using S-PLUS functions such as `fix`) while running `inspect`. In particular, do not edit functions that you are tracking, have marked, or have entered and not yet exited.
- Avoid using `inspect` on functions involving calls to `Recall`.



**Limitations of** `Inspect`

- You will see some extra frames and objects in the inspection mode that are not there in normal evaluation. These objects have names which are unlikely to conflict with those of the functions being inspected.

## OTHER DEBUGGING TOOLS

The `inspect` function provides a complete interactive debugging environment, and we recommend it for all your normal S-PLUS debugging needs. On occasion, however, you may find some of S-PLUS's other debugging tools of some use. This section briefly describes these other tools—`browser`, `debugger`, and `trace`.

### Using the S-PLUS Browser Function

The `browser` function is useful for debugging functions when you know an error occurs *after* some point in the function. If you insert a call to `browser` into your function at that point, you can check all assignments up to that point, and verify that they are indeed the correct ones. For example, to return to our `make.data.sets` example, we could have replaced our original `cat` statement with a call to `browser`:

```
make.data.sets <-
function(n)
{
    names <- paste("data", 1:n)
    browser()
    for(i in 1:n)
    {
        assign(names[i], runif(100), where = 1)
    }
}
```

When we call `make.data.sets`, we get a new prompt `b(2)>` to indicate we are in the browser, and a message telling us which function the browser was called from:

```
> make.data.sets(5)

Called from: make.data.sets(5)

b(2)>
```

Type `?` at the prompt to get a menu of the objects in the frame:

```
b(2)> ?

1: names
2: n
```

```
b(2)> names
```

```
[1] "data 1" "data 2" "data 3" "data 4" "data 5"
```

To leave the browser, type 0 at the prompt:

```
b(2)> 0
```

```
>
```

You can type arbitrary S-PLUS expressions at the browser prompt. These expressions are evaluated in the chosen frame, which is indicated by the number in parentheses within the prompt—thus, `b(2)>` indicates that you are in browser in frame 2. Thus, you can type alternative expressions and see if a possible fix will actually work.

## Using the S-PLUS Debugger

If a function is *broken*, so that it returns an error reliably when called, there is an alternative to all those `cat` and `browser` statements: the debugger function. To use debugger on a function, you must have the function's list of frames dumped to disk. You can do this in several ways:

- Call `dump.frames()` from within the function.
- Call `dump.frames()` from the browser.
- Set `options(error=expression(dump.frames()))` If you use this option, you should reset it to the default (`expression(dump.calls())`) when you are finished debugging, because dumped frames can be quite large.

Then, when an error occurs, you can call the debugger function with no arguments, which in turn uses the `browser` function to let you browse through the dumped frames of the broken function. Use the usual browser commands (`?` and frame numbers) to move through the dumped frames.

## Tracing Function Evaluation

Another way to use the `browser` function is with the `trace` function, which modifies a specified function so that some tracing action is taken whenever that function is called. You can specify that the action be to call the `browser` function (with the statement `tracer = browser`) providing yet another way to track down bugs.

**Warning:** `trace` and `inspect` clash

Do *not* use `trace` on *any* function if you intend to do your debugging with `inspect`.

For example, suppose we wanted to trace our `make.data.sets` function:

```
> trace(make.data.sets, browser)
> make.data.sets
function(n) {
  if(.Traceon)
  {
    .Internal(assign(".Traceon", F, where = 0),
               "S_put")
    cat("On entry: ")
    browser()
    .Internal(assign(".Traceon", T, where = 0),
               "S_put")
  } else
  {
    names <- paste("data", 1:n)
    for(i in 1:n)
    {
      assign(names[i], runif(100), where = 1)
    }
  }
}
```

The `trace` function copies an edited version of the traced function into the session frame, and maintains a list of all functions which are currently being traced. Since S-PLUS finds objects in the session frame before looking in directories, do *not* try to edit a function that is currently being traced. If, for instance, you call `fix(make.data.sets)` while `make.data.sets` is being traced, you overwrite the copy of `make.data.sets` in your working directory with the edited version, which contains several calls to `.Internal`. The additions include the call to the *tracer*, in this case `browser`. The object `.Traceon` specifies whether tracing is enabled; you can change this value with the `trace.on` function.

If we now call `make.data.sets`, we find ourselves in the browser, in the `make.data.sets` frame:

```
> make.data.sets(3)
On entry: Called from: make.data.sets(3)
b(2)> ?
1: n
b(2)>
```

However, `trace`, by default, puts the call to browser at the *beginning* of the function, so that we actually see less information in the browser than we hoped; in particular, we don't see the value of `names`. We can, however, run the expression to create the names:

```
b(2)> names <- paste("data", 1:n)
b(2)> names
[1] "data 1" "data 2" "data 3"
```

From this, we discover, as before, that our `paste` expression needs modification, and as before we can test our proposed change before implementing it. After leaving browser, type `untrace(make.data.sets)` to remove the traced function from the list.

If we had wanted the call to browser *after* the names assignment, we could have used the `at` argument to `trace`:

```
> trace(make.data.sets, browser, at=2)
> make.data.sets
function(n) {
  names <- paste("data", 1:n)
  { if(.Traceon)
    { .Internal(assign(".Traceon", F,
                      where = 0), "S_put")
      cat("At 2: ")
      browser()
      .Internal(assign(".Traceon", T,
                      where = 0), "S_put")
    }
    for(i in 1:n)
    { assign(names[i], runif(100), where = 1)
    }
  }
}
```

Now if we call `make.data.sets`, our browser session looks much like the one in the previous section:

```
> make.data.sets(3)
At 2: Called from: make.data.sets(3)
b(2)> ?
1: names
2: n
b(2)>
```

# EDITABLE GRAPHICS COMMANDS

# 7

---

<b>Getting Started with Editable Graphics</b>	<b>209</b>
<b>Graphics Objects</b>	<b>211</b>
Graph Sheets	211
Graphs	211
Axes	212
Plots	212
Annotation Objects	212
<b>Graphics Commands</b>	<b>213</b>
Getting Started With Simple Plots	213
Specifying Data for Plots	214
Setting Colors and Other Display Properties	215
Listing the Argument Names for an Object	216
<b>Multiple Plots on a Page</b>	<b>217</b>
Object Path Names	217
Combining Plots on a Graph	218
Laying Out Multiple Graphs on a Page	218
Laying Out Graphs on Multiple Pages	219
Conditioned Trellis Graphs	219
<b>Titles and Annotations</b>	<b>220</b>
<b>Plot Types</b>	<b>223</b>
2D Plots	223
3D Plot Palette	229
<b>Formatting Axes and Layout</b>	<b>232</b>
2D Plots	232
3D Plots	232
Creating Specialized Graphs Using Your Own Computations	233
Locating Positions on Your Graph	234

All of the graphics created using plot palettes can also be created via the Commands window or a Script window. And just as these graphs can be modified by using the toolbar and dialogs, there are corresponding function

calls that can make the equivalent modifications. Each field in a property dialog corresponds to a property of the object. Similarly, toolbar actions such as changing colors or line styles are changes in values of an object's properties. Every time a graph is created or modified the change is recorded in the History log. This provides an easy way to generate programming examples for yourself.

**Note**

The graphics produced by using the Statistics menus and dialogs are *traditional* graphics, see chapter 8 for details of these.



## GETTING STARTED WITH EDITABLE GRAPHICS

The `guiPlot` function emulates the action of interactively creating plots by selecting columns of data and clicking on a plot button on a plot palette. For example, the following command re-creates the familiar example of selecting the `Mileage` and `Weight` columns from the `fuel.frame` data set:

```
guiPlot("Scatter", DataSet = "fuel.frame", Columns =
  "Mileage, Weight")
```

A scatter plot of `Weight` versus `Mileage` is created. The plot type specified in `guiPlot` is the same as the name of the plot button, as displayed in its Tool Tip. The colors, symbol types, and line styles used by `guiPlot` are those specified in `Options/Graph Styles`, just as they are when you click on a plot button.

You can set the axes type in `guiPlot`, just as you can on the main toolbar before clicking on a plot button. For example, the following call creates a graph with a log Y axis:

```
guiPlot("Scatter", DataSet = "fuel.frame",
  Columns="Mileage, Weight", AxisType = "Log Y" )
```

Similarly, you can create a graph with two plots, one showing `Weight` versus `Mileage` and the other showing `Disp. Versus Mileage`, with the second plot scaled to a right Y axis, as follow:

```
guiPlot("Scatter", DataSet = "fuel.frame", Columns="Mileage,
  Weight, Disp.", AxisType = "Multiple Y")
```

Alternatively, the following call puts the plots in two separate panels, with the same X axis scaling but different Y axis scaling in each panel:

```
guiPlot("Scatter", DataSet = "fuel.frame",
  Columns="Mileage, Weight, Disp.",
  AxisType = "Vary Y Panel")
```

Trellis conditioning plots can also be created easily using `guiPlot`:

```
guiPlot("Scatter", DataSet = "fuel.frame", Columns =
  "Mileage, Weight, Type", NumConditioningVars = 1)
```

This creates a plot of `Weight` versus `Mileage`, with a separate panel for each type of car, just as if you had selected the three columns of data, pressed the conditioning mode button on the main toolbar, and clicked on the scatter plot button. The last column specified is always used as the conditioning variable.

The `gui Plot` function is written into the condensed history log when you create a graph interactively. You can create your own examples of `gui Plot` by creating the desired plot type and then looking at the history log.

# GRAPHICS OBJECTS

There are five main types of graphics objects: graph sheets, graphs, axes, plots, and annotation objects. Plots are contained in graphs which are contained in graph sheets. Most graphics objects cannot exist in isolation; if a graphics object is created in isolation, it will create an appropriate container. For example, when you create a plot, the appropriate graph, axes and graph sheet are automatically created. In general, the simplest way to do this is with the `gui Plot` function. In addition, each object can be created using the `gui Create` function, and the properties of each object can be modified using `gui Modify`.

## Graph Sheets

Graph sheets are the highest level graphics object. Graph sheets are documents that can be saved, opened, and exported. Graph sheet properties determine the orientation and shape of the graph sheet, the units used, the default layout to use when new graphs are added to the graph sheet, and the user-defined colors available for other objects to use. Graph sheets typically contain one or more graphs, and perhaps annotation objects such as text, line segments, arrows, or extra symbols.

## Graphs

There are six types of graphs: 2D, 3D, Polar, Matrix, Smith, and Text. The graph type determines the type of coordinate systems used within the graph. A 2D graph can have one or more 2D coordinate systems, each composed of an X and Y axis. A 3D graph has a single 3D coordinate system defined by a 3D axes object, a Polar graph has a single polar coordinate system, and a Matrix graph a single set of 2D coordinate systems drawn in a matrix layout. Smith plots are specialized graphs used in microwave engineering; they also have a single coordinate system. Text graphs have no coordinate systems other than the measurements of the graph sheet. They are used for pie charts.

Graph properties determine the size and shape of the graph area and the plot area. These areas can be filled with colors and have borders around them. All graphs allow the Trellis paradigm of having multiple panels. The graph properties determine the data to use for conditioning and the layout of the panels. 3D graphs also have properties that determine the shape and the view angle of the 3D workbox.

**Axes**

The characteristics of the coordinate systems within the graphs are set by the properties of the axes objects. Typically axis properties contain information about the axis range, the tick positions, and display characteristics such as line color and line weight. 2D axes also have properties that determine scaling and axis breaks. All axes other than 2D axes also contain information about tick labels and axis titles. 2D axes contain separate tick label and axis title objects, which have their own properties.

**Plots**

A plot contains data specifications and options related to how the plot is displayed, and in many cases, the type of calculations done to the data before drawing the plot. A plot is always contained within a graph, and is associated with a coordinate system.

**Annotation  
Objects**

Annotation objects can be placed directly on a graph sheet, or put within a graph. If they are contained within a graph, they will be repositioned as the graph is repositioned on the page. Annotation objects contain information about how they will be drawn (for example, line color and line style), and their position on the graph or graph sheet. Their units can be either document units, determined by the graph sheet, or axes units, determined by an axes coordinate system.

# GRAPHICS COMMANDS

This section describes the programming interface to the editable graphics system. For details on the plot types and plot properties, see the online help.

## Getting Started With Simple Plots

There are a large number of plot types, as shown on the plot palettes. These plot types are organized into a number of plot classes. All of the types within a plot class share a set of plot *properties*. Properties are components of graphics objects describing some aspect of the object, such as the line style to use. Variations on the basic plot types can be created by setting the appropriate properties. When creating a plot, properties are specified by name as arguments to the `gui Modify` or the `gui Create` function.

For example, `Line`, `Scatter`, and `Line Scatter` plots are all members of the plot class `LinePlot`. To create a line plot with symbols using all of the default values:

```
guiPlot("Line Scatter", DataSet="fuel.frame", Columns=
      "Weight, Mileage")
```

You can create the same plot with `gui Create` as follows:

```
guiCreate("LinePlot", DataSet="fuel.frame",
      xColumn="Weight", yColumn="Mileage")
```

The arguments `DataSet`, `xColumn`, and `yColumn` are properties of a `LinePlot` object. They are represented by the first three entries on the `Data to Plot` page of the `LinePlot` properties dialog.

You can create a scatter plot easily with either `guiPlot` or `gui Create`:

```
guiPlot("Scatter", DataSet="fuel.frame", Columns=
      "Weight, Mileage")
guiCreate("LinePlot", DataSet="fuel.frame",
      xColumn="Weight", yColumn="Mileage",
      LineStyle = "None")
```

Similarly, you can create a line plot without symbols:

```
guiPlot("Line", DataSet="fuel.frame", Columns=
      "Weight, Mileage")
guiCreate("LinePlot", DataSet="fuel.frame",
      xColumn="Weight", yColumn="Mileage",
      SymbolStyle="None")
```

In each case above, a new graph sheet is created with a 2D graph and a set of X and Y axes. The plot is placed within the graph.

## Specifying Data for Plots

You can specify data for plots either by name or by value. First, the names referring to the data can be used. In this case the plot is live; a plot updates when opened or brought into focus if the values in the data set are changed. The data names are always put in quotes. For `gui Plot`, you specify data by name using the `DataSet` and `Columns` arguments. For `gui Create` and `gui Modify`, you specify data by name using the properties `DataSet`, `xColumn`, `yColumn`, `zColumn` and `wColumn`.

Alternatively a plot can store the data internally. The expression used to specify the data is evaluated when the plot is created, and will not be updated thereafter. To specify the data values to be used permanently within the plot, for `gui Plot` use the argument `DataSetValues`; for `gui Create` and `gui Modify`, use the properties `DataSetValues`, `xValues`, `yValues`, `zValues`, and `wValues`.

For example, to create a scatter plot that stores a copy of the data internally in the graph sheet, use `gui Plot` as follows:

```
gui Plot("Scatter", DataSetValues=fuel.frame[, c("Weight",  
        "Mileage")])
```

If you are generating the plot from within a function you will want to pass the data by value if the data set was constructed within the function and will no longer exist upon termination of the function.

## Setting Colors and Other Display Properties

There are a number of display properties commonly used in plots and annotation objects.

*Table 7.1: Common display properties.*

Property	Settings
LineColor	"Transparent", "Black", "Blue", "Green", "Cyan", "Red", "Magenta", "Brown", "Lt Gray", "Dark Gray", "Lt Blue", "Lt Green", "Lt Cyan", "Lt Red", "Lt Magenta", "Yellow", "Bright White", "User1", ... "User16"
LineStyle	"None", "Solid", "Dots", "Dot Dash", "Short Dash", "Long Dash", "Dot Dot Dash", "Alt Dash", "Med Dash", "Tiny Dash"
LineWeight	point size
Symbol Color	See LineColor
Symbol Style	0,1, 2, ..., 27 "None"; "Circle, Solid"; "Circle, Empty"; "Box, Solid"; "Box, Empty"; "Triangle, Up, Solid"; "Triangle, Dn, Solid"; "Triangle, Up, Empty"; "Triangle, Dn, Empty"; "Diamond, Solid"; "Diamond, Empty"; "Plus"; "Cross"; "Ant"; "X"; "-"; " " ; "Box X"; "Plus X"; "Diamond X"; "Circle X"; "Box +"; "Diamond +"; "Circle +"; "Tri. Up Down"; "Tri. Up Box"; "Female"; "Male"
Symbol Size	point size

For example,

```
gui Modify("LinePlot", Name=gui GetPlotName(),
          LineColor="Red", LineStyle="Short Dash",
          LineWeight=2, Symbol Color="Lt Red",
          Symbol Style="Circle, Solid")
```

You can use the `gui DisplayDialog` function to get a property dialog after the object has been created. The properties for the plot may be modified using the dialog.

```
gui DisplayDialog("LinePlot", Name=gui GetPlotName())
```

This displays the dialog for the first plot of the first graph in the current graph sheet. Names are described in detail in a later section.

## Listing the Argument Names for an Object

All of the argument names that can be used in `gui Modify` or `gui Create` can be listed by using the `gui GetArgumentNames` function. It takes as its argument the class name for the graphics object.

```
gui GetArgumentNames("Li nePl ot")
```

Each argument has a corresponding field in the property dialog for the object. These fields are documented in detail in the online help.

Similarly, you can use the `gui Pri ntCl ass` function to list all of the arguments, the corresponding dialog prompt, the current default value, and any options available, if appropriate. For example, to see the arguments for `Li nePl ot` objects:

```
gui Pri ntCl ass("Li nePl ot")
```

A useful approach to determining what property to set and what value to use is to modify the desired plot and then look in the History Log to see what scripting commands correspond to modifying the object. (By default, the History Log shows a condensed version of the commands; properties which are left at their default values are not shown. You can see all the properties, with their default values, if you choose History Type "Full" in the Undo and History dialog under the Options menu. The property names are fairly descriptive and tend to correspond to the control labels in the dialog. Match the property name in the script to the control name in the dialog, and then look in the online help for a detailed description of the property.



---

## MULTIPLE PLOTS ON A PAGE

### Object Path Names

Every graph object has a path name that identifies the object. This is set by the property `Name`. A plot path name has three components. The first is the name of the graph sheet (preceded by `$$`), then the graph name or number, followed by the plot name or number. The components of the name are separated by `$`. For example, the name "`$$GS1$1$1`" refers to the first plot in the first graph of the graph sheet named `GS1`. If the graph sheet name is omitted, the current graph sheet is used. The current graph sheet is the graph sheet that was most recently created, modified, or viewed.

You can use the following functions to get the object name for a specific object type. Most of them take a graph sheet name and a `GraphNum` argument; you can use `gui GetGSName` to obtain the name of the current graph sheet:

- `gui GetAxisLabel sName`: returns the name of the `AxisLabels` for a specified axis (axis 1 by default).
- `gui GetAxis sName`: returns the name of the axis for a specified axis (axis 1 by default).
- `gui GetAxis sTitleName`: returns the name of the axis title for a specified axis (axis 1 by default).
- `gui GetGSName`: returns the name of the current graph sheet. (This function takes no arguments.)
- `gui GetGraphName`: returns the `GraphName` of the graph with the specified `GraphNum` in the specified graph sheet.
- `gui GetPlotName`: returns the `Name` of the plot with the specified `PlotNum` in the specified `GraphNum` in the specified graph sheet.

Because annotation objects such as extra text, lines, or symbols, can be placed directly on a graph sheet or in a graph, it is necessary to include the graph sheet name in the path when creating those objects.

## Combining Plots on a Graph

Multiple plots can be combined on a single graph by using a name denoting where the plot is to be contained.

First, create a new graph sheet with a new plot:

```
gsName <- gui Plot("Li ne Scatter", DataSet="car. mi l es",  
  Col umns="car. mi l es")
```

Now add a second plot of a different type, putting it in the first graph:

```
gui Plot("Li ne", DataSet="car. gal s",  
  Col umns="car. gal s", GraphSheet=gsName, Graph=1)
```

Since the data ranges are quite different for the two plots, let's put them in separate panels.

```
gui Modi fy("Graph2D", Name = gui GetGraphName(),  
  Panel Type = "By Pl ot")
```

and have the Y axis range vary across panels:

```
gui Modi fy("Axi s2dY", Name = gui GetAxi sName(),  
  VaryAxi sRange=T)
```

## Laying Out Multiple Graphs on a Page

Graph sheets can automatically resize and reposition the existing graphs on the page when a new graph is added. The layout parameters are properties of the graph sheet.

For example, create a new graph sheet with a line plot:

```
gsName <- gui Plot("Li ne", DataSet="car. mi l es",  
  Col umns="car. mi l es")
```

Now set the property of the graph sheet to stack all of the graphs one on top of each other, with one graph across each row. In other words, there will be one graph in each row.

```
gui Modi fy("GraphSheet", AutoArrange = "One Across")
```

Now add another graph to the graph sheet.

```
gui Plot("Li ne", DataSet="car. gal s", Col umns="car. gal s",  
  GraphSheet=gsName)
```

The first graph will be moved and resized to fit the top half of the page. The second graph will appear on the lower half of the page.

## Laying Out Graphs on Multiple Pages

Graphs can also be placed on different pages of the same graph sheet. For example, create a new graph sheet with a line plot:

```
gsName <- guiPlot("Line", DataSet="car.miles",  
  Columns="car.miles")
```

Now add a second graph on a second page:

```
guiPlot("Line", DataSet="car.gals", Columns="car.gals",  
  GraphSheet=gsName, Page="New")
```

## Conditioned Trellis Graphs

In conditioned Trellis graphs, conditioning is specified as part of the graph object. The specified conditioning variable or variables are used for all plots contained within the graph.

For example, to create a scatter plot with conditioning, do the following:

```
guiPlot("Scatter", DataSet="fuel.frame", Columns=  
  "Weight, Mileage, Type", NumConditioningVars=1)
```

Then modify the graph containing the plot to fine tune the Trellis layout.

```
guiModify("Graph2D", Name="1", NumberOfPages=3,  
  NumberOfPanelColumns=2)
```

The panels are now laid out two to a page over three pages.

## TITLES AND ANNOTATIONS

All graphs can contain titles and subtitles. Similarly, all 2D axes contain axis titles. To create a graph, and add a title, subtitle, and axes titles.

```
guiPlot("Scatter", DataSetValues=data.frame(car.miles,
car.gals))
guiCreate("MainTitle", Name = "1", Title = "Mileage Data")
guiCreate("Subtitle", Name = "1",
Title = "Miles versus Gallons")
guiCreate("YAxisTitle", Name = "1",
Title = "Miles per Trip")
guiCreate("XAxisTitle", Name = "1",
Title= "Gallons per Trip")
```

For a 3D graph using internal data:

```
xData <- 1:25
guiPlot("3D Line", DataSetValues=data.frame(xData,
cos(xData), sin(xData)))
guiModify("Axes3D", Name="1", xTitleText="x",
yTitleText="cos(x)", zTitleText="sin(x)")
```

### Legends

All graphs can also contain legends:

```
guiCreate("Legend", Name = "1$1$1", xPosition = 6.6,
yPosition = 6.1, Hide = F)
```

This legend can be hidden and unhidden using the Auto Legend button on the graph sheet toolbar.

Legends contain legend items that can be modified individually. To add text to the legend item:

```
guiModify("LegendItem", Name = "1$1$1",
Text = "Miles per Trip")
```

### Other Annotations

Because annotation objects such as extra text, lines, or symbols, can be placed directly on a graph sheet or in a graph, it is necessary to include the graph sheet name in the path when creating those objects. For example, to put a date stamp centered on the bottom of your graph sheet, first create the appropriate path name for the date stamp:

```
gsObjName <- paste("$", guiGetGSName(), "$1", sep = "")
```

If the new graph sheet is named, for example, GS2, then the comment Name will be \$\$GS2\$1. It will be positioned using the document units of the graph sheet, measured from the lower left-hand corner. If the position is set to "Auto", it will be centered.

```
gui Create("CommentDate", Name = gsObj Name,
          Title = "My Project", xPosition = "Auto",
          yPosition = 0.1, UseDate = T, UseTime = F)
```

Now put a box with rounded edges outside of the axes. This box will be contained by the graph sheet.

```
gui Create("Box", Name = gsObj Name, OriginX = .5,
          OriginY = .5, SizeX = 10, SizeY = 7.5,
          RoundCorners = T, UseAxesUnits = F)
```

Create an arrow on the graph, with the location specified in axes units. First create the appropriate path name, one level deeper than that used for the Box above.

```
obj Name <- paste( gsObj Name, "$1", sep = "" )
gui Create("Arrow", Name = obj Name, xStart = 44,
          yStart = 380, xEnd = 75, yEnd = 340, UseAxesUnits = T)
```

Similarly, other annotations such as extra symbols and lines can be added.

```
gui Create( "Symbol ", Name = obj Name,
          Symbol Style = "Box, Empty", xPosition = 78,
          yPosition = 334, Symbol Color = "Red",
          Symbol Height = .25,
          UseAxesUnits = T)
gui Create("Line", Name = obj Name, LineStyle = "Short Dash",
          xStart = 5, yStart = 334, xEnd = 90, yEnd = 334,
          UseAxesUnits=T)
```

Add a horizontal reference line at the mean of the data:

```
gui Create("ReferenceLine", Name = obj Name,
          LineColor = "Black", LineStyle="Long Dash",
          Orientation = "Horizontal ",
          Position = mean(car.miles))
```

Add an error bar showing the standard deviation of the data:

```
stddevy <- sqrt(var(car.miles))
gui Create("ErrorBar", Name = obj Name, xPosition = 110,
          yPosition = mean(car.miles), xMin = 0, yMin = stddevy,
          xMax = 0, yMax = stddevy, UseAxesUnits = T)
```

Other annotation objects, such as ellipses, radial lines, and arcs, can be used for specialize drawing. The following will create a new graph sheet and add annotation objects to it:

```
gui Create("Ellipse", FillColor = "Transparent",
          xCenter = 5.5, yCenter = 3.5, HorizontalRadius = 2.7,
          VerticalRadius = 3)
```

```
gui Create("Arc", Name = "1", xCenter = 5.5, yCenter = 2.0,
          HorizontalRadius = 1.2, VerticalRadius = .7,
          StartAngle = 180, EndAngle = 0)
```

```
gui Create("Radius", Name = "1", xCenter = 5.5,
          yCenter = 3.3, xStart = 3.0, xEnd = 3.5, Angle = 75)
```

```
gui Create("Radius", Name = "2", xCenter = 5.5,
          yCenter = 3.3, xStart = 3.0, xEnd = 3.5, Angle = 90)
```

```
gui Create("Radius", Name = "3", xCenter = 5.5,
          yCenter = 3.3, xStart = 3.0, xEnd = 3.5, Angle = 105)
```

# PLOT TYPES

S-PLUS has a wide variety of editable graphics plot types. The easiest way to determine what class corresponds to a particular editable graph is to generate the graph using the point-and-click interface, and then look at the History log to see what commands are used to generate the plot. To get a vector listing all of the GUI classes, including plot classes, you can use the function `gui GetClassNames`, which takes no arguments. To get a list of all the plot types that can be used in `gui Plot`, use `gui GetPlotClass`, which also takes no arguments.

In this section we present commands corresponding to the plot types on the 2D and 3D plot palettes. Commands corresponding to axes and layout operations are presented in a later section.

## 2D Plots

### 2D Line and Scatter Plots

The `LinePlot` class includes various kinds of line and scatter plots. You can determine the plot type to use for each plot either by hovering over the 2D plot palette and viewing the Tool Tips, or by using `gui PrintClass` to view the `PlotType` property of the `LinePlot` class.

#### Scatter

```
gui Plot("Scatter", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

#### Line

```
gui Plot("Line", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

#### Line Scatter

```
gui Plot("Line & Scatter", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

#### Isolated Points

```
gui Plot("Isolated Points", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

**High Density**

```
guiPlot("High Density", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

**Text as Symbols**

```
guiPlot("Text as Symbols", DataSetValues =  
  data.frame(util.mktbook, util.earn, 1:45))
```

**Bubble**

```
guiPlot("Bubble", DataSetValues =  
  data.frame(util.mktbook, util.earn, 1:45))
```

**Color Plot**

```
guiPlot("Color", DataSetValues =  
  data.frame(util.mktbook, util.earn, 1:45))
```

**Bubble Color**

```
guiPlot("BubbleColor", DataSetValues =  
  data.frame(util.mktbook, util.earn, 45:1, 1:45))
```

**Loess**

```
guiPlot("Loess", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

**Smoothing Spline**

```
guiPlot("Spline", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```

**Robust**

```
guiPlot("Robust LTS", DataSetValues =  
  data.frame(util.mktbook, util.earn))
```



**Dot Plot**

```
wt.means <- tapply(fuel.frame$Mileage, fuel.frame$Type,
                  mean)

guiPlot("Dot", DataSetValues= data.frame(wt.means,
                                           as.factor(names(wt.means))))
```

**Y Series Lines**

```
guiPlot("Y Series Lines", DataSet="hstart",
        Columns="hstart")
```

**Linear Curve Fit Plots**

The least-squares, polynomial, and exponential and log fit plots are all types of `LinearCF` plots. The plot types are "Linear Fit", "Poly Fit", "Exp Fit", "Power Fit", "Ln Fit", and "Log Fit".

**Linear Fit**

```
guiPlot("Linear Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

**Polynomial Fit**

```
guiPlot("Poly Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

**Exponential Fit**

```
guiPlot("Exp Fit", DataSetValues =
        data.frame(util.mktbook, util.earn))
```

**QQ Plots**

Quantile-Quantile plots comparing two samples or comparing one sample with a reference distribution are made by creating `QQPlot` objects. The QQ plot type compares one sample with (by default) a normal distribution and two samples with each other; the "QQ Normal" plot type compares one or more samples with the normal distribution.

**QQ Normal Plot**

```
guiPlot("QQ Normal", DataSetValues=data.frame(rnorm(25),
                                                runif(25)))
```

**QQ Plot**

```
guiPlot("QQ", DataSetValues=data.frame(rchisq(20, 5),  
rnorm(20)))
```

**Box Plots**

Box plots can be created for a single variable or a grouped variable.

**Box Plot**

```
guiPlot("Box", DataSetValues= util.earn)  
  
guiPlot("Box", DataSet="fuel.frame", Columns=  
"Type, Mileage")
```

**Pie Charts**

To create a pie chart specify the relative pie slice angles.

**Pie Chart**

```
guiPlot("Pie", DataSetValues= c(2, 4, 3))
```

**Histograms and  
Densities**

Histograms and density plots are created by making an object of class Histogram.

**Histogram**

```
guiPlot("Histogram", DataSetValues=util.earn)
```

**Density**

```
guiPlot("Density", DataSetValues=util.earn)
```

**Histogram/Density**

```
guiPlot("Histogram Density", DataSetValues=util.earn)
```

**Bar Plots**

The 2D Plots palette offers a variety of bar plots. For most ordinary comparisons, we recommend the bar plot with zero base.

**Bar Zero Base**

```
guiPlot("Bar Zero Base", DataSetValues =  
data.frame(as.factor(c("A", "B")), c(-20, 70)))
```

**Bar Y Min Base**

```
guiPlot("Bar Y Min Base", DataSetValues =
      data.frame(as.factor(c("A", "B")), c(-20, 70)))
```

**Grouped Bar**

```
guiPlot("Grouped Bar", DataSetValues =
      data.sheet(as.factor(c("A", "B")),
      c(20, 70, 30, 80)))
```

**Stacked Bar**

```
guiPlot("Stacked Bar", DataSetValues =
      data.sheet(as.factor(c("A", "B")),
      c(20, 70, 30, 80)))
```

**Bar w/Error**

```
guiPlot("Bar with Error", DataSetValues =
      data.frame(as.factor(c("A", "B")), c(20, 70), c(3, 6)))
```

**Grouped Bar w/Error**

```
guiPlot("Grouped Bar with Error", DataSetValues =
      data.sheet(as.factor(c("A", "B")),
      c(20, 70, 30, 80), c(3, 3, 6, 6)))
```

**Horizontal Bar**

```
guiPlot("Horizontal Bar", DataSetValues =
      data.frame(c(20, 70), as.factor(c("A", "B"))))
```

**Grouped Horizontal Bar**

```
guiPlot("Grouped Horizontal Bar", DataSetValues =
      data.sheet(c(20, 70, 30, 80),
      as.factor(c("A", "B"))))
```

**Stacked Horizontal Bar**

```
guiPlot("Stacked Horizontal Bar", DataSetValues =
      data.sheet(c(20, 70, 30, 80),
      as.factor(c("A", "B"))))
```

**High-Low Plots**

For a high-low plot specify the low and high values as `wValues` and `zValues`, respectively. Specify other quantities of interest, such as the daily open and close or the average value of the variable, using `yValues`:

**High-Low-Open-Close**

```
data.restore(  
  "C:\\Program Files\\sp2000\\samples\\exdjia87.sdd" )  
guiPlot( PlotType = "High Low", DataSet = "exdjia87",  
  Columns = "date, open, close, high, low")
```

**Error Bar Plots**

For an error bar plot specify the location of each point and the length of error bars about each point.

**Vertical Error Bar**

```
guiPlot("Error Bar", DataSetValues=data.frame(  
  as.factor(c("A", "B")), c(20, 70), c(3, 6)))
```

**Horizontal Error Bar**

```
guiPlot("Horizontal Error Bar", DataSetValues=data.frame(  
  c(20, 70), as.factor(c("A", "B")), c(3, 6)))
```

**Vertical and Horizontal Error Bars**

```
guiPlot("Error Bar - Both", DataSetValues=data.frame(  
  c(20, 43), c(20, 70), c(3, 6), c(5, 8)))
```

**Area Plots**

An area plot fills the area between points and the axis with color.

**Area**

```
guiPlot("Area", DataSetValues=data.frame(car.time,  
  car.gals))
```

**Scatter Plot  
Matrices**

A scatter plot matrix displays all pairwise plots of variables in a data set.

**Scatter Plot Matrix**

```
guiPlot("Scatter Matrix", DataSet="fuel.frame",  
  Columns = "Mileage, Weight, Type")
```

**Contour Plots**

Contour plots, filled contour plots, and levels plots are all cases of `ContourPlot`.

### Contour

```
guiPlot("Contour", DataSet="ethanol", Columns="C, E, NOx")
```

### Filled Contour

```
guiPlot("Filled Contour", DataSet="ethanol",  
        Columns="C, E, NOx")
```

### Levels Plot

```
guiPlot("Levels", DataSet="ethanol", Columns="C, E, NOx")
```

## 3D Plot Palette

For a 3D graph we need three variables, in this case from the **ozone** data set.

```
x <- ozone.xy$x  
y <- ozone.xy$y  
z <- ozone.median  
ozone.df <- data.frame(x, y, z)
```

First we will create a mesh surface plot.

```
guiPlot("Data Grid Surface", DataSetValues=ozone.df)
```

We can add the data points to the fitted surface.

```
guiCreate("Line3DPlot", Name="1$2", xValues=x, yValues=y,  
          zValues=z, SymbolStyle="Circle, Solid")
```

We can use `guiModify` to change the graph to use rotated 3D axes.

```
guiModify("Graph3D", Name="1", Rotate3Daxes=T)
```

If you would like to see the graph again without the 3D line plot, you can use the `guiRemove` function to remove the second plot.

```
guiRemove("Line3DPlot", Name="1$2")
```

## 3D Line and Scatter Plots

Line, scatter, drop-line, and regression plots are all of type `Line3DPlot`.

### Scatter

```
guiPlot("3D Scatter", DataSetValues=ozone.df)
```

**Line**

```
guiPlot("3D Line", DataSetValues=ozone.df)
```

**Line with Scatter**

```
guiPlot("3D Line Scatter", DataSetValues=ozone.df)
```

**Drop Line Scatter**

```
guiPlot("Drop Line Scatter", DataSetValues=ozone.df)
```

**Regression**

```
guiPlot("3D Regression", DataSetValues=ozone.df)
```

**Regression with Symbols**

```
guiPlot("3D Reg & Scatter", DataSetValues=ozone.df)
```

**Surface Plots**

Surface, filled surface, and bar plots are of class `SurfacePlot`.

**Coarse Surface**

```
guiPlot("Coarse Surface", DataSetValues=ozone.df)
```

**Data Grid Surface**

```
guiPlot("Data Grid Surface", DataSetValues=ozone.df)
```

**Spline**

```
guiPlot("Spline Surface", DataSetValues=ozone.df)
```

**Coarse Filled Surface**

```
guiPlot("Filled Coarse Surface", DataSetValues=ozone.df)
```

**Data Grid Filled Surface**

```
guiPlot("Filled Data Grid Surface", DataSetValues=ozone.df)
```

**Filled Spline Surface**

```
guiPlot("Filled Spline Surface", DataSetValues=ozone.df)
```

---

**8 Color Draped Surface**

```
guiPlot("8 Color Surface", DataSetValues=ozone.df)
```

**16 Color Draped Surface**

```
guiPlot("16 Color Surface", DataSetValues=ozone.df)
```

**32 Color Draped Surface**

```
guiPlot("32 Color Surface", DataSetValues=ozone.df)
```

**Bar**

```
guiPlot("3D Bar", DataSetValues=ozone.df)
```

**3D Contour Plots** Contour and filled contour plots are of type `ContourPlot`.

**Contour**

```
guiPlot("3D Contour", DataSetValues=ozone.df)
```

**Filled Contour**

```
guiPlot("3D Filled Contour", DataSetValues=ozone.df)
```

## FORMATTING AXES AND LAYOUT

### 2D Plots

#### Adding 2D Axes

Axes are added to a plot by creating `Axi s2dX` or `Axi s2dY` objects. The `Axi sPl acement` property may be set to "Left/Lower" or "Ri ght/Upper" to specify the side of the plot upon which to place the axis. The type of frame may be specified by setting `DrawFrame` to "None", "No ti cks", "Wi th ti cks", or "Wi th l abel s & ti cks".

```
gui Create("Li nePl ot", xVal ues = uti l.mktbook,  
           yVal ues = uti l.earn, Li neStyl e="None")  
  
gui Create("Axi s2dX", Name="1$1",  
           Axi sPl acement="Ri ght/Upper",  
           DrawFrame="Wi th l abel s & ti cks")
```

### 3D Plots

#### Projected Plots

Many 2D plots can be projected onto a 3D plane. For example, the following example creates a 3D line plot with a projected contour plot:

```
gsName <- gui Pl ot("Drop Li ne Scatter", DataSe tVal ues =  
                  ozone.df)  
gui Pl ot("Contour", DataSe tVal ues=ozone.df, GraphSheet =  
         gsName, Graph = 1)
```

#### Rotating 3D Graphs

To display a 3D graph in multiple rotations:

```
gui Pl ot("Data Gri d Surface", DataSe tVal ues=ozone.df,  
         Mul ti panel ="3DRotate4Panel ")
```

#### Conditioning 3D Graphs

Conditioning of 3D graphs is done in the same manner as for 2D graphs. Conditioning on a variable used in the plot results in slicing on that variable, while conditioning on a variable not used in the plot provides a way to look at how the surface varies based on the conditioning variable.



## Creating Specialized Graphs Using Your Own Computations

You can use the user-defined smoothing property of 2D line plots to create your own customized plot types. For example, to create a plot type that will draw crosshairs showing the mean and 95% confidence intervals of the x and y data, first write a function to do the calculations:

```
> crosshairs <- function(x, y, z, w, subscripts, panel.num)
{
  # Displays 95% confidence intervals for the means of x and y.
  # x,y,z,w correspond to the data fields in the plot dialog.
  # Currently only numeric columns are supported.
  # z and w can be empty -- NULL will be sent (in this function they
  # are ignored, but in others they might be useful).
  # Subscripts contains the row numbers for the x, y, etc. data.
  # This may be useful for conditioning.
  # panel.num will contain the panel number if conditioning,
  # otherwise it will contain 0.
  meanx <- mean(x)
  meany <- mean(y)
  stdx <- sqrt(var(x))
  stdy <- sqrt(var(y))
  crit.x <- qt(0.975, length(x) - 1)
  crit.y <- qt(0.975, length(y) - 1)
  xdata <- c(meanx - crit.x * stdx, meanx + crit.x * stdx, NA,
             meanx, meanx)
  ydata <- c(meany, meany, NA, meany - crit.y * stdy,
             meany + crit.y * stdy)
  list(x = xdata, y = ydata)
}
```

Notice that the first four arguments of the user function, x, y, z, and w, correspond to the data fields in the plot dialog. If no data are specified for one or more of these fields, NULL will be passed into the function. In this function z, w, and the arguments subscripts and panel.num are all ignored. These last two arguments may be useful when creating editable Trellis plots. The vector subscripts contains the row numbers for the data that are used for the current panel. The argument panel.num will contain the panel number if conditioning, otherwise it will contain 0.

The function must return a list containing x and y. These two vectors are the data that will be used in plotting. In the function above, the first two values of x and y are the points representing a line drawn at the mean of the input y from the mean of the input x minus the 97.5% confidence interval to the mean of the input x plus the 97.5% confidence level. The third output x and y values are missing values that are used to break the line. The last two values represent a line drawn at the mean of the input x showing the 95% confidence interval for the input y.

To create a plot that will draw the crosshairs, we use the function name in the `guiCreate` function for a line plot. The symbol style is set to solid circle so that we can see the original data points in the plot as well.

```
guiCreate("LinePlot", xValues = car.time,
          yValues=car.miles, LineStyle="Solid",
          SymbolStyle = "Circle, Solid", BreakForMissings = T,
          SmoothingType ="User", UserFunctionName = "crosshairs")
```

## Locating Positions on Your Graph

You can use the `guiLocator` function to prompt the user to click on locations on the graph. `guiLocator` takes as an argument the number of points that you would like returned. It will return a list with the elements `x` and `y`. By default `guiLocator` will use the current graph sheet.

For example, the following function uses `guiLocator` to let a user interactively rescale a plot. First a line plot is created with the input data. A comment is put on the graph prompting the user to click on the points that will determine the new `x` axis minimum and maximum. `guiLocator` is called requesting 2 points. Then the comment is removed and the `X` axis is rescaled with the `x` values returned from `guiLocator`.

```
my.rescale <-function(x, y) {
  guiCreate("LinePlot", xValues = x, yValues = y)
  gsName <- guiGetCurrMetaDoc("GraphSheet")
  commentName <- paste("$$", gsName, "$1", sep = "")
  guiCreate("CommentDate", Name = commentName, Title =
    "Click on two points to determine the \n
    new X axis minimum and maximum",
    xPosition = "Auto",
    yPosition = 0.12, FontSize = 15, UseDate = F,
    UseTime = F)
  a <- guiLocator(2)
  minXVal <- a$x[1]
  maxXVal <- a$x[2]
  guiRemove("CommentDate", Name = commentName)
  guiModify("Axis2dX", Name = "1$1", AxisMin = minXVal,
    AxisMax = maxXVal)
  invisible()
}
```

To use this function, try:

```
theta <- seq( 0, by=pi/10, len = 150)
y <- sin(theta)
my.rescale( theta, y )
```

One additional feature of `guiLocator` is often useful: if you call `guiLocator(-1)`, the current plot redraws.



---

<b>Introduction</b>	<b>240</b>
<b>Getting Started With Simple Plots</b>	<b>241</b>
Plotting a Vector Data Object	241
Plotting Mathematical Functions	242
Creating Scatter Plots	244
Plotting Time Series Data Objects	244
<b>Frequently Used Plotting Options</b>	<b>248</b>
Plot Shape	248
Multiple Plot Layout	248
Titles	249
Axis Labels	251
Axis Limits	251
Logarithmic Axes	252
Plot Types	252
Line Types	255
Plotting Characters	255
Plotting Characters and Line Types for Multiple Graphs	257
Controlling Plotting Colors	258
<b>Interactively Adding Information to Your Plot</b>	<b>259</b>
Identifying Plotted Points	259
Adding Straight Line Fits to a Current Scatter Plot	260
Adding New Data to a Current Plot	261
Adding Text to Your Plot	262
<b>Making Bar Plots, Dot Charts and Pie Charts</b>	<b>265</b>
Bar Plots	265
Dot Charts	267
Pie Charts	269
<b>Visualizing the Distribution of Your Data</b>	<b>271</b>
Box Plots	271
Histograms	272

Density Plots	273
Quantile-Quantile Plots	274
<b>Visualizing Higher Dimensional Data</b>	<b>278</b>
Multivariate Data Plots	278
Scatterplot Matrices	278
Plotting Matrix Data	279
Star Plots	280
Faces	280
<b>3-D Plots: Contour, Perspective, and Image Plots</b>	<b>282</b>
Contour Plots	282
Perspective Plots	284
Image Plots	285
<b>Customizing Your Graphics</b>	<b>287</b>
<b>Low-level Graphics Functions and Graphics Parameters</b>	<b>288</b>
<b>Setting and Viewing Graphics Parameters</b>	<b>290</b>
<b>Controlling Graphics Regions</b>	<b>293</b>
Controlling the Outer Margin	294
Controlling Figure Margins	295
Controlling the Plot Area	296
<b>Controlling Text in Graphics</b>	<b>297</b>
Controlling Text and Symbol Size	297
Controlling Text Placement	298
Controlling Text Orientation	299
Controlling Line Width	300
Plotting Symbols in Margin	300
<b>Text in Figure Margins</b>	<b>301</b>
<b>Controlling Axes</b>	<b>303</b>
Enabling and Disabling Axes	303
Controlling Tick Marks and Axis Labels	303
Controlling Axis Style	306
Controlling Axis Boxes	307
<b>Controlling Multiple Plots</b>	<b>308</b>
<b>Overlaying Figures</b>	<b>311</b>
High-level Functions That Can Act as Low-level Functions	311
Overlaying Figures by Setting new=TRUE	311
Overlay Figures by Using subplot	312

---

<b>Adding Special Symbols to Plots</b>	<b>315</b>
Arrows and Line Segments	315
Adding Stars and Other Symbols	316
Custom Symbols	318
<b>Writing Graphics Functions</b>	<b>320</b>
Modifying Existing Functions	320
Combining High-Level and Low-Level Functions	323
Using Graphical Parameters	325
<b>Traditional Graphics Summary</b>	<b>327</b>
References	329

## INTRODUCTION

Visualizing data is a powerful data analysis tool because it allows you to easily detect interesting features or structure in the data. This may lead you to immediate conclusions or guide you in building a statistical model for your data. This chapter shows you how to use S-PLUS to visualize your data.

The section *Getting Started With Simple Plots* (page 241) shows you how to plot vector and time series objects. Once you have read this first section, you will be ready to use any of the plotting options described in the section *Frequently Used Plotting Options* (page 248). These options, which can be used with many S-PLUS graphics functions, control most features in a plot, such as plot shape, multiple plot layout, titles, axes, etc.

The remaining sections of this chapter cover a range of plotting tasks:

- Interactively adding information to your plot.
- Barplot, pie chart and dot chart type presentation graphics.
- Visualizing the distribution of your data.
- Visualizing correlation in your time series data.
- Using multiple active graphics devices.

We recommend that you read the first two sections carefully before proceeding to any of the other sections.

In addition to the graphics features described in this chapter, S-PLUS includes the editable graphics described in Chapter 7, *Editable Graphics Commands*, and the Trellis Graphics library. Trellis Graphics features additional functionality such as multipanel layouts and improved 3-D rendering. See Chapter 9, *Traditional Trellis Graphics*, for more information.



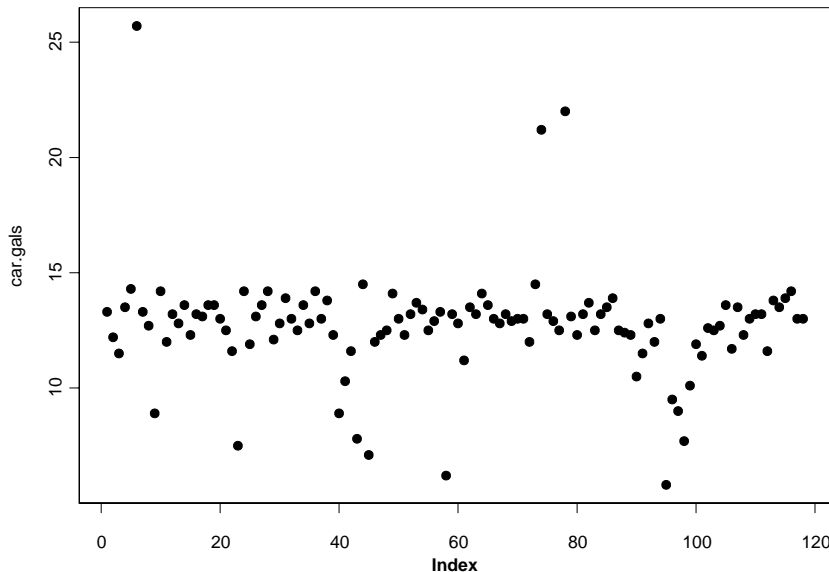
## GETTING STARTED WITH SIMPLE PLOTS

This section helps you get started with S-PLUS graphics by using two functions, `plot` and `ts.plot`, to make simple plots of your data. You use the function `plot` to make plots of vector data objects, plots of mathematical functions, and scatter plots of two vector data objects, i.e., plots of the values of one variable against the values of another variable. You use the function `ts.plot` to plot one or more time series of data against the observation times of the data.

### Plotting a Vector Data Object

You can graphically display the values of a batch of numbers, or “observations,” using the function `plot`. For example, you obtain a graph of the built-in vector data object `car.gals` using `plot` as follows:

```
> plot(car.gals)
```

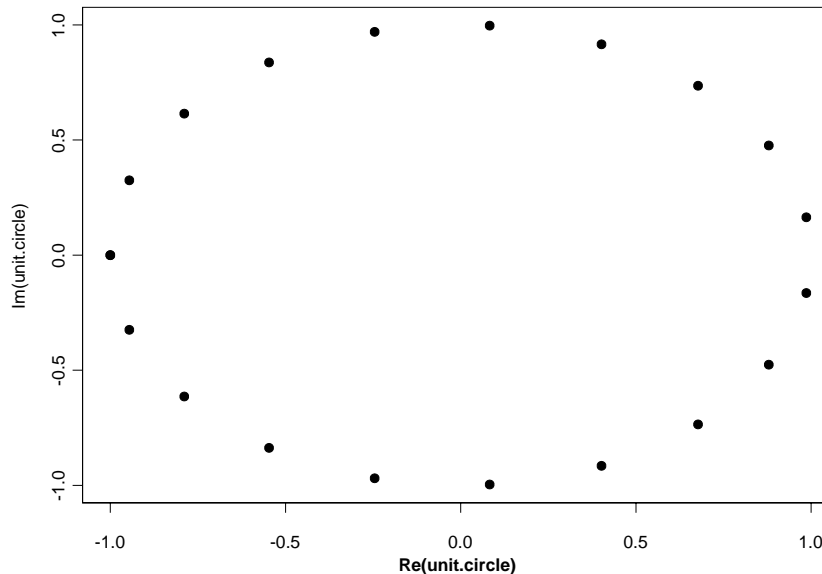


**Figure 8.1:** *Scatter plot of a single vector.*

The data are plotted as a set of isolated points. For each plotted point, the vertical axis location gives the data value and the horizontal axis location gives the observation number, or *index*.

If you have a vector  $x$  which is *complex*, `plot` plots the real part of  $x$  on the horizontal axis and the imaginary part on the vertical axis. For example, a set of points on the unit circle in the complex plane can be plotted as follows:

```
> unit.circle <- complex(arg=seq(-pi, pi, length=20))
> plot(unit.circle)
```



**Figure 8.2:** *Scatter plot of a single complex vector.*

## Plotting Mathematical Functions

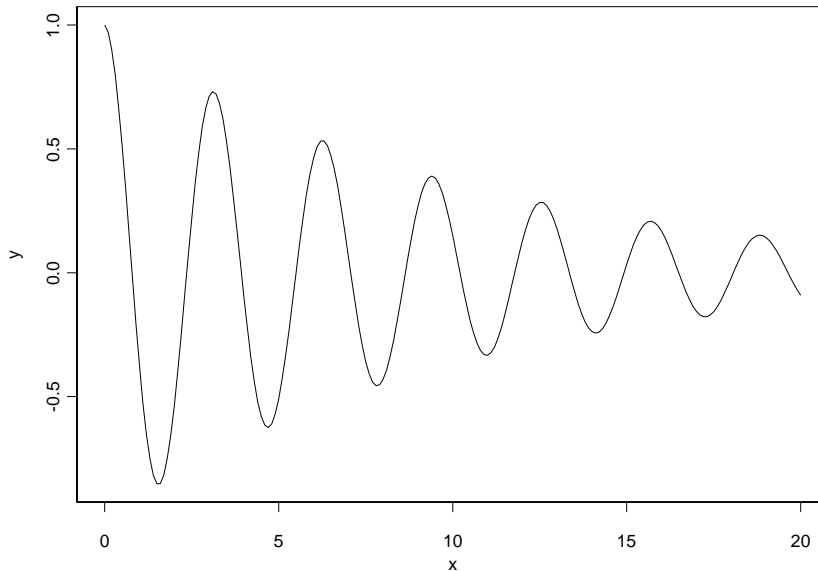
You can obtain smooth solid line plots of mathematical functions with `plot`, by using the optional argument `type="l"` to produce a plot with connected solid line segments rather than isolated points, provided you choose a sufficiently dense set of plotting points.

For example, to plot the mathematical function in the equation:

$$y = f(x) = e^{(-x)/10} \cos(2x) \quad (8.1)$$

for  $x$  in the range  $(0,20)$ , create a vector  $x$  with values ranging from 0 to 20 at intervals of 0.1, compute the vector  $y$  by evaluating the function at each value in  $x$ , then plot  $y$  against  $x$ :

```
> x <- seq(0, 20, .1)
> y <- exp(-x/10)*cos(2*x)
> plot(x, y, type="l")
```



**Figure 8.3:** *Plot of  $\exp(-x/10) * \cos(2x)$ .*

For a rougher plot, use fewer points; for a smoother plot, use more.

## Creating Scatter Plots

Scatter plots reveal relationships between pairs of variables. You create scatter plots in S-PLUS with the `plot` function applied to a pair of equal-length vectors, a matrix with two columns, or a list with components `x` and `y`. For example, to plot the built-in vectors `car.miles` versus `car.gals`, use the following S-PLUS expression:

```
> plot(car.miles, car.gals)
```

When using `plot` with two vector arguments, the first argument is plotted along the horizontal axis and the second argument is plotted along the vertical axis.

If `x` is a matrix with two columns, you use `plot(x)` to plot the second column versus the first. For example, you could combine the two vectors `car.miles` and `car.gals` into a matrix called `miles.gals` by using the function `cbind`:

```
> miles.gals <- cbind(car.miles, car.gals)
```

Then use

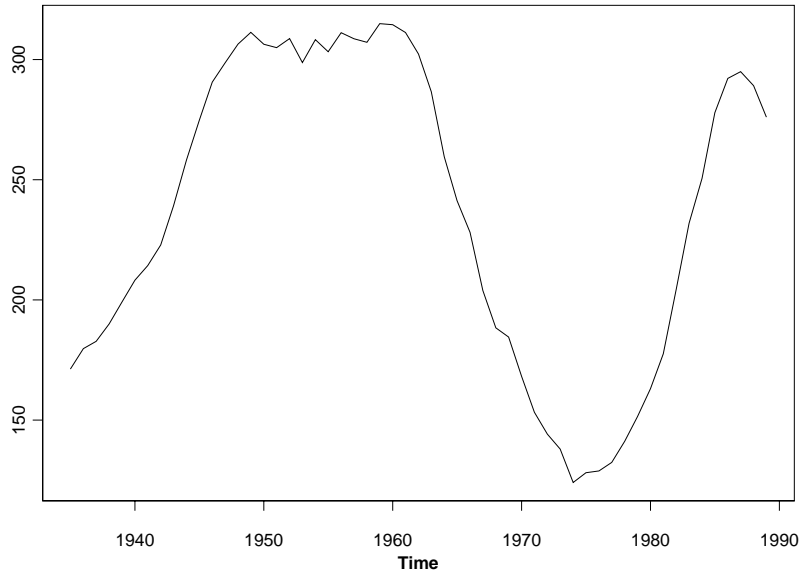
```
> plot(miles.gals)
```

## Plotting Time Series Data Objects

When data are gathered over time, it is natural to plot the observations against the time at which each observation was made. You can make such time series plots in S-PLUS for data collected at *equally spaced* time intervals, e.g., every month, every hour, etc. To do so, your data must be in the form of a *time series object* (see Chapter 5, Creating and Viewing Time Series, in the *Guide to Statistics, Volume 2*).

To plot a time series data object against its associated observation times, use the `ts.plot` function. For example, the `halibut` data set is a list with two components, each of which is a time series of length 55. To plot the first component, `halibut$biomass`, as a time series, use:

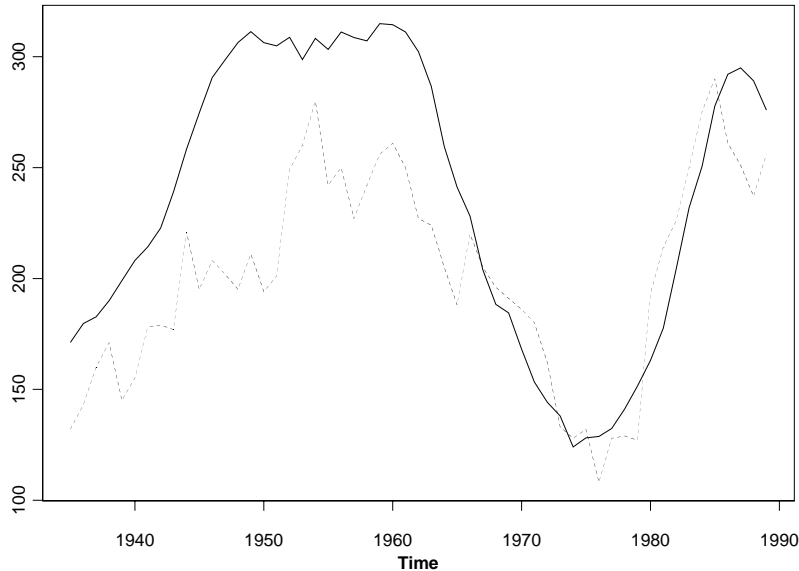
```
> ts.plot(halibut$biomass)
```



**Figure 8.4:** *Plotting time series data with **tsplot**.*

The function `ts.plot` also allows you to display two or more time series on a single plot. This is often useful if the time series have similar units or scales. For example, the `biomass` and `cpue` components of the `halibut` data set have different units, but have numerical values which are not too different. To plot them on the same graph, use `ts.plot` with both objects as arguments:

```
> ts.plot(hal i but$bi omass, hal i but$cpue)
```



**Figure 8.5:** *Plotting two time series simultaneously.*

The solid line always represents the first argument to `ts.plot` and the dashed line represents the second argument. You can get plotted points rather than lines when using `ts.plot`; read the section *Frequently Used Plotting Options* (page 248) for more information. You can plot as many time series as you want on a single graph, supplying each time series as an argument to `ts.plot`. However, a practical limit on the number of plots per graph is five to seven. Too many plots on a single graph causes clutter, making it difficult for you to extract information contained in the graph. If you want to plot a time series which is observed at *unequally* spaced time intervals, use `plot` with a vector of observation times as the `x` argument and the data as the `y` argument. To get a solid line plot, use the optional argument `type="l"`. You can make a time series plot of two or more time series with unequally spaced observations by using the function `matplot`. You must first combine the observations into a matrix. For example, suppose you have five observations, taken in January, March, April, July, and August of 1958, from each of two time series. The five observations from the first time series are stored in the

vector `data.1` and the five observations from the second series are stored in `data.2`:

```
> data.1
```

```
[1] 7.073236 10.942785 15.091415 20.973578 8.878581
```

```
> data.2
```

```
[1] 7.290172 12.308828 12.780996 19.764697 8.298905
```

You will need to recreate these vectors to run this example.

To use `matplot`, you first create a vector `times` to hold the observation times, use `cbind` to form a matrix of the time series data, then use `matplot` to plot the time series data against the observation times:

```
> times <- c(1958.000, 1958.167, 1958.250, 1958.500, 1958.667)
```

```
> data.ts <- cbind(data.1, data.2)
```

```
> matplot(times, data.ts, type="l")
```

You can also use `matplot` to plot two or more vector objects against their observation numbers. See the help file for further details on the use of `matplot`.

## FREQUENTLY USED PLOTTING OPTIONS

This section tells you how to make plots in S-PLUS with one or more of a collection of frequently used options. These options include:

- Controlling plot shape and multiple plot layout
- Adding titles and axis labels
- Setting axis limits and specifying logarithmic axes
- Choosing plotting characters and line types
- Choosing plotting colors

### Plot Shape

When you use an S-PLUS plotting function, the default shape of the box enclosing the plot is *rectangular*. Sometimes you prefer to have a *square* box around your plot. For example, a scatter plot is usually displayed as a square plot. You get a square box by using the global graphics parameter function `par` as follows:

```
> par(pty="s")
```

All subsequent plots are made with a square box around the plot. If you want to return to making rectangular plots, use

```
> par(pty="")
```

The `pty` stands for “plot type” and the “s” stands for square. However, you should think of `pty` as standing for “plot shape”, to avoid confusion with a different meaning for “plot type” in the section Plot Types (page 252).

### Multiple Plot Layout

You may want to display more than one plot on your screen, or on a single page of paper. To do so, you use the S-PLUS function `par` with the layout parameter `mfrow` to control the layout of the plots, as illustrated by the following example. In this example, you use `par` to set up a four plot layout, with two rows of two plots each. Following the use of `par`, we create four simple plots with titles:

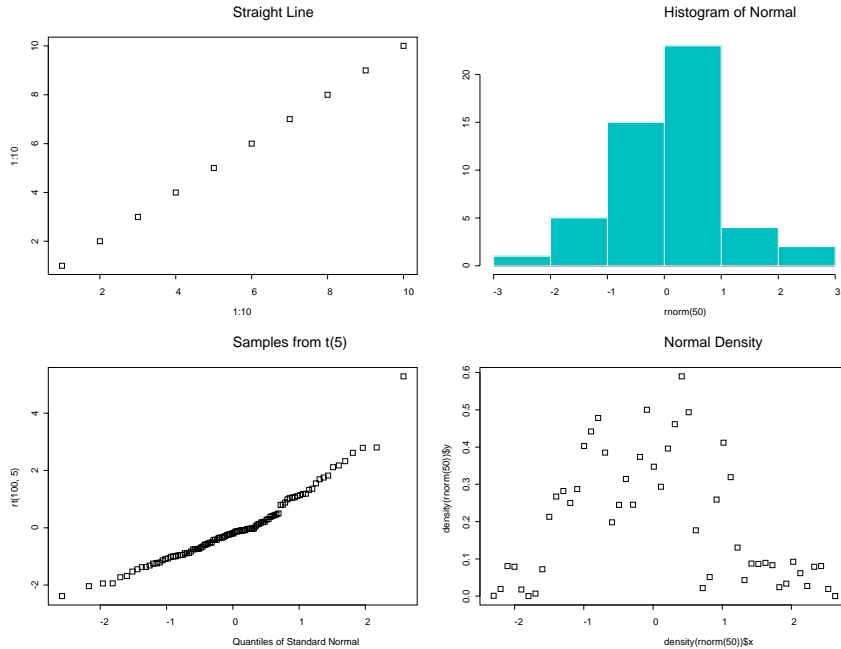
```
> par(mfrow=c(2,2))
```



```

> plot(1:10, 1:10, mai n="Straigh t Li ne")
> hist(rnorm(50), mai n="Hi stogram of Normal ")
> qqnorm(rt(100, 5), mai n="Sampl es from t(5)")
> plot(densi ty(rnorm(50)), mai n="Normal Densi ty")

```



**Figure 8.6:** *A four plot layout.*

When you are ready to return to one plot per figure, use

```
> par(mfrow=c(1, 1))
```

The function `par` is used to set many general parameters related to graphics. See the section *Setting and Viewing Graphics Parameters* (page 290) and the `par` help file for more information on using `par`. The section *Controlling Multiple Plots* (page 308) contains more information on using the `mfrow` parameter, and describes another method for creating multiple plots.

## Titles

You can easily add titles to any S-PLUS plot. You can add a *main title*, which goes at the top of the plot, or a *subtitle*, which goes at the bottom of the plot.

To get a main title on a plot of the `car.miles` versus `car.gals` data, use the argument `main` to `plot`. For example,

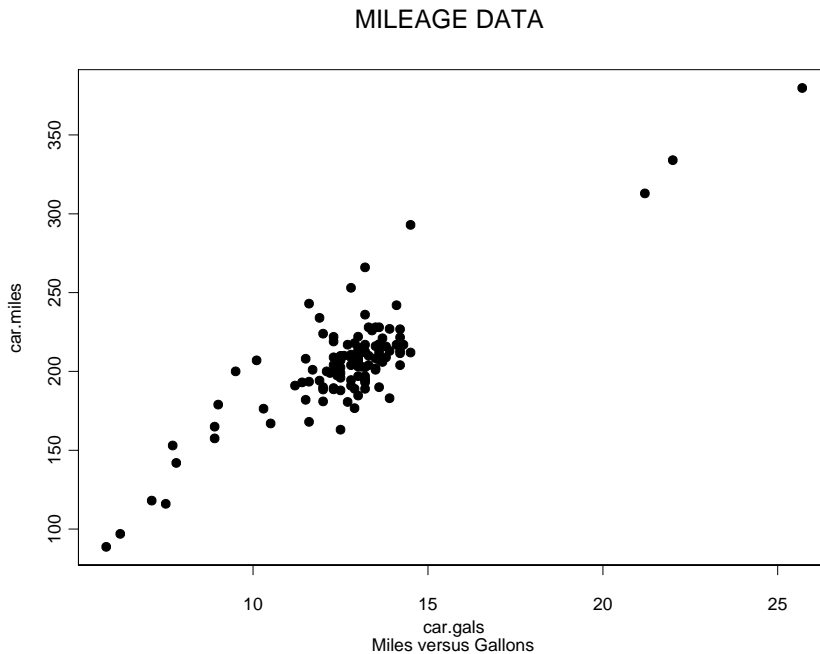
```
> plot(car.gals, car.miles, main="MILEAGE DATA")
```

To get a subtitle, use the `sub` argument:

```
> plot(car.gals, car.miles, sub="Miles versus Gallons")
```

To get both a main title and a subtitle, use both arguments:

```
> plot(car.gals, car.miles, main="MILEAGE DATA",  
+ sub="Miles versus Gallons")
```



**Figure 8.7:** *Putting main titles and subtitles on plots.*

Alternatively, you can add the titles after creating the plot using the function `title`, as follows:

```
> plot(car.gals, car.miles)  
> title(main="Mileage Data", sub="Miles versus Gallons")
```

## Axis Labels

When you use `plot`, S-PLUS provides axis labels which by default are the names of the data objects passed as arguments to `plot`. However, data object names, such as `car.gals` and `car.miles`, are chosen with brevity in mind. You may want to use more descriptive axis labels. For example, you may prefer “Gallons per Trip” and “Miles per Trip,” respectively, to “car.gals” and “car.miles.” To obtain your preferred labels, use the `xlab` and `ylab` arguments. For example,

```
> plot(car.gals, car.miles, xlab="Gallons per Trip",
+      ylab="Miles per Trip")
```

If you don't want the default labels, you can suppress them by using the arguments `xlab` and `ylab` with the value `""`, as follows:

```
> plot(car.gals, car.miles, xlab="", ylab="")
```

This gives you a plot with no axis labels. If desired, you can then add axis labels using `title`:

```
> title(xlab="Gallons per Trip", ylab="Miles per Trip")
```

## Axis Limits

The limits of the  $x$ -axis and the  $y$ -axis are set automatically by the S-PLUS plotting functions. However, you may wish to choose your own axis limits to make room for adding text in the body of a plot (as described in the section *Interactively Adding Information to Your Plot* (page 259)). For example,

```
> plot(co2)
```

automatically determines  $y$ -axis limits of roughly 310 and 360, giving just enough vertical room for the plot to fit inside the box.

You can make more vertical or horizontal room in the plot by using the optional arguments `ylim` and `xlim`. To get  $y$ -axis limits of 300 and 370, use

```
> plot(co2, ylim=c(300, 370))
```

You can change the  $x$ -axis limits as well; for example:

```
> plot(co2, xlim=c(1955, 1995))
```

You can use both `xlim` and `ylim` at the same time. S-PLUS rounds your specified axis limits to sensible values. You may also want to set axis limits when you are making multiple plots, as described in the section *Multiple Plot Layout* (page 248). For example, after creating one plot, you may wish to make the  $x$ -axis and  $y$ -axis limits the same for all of the plots in the set. You can do so by using the function `par` as follows:

```
> par(xaxs="d", yaxs="d")
```

If you want to control the limits of only one of the axes, you drop one of the two arguments, as appropriate. Using the `xaxs="d"` and `yaxs="d"` arguments sets all axis limits to the values for the most recent plot in a sequence of plots. If those limits are not the widest required in the sequence, points outside the limits are not plotted and you receive the message `Points out of bounds`. To avoid this error, you can first make all plots in the usual way, without specifying axis limits, to find out which plot has the largest range of axis limits. Then, create your first plot using `xlim` and `ylim` with values determined by the largest range. Now set the axes with `xaxs="d"` and `yaxs="d"` as described above. To return to the usual default state, in which each plot determines its own limits in a multiple plot layout, use

```
> par(xaxs="", yaxs="")
```

The change goes into effect on the next “page” of figures.

## Logarithmic Axes

Often, a data set you are interested in does not reveal much detail when graphed on ordinary axes. This is particularly true when many of the data points bunch up at small values, making it difficult to see any potentially interesting structure in the data. Such data sets yield more informative plots if you graph them using a *logarithmic scale* for one or both of the axes.

To put the horizontal axis on a logarithmic scale, use `log="x"`, similarly for the vertical axis use `log="y"`, and to put both the horizontal and vertical axes on logarithmic scales, use `log="xy"`.

## Plot Types

You can plot data in S-PLUS in any of the following ways:

- As points
- As lines (i.e., as connected straight line segments)
- As both points and lines (with points isolated)
- As “overstruck” points and lines (points not isolated)
- As a vertical line for each data point (this is known as a “high-density” plot)

- As a stairstep plot
- As an *empty* plot, with axes and labels but no data plotted

The method used for plotting data on a graph is called the graph's *plot type*. Scatter plots typically use the first plot type, while time series plots typically use the second. In this section, we give examples of the other plot types. You choose your plot type by using the optional argument *type*. The possible values for this argument correspond to the choices listed above:

**Table 8.1:** *Possible values of the plot **type** argument.*

Setting	Plot type
<code>type="p"</code>	points
<code>type="l "</code>	lines
<code>type="b"</code>	both points and lines
<code>type="o"</code>	lines with points overstruck
<code>type="h"</code>	high-density plot
<code>type="s"</code>	stairstep plot
<code>type="n"</code>	no data plotted

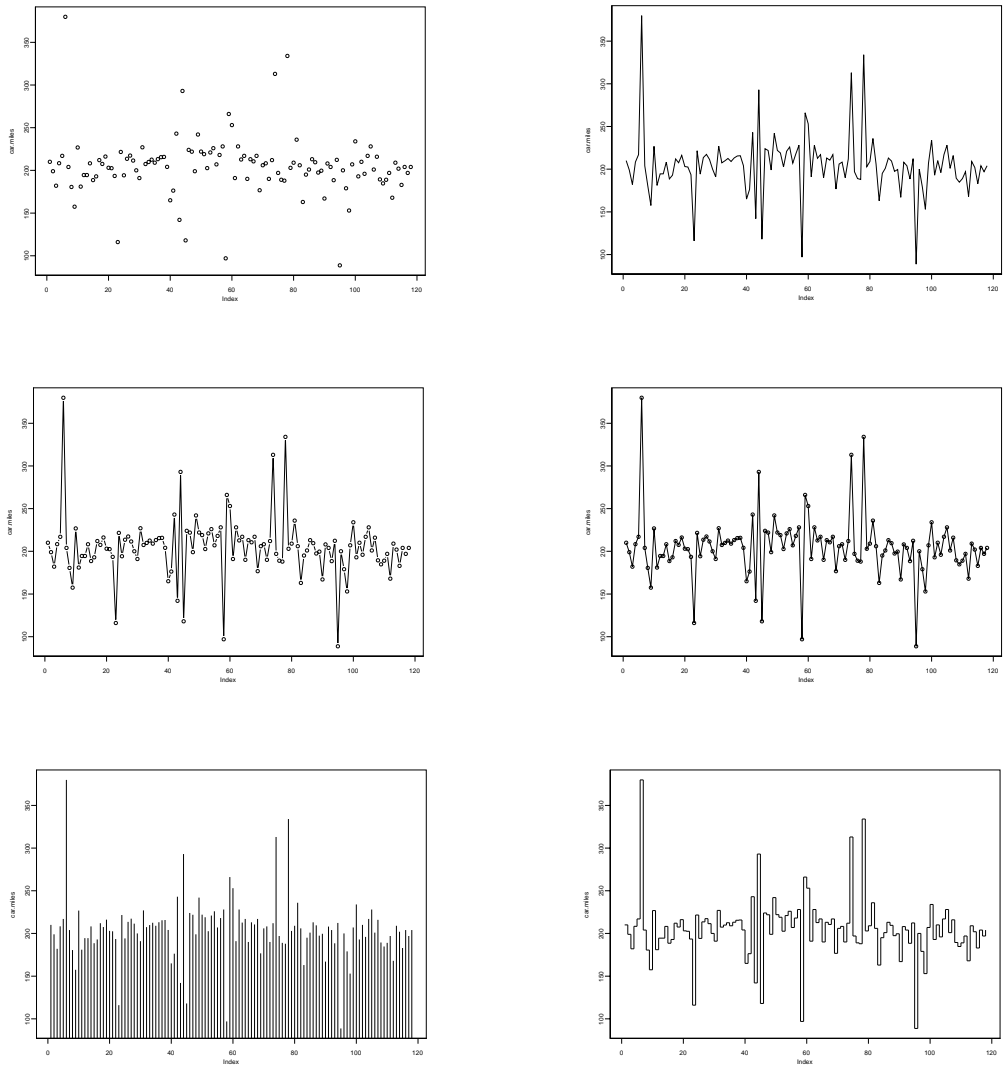
Different graphics functions have different default choices. For example, `plot` and `matplot` use the default `type="p"`, while `ts.plot` uses the default `type="l "`. Although you can use any of the plot types with any plotting function, some combinations of plot function and plot type may result in an ineffective display of your data. The option `type="n"` is useful for obtaining precise control over axis limits and box line types. For example, you might want to have the axes and labels in one color, and the data plotted in another. You could do this easily as follows:

```
> plot(x, y, type="n")
> points(x, y, col=3)
```

Figure 8.8 shows the different plot types for the built-in data set `car.miles`, plotted with the `plot` function:

```
> plot(car.miles)
> plot(car.miles, type = "l")
> plot(car.miles, type = "b")
```

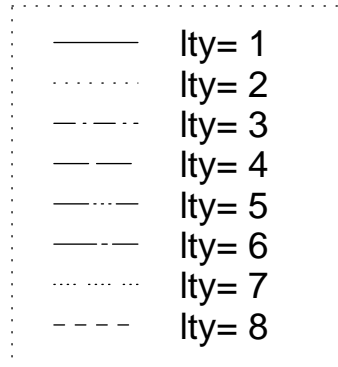
```
> plot(car.miles, type = "o")  
> plot(car.miles, type = "h")  
> plot(car.miles, type = "s")
```



**Figure 8.8:** Plot types for the function *plot*. Top row: points and lines, second row: both points and lines, and lines with points overstruck, third row: high density plot and stairstep plot.

## Line Types

When your plot type involves lines, you can choose the *line type* for the lines. By default, the line type for the first line on a graph is a solid line. If you prefer a different line type, you can use the argument `lty=n`, where *n* is an integer, to specify a different one. On most devices, there are eight distinct line types; Figure 8.9 illustrates the various types.



**Figure 8.9:** *Line types.*

If you specify a higher value, S-PLUS produces the line type corresponding to the *remainder* on division by the number of line types. For example, if you specify `lty=26` on the graphicsheet graphics device, S-PLUS produces the line type shown as `lty=2`.

### Warning

The value of `lty` must be an integer. This contrasts with the value of `type`, which is of character mode and is therefore enclosed in quotes. For example, to plot the time series `halibut$cpue` using `ts.plot` with `lty=2`:

```
> ts.plot(halibut$cpue, lty=2)
```

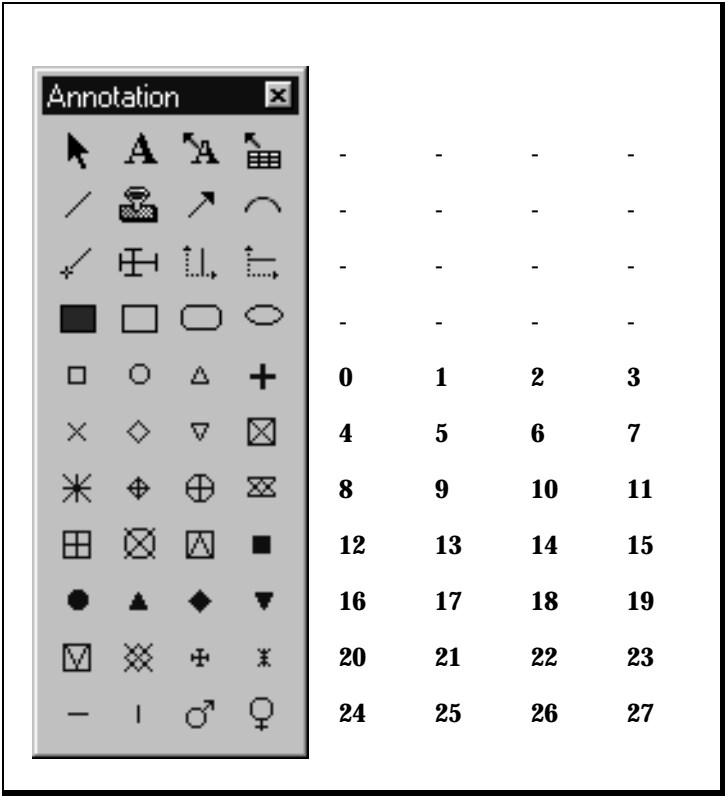
## Plotting Characters

When your plot type involves points, you can choose the *plotting character* for the points. By default, the plotting character is usually a circle (o), depending on your graphics device and the `plot` function you use. For `matplot` (and `ts.plot` if `type="p"`) the default plotting character is the number 1, because `ts.plot` and `matplot` are often used to plot more than one time series or more than one vector. In such cases, more than one plotting

character is needed to distinguish the separate graphs (one plotting character for each time series or vector to be plotted). The default plotting characters in such cases are the numbers 1, 2,... However, you can choose alternative plotting characters when making a points-type plot with any of the above plotting functions by using the optional argument `pch`. Any printing character can be used as a plotting character. The plotting character is specified as a character string, so it must be enclosed in quotes. For example:

```
> plot(hal i but$bi omass, pch="B")
> ts.plot(hal i but$cpue, type="p", pch="C")
```

You can also choose any one of a range of plotting symbols by using `pch=n`. Here you must use *numeric* mode for the value of `pch`. The symbol corresponding to each of these integers is shown in Figure 8.10.



**Figure 8.10:** Plotting symbols are on the Annotation palette, and can be accessed from the language by using the argument ***pch=*n****.



## Plotting Characters and Line Types for Multiple Graphs

When you use `ts.plot` to make graphs of two or more vector objects or time series objects in a single composite plot, you want a unique line type or plotting character for each data set to make it easy to distinguish the data sets from one another. This is taken care of automatically by `ts.plot`, which uses up to five different line types as needed when you plot five or fewer data objects (vectors or time series). To observe this default behavior, use `ts.plot` to plot the vectors `y1`, `y2`, `y3`, `y4`, and `y5` consisting of random numbers with means 1, 2, 3, 4, and 5, respectively:

```
> ts.plot(y1, y2, y3, y4, y5)
```

You can choose alternative line types by using the argument `lty` with a *numeric vector* as its value. For example:

```
> ts.plot(y1, y2, y3, lty=c(1, 3, 5))
```

You can also make a points-type plot with `ts.plot`, using either numbers or letters as points. To do so, you use both the `type="p"` and the `pch` optional arguments, with the value of `pch` either a character string consisting of the numbers and letters you want to use or a numeric vector representing the special plotting symbols you want to use. For example:

```
> ts.plot(y1, y2, y3, type="p", pch="A3B")
```

You can also mix the plot types as follows

```
> ts.plot(y1, y2, y3, y4, y5, type="pl pl p",  
+ pch="AABBC", lty=c(1, 1, 5, 5, 5))
```

In the above example, you specify the plot types for each of the data sets `y1`, `y2`, `y3`, `y4`, and `y5`, by using the character string `"pl pl p"` as the value of `type`. You construct the values of `pch` and `lty` as a character vector of length 5 and a numeric vector of length five, respectively. Only the first, third, and fifth values of `"AABBC"` matter, but you must provide the second and fourth characters to force the character string to length five.

The character string `"A B C"`, where the second and fourth characters are blanks, works also, as would any character string of length 5 with A in the first place, B in the third place, and C in the fifth place. Similarly, only the second and fourth values of `c(1, 1, 5, 5, 5)` matter, but you must provide first, third, and fifth values in order to make this numeric vector have length five. You may also use `matplot` to plot several vector objects in matrix form against their observation numbers (in the form of the vector object `1:n`) or against the values of some independent variable (in the form of a vector object). When using `matplot`, the default plotting type is `"points"`, and the default plotting characters are the integers 1, 2,... You can choose alternative plotting types (e.g., `"lines"`) with the `type` argument. If `type="l"`, you can choose alternative line types with the `lty` argument. If the plotting type

involves points, you can choose alternative plotting characters with the `pch` argument as described above for `ts.plot`.

## Controlling Plotting Colors

To specify the *color* in which your graphics are plotted, use the `col` parameter. You can use color to distinguish between sets of overlaid data:

```
> plot(co2)
> lines(smooth(co2), col = 2)
```

The colors available are determined by the device's *color map*. The default color map for `graphsheet` has sixteen colors: fifteen foreground and one background color. To see all the colors in the default color map, use the following expression:

```
> pie(rep(1, 15), col = 1: 15)
```

This expression plots a pie chart with 15 colors on the background color, color 0, for a total of 16 colors. You specify the color map for the `graphsheet` device using the Color Schemes dialog box, which lists the default color map, or *scheme*, together with several other predefined schemes and any color schemes you define. From the Color Schemes dialog box, you can select an alternate color scheme, modify existing color schemes, or define new color schemes. See the online help for details on working with color schemes. You may want to experiment with many values to find the most pleasing color map. For other graphics devices, see the device's help file for a description of the color map. S-PLUS uses the color map cyclically, that is, if you specify `col = 9` and your color map has only 8 colors, S-PLUS prints color 1. Color 0 is the background color; over-plotting items using color 0 erases them on most graphics devices.

## INTERACTIVELY ADDING INFORMATION TO YOUR PLOT

The functions described so far in this chapter create complete plots. Often, however, you want to build on an existing plot in an interactive way. For example, you may want to identify individual points in a plot and label them for future reference. Or you may want to add some text or a legend, or overlay some new data. In this section, we describe some simple techniques for interactively adding information to your plots. More involved techniques for producing customized plots are described in the section Customizing Your Graphics (page 287).

### Identifying Plotted Points

While examining a plot, you may notice that some of the plotted points are unusual in some way. To identify the observation numbers of such points, use the `i d e n t i f y` function, which lets you “point and click” with a mouse on the unusual points. For example, consider the plot of  $y$  versus  $x$ , plotted as follows:

```
> set.seed(12)
> x <- runif(20)
> y <- 4*x + rnorm(20)
> x <- c(x, 2)
> y <- c(y, 2)
> plot(x, y)
```

You immediately notice one point separated from the bulk of the data. (Such a data point is called an *outlier*.) To identify this point by observation number, use `i d e n t i f y` as follows:

```
> i d e n t i f y(x, y, n=1)
```

After pressing return, you *do not* get a prompt. Instead, S-PLUS waits for you to identify points with the mouse. Now move the mouse cursor into the graphics window so that it is adjacent to the data point to be identified and click the left mouse button. The observation number appears next to the point. If you click when the cursor is more than 0.5 inch from the nearest point in the plot, a message appears on your screen to tell you there are no points near the cursor. After identifying all the points that you requested (in our example,  $n=1$ ), S-PLUS prints out the observation numbers of the identified points and returns your prompt:

```
> i d e n t i f y(x, y, n=1)
```

```
[1] 21
```

If you omit the optional argument `n=n` you can identify as many points as you wish. In this case, you must signal S-PLUS that you've finished identifying points by taking an appropriate action, for example, pressing the right mouse button. You can use `identify` with any plot of data (as opposed to summaries of data, such as histograms). For example, consider the time series plots of the time series objects `hali but$biomass` and `hali but$cpue`.

The solid line is `hali but$biomass` and the dashed line is `hali but$cpue`. To identify the points in both series which correspond to the local minimum of `hali but$biomass` in 1974 and the maximum of `hali but$biomass` which occurs around 1987, use `identify` as follows:

```
> identify(time(hali but$biomass), hali but$biomass, n=2)
```

(The function `time` extracts the vector of "times" from `hali but$biomass` to provide a vector of x coordinates to `identify`.)

Point the mouse cursor at the appropriate points of `hali but$biomass` in the plot and click. S-PLUS returns their observation numbers. You can then use `identify` again:

```
[1] 40 53
```

```
> identify(time(hali but$biomass), hali but$cpue, n=2)
```

```
[1] 40 53
```

In this second use of `identify`, you point the mouse cursor at the appropriate points of `hali but$cpue` in the plot and click.

## Adding Straight Line Fits to a Current Scatter Plot

When you make a scatter plot, you may notice an approximately linear association between the vertical-axis variable and the horizontal-axis variable. In such cases you may find it helpful to display a straight line which has been fit to the data. You can use the function `abline(a, b)` to add a straight line with intercept `a` and slope `b`, on the current plot.

## Adding a Least- Squares Straight Line

The best-known method of fitting a straight line to a scatter plot is the method of least squares. The S-PLUS function `lm` fits a linear model using the method of least-squares. The `lm` function requires a formula argument, expressing the dependence of the response variable `y` on the predictor variable `x`. See the *Guide to Statistics, Volumes 1 and 2* for a complete description of formulas and statistical modeling. To get a least-squares line, simply use `abline` on the results of `lm`. For example, use the following S-PLUS expressions to obtain a scatter plot and dotted line least-squares fit:

```
> plot(x, y)
> abline(lm(y ~x), lty=2)
```

## Adding a Robust Straight Line Fit

While the fitting of a least-squares line to data in the plane is probably the most common data fitting procedure in the world, the least-squares approach has a fundamental weakness: it lacks robustness, in the sense that the least-squares method is very sensitive to outliers. A robust method is one which is not affected very much by outliers, and which gives a good fit to the bulk of the data.

If you want a highly robust straight line fit to your scatterplot data, use the S-PLUS function `ltsreg` in place of `lm`:

```
> abline(ltsreg(x, y))
```

See Rousseeuw and Leroy (1987) for details on the LTS method.

## Adding New Data to a Current Plot

Once you have created a plot, you may want to add additional data to it. For example, you might plot an additional data set with a different line type or plotting character. Or you might add a statistical function such as a smooth curve fit to the data already in the plot. To add data to a plot created by `plot`, you use one of the two functions `points` or `lines`. These functions are virtually identical to `plot` except that they plot without creating a new set of axes. The `points` function is used to add data points, while `lines` is used to add lines. All the arguments to `plot` that we've discussed so far (including `type`, `pch`, and `lty`) work with `points` and `lines` exactly as before. This means that you can choose line types and plotting characters as you wish. (You can even make line-type plots with `points` and points-type plots with `lines`!) For example, suppose you plot the built-in data set `co2`, which gives monthly levels of carbon dioxide at the Mauna Loa volcano from January 1959 to December 1990:

```
> plot(co2)
```

By default, `plot` uses "points" to plot the data. The `plot` function recognizes that `co2` is a time series data set consisting of monthly measurements, and provides appropriate yearly labels on the horizontal axis. The series `co2` has an obvious seasonal cycle and an increasing trend. It is often useful to smooth such data, and display the smoothed version in the same plot. The function `smooth` produces a smoothed version of an S-PLUS data object. You can use `smooth` as an argument to `lines` to add a plot of the smoothed version of `co2` to the existing plot:

```
> lines(smooth(co2))
```

You might also start with a connected plot of `co2`, such as that created by `ts.plot`, and then decide to add `co2` with `points`:

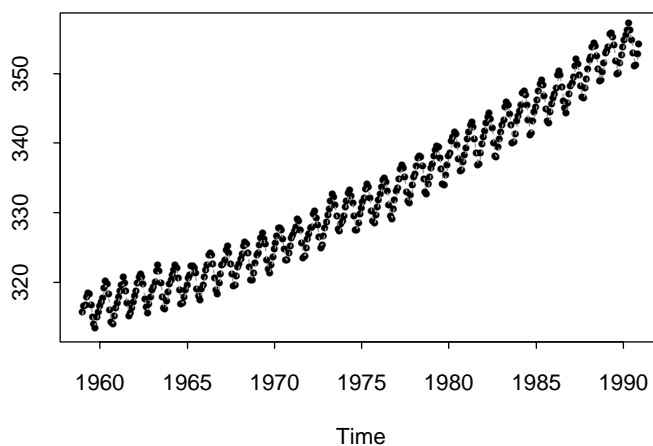
```
> ts.plot(co2)
> points(co2)
```

### Warning

If the data you add with `points` or `lines` have a range greater than the axis limits of the original plot, not all the data you want will be added. In such cases you get an “out of bounds” warning message. You can avoid this by appropriate use of the optional argument `ylim`. It is usually easier, however, once you see which of two or more graphs you want, to do them all at once with `ts.plot`. For example:

```
> ts.plot(co2, smooth(co2), type="pl", pch="*")
```

If your original plot was created with `ts.plot` or `matplot`, you can add new data with functions analogous to `points` and `lines`. To add data to a plot created with `ts.plot`, use `ts.points` or `ts.lines`. To add data to a plot created with `matplot`, use `matpoints` or `matlines`. See the corresponding help files for further details.



**Figure 8.11:** *Multiple plots with `ts.plot`.*

## Adding Text to Your Plot

Suppose you want to add some text to an existing plot. For example, consider the automobile mileage data plot in Figure 8.1. To add the text “Outliers” near the three outlying data points in the upper right hand corner of the plot,

use the `text` function. To use `text`, you specify the `x` and `y` coordinates (the same coordinate system used by the plot itself) at which you want the text to appear, and the text itself. More generally, you can specify vectors of `x` and `y` coordinates and a vector of text labels. Thus, in our example you type:

```
> plot(car.miles, car.gals)
> text(275, 22, "Outliers")
```

The text "Outliers" is *centered* on the `xy`-coordinates (275,22). You can guess the coordinate values by "eyeballing" the spot on the plot where you want the text to go. However, this approach to locating text is not very accurate, and you can do better using the `locator` function within `text`. The `locator` function allows you to use the mouse cursor to accurately identify the location of any number of points on your plot. When you use `locator`, S-PLUS waits for you to position the mouse cursor and click the left mouse button, and then it calculates the coordinates of the selected point. The argument to `locator` specifies the number of times the text is to be positioned. For example, we could have applied `text` and `locator` together as follows to obtain much the same result as before:

```
> text(locator(1), "Outliers")
```

### Connecting Text and Data Points with Straight Lines

Suppose that you want to improve the graphical presentation by drawing a straight line from the text "Outliers" to each of the three data points which you regard as outliers. You can add each such line, one at a time, with the following expression:

```
> locator(n=2, type="l")
```

S-PLUS now awaits your response. Locate the mouse cursor at the desired starting point for the line and click the left button. Move the mouse cursor to the desired ending point for the line and click the left button again. S-PLUS then draws a straight line between the two points.

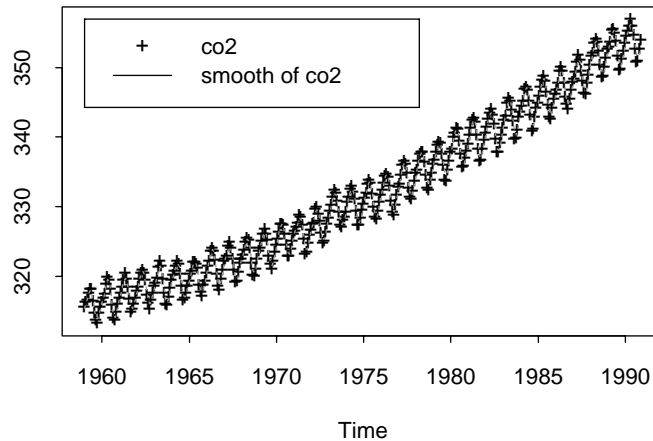
### Adding Legends

Often you make plots which contain one or more sets of data displayed with different plotting characters or line types. In such cases, you probably want to provide a legend which identifies each of the plotting characters or line types. For example, if you use

```
> ts.plot(co2, smooth(co2), type="pl", pch="+")
```

to plot the data shown in Figure 8.12, you probably want to add the legend shown in the figure. To do this, first make a vector `leg.names`, which contains the character strings "co2" and "smooth of co2", and then use `legend` as follows:

```
> leg.names <- c("co2", "smooth of co2")  
> legend(locator(1), leg.names, pch="+ ", lty=c(0, 1))
```



**Figure 8.12:** *Plot with added legend.*

S-PLUS now waits for you to respond. Move the mouse cursor to the location on the plot where you want to place the *upper left corner* of the legend box to go, then click the left mouse button.



## MAKING BAR PLOTS, DOT CHARTS AND PIE CHARTS

Bar plots and pie charts are familiar methods of graphically displaying data for oral presentations, reports and publications. In this section we show you how to use S-PLUS to make these plots. We also show you how to make another type of chart, called a dot chart, that is less widely known but often more useful than the more familiar bar plots and pie charts. We illustrate each of the above types of plots with the following 5 x 3 matrix `di gi ts`:

```
> di gi ts
      sampl e 1 sampl e 2 sampl e 3
di gi t 1      20      15      30
di gi t 2      16      17      30
di gi t 3      24      16      17
di gi t 4      21      24      20
di gi t 5      19      13      28
```

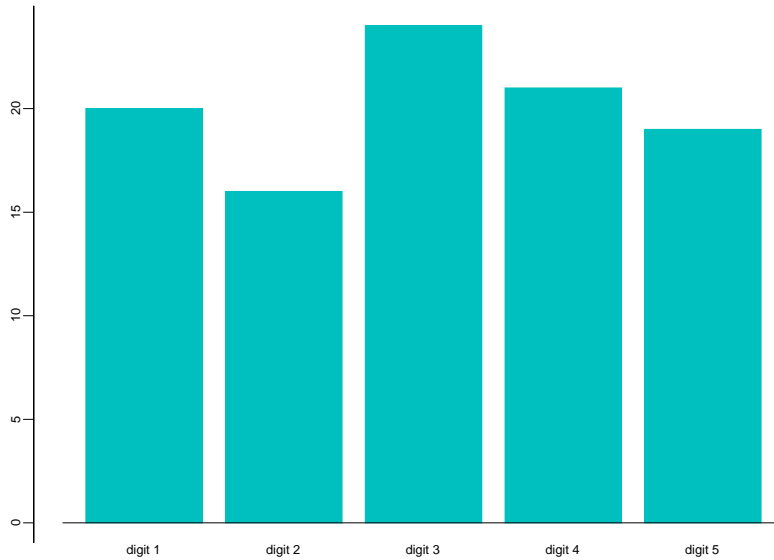
For convenience in what follows, create this matrix, and take the row labels and the column labels from this matrix as follows:

```
di gi ts <- matrix( c( 20, 15, 30, 16, 17, 30, 24, 16, 17, 21, 24, 20,
+ 19, 13, 28), nrow=5, byrow=T)
> di mnames(di gi ts) <- list( paste("di gi t", 1:5,
+ sep=" "), paste("sampl e", 1:3, sep=" "))
di gi t. names <- di mnames(di gi ts)[[1]]
sampl e. names <- di mnames(di gi ts)[[2]]
```

### Bar Plots

The function `barplot` is a flexible function for making bar plots. The simplest use of `barplot` is with a vector or a single column of a matrix. For example, using the first column of `di gi ts` gives the result in Figure 8.13:

```
> barplot(digits[, 1], names=digit.names)
```



**Figure 8.13:** *A bar plot of the **digits** data.*

In this case, the height of each bar is the value (usually a count) occurring in the corresponding component of the vector (or matrix column). To make a bar plot of the entire `digits` data matrix, use `barplot` in a more powerful way in which each bar represents a sample (i.e., a column of the matrix), and each bar is divided into a number of blocks representing the digits, with different shadings in each of the blocks. You do this as follows:

```
> barplot(digits, angle=seq(45, 135, len=5),
+ density=16, names=sample.names)
```

Using the optional argument `angle=seq(45, 135, len=5)` establishes five angles for the shading fill for each of the five blocks in each bar, with the angles equally spaced between 45 degrees and 135 degrees. Setting the `density` optional argument at the value 16 causes the shading fill lines to have a density of 16 lines per inch. If you want the density of the shading fill lines to vary cyclically, you need to set `density` at a vector value, with the vector of length five in the case of the `digits` data. For example:

```
> barplot(digits, angle=seq(45, 135, len=5),  
+ density=(1:5)*5, names=sample.names)
```

To produce a legend that associates a name to each block of bars, use the `legend` argument, with an appropriate character vector as its value. For the `digits` data example, you use `legend=digits.names` to associate a digit name with each of the blocks in the bars:

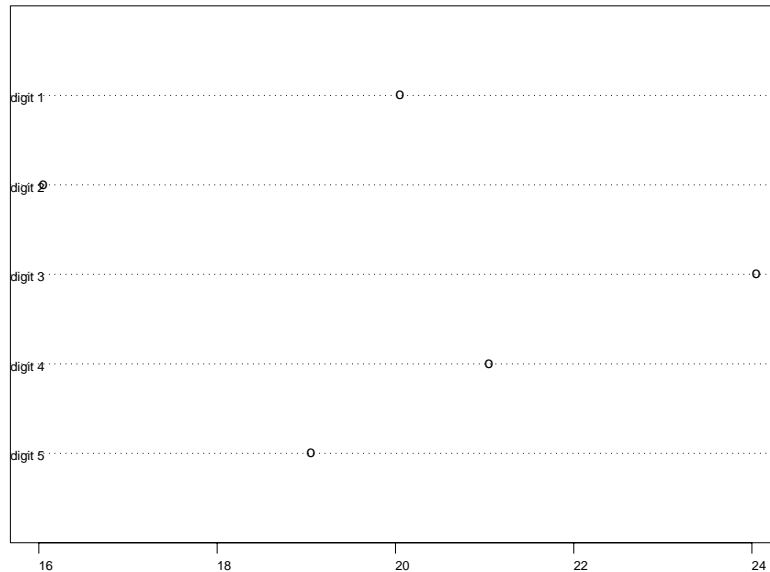
```
> barplot(digits, angle=c(45, 135), density=(1:5)*5,  
+ names=sample.names, legend=digits.names, ylim=c(0, 270))
```

To make room for the legend, you usually need to increase the range of the vertical axis, so we use `ylim=c(0, 270)`. You can obtain greater flexibility for the positioning of the legend by using the function `legend` after you have made your bar plot (rather than relying on the automatic positioning that results from using the optional argument `legend`). See the section [Adding Legends](#) (page 263) for more information. Many other options are available to you as arguments to `barplot`; see the help file for complete details.

## Dot Charts

The dot chart was first described by Cleveland (1985) as an alternative to bar plots and pie charts. The dot chart displays the same information as the bar plot or pie chart, but in a form that is often easier to grasp. In particular, the dot chart reduces most data comparisons to straightforward length comparisons on a common scale. The simplest use of `dotchart` is analogous to the simplest use of `barplot`, as you can see by applying `dotchart` to the first column of the `digits` matrix:

```
> dotchart(di gi ts[, 1], di gi t. names)
```



**Figure 8.14:** *Making dot charts with the **digits** data.*

To get a display of all the data in the matrix `di gi ts`, you could use the following command:

```
> dotchart(di gi ts, di gi t. names)
```

Or, you could use the following command:

```
> dotchart(t(di gi ts), sampl e. names)
```

The argument `t(di gi ts)` uses the function `t` to transpose the matrix `di gi ts`, i.e., to interchange the rows and columns of `di gi ts`. To get a display with both the sample labels and the digit labels, you need to create a factor object, a *grouping* variable to use as an additional argument. For example, if you wish to use the sample number as the grouping variable then create the factor object `sampl e. fac` as follows:

```
> sampl e. fac <- factor(col (di gi ts), l ab=sampl e. names)
```

and use this factor object as the third argument to `dotchart`:

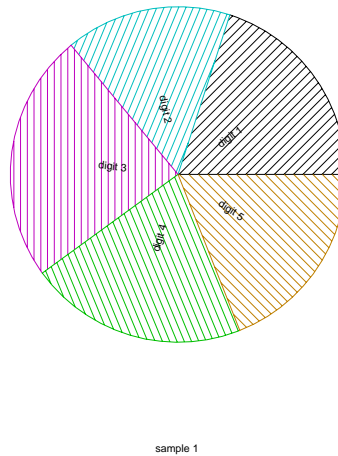
```
> dotchart(digits, digit.names, sample.fac)
```

For more information on factor objects, see Chapter 2, Data Objects. Several other options are available with the `dotchart` function; see the help file for complete details.

## Pie Charts

You can make pie charts with the function `pie`. For example, you can display the first sample of the `digits` data as a pie chart, and add the subtitle “sample 1,” by using `pie` as follows:

```
> pie(digits[, 1], names=digit.names,
+ angle=seq(45, 135, len=5), density=10, sub="sample 1")
```



**Figure 8.15:** *A pie charts of the **digits** data.*

As an alternative, try replacing `digits[, 1]` by `digits[, 2]` and `digits[, 3]`, and replacing “sample 1” by “sample 2” and “sample 3”, respectively. Several other options are available with the `pie` function; see the help file for complete details.

**Recommendation**

Although pie charts display all the information about the three samples of random digits, they are not as easy to interpret as dot charts and bar plots. Bar plots, too, introduce perceptual ambiguities, particularly in the “divided bar chart.” For these reasons, we recommend the dot chart.

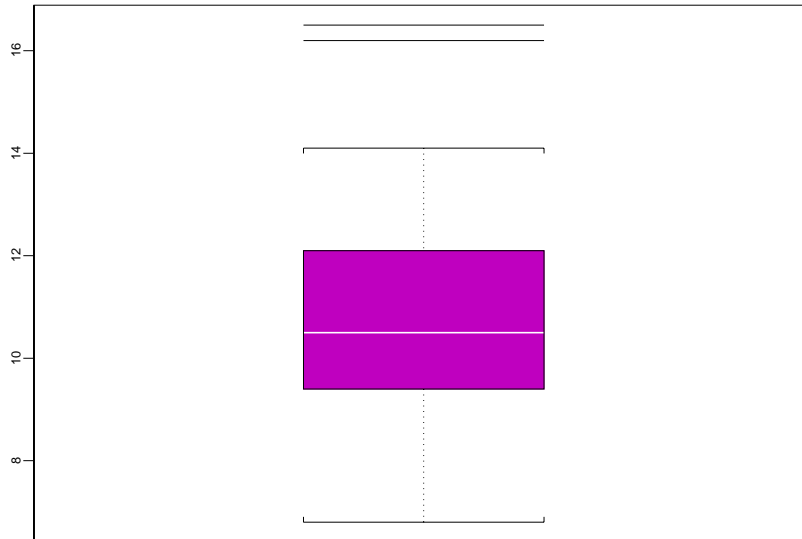
# VISUALIZING THE DISTRIBUTION OF YOUR DATA

For any data set you need to analyze, you should try to get a visual picture of the shape of its distribution. The distribution shape is readily visualized from such familiar plots as box plots, histograms, and density plots. Less familiar, but equally useful, are quantile-quantile plots (qqplots). In this section we show you how to use S-PLUS functions to make these kinds of plots.

## Box Plots

A box plot is a simple graphical representation showing the center and spread of a distribution, along with a display of unusually deviant data points, called outliers. To create a box plot in S-PLUS, you use the `boxplot` function:

```
> boxplot(corn.rain)
```



**Figure 8.16:** *Box plot from **corn.rain** data.*

The horizontal line in the interior of the box is located at the median of the data. This estimates the center of the distribution for the data. The height of

the box is equal to the *interquartile distance*, or IQD, which is the difference between the third quartile of the data and the first quartile. The IQD indicates the spread or width of the distribution for the data. The whiskers (the dotted lines extending from the top and bottom of the box) extend to the extreme values of the data or a distance  $1.5 \times \text{IQD}$  from the center, whichever is less. For data having a Gaussian distribution, approximately 99.3% of the data falls inside the whiskers. Data points which fall outside the whiskers may be outliers, and so they are indicated by horizontal lines. In our example, the two horizontal lines at the top of the graph represent outliers. Box plots provide a very powerful method for visualizing the rough distributional shape of two or more samples of data.

For example, to compare the distributions of the New Jersey lottery payoffs `lottery.payoff`, `lottery2.payoff`, and `lottery3.payoff` in each of three different years, use

```
> boxplot(lottery.payoff, lottery2.payoff, lottery3.payoff)
```

You can modify the style of your box plots, and many other features as well, using arguments to `boxplot`; see the help file for complete details.

## Histograms

A histogram shows the number of data points that fall in each of a number of intervals. You create histograms in S-PLUS with the `hist` function:

```
> hist(corn.rai n)
```

Notice that a histogram gives you an indication of the relative density of the data points along the horizontal axis. For example, there are 10 data points in the interval 8 to 10, and only one data point in the interval 14 to 16. The histogram produced by the above simple use of `hist` always spans the range of the data, i.e., the smallest data value falls in the leftmost interval and the largest data point falls in the rightmost interval.

The number of intervals produced by `hist`, e.g., six intervals in the above example, is determined automatically by `hist` to balance the tradeoff between obtaining smoothness and preserving detail. However, no automatic rule is completely satisfactory. Thus, `hist` allows you to choose the number of intervals yourself, by using the optional argument `nclass`. Choosing a larger number of intervals produces a “rougher” histogram with more detail, and choosing a smaller number produces a “smoother” histogram with less detail. For example:

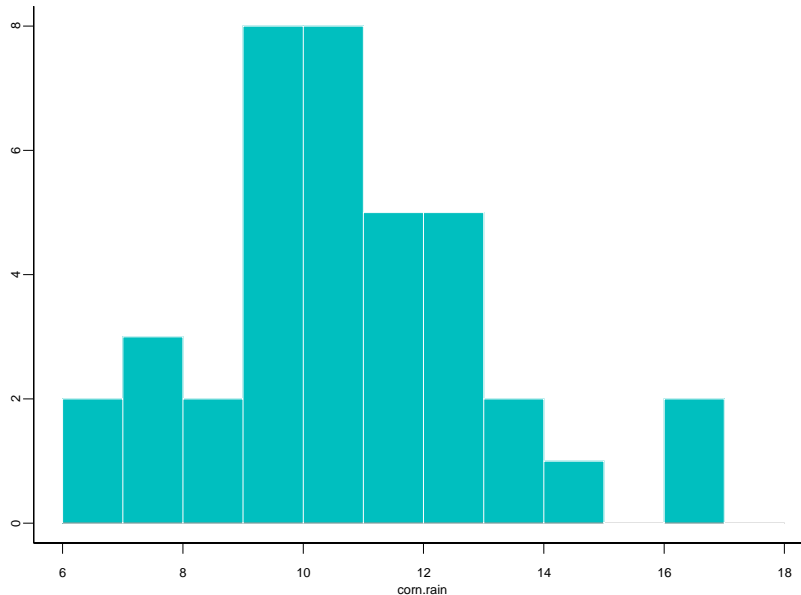
```
> hist(corn.rai n, nclass=10)
```

gives the rougher but more detailed histogram.



You can also use `hist` to make a histogram in which you specify the number of intervals and their locations. You do this by using the optional argument `breaks`, with value a vector whose values give the interval boundary points. The length of this vector is one plus the number of intervals you want. For example, to specify 12 intervals for the `corn.rain` histogram, with interval boundaries at the integers 6 through 18, use

```
> hist(corn.rain, breaks=6:18)
```



**Figure 8.17:** *Histogram of `corn.rain` with specified break points.*

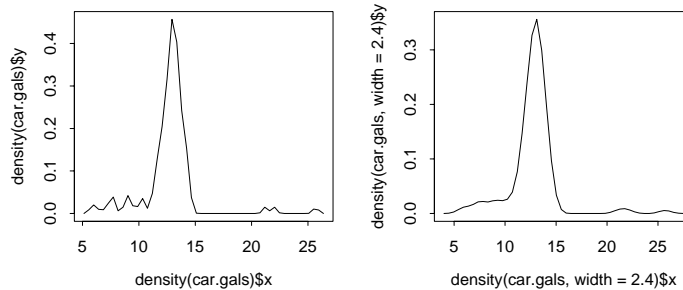
Many other options are available with `hist`, including many of the arguments to `barplot`. See the help files for `hist` and `barplot` for complete details.

## Density Plots

A histogram for continuous numeric data is a rough estimate of a smooth underlying (population) density curve which gives the relative frequency with which the data falls in different intervals. This underlying density curve, which is formally called a probability density function, allows you to compute the probability that your data falls in any interval. Thus you may

prefer a smooth estimate of this density to a rough histogram estimate. To get such a smooth density estimate in S-PLUS, use `plot` with the function `density`. The optional argument `width` controls the smoothness of the plot. For example:

```
> plot(density(car.gals), type="l")
> plot(density(car.gals, width=2.4), type="l")
```



**Figure 8.18:** *Probability density plots.*

The default value for `width` results in a somewhat rough density estimate in the tail, whereas the choice `width=2.4` produces a smoother density estimate. The value 2.4 in the second plot is obtained by applying the choice `width=2*iqd` to the `car.gals` data, where `iqd` is the interquartile distance. You can obtain the IQD from `summary` by subtracting the value 1st Qu. from the value 3rd Qu.:

```
> summary(car.gals)

Min.   1st Qu.   Median   Mean   3rd Qu.   Max.
 5.80    12.30    13.00  12.72    13.50   25.70
```

Here,  $\text{IQD} = 13.50 - 12.30 = 1.20$ .

A width of twice the interquartile distance generally gives a smooth plot, but may obscure local details of the density. On the other hand, rougher density estimates may highlight random effects. See Silverman (1986) for a discussion of the issues involved in choosing a width parameter.

## Quantile-Quantile Plots

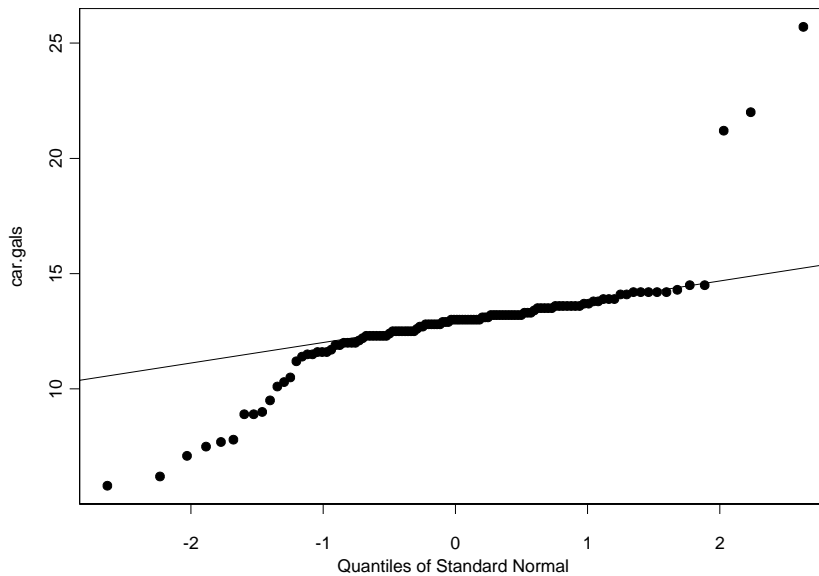
A quantile-quantile plot, or `qqplot`, is a plot of one set of quantiles against another set of quantiles. There are two main forms of `qqplots`. The most frequently used form checks whether a data set comes from a particular hypothesized distribution shape. In this case, one set of quantiles consists of

the ordered set of data values (which are in fact quantiles for the empirical distribution for the data), and the other set of quantiles are quantiles for your hypothesized distribution. If the points in this plot cluster along a straight line, the data set probably has the hypothesized distribution. The second form of qqplot is used when you want to find out whether two data sets have the same distribution shape. If the points in this plot cluster along a straight line, the two data sets probably have the same distribution shape.

### QQplots for Checking Distribution Shape

To produce the first type of qqplot when your hypothesized distribution is normal, use the function `qqnorm`:

```
> qqnorm(car.gals)
> qqline(car.gals)
```



**Figure 8.19:** A *qqnorm* plot.

The `qqline` function gives the highly robust straight line fit, which is not much influenced by outliers. You can also make qqplots to check whether or not your data comes from any of a number of other distributions. To do so, you need to create a simple S-PLUS function for each distribution, which we illustrate for the case of a hypothesized uniform distribution. Create the function `qqunif` as follows:

```
> qqunif <- function(x){ plot(qunif(ppoints(x)), sort(x)) }
```

The function `qunif` computes quantiles for the uniform distribution at probability values  $pi=(i-.5)n$  computed by `ppoints`, and `sort` orders the data `x`.

```
> qqunif(car.gals)
```

Now you can create a `qqplot` for other hypothesized distributions by replacing `qunif` by one of the functions from Table 8.2.

### QQplots for Comparing Two Sets of Data

When you want to check whether two sets of data have the same distribution, use the function `qqplot`. If the two data sets have the same number of observations, `qqplot` plots the ordered data values of one data set versus the ordered data values of the other data set. If the two data sets have different numbers of observations, then the ordered data values for one data set are plotted against interpolates of the ordered values of the other data set.

For example, to compare the distributions of the two New Jersey lottery data sets `lottery.payoff` and `lottery3.payoff`, use the following expression:

```
> qqplot(lottery.payoff, lottery3.payoff)
```

**Table 8.2:** *Distributions for qqplots.*

Function	Distribution	Required Arguments	Optional Arguments	Defaults
<code>qbeta</code>	beta	<code>shape1, shape2</code>	<code>none</code>	
<code>qcauchy</code>	Cauchy	<code>none</code>	<code>location, scale</code>	<code>0, 1</code>
<code>qchisq</code>	chi-square	<code>df</code>	<code>none</code>	
<code>qexp</code>	exponential	<code>none</code>	<code>rate</code>	<code>1</code>
<code>qf</code>	F	<code>df1, df2</code>	<code>none</code>	
<code>qgamma</code>	Gamma	<code>shape</code>	<code>none</code>	
<code>qlnorm</code>	log-normal	<code>none</code>	<code>mean, sd</code>	<code>0, 1</code>
<code>qnorm</code>	normal	<code>none</code>	<code>mean, sd</code>	<code>0, 1</code>
<code>qt</code>	Student's t	<code>df</code>	<code>none</code>	
<code>qunif</code>	uniform	<code>none</code>	<code>min, max</code>	<code>0, 1</code>

**Note**

For functions which require a parameter argument, you must allow your qqplot function to pass the required argument. For example, you create qqchi sq as follows:

```
> qqchi sq<-function(x, df) { plot(qchi sq(ppoi nts(x), df), sort(x)) }
```

## VISUALIZING HIGHER DIMENSIONAL DATA

For data with three or more variables, many methods of graphical visualization have been developed. Some of these are highly interactive, and take full advantage of the power of personal computers. The following sections describe how to use S-PLUS functions in analyzing multi-dimensional data.

### Multivariate Data Plots

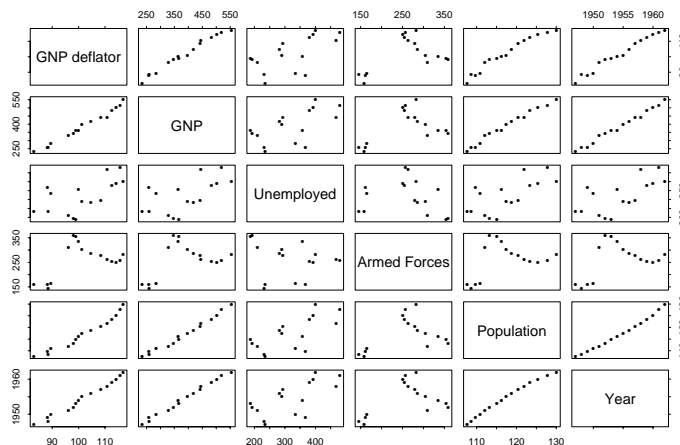
This section describes several methods for *static* data visualization that are widely considered useful: scatterplot matrices, matplots, star plots, and Chernoff's faces.

### Scatterplot Matrices

A *scatterplot matrix* is an array of pairwise scatter plots showing the relationship between any pair of variables in a multivariate data set. To produce a static scatterplot matrix in S-PLUS, you use the `pairs` function with an appropriate data object as its argument.

For example, the following S-PLUS expression generates a scatterplot matrix:

```
> pairs(longley.x)
```



**Figure 8.20:** A scatterplot matrix.

## Plotting Matrix Data

For visualizing several vector data objects at once or for visualizing some kinds of multivariate data, you can use the function `matplot` to plot columns of one matrix against the columns of another.

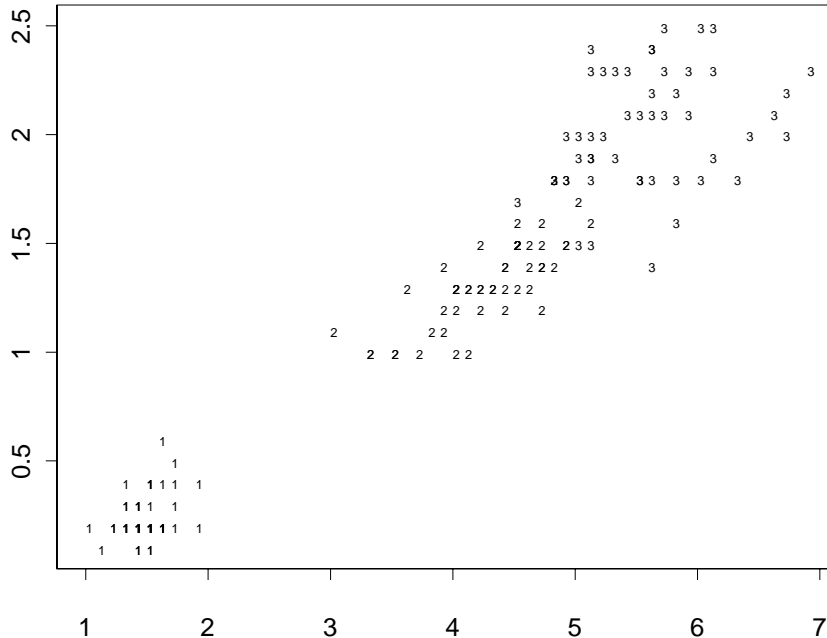
For example, S-PLUS has a built-in multivariate data set `iris`. The `iris` data is in the form of a data *array*, which is a generalized matrix. Let's extract two particular  $50 \times 3$  matrices from the `iris` array:

```
> pet.length <- iris[, 3, ]
> pet.width <- iris[, 4, ]
```

The matrix `pet.length` contains 50 observations (the rows) of petal lengths for each of three species of iris (the columns): Setosa, Versicolor and Virginica. The matrix `pet.width` contains 50 observations of petal widths for each of the same three species.

To graphically explore the relationship between petal lengths and petal widths, use `matplot` to display widths versus lengths simultaneously on a single plot:

```
> matplot(pet.length, pet.width)
```



**Figure 8.21:** *Simultaneous plots of petal heights versus widths for three species of iris.*

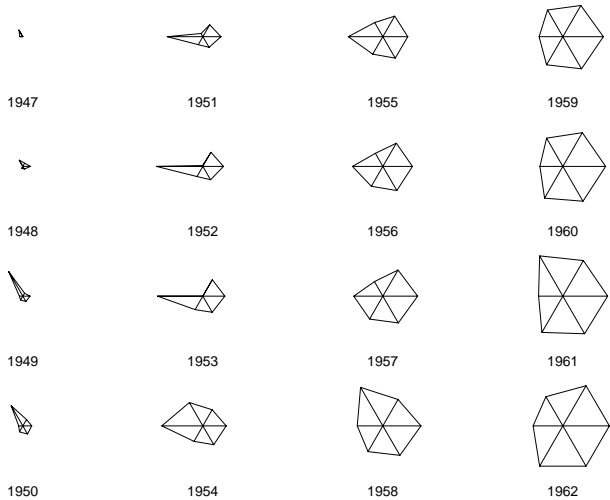
If the matrices  $x$  and  $y$  you are plotting with `matplot` do not have the same number of columns, then the columns of the smaller matrix are cycled so that every column in the larger matrix is plotted. Thus, if  $x$  is a vector, i.e., a matrix with a single column, then `matplot(x, y)` plots every column of the matrix  $y$  against the vector  $x$ .

# Star Plots

A *star plot* represents multivariate data as a set of stars, with each star representing one case, or row, and each point (or *radial*) of a star representing a particular variable, or column. The length of each radial is proportional to the data value of the corresponding variable. Thus, both the size and the shape of the stars have meaning: size reflects the overall magnitude of the data, and shape reveals the relationships between variables. Comparing two stars gives a quick graphical picture of similarities and differences between two cases—similarly shaped stars indicate similar cases.

For example, to create a star plot from the data used to create our scatterplot matrix:

```
> stars(longley, x)
```



**Figure 8.22:** *A star plot.*

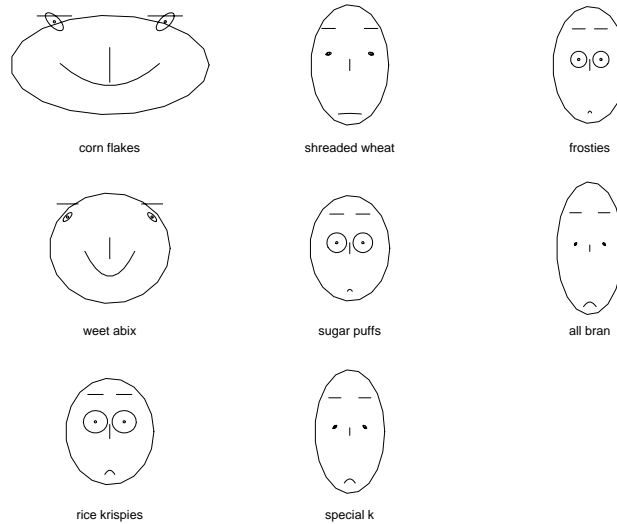
# Faces

Chernoff introduced the idea of using faces to represent multivariate observations. Each variable in a given observation is associated to one feature



of the face. Two cases can be compared using a feature-by-feature comparison. You can create Chernoff's faces with the S-PLUS `faces` function:

```
> faces(t(cereal.attitude), labels =  
+ dimnames(cereal.attitude)[[2]], ncol=3)
```



**Figure 8.23:** *A faces plot.*

See the `faces` help file and Chernoff (1973) for complete details on interpreting Chernoff faces.

## 3-D PLOTS: CONTOUR, PERSPECTIVE, AND IMAGE PLOTS

Many types of data are usefully viewed as *surfaces* generated by functions of two variables. Familiar examples are meteorological data, topographic data, and other data gathered by geographical location.

S-PLUS provides three functions for viewing such data. The simplest, `contour`, represents the surface as a set of contour plots lines on a grid representing the other two variables. The perspective plot, `persp`, creates a perspective plot with hidden line removal. The `image` function plots the surface as a color or grayscale variation on the base grid.

All three functions require similar input—a vector of  $x$  coordinates, a vector of  $y$  coordinates, and a length  $x$  by length  $y$  matrix of  $z$  values. In many cases, these arguments are all supplied by a single list, such as the output of the `interp` function. The `interp` function *interpolates* the value of the third variable onto an evenly spaced grid of the first two variables. For example, the built-in data set `ozone` contains the objects `ozone.xy`, a list of latitudes and longitudes for each observation site, and `ozone.median`, a vector of the medians of daily maxima ozone concentrations at all sites. To create a contour or perspective plot, we can use `interp` to interpolate the data as follows:

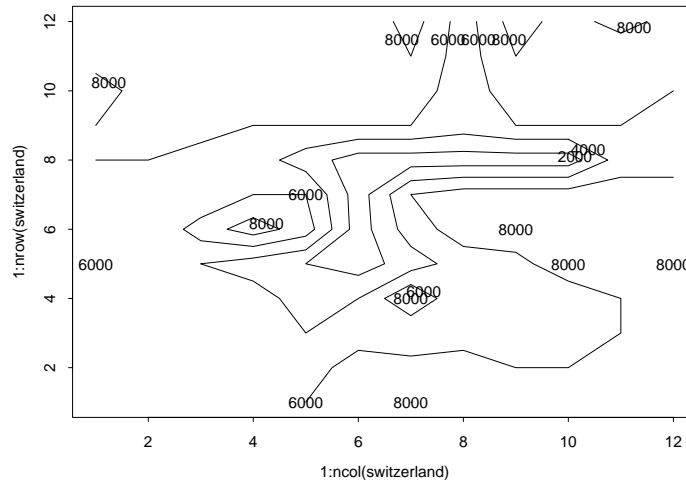
```
ozone.fi t<-i nterp(ozone.xy$x, ozone.xy$y, ozone.medi an)
```

For `contour` and `persp`, but not `image`, you can also provide a single matrix argument, which `contour` and `persp` interpret as the  $z$  matrix. The two functions then automatically generate an  $x$  vector `1:nrow(z)` and a  $y$  vector `1:ncol(z)`. See the `persp` and `contour` help files for more information.

### Contour Plots

To generate a contour plot, use the `contour` function. For example, the built-in data set `swi tzerl` and contains elevation data for Switzerland.

```
> contour(swi tzerl and)
```



**Figure 8.24:** *Contour plot of Switzerland.*

By default, `contour` draws contour lines for each of five levels, and labels each one. You can change the number of levels with either the `nlevels` or the `levels` argument. The `nlevels` argument specifies the approximate number of contour intervals desired, while `levels` specifies a vector of heights for the contour lines.

You control the size of the labels for the contour lines with the `labex` argument. You specify the size as a relative value to the current axis-label font, so that `labex=1` (the default) yields labels which are the same size as the axis labels. Setting `labex=0` gives you unlabeled contour lines.

For example, to view a voice spectrogram for the word “five,” use `contour` on the built-in data object `voic.five`. Because `voic.five` generates many contour lines, we suppress the labels with `labex=0`:

```
> contour(voic.five, labex=0)
```

If you have an equal number of observations for each of three variables, you can use `interp` to generate interpolated values for  $z$  on an equally-spaced  $xy$  grid. For example, to create a contour plot of the ozone data, you can use `interp` and `contour` as follows:

```
> ozone.fitted <- interp(ozone.xy$x, ozone.xy$y, ozone.median)
> contour(ozone.fitted)
```

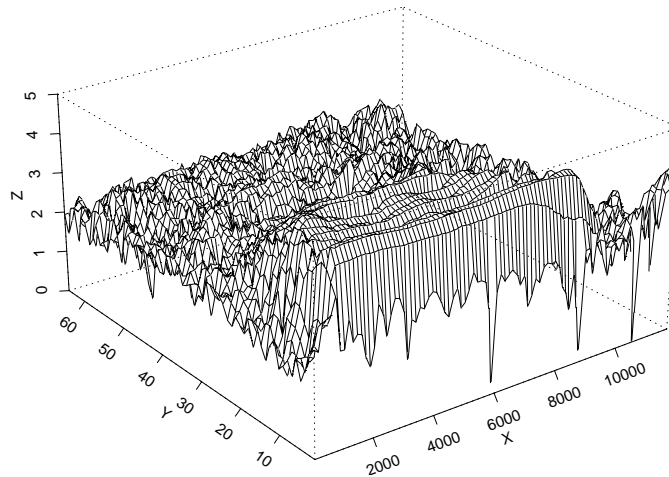
## Perspective Plots

Perspective plots give a three-dimensional view of data in the form of a matrix of heights on an evenly spaced grid. The heights are connected by line segments to produce the familiar mesh appearance of such plots.

As a simple example, consider again the voice spectrogram for the word “five.” The contour plot of the voice data was difficult to interpret because the number of contour lines forced us to omit the height labels. Had we included the labels, the clutter of labels would have made the graph unreadable.

The perspective plot in Figure 8.25 gives a much clearer view of how the spectrogram varies. To create the plot function, use the following S-PLUS expression:

```
> persp(voi ce. fi ve)
```



**Figure 8.25:** *Perspective plot of a voice spectrogram.*

You can modify the perspective by choosing a different “eye” location. You do this with the `eye` argument. By default, the eye is located  $c(-6, -8, 5)$  times the range of the  $x$ ,  $y$ , and  $z$  values. For example, to look at the voice data from “the other side,” we could use the following command:

```
> persp(voi ce. fi ve, eye=c(72000, 350, 30))
```

If you have an equal number of observations for each of three variables, you can use `interp` to generate interpolated values for  $z$  on an equally-spaced  $xy$  grid. For example, to create a perspective plot of the ozone data, you can use `interp` and `persp` as follows:

```
> ozone.fitted <- interp(ozone.xy$x, ozone.xy$y, ozone.median)
> persp(ozone.fitted)
```

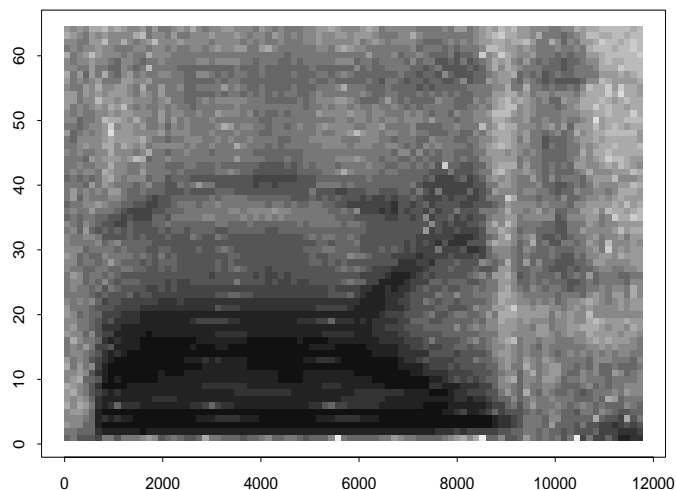
### Warning

It is not a good idea to convert a `persp` plot to objects; so many objects can result that the conversion takes a considerable time.

## Image Plots

An image plot is a two-dimensional plot that represents three-dimensional data as shades of color or gray-scale. You produce image plots with the `image` function:

```
> image(voice.five)
```

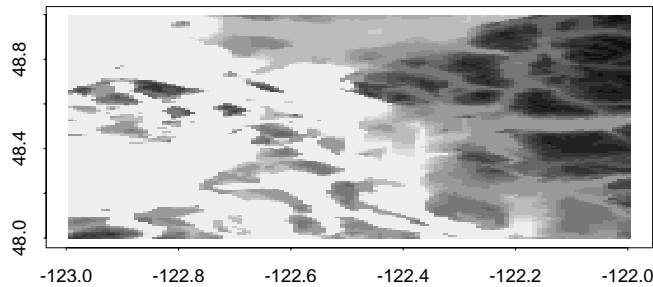


**Figure 8.26:** *Image of the voice spectrogram.*

A more conventional use of `image` is to produce images of topological data, as in the following example:

```
> image(pugetN)
```

The data set `pugetN` contains elevations in and around Puget Sound. It is not part of the standard S-PLUS distribution.



**Figure 8.27:** *Image plot of Puget Sound.*

If you have an equal number of observations for each of three variables, you can use `interp` to generate interpolated values for  $z$  on an equally-spaced  $xy$  grid. For example, to create an image plot of the ozone data, you can use `interp` and `image` as follows:

```
> ozone.fi t <- i nterp(ozone.xy$x, ozone.xy$y, ozone.medi an)
> i mage(ozone.fi t)
```

---

## CUSTOMIZING YOUR GRAPHICS

For most exploratory data analysis, the complete graphics created by S-PLUS, with their automatically generated axes, tick marks, and axis labels, serve your needs well. Most of the graphics described in the previous sections, were created with one-step functions such as `plot` and `hist`. These one-step functions are called *high-level graphics functions*. If you are preparing graphics for publication or a presentation, you need more control over the graphics that S-PLUS produces.

The following sections describe how to customize and fine-tune your S-PLUS graphics with *low-level graphics functions* and *graphics parameters*. Low-level graphics functions do not generate a complete graphic, but rather one specific part of a graphic. Graphics parameters control the details of the graphics that are produced by the graphics functions, including where the graphics appear on the graphics device. The customization features described here refer to traditional graphics, using the editable graphics described in Chapter 7 may be the preferable method of performing customization.

Many of the examples in this chapter use the following data:

```
> set.seed(12)
> x <- runif(12)
> y <- rnorm(12)
```

If you use these statements, you will be able to reproduce exactly the plots that use `x` and `y`. We also use the following data from the built-in data set `auto.stats`:

```
> price <- auto.stats[, "Price"]
> mileage <- auto.stats[, 2]
```

## LOW-LEVEL GRAPHICS FUNCTIONS AND GRAPHICS PARAMETERS

The section *Frequently Used Plotting Options* (page 248) introduced several low-level graphics functions, including `points`, which adds a scatter of points to an existing plot, and `abline`, which adds a specified line to an existing plot. Low-level graphics functions, unlike high-level graphics functions, do not automatically generate a new coordinate system. Thus, you can use several low-level graphics functions in succession to create a single finished graphic.

Some functions, such as `image` and `contour`, which are described in the section *3-D Plots: Contour, Perspective, and Image Plots* (page 282), can be used as either high- or low-level graphics functions.

*Graphics parameters* add to the flexibility of graphics, by controlling virtually every detail of a page of graphics. There are about 60 parameters, which fall into four classes:

- *High-level* graphics parameters can be used only as arguments to high-level graphics functions. An example is `xlim`, which gives the approximate limits for the *x*-axis.
- *Layout* graphics parameters can be set only with the `par` function. These parameters typically affect quantities that concern the page as a whole. The `mflow` parameter is an example; this states how many rows and how many columns of plots are placed on a single page.
- *General* graphics parameters may be set either in a call to a graphics function, or with the `par` function. When used in a graphics function, the change is valid only for that function call. If you set a parameter with `par`, the change lasts until you change it again. Graphics parameters are initialized whenever a graphics device is started; a change via `par` applies only to the *current* device. (You can write your own `Device.Default` function to have one or more parameters set automatically when you start a graphics device—see the `Device.Default` help file.)
- *Information* parameters give information about the state of the device, but may not be changed directly by the user. An example is `din`, the size of the current device in inches. See the `par` help file for descriptions of the information parameters.



The arguments to `title(main, sub, xlab, and ylab)`, while not graphics parameters, are quite similar to them. They are accepted as arguments by several graphics functions as well as the `title` function.

Table 8.10 (on page 327) summarizes the S-PLUS graphics parameters.

<b>Warning</b>
Some graphics functions do not recognize certain high-level or general graphics parameters. The help files for these functions describe which graphics parameters the functions will accept.

## SETTING AND VIEWING GRAPHICS PARAMETERS

There are two ways to set graphics parameters:

1. Use the *name=value* form either within a graphics function call or with the `par` function. For example:

```
> par(mfrow=c(2,1), cex=.5)
> plot(x, y, pch=17)
> plot(price, mileage, log="y")
```

Note that you can set several graphics parameters simultaneously in a single call to `par`.

2. Supply a list to the `par` function. The names of the list components are the names of the graphics parameters you want to set. For example,

```
> my.list <- list(mfrow=c(2,1), cex=.5)
> par(my.list)
```

When you change graphics parameters with `par`, it returns a list containing the *original* values of the graphics parameters that you changed. This list will not print out on your screen; you must assign the result of calling `par` to a variable name if you want to see it:

```
> graphicsheet()
> par.orig <- par(mfrow=c(2,1), cex=.5)
> par.orig
```

```
$mfrow:
[1] 1 1
$cex:
[1] 1
```

You can use this list returned by `par` to restore parameters after you have changed them:

```
> graphicsheet()
> par.orig <- par(mfrow=c(2,1), cex=.5)
> # Now make some plots
> par(par.orig)
```

When setting multiple parameters with `par`, check for possible interactions between parameters. Such interactions are indicated in Table 8.3 and in the `par` help file. In a single call to `par`, general graphics parameters are set first,

then layout graphics parameters. If a layout graphics parameter affects the value of a general graphics parameter, what you specify for the general graphics parameter may get overridden. For example, changing `mfrow` automatically resets `cex` (see the section Controlling Multiple Plots (page 308)). If you type

```
> par(mfrow=c(2, 1), cex=. 75)
```

**Table 8.3:** *Interaction between graphics parameters.*

Parameters	Interaction
<code>cex</code> , <code>mex</code> , <code>mfrow</code> , <code>mfcol</code>	If <code>mfrow</code> or <code>mfcol</code> specify a layout with more than two rows or columns, <code>cex</code> and <code>mex</code> are set to 0.5, otherwise <code>cex</code> and <code>mex</code> are both set to 1.
<code>crt</code> , <code>srt</code>	When <code>srt</code> is set, <code>crt</code> is set to the same value, unless <code>crt</code> appears later in the command than <code>srt</code> .

S-PLUS will first set `cex=. 75` (because `cex` is a general graphics parameter), then set `mfrow=c(2, 1)` (because `mfrow` is a layout graphics parameter), but setting `mfrow=c(2, 1)` automatically sets `cex` back to 1. To set both `mfrow` and `cex`, you need to call `par` twice:

```
> par(mfrow=c(2, 1))
> par(cex=. 75)
```

You can also use the `par` function to view the current setting of any or all graphics parameters. To view the current values of parameters give `par` a vector of character strings of the names of the parameters:

```
> par("usr")
```

or

```
> par(c("mfrow", "cex"))
```

To get a list of all of the parameters call `par` with no arguments:

```
> par()
```

During an extended S-PLUS session, you may make repeated calls to `par` to change graphics parameters. Sometimes, you may forget what you have changed and may just want to restore the device to its original defaults. It is often a good idea to save the original values of the graphics parameters as soon as you start a device. You can then call `par` to restore the device to its original state:

```

> graphsheet()
> par.orig.wg <- par()
> par(mfrow=c(3,1), col=4, lty=2)
> # create some plots
> # several more calls to par
> par(par.orig.wg)

```

### Warning

When a device is first started, before any plots are produced, the graphics parameter `new` is set equal to `T`. In this case, a call to a high-level graphics function will not clear the device before putting up a new plot (see the section *Overlaying Figures* (page 311)). Thus, if you follow the above commands to restore all graphics parameters to their original state, you need to call `frame` before issuing the next plotting command.

Separate sets of graphics parameters are maintained for each active graphics device. When you change graphics parameters with the `par` function, you are changing their value only for the *current* graphics device. For example, if you have both a `graphsheet` and a `postscript` graphics device active, and the `postscript` device is the current device, then calling `par` to change graphics parameters will affect only the graphics parameters for the `postscript` device:

```

> graphsheet()
> postscript()
> dev.list()
graphsheet postscript
      2      3
> dev.cur()
postscript
      3
> par(mfrow=c(2,2))
> par("mfrow")

[1] 2 2

> dev.set()
graphsheet
      2
> par("mfrow")

[1] 1 1

```

## CONTROLLING GRAPHICS REGIONS

The location and size of a figure are determined by parameters that control *graphics regions*. The surface of any graphics device can be divided into two regions: the *outer margin* and the *figure region*. The figure region contains one or more *figures*, each of which is composed of a *plot area* (or region) surrounded by a *margin*. By default, a device is initialized with one figure and the outer margin has zero area; that is, typically there is just a plot area surrounded by a margin.

The plot area is where the data is shown. In the typical plot, the axis line is drawn on the boundary between the plot area and the margin. Each margin, whether the outer margin or a figure margin, is divided into four parts, as shown in Figure 8.28: bottom (side 1), left (side 2), top (side 3) and right (side 4).



**Figure 8.28:** *The four sides of a margin.*

You can change the size of any of the regions. Changing one area causes S-PLUS to automatically resize the regions within and surrounding the one that you have changed. For example, when you specify the size of a figure, the margin size is subtracted from the figure size to obtain the size of the plot area—S-PLUS does not allow a figure with a margin that takes more room than the figure.

Most often, you change the size of regions with the `mfrow` or `mfcol` layout parameters—when you specify the number of rows and columns, S-PLUS automatically determines the appropriate figure size. To control region size explicitly, work your way inward by specifying first the outer margins and then the figure margins.

## Controlling the Outer Margin

You usually specify an outer margin only when creating multiple figures per page. You can use the outer margin to hold a title for an entire page of plots or to label different pages consistently when some pages have multiple plots and others have a single plot.

You must specify a size for the outer margin if you want one—the default size is 0. To specify the size of the outer margin, use any one of three equivalent layout parameters: `oma`, `omi`, or `omd`.

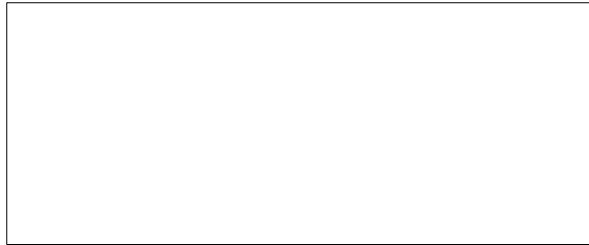
The most useful of these is `oma`, specified as a numeric vector of length four (one element for each side), where the values are expressed in `mex` (the size of the font for one line of text in the margins). If you specify the outer margin with `oma`, the specified values correspond to the number of lines of text that will fit in each margin. For example, to leave room for a title at the top of a page of plots, we could set the outer margin as follows:

```
> par(oma=c(0, 0, 5, 0))
```

You can then use `mtext` as follows to add a title, to obtain Figure 8.29:

```
> mtext("A Title in the Outer Margin", side=3,  
+ outer=T, cex=1.5)  
> box()
```

## A Title in the Outer Margin



**Figure 8.29:** *A plot with an outer margin.*

Setting the parameter `oma` automatically changes both `omi` (the outer margin in inches) and `omd` (the outer margin as a fraction of the device surface). See the `par` help file for more information on `omi` and `omd`.

### Warning

If you set `oma` to something other than the default value `c(0, 0, 0, 0)` and then later reset *all* of the graphics parameters in a call to `par` (e.g., `par(ori g. par)`), you will see the warning message:

Warning messages:

Graphics error: Figure specified in inches too large (in `zzfigz`) in:...

This message can be safely ignored.

## Controlling Figure Margins

To specify the size of the figure margins, use one of two equivalent graphics layout parameters: `mar` or `mai`. The `mar` parameter, specified as a numeric vector of length four with values expressed in `mex`, is generally the more useful of the two, because it can be used to specify *relative* margin sizes. The `mai` parameter measures the size of each side of the margin in inches, and is thus useful for specifying *absolute* margin sizes. If, for example, `mex` is 1 (the default) and `mar` equals `c(5, 5, 5, 5)`, there is room for five lines of default-font text (`cex=1`) in each margin. If `mex` is 2 and `mar` is `c(5, 5, 5, 5)`, there is room for 10 lines of default-font text in each margin.

The `mex` parameter specifies the size of font that is to be used to measure the margins. When you change `mex`, S-PLUS automatically resets some margin parameters to decrease the size of the figure margins to correspond to smaller text without changing the size of the outer margin. Table 8.4 shows the effects on the various margin parameters of a change in `mex` from 1 to 2.

**Table 8.4:** *Effect of changing `mex`.*

Parameter	<code>mex=1</code>	<code>mex=2</code>
<code>mar</code>	5.1 4.1 4.1 2.1	5.1 4.1 4.1 2.1
<code>mai</code>	0.714 0.574 0.574 0.294	1.428 1.148 1.148 0.588
<code>oma</code>	0 0 5 0	0.0 0.0 2.5 0.0
<code>omi</code>	0.000 0.000 0.699 0.000	0.000 0.000 0.699 0.000

From the table, we see that an increase in `mex` leaves `mar` and `omi` unchanged, while `mai` is increased and `oma` is decreased. When you shrink margins with `mar`, be sure to check the `mgp` parameter which determines where axis and tick labels are placed; if the margins don't provide room for those labels, the labels are not printed and you receive a warning from S-PLUS.

## Controlling the Plot Area

To determine the *shape* of the plot use the `pty` layout graphics parameter ("`plot type`"). The `pty` parameter has two possible values: "`m`" for maximal and "`s`" for square. By default, plots fill the entire space allowed for the `plot` (`pty="m"`). Another way to control the shape of a plot is with `pin`, which gives the width and height of the plot in inches.



---

## CONTROLLING TEXT IN GRAPHICS

The section Interactively Adding Information to Your Plot (page 259) described how to add text and legends to existing plots. This section describes how to control the size of text and plotting symbols, the placement of text within the plot area, and the width of lines in the plot area.

### Controlling Text and Symbol Size

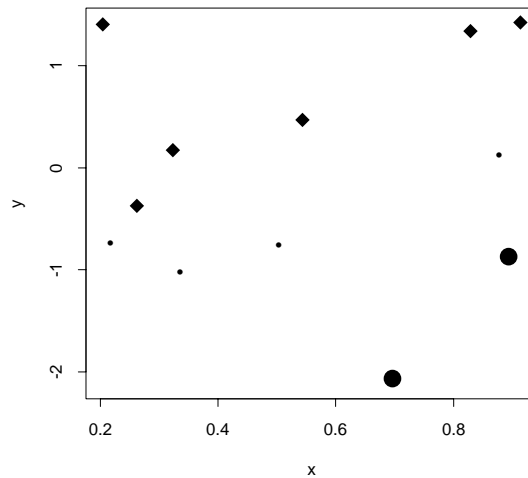
The size of text and most plotting symbols is controlled by the general graphics parameter `cex` (character expansion). The *expansion* refers to expansion with respect to the graphics device's default font. By default, `cex` is set to 1, so graphics text and symbols appear in the default font size. When `cex=2`, text appears at twice the default font size. Some devices, however, have only a few fonts available, so that all values of `cex` in a certain range produce the same font. See the online help for information on how to control available fonts on your display device.

Many graphics functions and parameters use or modify `cex`. For example, main titles are written with a `cex` of 1.5 times the current `cex`. The `mfrow` parameter sets `cex` to 1 for a small number of plots (fewer than three per row or column), but sets it to 0.5 for a larger number of plots.

The `cex` parameter controls the size of plotting symbols. Plotting symbols of various sizes can be shown on a single figure, as shown in Figure 8.30, which shows how symbols of different sizes can be used to highlight groups of data. Figure 8.30 is produced with the following expressions:

```
> plot(x, y)
> points(x[x-y>2*median(x-y)], y[x-y>2*median(x-y)], cex=2)
```

```
> points(x[x-y<median(x-y)],
+ y[x-y<median(x-y)], pch=18, cex=2)
```



**Figure 8.30:** *Symbols of different sizes.*

A parameter equivalent to `cex` is `csi`, which gives the height (interline space) of text with the current `cex` measured in inches. Changing either `cex` or `csi` changes the other. The `csi` parameter is useful when creating the same graphics on different devices since the absolute size of graphics is device dependent.

## Controlling Text Placement

When you add text to the plot area, you specify its coordinates in terms of the plotted data—in essence, S-PLUS treats the added text as a data point. If axes have been drawn and labeled, you can read the coordinates off the plot. If not, you can obtain the desired coordinates by interpolating from the values in the layout parameter `usr`. For example, Figure 8.30 has an *x*-axis with values from 0 to 1 and a *y*-axis with values running from approximately -2.5 to 1. To add the text “Different size symbols” we could specify any point within the grid determined by these *x* and *y* limits, as follows:

```
> text(.4, .7, "Different size symbols")
```

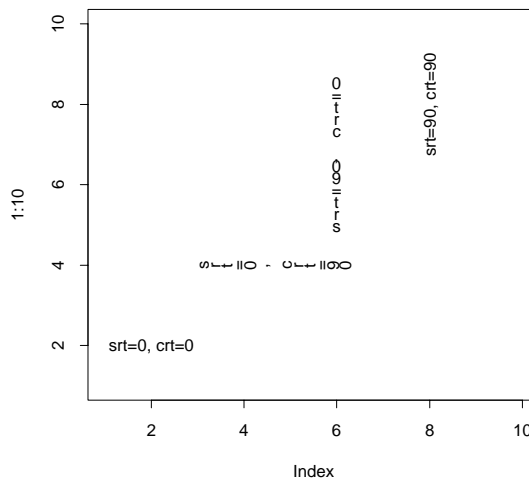
By default, the text is centered at the specified point. However, you can left- or right-justify the text at the specified point by using the general parameter `adj`. The `adj` parameter determines the fraction of the text string that appears to the left of the specified *xy*-coordinate. The default is 0.5. Set `adj=0` to left-justify, `adj=1` to right-justify.

If no axes have been drawn and you can't determine the coordinates by looking at your graphic, you can obtain the desired coordinates by interpolating from the values in the layout parameter `usr`. The `usr` parameter gives the minimum and maximum of the  $x$  and  $y$  coordinates.

## Controlling Text Orientation

Two graphics parameters, `crt` (character rotation) and `srt` (string rotation) control the orientation of text in the plot region and the figure and outer margins. Figure 8.31 shows the result of typing the following commands after starting a postscript device:

```
> plot(1:10, type = "n")
> text(2, 2, "srt=0, crt=0", srt = 0, crt = 0)
> text(4, 4, "srt=0, crt=90", srt = 0, crt = 90)
> text(6, 6, "srt=90, crt=0", srt = 90, crt = 0)
> text(8, 8, "srt=90, crt=90", srt = 90, crt = 90)
```



**Figure 8.31:** *Character and string rotation.*

The postscript device is the only graphics device that uses both the `crt` and `srt` graphics parameters. All other graphics devices ignore `crt`, so you can rotate only the whole string with `srt`.

**Warning**

If you use both `crt` and `srt` in a plotting command while running the postscript device, you must supply `crt` *after* `srt`, otherwise it will be ignored

**Controlling  
Line Width**

The width of lines, both within a plot and in the axes, is controlled by the general graphics parameter `lwd`. The default value of `lwd` is 1—larger numbers produce wider lines, while smaller numbers produce narrower lines. Some graphics devices can produce only one width.

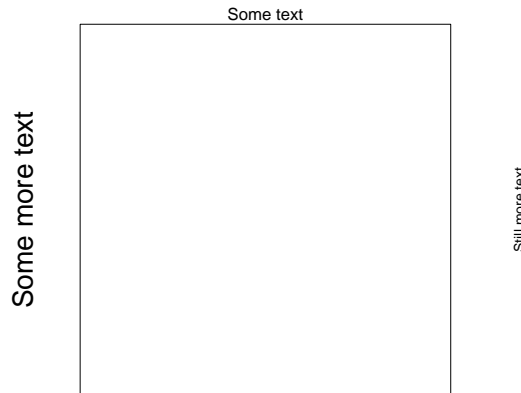
**Plotting  
Symbols in  
Margin**

Generally, plotting symbols are “clipped” so that the symbols don’t appear in the margin. You can allow plotting in the margin by setting `xpd` to `TRUE` (the allowable plotting area is expanded).

## TEXT IN FIGURE MARGINS

To add text in margins, use the `mtext` marginal text function. You specify which of the four margins with the `side` argument, which is a number from 1 to 4 (the default is 3). The `line` argument to `mtext` gives the distance in `mex` between the text and the plot. You may specify non-integer values for `line` in `mtext`. For example, Figure 8.32 shows the placement of the following marginal text:

```
> par(mar=c(5, 5, 5, 5)+.1)
> plot(x, y, type = "n", axes=F, xlab="", ylab="")
> box()
> mtext("Some text", line=0)
> mtext("Some more text", side=2, cex=1, line=2)
> mtext("Still more text", side=4, cex=.5, line=3)
```



**Figure 8.32:** *Placing text in margins.*

Text is not placed in the margin if there is not room for it; this usually happens only when the margin sizes or `cex` have been reset, or with long axis labels. For example, suppose `mex=1` (the default), and you reset the figure margins with `mar=c(1, 1, 1, 1)` to allow precisely one line of text in each margin. If you try to write text with `cex=2`, it will not fit, because the text is twice as high as the specified margin line.

To specify the position of the text *along* the margin, you can use the `at` argument with the `mtext` command argument. The value of the `at` argument is in units of the  $x$  or  $y$  coordinates, depending on whether you are placing text on the top or bottom margin (sides 1 and 3), or the left or right

margin (sides 2 and 4). As described in the section Controlling Text Placement (page 298), if you can't determine the appropriate value of the `at` argument, you can look at the `usr` coordinates graphics parameter. For example, the following command puts text in the lower left-hand corner of the figure margin of Figure 8.32:

```
> par("usr")

[1] 0.1758803 0.9420847 -2.2629721 1.5655365

> mtext("A comment", line=3, side=1, at=.3)
```

By default, `mtext` centers text along the margin or, if the `at` argument is supplied, at the `at` coordinate. You can also use the `adj` parameter to place text along the margin. The default setting is `adj=0.5` (centered text). Set `adj=0` to set the text flush with the left side of the margin or `at` coordinate, `adj=1` to set the text flush right. Values between 0 and 1 set the text with the specified fraction of white space placed before the text, the remaining white space placed after the text.

#### Note

The `adj` parameter is generally more useful than `usr` coordinates when writing in the outer margin of multiple figures, because the `usr` coordinates are the coordinates from the most recent plot created in the figure region.

By default, `mtext` rotates text to be parallel to the axis. To control the orientation of text in the margins use the `srt` argument along with the `at` argument. For example, the following command displays upside-down text in the top figure margin:

```
> mtext("Title with srt=180", line = 2, at = .5, srt = 180)
```

#### Warning

If you supply `mtext` with the `srt` argument, you must supply the `at` argument, otherwise `srt` will be ignored.

## CONTROLLING AXES

The high-level graphics commands described in the section *Getting Started With Simple Plots* (page 241) create complete graphics, including labeled axes. Often, however, you need to create graphics with axes different from those provided by S-PLUS. You may need to specify a different choice of axes, or different tick marks, or different plotting characteristics. This section describes how to control these characteristics.

### Enabling and Disabling Axes

Whether axes appear on a plot is determined by the high-level graphics parameter `axes`, which takes a logical value. If `axes=FALSE`, no axes are drawn on the plot. If axes are not drawn on the original plot, they can be added afterward with one or more calls to the `axis` function.

You can use `plot` with `axes=F` together with the `axis` function to create plots of mathematical functions on a standard Cartesian coordinate system. For example, you can define the following simple function to plot a set of points from the *domain* of a function against the set's *image* on a Cartesian grid:

```
> mathplot <- function(domain, image) {
+   plot(domain, image, type="l", axes=F)
+   axis(1, pos=0)
+   axis(2, pos=0)
+ }
```

### Controlling Tick Marks and Axis Labels

To control the *length* of tick marks use the `tck` general parameter. This parameter is a single number which is interpreted as a fraction of a plot dimension. If `tck` is less than one-half, the tick marks on each axis have the same length; this length is the fraction `tck` of the smaller of the width and height of the plot area. Otherwise, the length of the tick marks on each axis are a fraction of the corresponding plot dimension. Use `tck=1` to draw grid lines. The default is `tck=-.02`, meaning tick marks of equal length on each axis are drawn pointing out from the plot. Try the following expressions:

```
> par(mfrow=c(2, 2))
> plot(x, y, main="tck = -.02")
> plot(x, y, main="tck = .05", tck=.05)
> plot(x, y, main="tck = 1", tck=1)
```

You can have tick marks of different lengths on each axis. The following code draws a plot with no axes, then adds each axis individually with different values of `tck` (and `lty`, the line type):

```
> plot(x, y, axes=F, main="Different tick marks")
> axis(1)
> axis(2, tck=1, lty=2)
> box()
```

To control the *number* of tick marks number on an axis, you can set the `lab` parameter. The `lab` parameter is an integer vector of length three that gives the approximate number of tick marks on the *x*-axis, the approximate number of tick marks on the *y*-axis, and the number of characters for tick labels. (The number is only approximate because S-PLUS tries to use round numbers for tick labels.) It may take some experimentation with `lab` to get just the axis that you want.

To control the *format* of tick labels in exponential notation, use the `exp` graphics parameter, as follows:

**Table 8.5:** *Controlling the format of tick labels.*

Setting	Effect
<code>exp=0</code>	Exponential tick labels are printed on two lines, so that 1e6 is printed with the “1” on one line and the “e6” on the next.
<code>exp=1</code>	Exponential tick labels are printed on a single line, in the form 1e6.
<code>exp=2</code>	(Default value.) Exponential tick labels are printed on a single line, in the form 106.

Uses of the `lab` and `exp` parameters are illustrated with the following code:

```
> par(mfrow=c(2, 2))
> plot(price, mileage, main="lab = c(5, 5, 7)")
> plot(price, mileage, lab=c(10, 3, 7),
+ main="lab = c(10, 3, 7)")
> plot(price, mileage, lab=c(5, 5, 4),
+ main="lab = c(5, 5, 4), exp = 0")
> plot(price, mileage, lab=c(5, 5, 4), exp=1,
+ main="lab = c(5, 5, 4), exp = 1")
```

To control the *orientation* of the axis labels, use the `las` graphics parameter. You can choose between labels that are written parallel to the axes (the default, `las=0`), horizontally (`las=1`), or perpendicular to the axes (`las=2`).



Try the following commands:

```
> par(mfrow=c(2, 2))
> plot(x, y, las=0, mai n="Paral lel , las = 0")
> plot(x, y, las=1, mai n="Hori zontal , las = 1")
> plot(x, y, las=2, mai n="Perpendi cular, las=2")
> plot(x, y, axes=F, mai n="Customi zed")
> axis(2)
> axis(1, at=c(. 2, . 4, . 6, . 8), labe l s=c("2/10",
+ "4/10", "6/10", "8/10"))
> box()
```

The command `box`, ensures that a complete rectangle is drawn around the plotted points (see the section Controlling Axis Boxes (page 307)). The `xaxt` and `yaxt` parameters also control axis plotting. If one of these parameters is equal to "n", the tick marks for the corresponding axis are not drawn. For example, you could also create the last panel produced by the code above, with the following commands:

```
> plot(x, y, xaxt="n")
> axis(1, at=c(. 2, . 4, . 6, . 8), labe l s=c("2/10",
+ "4/10", "6/10", "8/10"))
```

To set the distance from the plot to the axis title, use the `mgp` general parameter. The parameter `mgp` is a numeric vector with three elements in units of `mex`: the first element gives the location of the axis title, the second the location of the tick labels, and the third the location of the axis line. The default value is `c(3, 1, 0)`. You can use `mgp` to control how much space the axes consume.

For example, if you have small margins, you might create a plot with:

```
> plot(x, y, tck=. 02, mgp=c(2, . 1, 0))
```

which draws the tick marks inside the plot, and brings the labels closer to the axis line.

Controlling  
Axis Style

The `xaxs` and `yaxs` parameters determine the style of the axes. The available styles are as follows:

Table 8.6: *Axis styles.*

Setting	Style
"r"	The default axis style; this extends the range of the data by 4% and then labels internally. An <i>internally labeled</i> axis has labels that are inside the range of the data.
"i "	Internally the axis without expanding the range. Thus there will be at least one datapoint on each boundary of an "i " style axis (if <code>xlim</code> and <code>ylim</code> are not used).
"e"	<i>Extended axes</i> label externally (that is, a "pretty" value beyond the range of the data is included) and expand the range by half a character, if necessary, so that no point is precisely on a boundary.
"s"	<i>Standard axes</i> are similar to extended axes but do not expand the range. A plot with standard axes will be exactly the same as a plot with extended axes for some data sets, but for other data sets the extended axes will contain a slightly wider range.
"d"	<i>Direct axis</i> retains the axis from the previous plot. For example, you can make several plots that have precisely the same <i>x</i> -axis or <i>y</i> -axis by giving <code>xaxs="d"</code> or <code>yaxs="d"</code> as an argument to the second and subsequent plot commands. (You can also set it with <code>par</code> , but then you need to remember to release the axis afterwards.)

Axis styles can be illustrated with the following expressions:

```
> par(mfrow=c(2,2))
> plot(x, y, main="Rational axes")
> plot(x, y, xaxs="i ", yaxs="i ", main="Internal axes")
> plot(x, y, xaxs="e", yaxs="e", main="Extended axes")
> plot(x, y, xaxs="s", yaxs="s", main="Standard axes")
```

## Controlling Axis Boxes

You control boxes around the plot area using the `bty` (“box type”) parameter which specifies the type of box to be drawn around a plot. The available types are as follows:

**Table 8.7:** *Specifying the type of box around a plot, using the **bty** parameter.*

Setting	Effect
"n"	No box is drawn around the plot, although the $x$ and $y$ axes are still drawn.
"o"	The default box type; draws a four-sided box around the plot. (The box resembles an uppercase “O,” hence the option name.)
"c"	Draws a three-sided box around the plot in the shape of an uppercase “C.”
"l"	Draws a two-sided box around the plot in the shape of an uppercase “L.”
"7"	Draws a two-sided box around the plot in the shape of a square numeral “7.”

The `box` function draws a box of given thickness around the plot area. The shape of the box is determined by the `bty` parameter. You use `box` to draw full boxes on plots with customized axes, for example:

```
> par(mfrow=c(2, 2))
> plot(x, y, main=' bty = "o" ')
> plot(x, y, bty="l ", main=' bty = "l" ')
> plot(x, y, bty="n", main=' bty = "n" ')
> plot(x, y, main="heavy box")
> box(20)
```

## CONTROLLING MULTIPLE PLOTS

Multiple figures can be created using `par` and `mfrow`. For example, to set a three row by two column layout:

```
> par(mfrow=c(3, 2))
```

In this section, we describe controlling multiple plots in more detail.

When you specify `mfrow` or `mfcol`, S-PLUS automatically changes several other parameters, as follows:

**Table 8.8:** *Changes induces by specifying mfrow or mfcol.*

Parameter	Effects
<code>ftyp</code>	Set to "c" by <code>mfcol</code> and to "r" by <code>mfrow</code> . (This is how S-PLUS knows to go along rows or columns.)
<code>mfg</code>	Contains the row and column of the current figure, and the number of rows and columns in the current array of figures.
<code>cex</code> and <code>mex</code>	If either the number of rows or the number of columns is greater than 2, then both <code>cex</code> and <code>mex</code> are set to 0.5.

To override `mfrow`'s choice of `mex` and `cex`, you must issue separate calls to `par`:

```
> par(mfrow=c(2, 2))
> par(mex=.6, cex=.6)
```

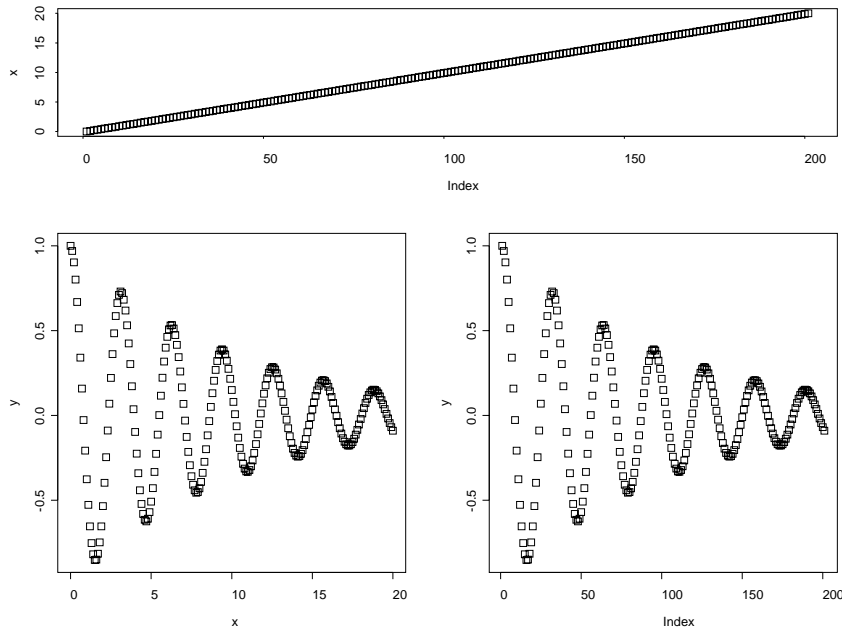
The `mfrow` and `mfcol` layout parameters automatically create multiple figure layouts in which all figures are the same size. You can create multiple figure plots in which the figures are different sizes by using the `fig` layout graphics parameter. The `fig` parameter gives the coordinates of the corners of the current figure as fractions of the device surface. An example is given in Figure 8.33, in which the first plot uses the top third of the device, the second plot uses the left half of the bottom two thirds of the device, and the last plot uses the right half of the bottom two thirds. The example begins with the `frame` function, which tells the graphics device to begin a new figure. You use `frame` frequently when creating graphics from low-level graphics functions:

```
> frame()
> par(fig=c(0, 1, .66, 1), mar=c(5, 4, 2, 2)+.1)
> plot(x)
```

```

> par(fig=c(0, .5, 0, .66))
> plot(x, y)
> par(fig=c(.5, 1, 0, .66))
> plot(y, yaxs="d")
> par(fig=c(0, 1, 0, 1))

```



**Figure 8.33:** *Controlling the layout of multiple plots on one page.*

Once you create one figure with `fig`, you must use it to specify the layout of the entire page of plots. When you complete your custom plot, reset `fig` to `c(0, 1, 0, 1)`.

An easy way to use `fig` with a display device is through the functions `split.screen` and `prompt.screen`. These functions used together let you specify the figure regions interactively with your mouse. When you type:

```
> split.screen(prompt.screen())
```

S-PLUS responds with:

Click at 2 opposite corners

Now move your mouse cursor into your graphics window and click at two opposite corners. After you do this, the region you indicated will be colored in and labeled with the number 1. This is the first screen. In the command window, S-PLUS responds again with:

Click at 2 opposite corners

Repeat this action until you have created all the screens you want, then click on the right mouse button. Once you have divided up the graphics device into separate screens, use the `screen` function to move between screens. See the help file for `split.screen` for more information on using these functions.

<b>Warning</b>
If you want to issue a high-level plotting command in a screen that already has a plot in it, but you don't want the plots in the other screens to disappear, use the <code>erase.screen</code> function before calling the high-level plotting command.

# OVERLAYING FIGURES

It is often desirable to include more than one data set in the same plot. Simple additions can be made with the `lines` and `points` functions. The `matplot` function plots a number of columns of data at once. These all assume, however, that the data are all on the same scale.

There are three general ways to overlay figures in S-PLUS :

1. Call a high-level plotting function, then call one of the high-level plotting functions that can be used as a low-level plotting function by specifying the argument `add=T`.
2. Call a high-level plotting function, set the graphics parameter `new=T`, then call another high-level plotting function.
3. Use the `subplot` function.

We discuss each of these methods below.

## High-level Functions That Can Act as Low-level Functions

There are currently four plotting functions that can act as either high-level or low-level plotting functions: `usa`, `symbols`, `image` and `contour`. By default, these functions act like high-level plotting functions; to make them act like low-level plotting functions set the argument `add=T`. For example, you can put up a map of the northeastern US with a call to `usa`, then overlay a contour plot of ozone concentrations with a call to `contour` by setting `add=T`:

```
> usa(xlim=range(ozone.xy$x), ylim=range(ozone.xy$y),
+ lty=2, col=2)
> contour(interp(ozone.xy$x, ozone.xy$y, ozone.median),
+ add=T)
> title("Median Ozone Concentrations in the North East")
```

## Overlaying Figures by Setting new=TRUE

Another way to overlay figures is to reset the `new` graphics parameter. Whenever a graphics device is initialized, the graphics parameter `new` is set to `TRUE`, meaning that this is a new graphics device, so it is assumed there are currently no plots on it. In this case, a call to a high-level plotting function will *not* erase the canvas before putting up a plot. As soon as a high-level graphics function is called, `new` is set to `FALSE`. In this case, high-level

graphics functions such as `plot` move to the next figure (or erase the current figure if there is only one) in order to avoid overwriting a plot.

You can take advantage of the new `graphics` parameter to call two high-level plotting functions in succession without having the first plot disappear. The code below produces an example of a plot with the same *x*-axis but different *y*-axes. We first set `mar` so that there is room for a labeled axis on both the left and the right, then produce the first plot and the legend:

```
> par(mar=c(5, 4, 4, 5)+.1)
> ts.plot(hstart, ylab="Housing Starts")
> legend(1966.3, 220, c("Housing Starts",
+ "Manufacturing Shipments"), lty=1:2)
```

Now, we set `new` to `TRUE` so that the first plot won't be erased, and specify *direct* axes for *x*-axis in the second plot:

```
> par(new=T, xaxs="d")
> ts.plot(ship, axes=F, lty=2)
> axis(side=4)
> mtext(side=4, line=3.8,
+ "Manufacturing (millions of dollars)")
> par(xaxs="r") # release the direct axis
```

## Overlay Figures by Using subplot

The `subplot` function is another way to overlay plots with different scales. The `subplot` function allows you to put any S-PLUS graphic (except `brush` and `spin`) into another graphic. You specify the graphics function and the coordinates of the subplot. The following code will produce a plot showing selected cities in New England, and New England's position relative to the rest of the United States. To do this, `subplot` is called several times.

To create the main plot, use the `usa` function with the arguments `xlim` and `ylim` to restrict attention to New England.

```
> usa(xlim=c(-72.5, -65), ylim=c(40.4, 47.6))
```

The coordinates shown in the example were obtained by trial-and-error, using as a starting point the coordinates of New York. These were obtained from the three built-in data sets `ci.ty.x`, `ci.ty.y`, and `ci.ty.name` as follows:

```
> names(ci.ty.x) <- ci.ty.name
> names(ci.ty.y) <- ci.ty.name
> nyc.coord <- c(ci.ty.x["New York"], ci.ty.y["New York"])
> nyc.coord
```



```
New York New York
-73.9667 40.7833
```

To plot the city names, we first use `city.x` and `city.y` to determine which cities are contained in the plotted area:

```
> ne.cities <- city.x > -72.5 & city.y > 40.4
```

We then use this criterion to select cities to label:

```
> text(city.x[ne.cities], city.y[ne.cities],
+ city.name[ne.cities])
```

For convenience in placing the subplot, retrieve the `usr` coordinates:

```
> usr <- par("usr")
```

Now, create a subplot of the entire U.S. in a blank spot, and save the value of this call to `subplot` so that information can be added to it:

```
> subpars <- subplot(x=c(-69, usr[2]), y=c(usr[3], 43),
+ usa(xlim=c(-130, -50)))
```

The rest of the commands add to the small map of the entire U.S. First, draw the map with a box around it:

```
> subplot(box(), pars=subpars)
```

Next, draw a box around New England:

```
> subplot(polygon(c(usr[1], -65, -65, usr[1]),
+ c(usr[3], usr[3], usr[4], usr[4]), density=0),
+ pars=subpars)
```

Finally, add text to indicate that the boxed region just created corresponds to the enlarged region:

```
> subplot(text((usr[1]+usr[2])/2, usr[4]+4,
+ "Enlarged Region"), pars=subpars)
```

The `subplot` function can also be used to create *composite* figures. For example, to plot density estimates of the marginal distributions in the margins of a plot of Mileage against Price, enter the following code. First, we set up the coordinate system with `par` and `usr`, and create and store the main plot with `subplot`:

```
> frame()
> par(usr=c(0, 1, 0, 1))
> o.par <- subplot(x=c(0, .85), y=c(0, .85),
+ fun=plot(price, mileage, log="x"))
```

We next find the `usr` coordinates from the main plot and calculate the density estimate for both variables:

```
> o.usr <- o.par$usr
> den.p <- density(price, width=3000)
> den.m <- density(mileage, width=10)
```

Finally, we plot the two marginal densities with two calls to `subplot`. The first plots the density estimate for price along the top of the main plot:

```
> subplot(x=c(0, .85), y=c(.85, 1),
+ fun={par(usr=c(o.usr[1:2], 0, 1.04*max(den.p$y)),
+ xaxt="l"); lines(den.p); box()})
```

The `xaxt="l"` parameter is necessary in the first marginal density plot since price is plotted with a logarithmic axis. To plot the density estimate for mileage along the right of the main plot, use `subplot` as follows:

```
> subplot(x=c(.85, 1), y=c(0, .85),
+ fun={par(usr=c(0, 1.04*max(den.m$y), o.usr[3:4]));
+ lines(den.m$y, den.m$x); box()})
```

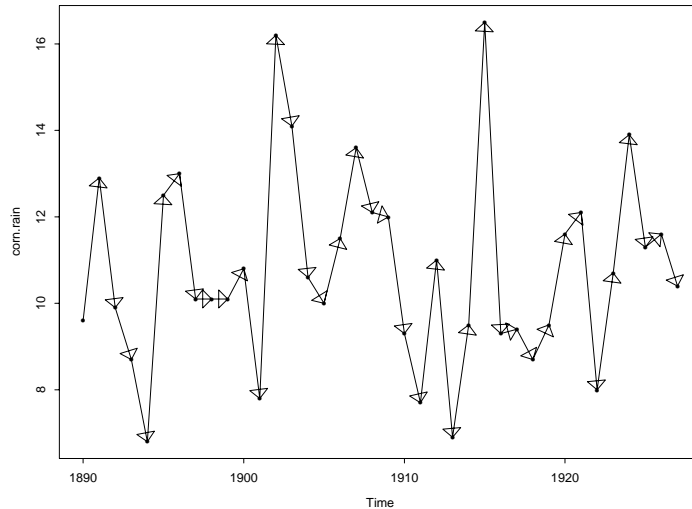
# ADDING SPECIAL SYMBOLS TO PLOTS

In the section Interactively Adding Information to Your Plot (page 259), we saw how to add lines and new data to existing plots. In this section, we describe how to add arrows, stars, and other special symbols to existing plots.

## Arrows and Line Segments

To add one or more arrows to an existing plot, use the `arrows` function. To add a line segment, which is essentially an unpointed arrow, use the `segments` function. Both `segments` and `arrows` take beginning and ending coordinates so that one or more line segments are drawn on the plot. For example, the following commands plot the `corn.rain` data and draw arrows from the *i*th to *i+1*th observation:

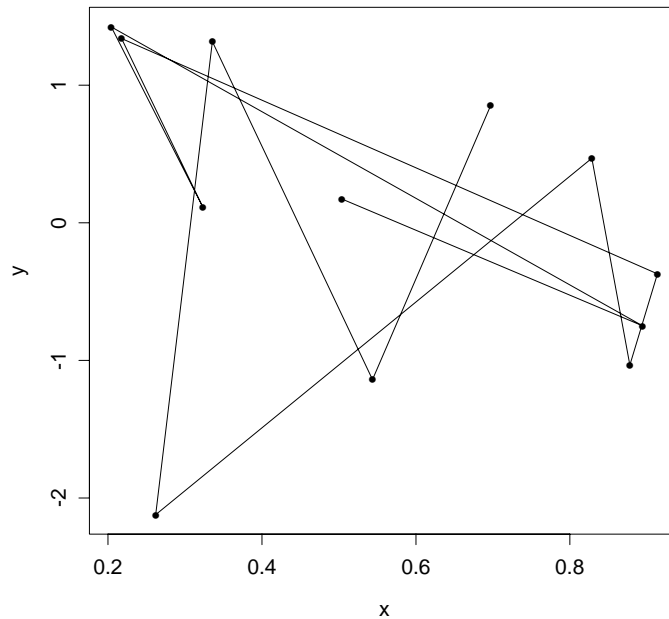
```
> plot(corn.rain)
> for (i in seq(along=corn.rain))
+   arrows(1889+i, corn.rain[i], 1890+i, corn.rain[i + 1])
```



**Figure 8.34:** *Adding arrows to plots.*

Use the `segments` function similarly:

```
> plot(x, y)
> for (i in seq(along=x))
+   segments(x[i], y[i], x[i+1], y[i+1])
```



**Figure 8.35:** *Adding segments to plots.*

## Adding Stars and Other Symbols

You can add a third dimension of data to your plots by using the `symbol.s` function to encode it as stars, circles, or other special symbols. To plot cities with circles whose area represent the population, the steps involved are described below.

First, create the data. We select twelve cities reasonably well-distributed across the country, from among those listed in the built-in data set `ci.ty.name`:

```
> select <- c("Atlanta", "Atlantic City", "Bismarck",
+ "Boise", "Dallas", "Denver", "Lincoln", "Los Angeles",
+ "Miami", "Milwaukee", "New York", "Seattle")
```

As described in the section *Overlaying Figures* (page 311), use names to assign the city names as vector names for the data sets `ci.ty.x`, `ci.ty.y`, and `ci.ty.name`:

```
> names(ci.ty.x) <- ci.ty.name
> names(ci.ty.y) <- ci.ty.name
> names(ci.ty.name) <- ci.ty.name
```

By assigning names in this way, we can access the information necessary to plot the cities without learning their vector indices. From an almanac or similar reference, look up the populations of the selected cities, and create a vector to hold the information (in thousands):

```
> pop <- c(425, 60, 28, 34, 904, 494, 129, 2967, 347, 741,
+ 7072, 557)
```

Use the `usa` function to plot the map:

```
> usa()
```

Next, add the circles representing the cities:

```
> symbols(city.x[select], city.y[select],
+ circles=sqrt(pop), add=T)
```

The next two lines use the `ifelse` command to create a size vector for controlling the text size:

```
> size <- ifelse(pop>1000, 2, 1)
> size <- ifelse(pop<100, .5, size)
```

Taken together, these two lines specify a size of 2 for cities with population greater than one million, a size of 1 for cities with population between one hundred thousand and one million, and a size of 0.5 for cities with population less than one hundred thousand. Finally, we add the text, using the size just determined to specify the text size:

```
> text(city.x[select], city.y[select], city.name[select],
+ cex=size)
```

You can use any one of the following shapes as an argument to `symbol`, with values as indicated:

**Table 8.9:** *Using shapes as an argument to the function **symbol**.*

Shape	Values
<code>circles</code>	Vector or matrix with one column containing the radii of the circles.
<code>squares</code>	Vector or matrix with one column containing the lengths of the sides of the squares.
<code>rectangles</code>	Matrix with two columns giving widths and heights of rectangles. Missing values are allowed, points containing missing values are not plotted.

**Table 8.9:** *Using shapes as an argument to the function **symbol**.*

Shape	Values
<code>stars</code>	Matrix with $n$ columns, where $n$ is the number of points to a star. The matrix must be scaled from 0 to 1.
<code>thermometers</code>	Matrix with 3 or 4 columns. The first two columns give the widths and heights of the rectangular thermometer symbols. If the matrix has 3 columns, the third column gives the fraction of the symbol that is filled (from the bottom up). If the matrix has 4 columns, the third and fourth columns give the fractions of the rectangle between which it is filled.
<code>boxplots</code>	Matrix with 5 columns of positive numbers, giving the width and height of the box, the amount to extend on the top and bottom, and the fraction of the way up the box to draw the median line.

Note: Missing values are allowed, points containing missing values are not plotted, except in `stars`, where they are treated as zeros.

**Custom Symbols**

The following functions provide a simple way to add your own symbols to a plot. The `make.symbol` function facilitates creating a symbol:

```
> make.symbol <- function()
+ {
+   on.exit(par(p))
+   p <- par(pty = "s")
+   plot(0, 0, type = "n", xlim = c(-0.5, 0.5),
+        ylim = c(-0.5, 0.5))
+   cat("Now draw your symbol using the mouse,
+       Continue string: clicking at corners\n ")
+   locator(type = "l")
+ }
```

This returns a list with components named `x` and `y`. The `Continue string:` prompt is given because there was a new line while in the middle of a character string. The most important feature of this function is that it uses `pty="s"` so that the figure will be drawn to proper scale when used with `draw.symbol`. The `draw.symbol` function takes some locations and a symbol given in the form of a list with `x` and `y` components:

```
> draw.symbol <-
+ function(x, y, sym, size = 1, fill=F,...)
+ {
+   uin <- par()$uin # inches per user unit
+   sym$x <- sym$x/uin[1] * size
+   sym$y <- sym$y/uin[2] * size
+   if (!fill)
+     for(i in 1:length(x))
+       lines(x[i] + sym$x, y[i] + sym$y,...)
+   else
+     for(i in 1:length(x))
+       polygon(x[i] + sym$x, y[i] + sym$y,...)
+ }
```

The `uin` graphics parameter is used to scale the symbol into user units. The `make.symbol` and `draw.symbol` functions are examples of how to create your own graphics functions using the built-in graphics functions and graphics parameters.

## WRITING GRAPHICS FUNCTIONS

S-PLUS was designed to provide a convenient graphical environment for data analysis, so it is rich in graphics functions for creating many types of plots quickly and easily. These graphics functions are of two main types: *high-level* which start a new plot and select scales, shape, and position of the plot, and *low-level* which add individual elements to a plot. For much exploratory data analysis, you can work with simple calls to high-level functions, but for production-quality graphics you generally need to combine high-level functions with low-level functions to add special features. Or you can gain complete control over the structure of your plot by using only low-level graphics functions.

Because any combination of S-PLUS expressions can be combined into an S-PLUS function, you can easily create new graphics functions to produce virtually any type of plot you want. Also, since most of the graphics functions are largely written in the S-PLUS language, you can easily modify existing functions to produce plots similar to, but slightly different from, the plots created by the built-in function. This section shows several possible strategies.

### Note

This section describes writing functions that use traditional graphics. The object-oriented graphics functions are described in Chapter 7, Editable Graphics Commands.

### Modifying Existing Functions

Sometimes an existing function does most of what you want it to do, but there is some plot detail that you are unhappy with. For example, consider the `hist` function. The `hist` function allows you to specify the cell width by means of the `breaks` argument. By default, however, the axis labels are chosen using the `pretty` function, and thus they will not, in general, coincide with the cell divisions specified by `breaks`. Figure 8.36 illustrates the effect, showing the result of the following call to `hist`:

```
> hist(corn.rai n, breaks=c(6, 9, 12, 15, 18))
```

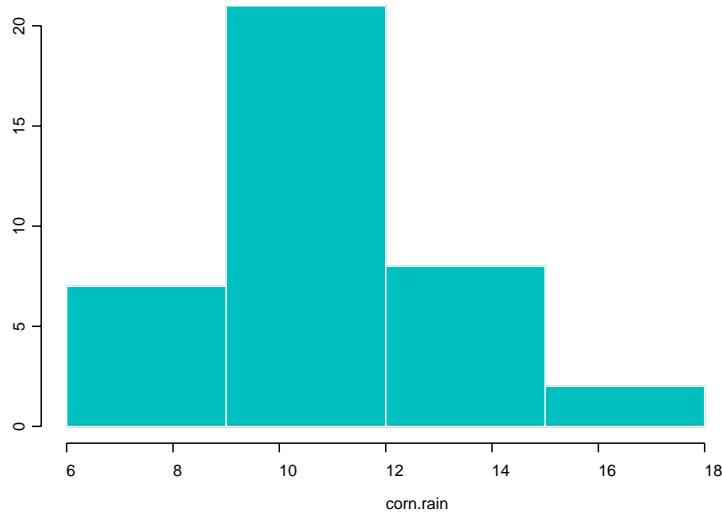
It is a relatively easy matter to make the axis labels coincide with the specified breaks. The necessary modifications are the following:

- Prevent `hist` from drawing the default axes.
- Determine whether the histogram is vertical or horizontal; we want



to modify the  $x$ -axis if horizontal and the  $y$ -axis if vertical. The other axis we want to leave alone.

- Draw both axes explicitly.



**Figure 8.36:** *Note how the axis labels and the bar columns do not match up.*

In deciding how to implement the modification, you have two choices: either modify (a copy of) the `hist`, or write a *wrapper* function that wraps the desired behavior around the `hist` function's default behavior. MathSoft highly recommends the wrapper approach, because it allows you to build on existing functionality while at the same time allowing your modified function to pass on future enhancements to the original function. If you modify the existing function, you must modify the improved function when updating S-PLUS. By using the S-PLUS `get` function within your wrapper, you can build a wrapper with the same name as the modified function that actually calls the *unmodified* version. If you want the modified behavior only sporadically, a wrapper function with a different name is the way to go. However, if the modification requires changes at several points in the execution of the original function, it is probably easier to start from the

existing function and modify it as necessary. When the modifications are very extensive, you are probably better off renaming the resulting function.

### Hint: modifying Built-in Functions

If you modify a built-in function, such as `hist`, do *not* assign your modified function to the directory in which it was originally stored. Initially, all your modifications should be done in your working data directory. If you then want to make your modified function more widely available, use an S-PLUS library to hold your modified function, and then attach the library at a suitable place in the search path (ahead of the system function directories). In this way, you retain the original, unmodified function and can recover it at any time.

No matter which approach you use, you should make sure that your modified function accepts all the original arguments and returns the same value. This is particularly important if you are using the same name for your modification.

In our example, all of the necessary changes occur in the last part of the `hist` function, indicating that a wrapper function will serve us well. Here is the wrapper version of our modification:

```
my.hist <-
function(x, breaks, horiz = F, plot = T, ..., xlab =
    deparse(substitute(x)))
{
    # Makes axis-labels coincide with user-specified breaks
    if(horiz)
    {
        breaksaxis <- 2
        otheraxis <- 1
    } else
    {
        breaksaxis <- 1
        otheraxis <- 2
    }

    if(!missing(breaks))
    {
        val <- hist(x, breaks = breaks,
            horiz = horiz, plot = plot, axes = F, ...,
            xlab = xlab)
    } else
    {
        val <- hist(x, horiz = horiz, plot = plot,
            axes = F, ..., xlab = xlab)
    }
}
```

```

        breaks <- val
    }

    if(plot)
    {
        axis(breaksaxis, at = breaks)
        axis(otheraxis)
        invisible(val)
    } else val
}

```

Figure 8.37 shows the histogram created for the following S-PLUS expression:

```
> my.hist(corn.rain, breaks=c(6, 9, 12, 15, 18))
```

As desired, the breaks and axis labels coincide.

## Combining High-Level and Low-Level Functions

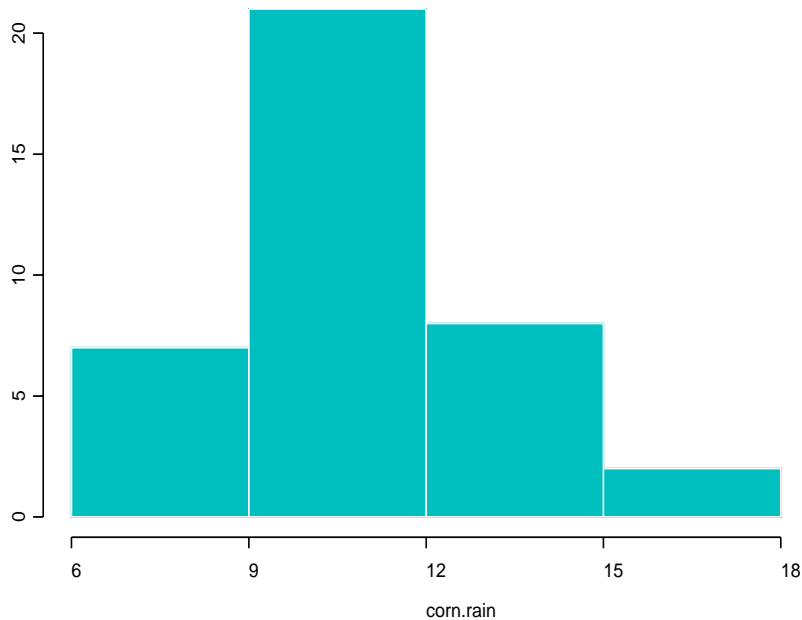
Graphics functions such as `hist` produce specialized output, so changes in their output format can easily be incorporated into the function itself. On the other hand, a graphics function such as `plot` is used to create a wide variety of plots, so you will probably not want to modify the function itself to create a special function. In such cases, you will most likely build a wrapper function, as we did with `my.hist` in the previous section. The wrapper function combines the effects of a high-level function with some low-level function calls to create a customized graphics function.

For example, a query to the electronic mailing list S-News asked how to create plots with the plot origin in the upper left corner, rather than the lower right (the default case). To get the data plotted correctly, you just need to plot  $-y$  rather than  $y$  and draw the  $x$ -axis at the top of the plot rather than the bottom. That part is easy. Getting the  $y$ -axis labels to come out right requires a trick: the `yaxp` graphics parameter stores the coordinates of the uppermost tick mark and the lowermost tick mark, along with the total number of tick marks on the  $y$ -axis. We can use this information to construct correct tick marks and labels for our  $y$ -axis. Note that things get more complicated if your graph requires log axes.

```

ul origin <-
function(x, y, ...)
{
  plot(x, ~ y, axes = F, ..., xlab = "", ylab = "")
  axis(3)
  yaxp <- par("yaxp")
  ticks <- seq(yaxp[1], yaxp[2], length = yaxp[3])
  axis(2, at = ticks, labels = ~ ticks, srt = 90)
  box()
}

```



**Figure 8.37:** *The x-axis labels and bar columns now match perfectly.*

Another common modification to the default plot type is to create axes at  $x=0$  and  $y=0$ , such as those commonly seen when plotting mathematical functions. To be sure these axes are printed, you may need to expand the range of plotted data to include the origin. Here is a very simple function to produce such plots:

```

mathplot <-
function(x, y, ..., xlim, ylim)
{
  if(missing(xlim))
    xlim <- c(min(0, x), max(0, x))
}

```

```

    if(missing(ylim))
      ylim <- c(min(0, y), max(0, y))
    plot(x, y, type = "l", axes = F, ...,
         xlim = xlim, ylim = ylim)
    axis(1, pos = 0)
    axis(2, pos = 0)
  }

```

Both of the above functions can be enhanced with labels reflecting the input data. These enhancements are discussed in Chapter 20, Computing on the Language.

## Using Graphical Parameters

*Graphical parameters* govern many aspects of S-PLUS's plotting behavior. These parameters fall into four broad categories:

- *High-level* graphical parameters, such as `xlab` and `type`, must, if specified at all, be specified as arguments to a high-level graphics function such as `plot`.
- *General* parameters, such as `cex` and `col`, can be specified either as arguments to high-level functions or through the parameter-setting function `par`.
- *Layout* parameters, such as `fig` and `mfrow`, can be specified only through `par`.
- *Information* parameters, such as `ui`, provide information about the graphics environment, but cannot be changed.

In building your own graphics functions, you will probably need some of these parameters. This section gives some examples of using these parameters within function definitions. For a thorough discussion of the individual parameters and the effects they can be used to achieve, see the `par` help file

As a first example, consider the problem of producing publication-quality graphics, such as the figures in this book. Typically, a book designer specifies a page size, the page margins, and many other details of the book's look. To obtain the best reproduction possible, you want to produce figures at an appropriate scale to fit the specified page size. This is particularly important in figures containing multiple plots. To produce multiple-plot figures in S-PLUS, you use the `mfrow` layout parameter. When you reset `mfrow`, S-PLUS automatically resets several other parameters, including `cex` and `mex`. Often,

however, S-PLUS's choice of value is not optimal. At such times, a function can give you more control over the finished graphic.

Here is a function, `fourplot`, used to create two-by-two plots. It calls the `postscript` device with a specified size, sets `mfrow` for two-by-two plots, then resets `cex`, `mex`, and `mgp` to appropriate values:

```
fourplot <-  
function(filename = "ps.out")  
{  
  postscript(height = 5, width = 5, horizo = F,  
             file = filename)  
  par(mfrow = c(2, 2))  
  par(cex = 0.6)  
  par(mex = 0.6)  
  par(mgp = c(2.5, 1, 0))  
  invisible()  
}
```

The `fourplot` function does no plotting, it merely sets up the `postscript` graphics device and sets a few graphics parameters. The `fourplot` function has the side effect that it modifies the graphics state for this instance of the `postscript` device; subsequent calls to graphics functions are governed by the new values of `mfrow`, `cex`, `mex`, and `mgp`. This side effect is intentional.

If a function does perform plotting, however, the finished plot is usually the intended side effect. If graphical parameters must be modified to create the plot, they should be restored to their earlier condition before exiting the function. This can easily be done by including lines like the following in the function:

```
opar <- par(mfrow = c(2, 2))  
on.exit(par(opar))
```

# TRADITIONAL GRAPHICS SUMMARY

**Table 8.10:** *Summary of the most useful graphics parameters.*

Name	Type	Mode	Description	Example
<b>MULTIPLE FIGURES</b>				
<code>fig</code>	layout	numeric	figure location	<code>c(0, .5, .3, 1)</code>
<code>fin</code>	layout	numeric	figure size	<code>c(3.5, 4)</code>
<code>fty</code>	layout	character	figure type	<code>"r"</code>
<code>mfg</code>	layout	integer	location in figure array	<code>c(1, 1, 2, 3)</code>
<code>mfc0l</code>	layout	integer	figure array size	<code>c(2, 3)</code>
<code>mflow</code>	layout	integer	figure array size	<code>c(2, 3)</code>
<b>TEXT</b>				
<code>adj</code>	general	numeric	text justification	<code>.5</code>
<code>cex</code>	general	numeric	height of font	<code>1.5</code>
<code>crt</code>	general	numeric	character rotation	<code>90</code>
<code>csi</code>	general	numeric	height of font	<code>.11</code>
<code>mai n</code>	title	character	main title	<code>"Y versus X"</code>
<code>srt</code>	general	numeric	string rotation	<code>90</code>
<code>sub</code>	title	character	subtitle	<code>"Y versus X"</code>
<code>xl ab</code>	title	character	axis titles	<code>"X (i n d o l l a r s)"</code>
<code>yl ab</code>	title	character	axis title	<code>"Y (i n s i z e)"</code>
<b>SYMBOLS</b>				
<code>l ty</code>	general	integer	line type	<code>2</code>
<code>l wd</code>	general	numeric	line width	<code>3</code>
<code>pch</code>	general	character, integer	plot symbol	<code>"*", 4</code>

**Table 8.10:** *Summary of the most useful graphics parameters.*

Name	Type	Mode	Description	Example
<code>smo</code>	general	integer	curve smoothness	1
<code>type</code>	general	character	plot type	"h"
<code>xpd</code>	general	logical	symbols in margins	TRUE
<b>AXES</b>				
<code>axes</code>	high-level	logical	plot axes	FALSE
<code>bty</code>	general	integer	box type	4
<code>exp</code>	general	numeric	format for exponential numbers	1
<code>lab</code>	general	integer	tick marks and labels	<code>c(3, 7, 4)</code>
<code>las</code>	general	integer	label orientation	1
<code>log</code>	high-level	character	logarithmic axes	"xy"
<code>mgp</code>	general	numeric	axis locations	<code>c(3, 1, 0)</code>
<code>tck</code>	general	numeric	tick mark length	1
<code>xaxs</code>	general	character	style of limits	"i"
<code>yaxs</code>	general	character	style of limits	"i"
<code>xart</code>	general	character	axis type	"n"
<code>yart</code>	general	character	axis type	"n"
<b>MARGINS</b>				
<code>mai</code>	layout	numeric	margin size	<code>c(.4, .5, .6, .2)</code>
<code>mar</code>	layout	numeric	margin size	<code>c(3, 4, 5, 1)</code>
<code>mex</code>	layout	numeric	margin units	.5
<code>oma</code>	layout	numeric	outer margin size	<code>c(0, 0, 5, 0)</code>
<code>omd</code>	layout	numeric	outer margin size	<code>c(0, .95, 0, 1)</code>



**Table 8.10:** *Summary of the most useful graphics parameters.*

Name	Type	Mode	Description	Example
omi	layout	numeric	outer margin size	c(0, 0, . 5, 0)
<b>PLOT AREA</b>				
pi n	layout	numeric	plot area	c(3. 5, 4)
pl t	layout	numeric	plot area	c(. 05, . 95, . 1, . 9)
pty	layout	character	plot type	"s"
ui n	information	numeric	inches per usr unit	c(. 73, . 05)
usr	layout	numeric	limits in plot area	c(76, 87, 3, 8)
xl im	high-level	numeric	limits in plot area	c(3, 8)
yl im	high-level	numeric	limits in plot area	c(3, 8)
<b>MISCELLANEOUS</b>				
col	general	integer	color	2
err	general	integer	print warnings?	-1
new	layout	logical	is figure blank?	TRUE

## References

- Chernoff, H. (1973). The Use of Faces to Represent Points in k-Dimensional Space Graphically. *Journal of American Statistical Association* **68**, 361-368.
- Cleveland, W. S. (1985). *The Elements of Graphing Data*. Monterey, California: Wadsworth.
- Martin, R. D., Yohai, V. J., and Zamar, R. H. (1989). Min-max bias robust regression. *Annals of Statistics* **17**, 1608-30.
- Rousseeuw, P. J. and Leroy, A. M. (1987). *Robust Regression and Outlier Detection*. New York: Wiley.
- Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis*. London: Chapman and Hall.



# TRADITIONAL TRELLIS GRAPHICS

# 9

---

<b>A Roadmap of Trellis Graphics</b>	<b>333</b>
<b>Giving Data to General Display Functions</b>	<b>335</b>
A Data Set: gas	335
formula Argument	335
subset Argument	337
Data Frames	338
<b>Aspect Ratio</b>	<b>339</b>
<b>General Display Functions</b>	<b>341</b>
A Data Set: fuel.frame	341
A Data Set: gauss	354
The Display Functions and Their Formulas	357
<b>Arranging Several Graphs on One Page</b>	<b>359</b>
<b>Multipanel Conditioning</b>	<b>361</b>
A Data Set: barley	361
About Multipanel Display	361
Columns, Rows, and Pages	361
Packet Order and Panel Order	362
layout Argument	364
Main-Effects Ordering	366
Summary: The Layout of a Multipanel Display	368
A Data Set: ethanol	368
Conditioning on Discrete Values of a Numeric Variable	368
Conditioning on Intervals of a Numeric Variable	370
<b>Scales and Labels</b>	<b>374</b>
3-D Display: aspect Argument	376
Changing the Text in Strip Labels	376
<b>Panel Functions</b>	<b>378</b>
How to Change the Rendering in the Data Region	378
Passing Arguments to a Default Panel Function	378
A Panel Function for a Multipanel Display	379

Special Panel Functions	380
Commonly-Used S-PLUS Graphics Functions and Parameters	380
<b>Panel Functions and the Trellis Settings</b>	<b>381</b>
<b>Superposing Two or More Groups of Values on a Panel</b>	<b>384</b>
<b>Data Structures</b>	<b>391</b>
<b>More on Aspect Ratio and Scales: Prepanel Functions</b>	<b>394</b>
More on Multipanel Conditioning	395
<b>Summary of Trellis Functions and Arguments</b>	<b>398</b>

---

# A ROADMAP OF TRELLIS GRAPHICS

Trellis Graphics provide a comprehensive set of display functions, that are a popular alternative to using the traditional S-PLUS graphics functions described in the previous chapter. The Trellis functions are particularly geared towards multipanel and multipage plots. The concepts have been used extensively in the displaying of graphs using the menus, toolbars and dialogs of the graphical user interface (described in the *User's Guide*). This chapter describes the Trellis system based on traditional S-PLUS graphics. This gives the user a more detailed level of control than the editable graphics.

## Getting Started With Trellis

Open a Trellis Graphics device with the command `trellis.device`. If no device is open, Trellis commands will open one by default, but by using this command you ensure the open graphics device is compatible with Trellis Graphics.

```
> trellis.device()
```

## General Display Functions

The Trellis library has a collection of *general display functions* that draw different types of graphs. For example, `xyplot` makes x-y plots, `dotplot` makes dot plots, and `wireframe` makes 3-D wireframe displays. The functions are *general* because they have the full capability of Trellis Graphics including multipanel conditioning.

These functions are introduced in the section General Display Functions (page 341).

## Common Arguments

There are a set of common arguments that all general display functions employ. The usage of some of these arguments varies, but each has a common purpose across all functions. Many of the general display functions also have arguments that are specific to the types of graphs that they draw.

The common arguments, which are listed in the section Summary of Trellis Functions and Arguments (page 398), are discussed in many sections.

## Panel Functions

Panel functions are a critical aspect of Trellis Graphics. They make it easy to tailor displays to your data even when the displays are quite complicated ones with many panels.

The data region of a panel on a graph resulting from a general display function is a rectangle that just encloses the data. The sole responsibility for drawing in a data region is given to a panel function that is an argument of the general display function. The other arguments of the general display

function manage the superstructure of the graph—scales, labels, boxes around the data region, and keys. The panel function manages the symbols, lines, and so forth that encode the data in the data regions.

Panel functions are discussed in the section Panel Functions (page 378).

## **Core S-PLUS Graphics**

Trellis Graphics is implemented in the core traditional S-PLUS graphics. Also, when you write a panel function you use functions and graphics parameters from the traditional graphics system.

Some core S-PLUS graphics features are discussed in the section Commonly-Used S-PLUS Graphics Functions and Parameters (page 380).

## **Printing, Devices and Settings**

To send a graph to the printer, first open a hardcopy device, for example with `trellis.device(postscript)`, or `trellis.device(pdf.graph)`. To actually send the graphics to the printer, enter the command `dev.off()`. For color graphics printing, set the `color=T` flag (the default is black and white), when opening the device, for example:

```
> trellis.device(postscript, color=T)
```

Trellis Graphics has many settings for graph rendering details—plotting symbols, colors, line types and so forth—that are automatically chosen depending on the device you select.

The section Panel Functions and the Trellis Settings (page 381) discusses the Trellis settings.

## **Data Structures**

The general display functions take in data just like many of the S-PLUS modeling functions such as `lm`, `aov`, `glm`, and `loess`. This means that there is a heavy reliance on data frames. The Trellis library contains several functions that change data structures of certain types to a data frame, which makes it easier to pass the data on to the general display functions (or, in fact, on to the modeling functions).

The section Data Structures (page 391) discusses these functions that create data frames.

## GIVING DATA TO GENERAL DISPLAY FUNCTIONS

For a graphics function to draw a graph, it needs to know the data on which the drawing is based. This section is about arguments to the Trellis drawing functions that allow you to specify the data.

**A Data Set: `gas`** The data frame `gas` contains two variables from an industrial experiment with twenty two runs in which the concentrations of oxides of nitrogen (NO<sub>x</sub>) in the exhaust of an engine were measured for different settings of equivalence ratio (E).

```
> names(gas)

[1] "NOx" "E"

> dim(gas)

[1] 22 2
```

### formula Argument

The function `xyplot` makes an x-y plot, a graph of two numerical variables; the result might be scattered points, curves, or both. A full discussion of `xyplot` is in the section General Display Functions (page 341), but for now we will use it to illustrate how to specify data.

The plot in Figure 9.1 is a scatterplot of `gas$NOx` against `gas$E`:

```
> xyplot (formula = gas$NOx ~ gas$E)
```

The argument `formula` specifies the variables that are to be graphed. In this case they are `gas$NOx` and `gas$E`. For `xyplot`, the variable to the left of the `~` goes on the vertical axis, and the variable to the right of the `~` goes on the horizontal axis. The formula `gas$NOx ~ gas$E` is read as `gas$NOx` “is graphed against” `gas$E`.

The use of `formula` here is the same as that in the S-PLUS statistical modeling functions such as `lm` and `aov`. To the left or right of the `~` you can use any S-PLUS expression. For example, if you want to graph the log base 2 of `gas$NOx`, you can use the formula

```
log(gas$NOx, base=2) ~ gas$E
```

The argument `formula` is a special one in Trellis Graphics. It is always the first argument of a general display function such as `xyplot`. We can omit typing `formula` provided the formula is the first argument. Thus the expression `xyplot(gas$NOx ~ gas$E)` also produces Figure 9.1.

The argument `formula` is the only one that should be given by position; all others must be given by name.

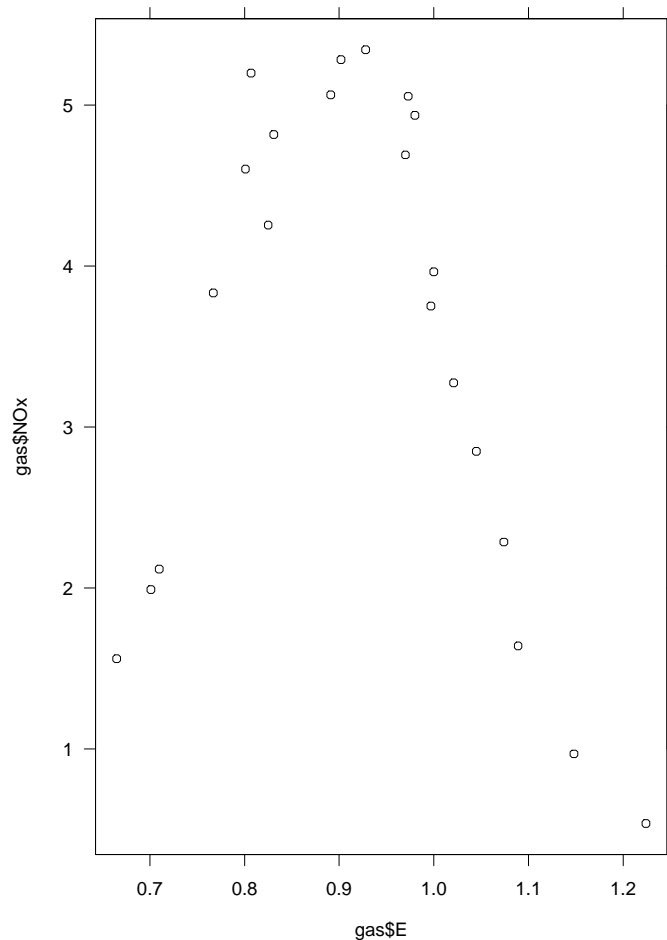


Figure 9.1: Scatterplot of ***gas\$NOx*** against ***gas\$E***.

Certain single-symbol operators that perform functions in S-PLUS have a special meaning in the formula language (for example, `+`, `*`, `/`, `|`, and `:`), although Trellis, as we will see, uses only `*` and `|`. If you want to use any of these operators for their conventional meaning in any formula expression—



for example, if you want to use `*` as multiplication—you must put the expression inside the identity function `l()` unless it is already given as an argument to a function. Here is an example:

```
log(2*gas$NOx, base=2) ~ l(2*gas$E)
```

We use `l` on the right of the formula to protect against the `*` in `2*gas$E`, but not on the left because `2*gas$NOx` sits inside a function data argument

One annoyance in the use of the above formulas is that we had to continually refer to the data frame `gas`. This is not necessary if we attach `gas` to the search list of databases. We can draw Figure 9.1 by

```
> attach(gas)
> xyplot(NOx~E)
```

Another possibility is to use the argument `data`:

```
> xyplot(NOx~E, data = gas)
```

In this case, the variables of `gas` are available for use in the formula argument just during the execution of `xyplot`. The effect is the same as

```
> attach(gas)
> xyplot(NOx ~ E)
> detach(gas)
```

The use of the `data` argument has another benefit. In the call to `xyplot` we see explicitly that the data frame `gas` is being used; this can be helpful for understanding, at some future point, how the graph was produced.

## subset Argument

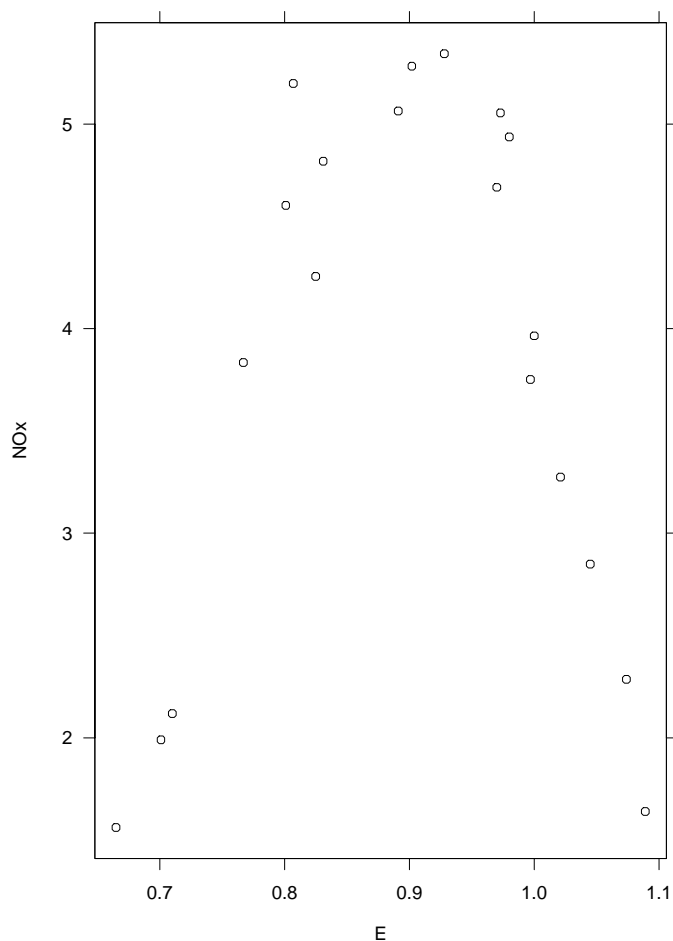
Suppose you want to redo Figure 9.1 and omit the observations for which `E` is 1.1 or greater. You could do this by

```
> xyplot(NOx[E < 1.1] ~ E[E < 1.1], data = gas)
```

But it is a nuisance to repeat the logical subsetting, `E < 1.1`. And the nuisance would be much greater if there were many variables in the formula instead of just two. It is typically easier to use the argument `subset` instead:

```
> xyplot(NOx~E, data = gas, subset = E < 1.1)
```

The result is shown in Figure 9.2. The argument `subset` can be a logical or numerical vector.



*Figure 9.2: Using the subset argument on the gas data.*

## Data Frames

You can keep variables as vectors and draw Trellis displays without using data frames. Still, data frames are very convenient. But data sets are often stored, at least initially, in data structures other than data frames, so we need ways to go from data structures of various types to data frames. Functions to do this are discussed in the section Data Structures (page 391).

---

## ASPECT RATIO

The aspect ratio of a graph, the height of a panel data region divided by its width, is a critical factor in determining how well a data display shows the structure of the data. There are situations where choosing the aspect ratio to carry out banking to 45 degrees shows information in the data that cannot be seen if the graph is square, that is, has an aspect ratio of 1. More generally, any time we graph a curve, or a scatter of points with an underlying pattern that we want to assess, controlling the aspect ratio is vital. One advance of Trellis Graphics is the direct control of the aspect ratio through the argument `aspect`.

**aspect argument** You can use the `aspect` argument to set the ratio to a specific value. In Figure 9.3, the aspect ratio has been set to 3/4:

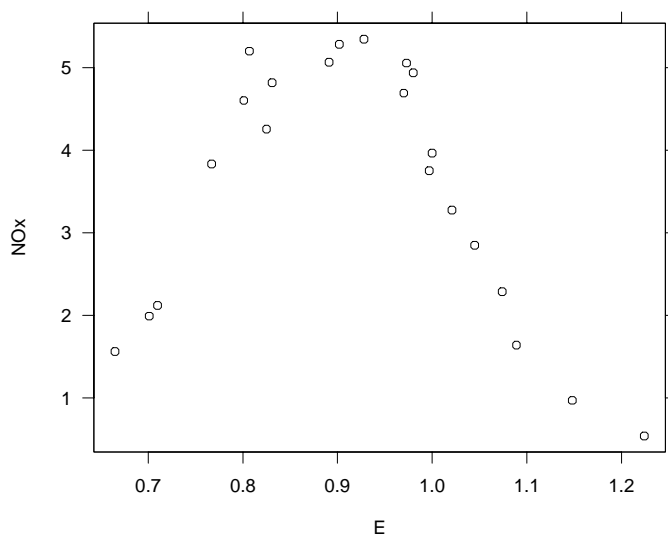
```
> xyplot(NOx~E, data = gas, aspect = 3/4)
```

Setting the `aspect` argument, "`xy`" banks line segments to 45 degrees. Here is how it works. Suppose `x` and `y` are data points to be plotted. Consider the line segments that connect successive points. The aspect ratio is chosen so that the absolute values of the slopes of these segments are centered on 45 degrees. This is done in Figure 9.4 by the expression

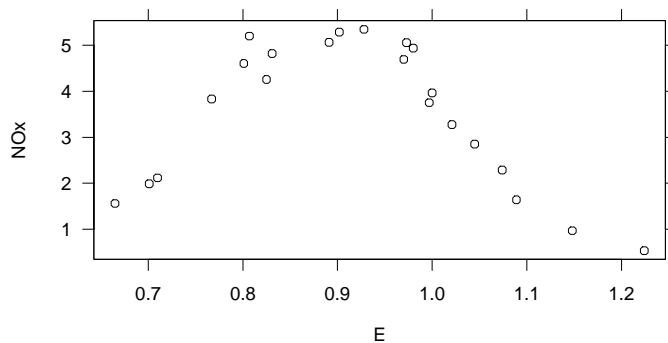
```
> xyplot(NOx~E, data = gas, aspect = "xy")
```

We have used the data themselves in this example to carry out banking, just to illustrate how it works. The resulting aspect ratio is about 0.4. Ordinarily, though, we should bank based on a smooth underlying pattern in the data;

that is, we should bank based on the line segments of a fitted curve. You can do that with Trellis Graphics as well; an example is in the section More on Aspect Ratio and Scales: Prepanel Functions (page 394).



*Figure 9.3: The scatterplot of the gas data with an aspect ratio of 3/4.*



*Figure 9.4: The scatter plot of the gas data with line segments banked to 45 degrees.*

## GENERAL DISPLAY FUNCTIONS

Each *general display function* draws a particular type of graph. For example, `dotplot` makes dot plots, `wireframe` makes 3-D wireframe displays, `histogram` makes histograms, and `xyplot` makes x-y plots. This section describes a collection of general display functions.

### A Data Set: fuel.frame

The data frame `fuel.frame` contains five variables that measure characteristics of 60 automobile models:

```
> names(fuel.frame)

[1] "Weight" "Displ." "Mileage" "Fuel" "Type"

> dim(fuel.frame)

[1] 60 5
```

The variables are weight, displacement of the engine, fuel consumption in miles per gallon, fuel consumption in gallons per mile, and a classification into type of vehicle. The first four variables are numeric. The fifth variable is a factor:

```
> table(fuel.frame$Type)

Compact Large Medium Small Sporty Van
      15      3      13      13      9      7
```

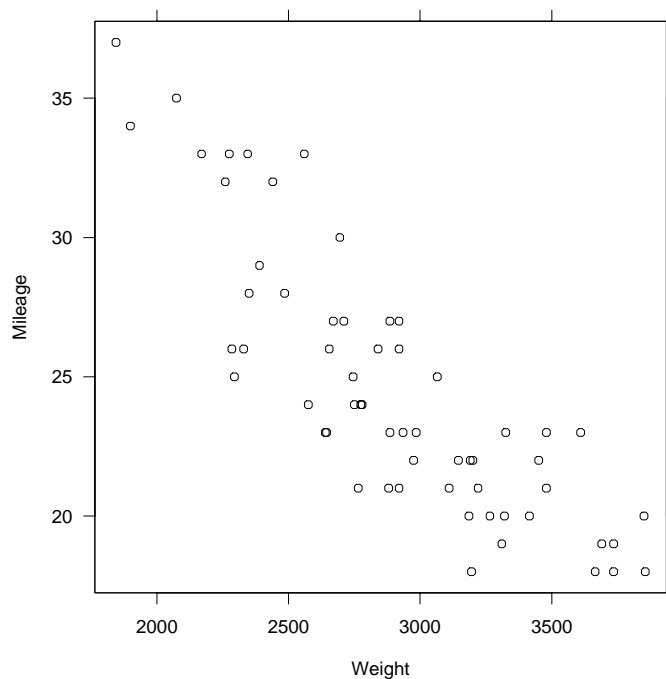
**xyplot**

We have already seen `xyplot` in action in many of our previous examples. This function is a basic graphical method—graphing one set of numerical values on a vertical scale against another set of numerical values on a horizontal scale.

Figure 9.5 is a scatterplot of mileage against weight:

```
> xyplot(Mileage ~ Weight, data = fuel.frame, aspect = 1)
```

The variable on the left of the `~` goes on the vertical, or y, axis and the variable on the right goes on the horizontal, or x, axis.



*Figure 9.5: Scatterplot.*

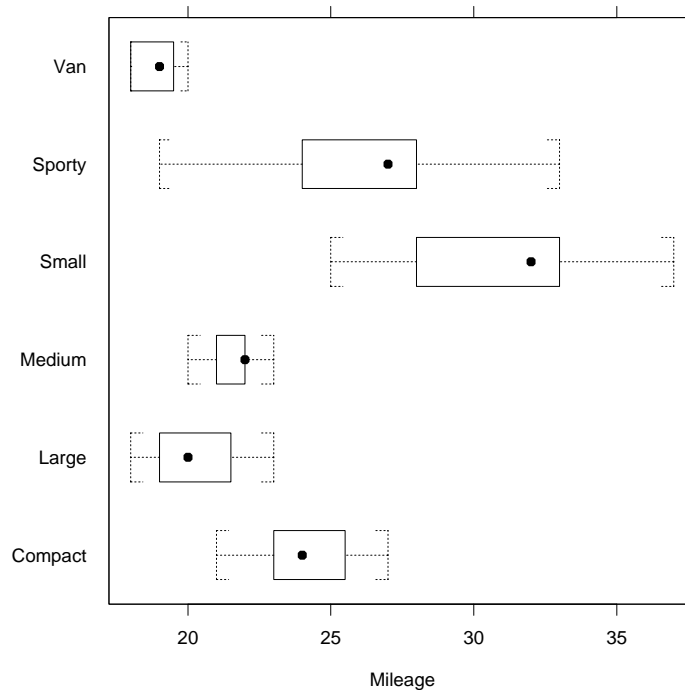
**bwplot**

The box and whisker plot, or box plot, is a very clever invention of John Tukey that is widely used for comparing the distributions of several data sets.

Figure 9.6 is a box plot of mileage classified by vehicle type:

```
> bwplot(Type ~ Mileage, data = fuel.frame, aspect = 1)
```

The factor `Type` is on the left of the formula because it goes on the vertical axis and the numeric vector `Mileage` is on the right because it goes on the horizontal axis.



*Figure 9.6: Box plot.*

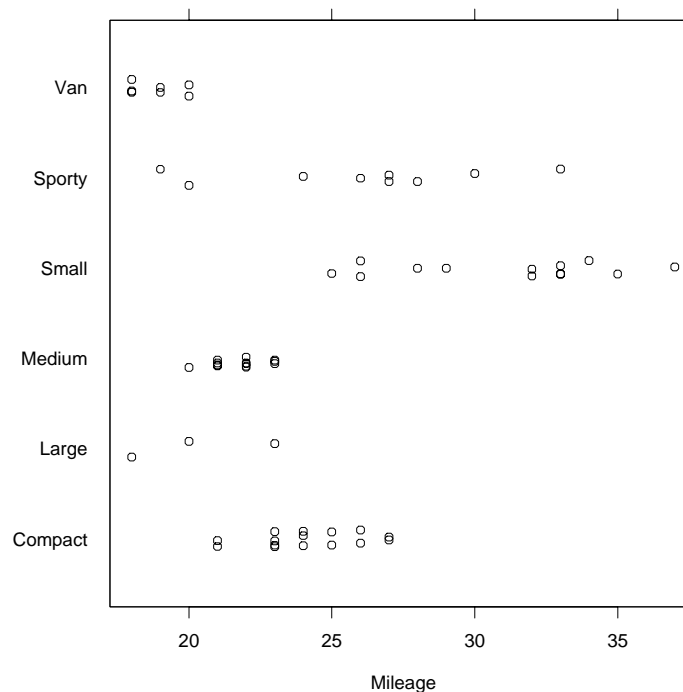
**stripplot**

A strip plot, sometimes called a one-dimensional scatterplot, is similar to a box plot in general layout but the individual data points are shown instead of the box plot summary.

Figure 9.7 is a stripplot:

```
> stripplot(Type ~ Mileage, data = fuel.frame,
+ jitter = TRUE, aspect = 1)
```

Setting `jitter = TRUE` causes some random noise to be added vertically to the points to alleviate the overlap of the plotting symbols. When `jitter = FALSE`, the default, the points for each level lie on a horizontal line.



*Figure 9.7: Strip plot.*



**qq**

The quantile-quantile plot, or qqplot, is an extremely powerful tool for comparing the distributions of two sets of data. The idea is quite simple; quantiles of one data set are graphed against corresponding quantiles of the other data set.

The variable `fuel.frame$Type` has five levels:

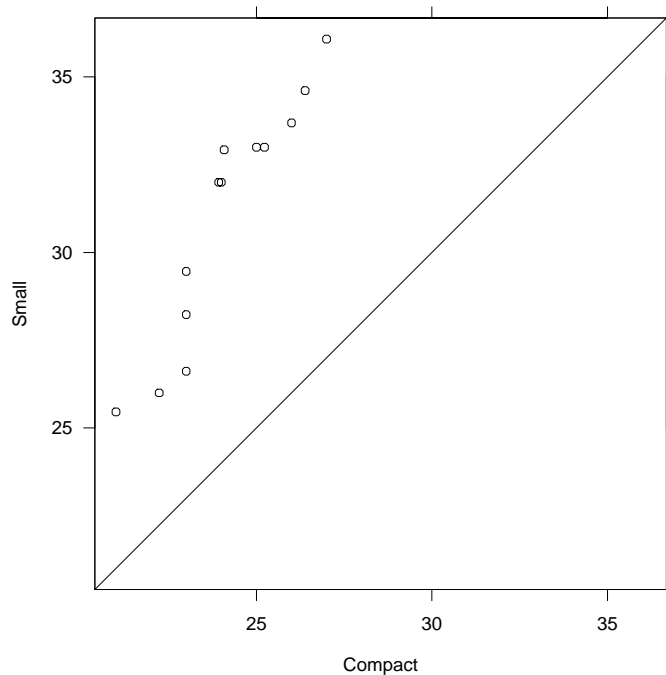
```
> table(fuel.frame$Type)

Compact Large Medium Small Sporty Van
      15      3      13      13      9      7
```

Figure 9.8 is a qqplot comparing the quantiles of mileage for compact cars with the corresponding quantiles for small cars:

```
> qq(Type ~ Mileage, data = fuel.frame, aspect = 1,
+ subset = (Type == "Compact") | (Type == "Small"))
```

The factor on the left side of the formula must have two levels. The default labels for the two scales are the names of the levels.



*Figure 9.8: qqplot.*

**qqmath**

Normal probability plots, or normal qqplots, are the single most powerful tool for determining if the distribution of a set of measurements is well approximated by the normal distribution.

Figure 9.9 is a normal probability plot of the mileages for small cars:

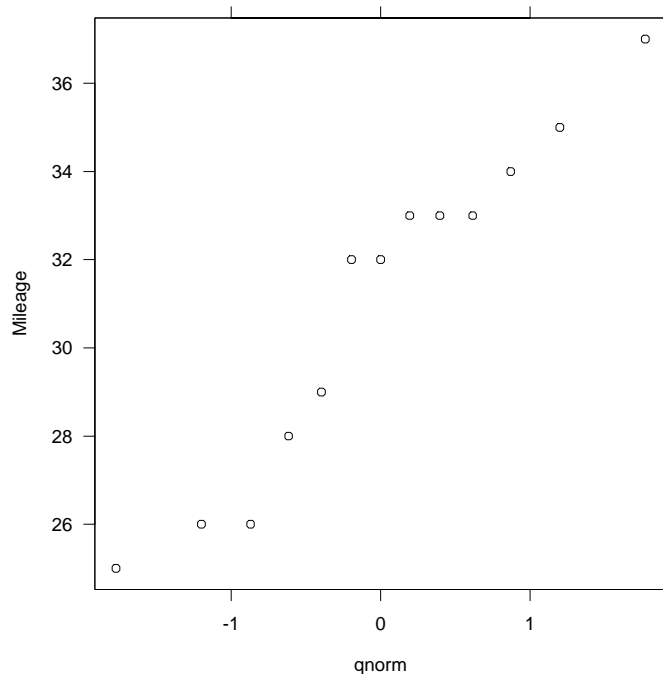
```
> qqmath( ~Mileage, data = fuel.frame,  
> subset = (Type == "Small"))
```

That is, the ordered data are graphed against quantiles of the standard normal distribution. The formula for `qqmath` is used in a way unlike any of the previous examples. Only one data object appears in the formula, to the right of the `~`, because this graphical method utilizes only one data object.

If we used

```
> qqmath( ~Mileage, data = fuel.frame,  
+ subset = (Type == "Small"), aspect = 1,  
+ distribution = qexp)
```

the result would be an exponential probability plot. Note that the name of the function appears as the default label on the horizontal scale of the plot.



*Figure 9.9: Normal probability plot.*

**dotplot**

The dot plot, which displays data with labels, provides highly accurate visual decodings, typically far more accurate than other methods for displaying labeled data. Let us compute the mean mileage for each vehicle type:

```
> mileage.means <- tapply(fuel.frame$Mileage,
+ fuel.frame$Type, mean)
```

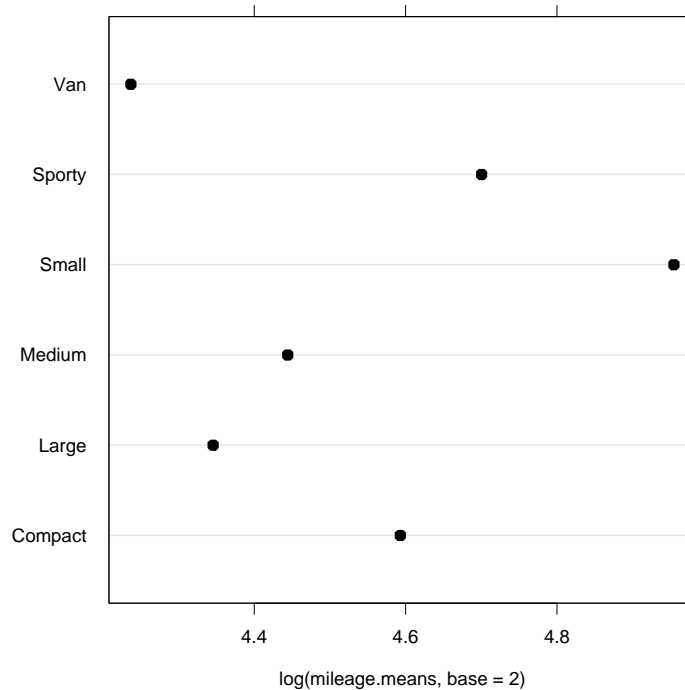
Figure 9.10 is a dot plot of the log base 2 means:

```
> dotplot(names(mileage.means) ~ log(mileage.means, base=2),
+ aspect = 1, cex = 1.25)
```

The argument `cex` is passed to the `panel` function to change the size of the dot of the dot plot; more on this in the section `Panel Functions` (page 378).

Notice that the vehicle types in Figure 9.10 are ordered, from bottom to top, by the order of the elements of the vector `mileage.means`. If you wanted the graph to show the values from smallest to largest going from bottom to top, you could first redefine `mileage.means`:

```
> mileage.means <- sort(mileage.means)
```



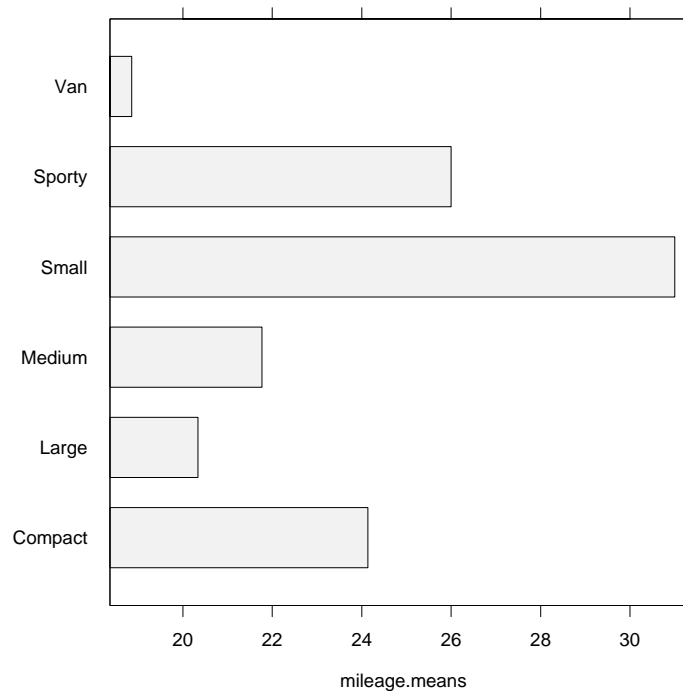
*Figure 9.10: Dot plot.*

**barchart**

Overall, dot plots are a more effective display method than bar charts, avoiding some of the perceptual problems of bar charts. Still, there are circumstances where bar charts are harmless.

Figure 9.11 is a bar chart of the mileage means (without logs):

```
> barchart(names(mileage.means) ~ mileage.means, aspect = 1)
```



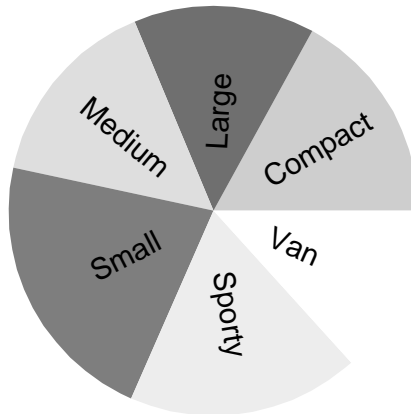
*Figure 9.11: Bar chart.*

**piechart**

Pie charts have severe perceptual problems. Experiments in graphical perception have shown that compared with dot plots, they convey information far less reliably. But if you want to display some data, and perceiving the information is not so important, then a pie chart is fine.

Figure 9.12 is a pie chart of the mileage means:

```
> piechart(names(mileage.means) ~ mileage.means)
```



*Figure 9.12: Pie chart.*

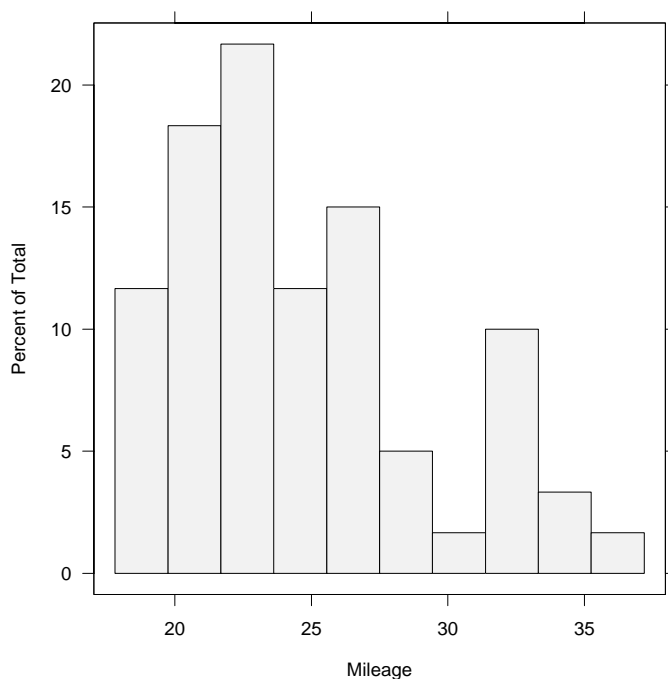
**histogram**

A histogram can be useful for showing the distribution of a single set of data, but two or more histograms are typically not nearly as powerful as a box plot or qqplot for comparing data distributions.

Figure 9.13 is a histogram of mileage:

```
> histogram( ~Mileage, data= fuel.frame, aspect= 1,  
+ nint= 10)
```

The argument `nint` determines the number of intervals. The histogram algorithm chooses the intervals to make the bar widths be simple numbers while trying to make the number of intervals as close to `nint` as possible.



*Figure 9.13: Histogram.*

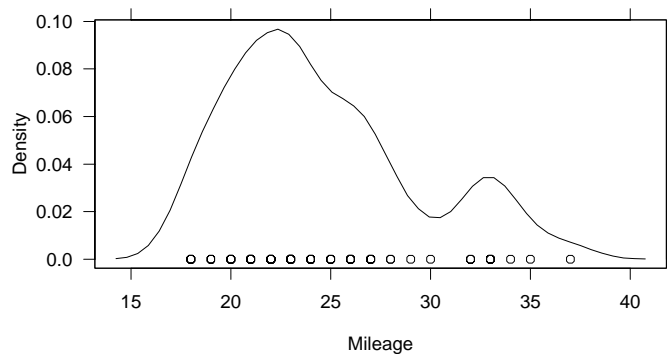
**densityplot**

Like histograms, density plots can be of help in understanding the distribution of a single set of data, but box plots and qqplots typically give more incisive comparisons of distributions.

Figure 9.14 is a density plot of mileage:

```
> densityplot( ~Mileage, data = fuel.frame, aspect = 1/2,  
+ width = 5)
```

The argument `width` controls the width of the smoothing window in the same units as the data, mpg here; as the width increases, the smoothness increases.



*Figure 9.14: Density plot.*

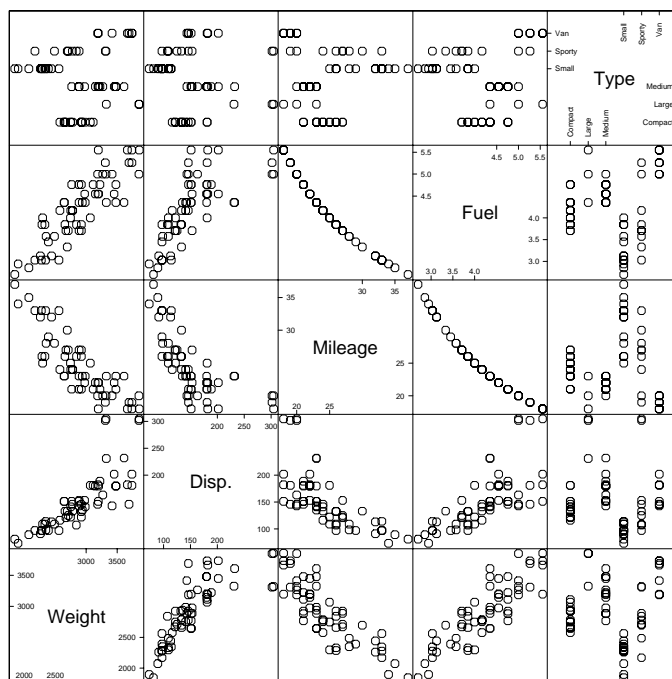
**splom**

The scatterplot matrix is an exceedingly powerful tool for displaying measurements of three or more variables.

Figure 9.15 is a scatterplot matrix of the variables in `fuel.frame`:

```
> splom( ~fuel.frame)
```

Note that the factor `Type` has been converted to a numeric variable and plotted just like the other variables, which are numeric. The six levels of `Type` simply take the values 1 to 6 in this conversion.



*Figure 9.15: Scatterplot matrix.*

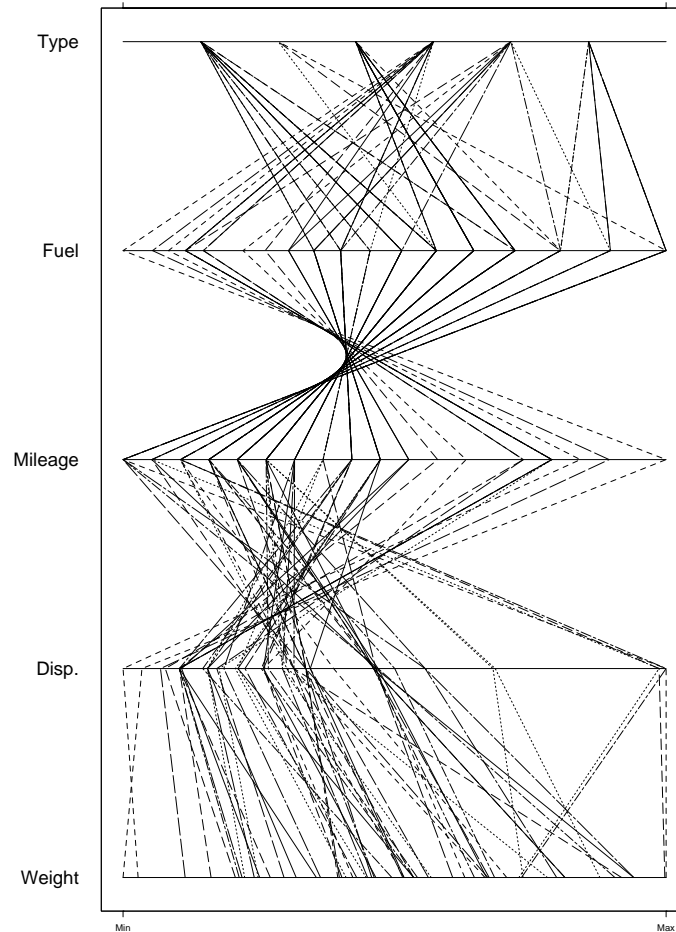


**parallel**

Parallel coordinates are an interesting method, but it is unclear at the time of this writing whether they have the power to uncover structure that is not more readily apparent using other graphical methods.

Figure 9.16 is a parallel coordinates display of the variables in `fuel.frame`:

```
> parallel (~fuel.frame)
```



*Figure 9.16: Parallel coordinates display.*

## A Data Set: gauss

To further illustrate the general display routines, we will compute a function of two variables over a grid.

```
> datax <- rep(seq(-1.5, 1.5, length = 50), 50)
> datay <- rep(seq(-1.5, 1.5, length = 50), rep(50, 50))
> dataz <- exp(-(datax^2 + datay^2 + datax*datay))
> gauss <- data.frame(datax, datay, dataz)
```

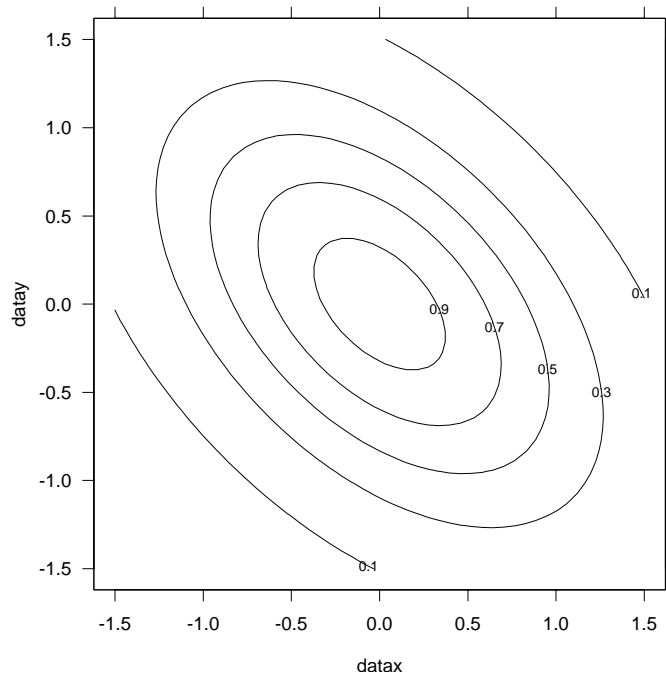
Thus `dataz` is the exponential of a quadratic function defined over a 50x50 grid; in other words, the surface is proportional to a bivariate normal density.

## contourplot

Contour plots are helpful displays for studying a function,  $f(x,y)$ , when we have no need to study the conditional dependence of  $f$  on  $x$  given  $y$  or of  $f$  on  $y$  given  $x$ . Conditional dependence is revealed far better by multipanel conditioning. Figure 9.17 is a contour plot of the gaussian surface:

```
> contourplot(dataz ~ datax * datay, data = gauss,
+ aspect = 1, at = seq(.1, .9, by = .2))
```

The argument `at` specifies the values as which the contours are to be computed and drawn. If no argument is specified, default values are chosen.



*Figure 9.17: Contour plot.*

**levelplot**

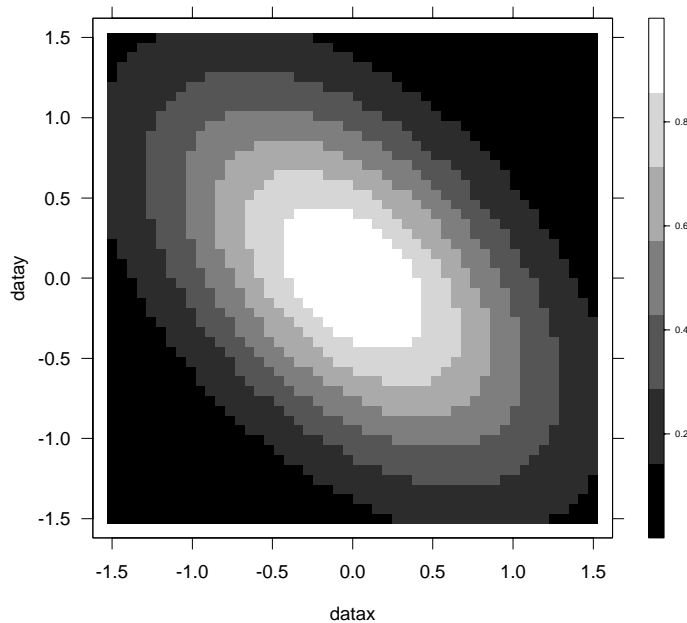
Level plots are also helpful displays for studying a function,  $f(x,y)$ . They are no better than contour plots when the function is simple, but often are better when there is much fine detail, for example, many peaks and valleys.

Figure 9.18 is a level plot of the gauss surface:

```
> levelplot(dataz ~ datax * datay, data = gauss, aspect = 1,
+ cuts = 6)
```

The values of the surface are encoded by color, a gray scale in this case. For devices with full color, the scale goes from pure magenta to white and then to pure cyan. If the device does not have full color, a gray scale is used.

For a level plot, the range of the function values is divided into intervals and each interval is assigned a color. A rectangle centered on each grid point is given the color of the interval containing the value of the function at the grid point. In Figure 9.18, there are six intervals. The argument `cuts` specifies the number of breakpoints between intervals.



*Figure 9.18: Level plot.*

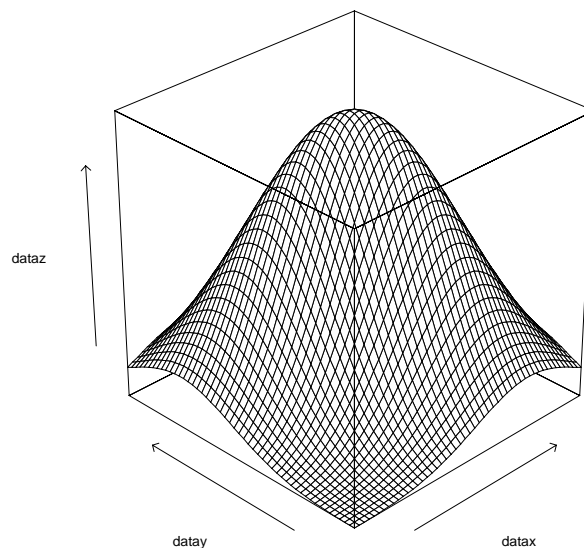
**wireframe**

Wireframe displays can be quite useful for displaying  $f(x,y)$  when we have no need to study conditional dependence. Figure 9.19 is a 3-D wireframe plot of the gauss surface:

```
> wireframe(dataz ~ datax * datay, data = gauss, drape = F,
+ screen = list(z=45, x=-60, y=0))
```

The argument `screen` is a list. The three components of the list— `x`, `y`, and `z`—refer to screen axes. The first component is horizontal and the second is vertical, both in the plane of the screen. The third component is perpendicular to the screen. The surface is rotated about these axes in the order given in the list. Here is how it worked for Figure 9.19. The surface began with `datax` as the horizontal screen axis, `datay` as the vertical, and `dataz` as the perpendicular. The origin was at the lower left in the back. First, the surface was rotated 45 degrees about the perpendicular screen axis, where a positive rotation is counterclockwise. Then, there was a -60 degrees rotation about the horizontal screen axis, where a negative rotation brings the picture at the top of the screen away from the viewer and the bottom toward the viewer. Finally, there was no rotation about the vertical screen axis; had there been one with a positive number of degrees, then the left side of the picture would have moved toward the viewer and the right away.

If `drape=T`, a color encoding is added to the surface using the same encoding method of the level plot.



*Figure 9.19: 3D wireframe plot.*

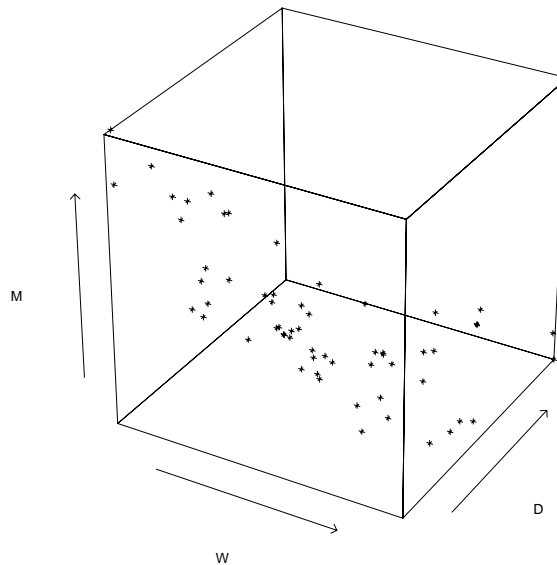
**cloud**

A static 3-D plot of a scatter of points is typically not effective because the depth cues are insufficient to give a strong 3-D effect. Still, on rare occasions, such a plot can be useful, sometimes as a presentation or teaching tool.

Figure 9.20 is a 3-D scatterplot of the first three variables in the data frame `fuel.frame`:

```
> cloud(Mileage ~ Weight * Disp., data = fuel.frame,
+ screen = list(z=-30, x=-60, y = 0), xlab = "W", ylab = "D",
+ zlab = "M")
```

The behavior of the argument `screen` is the same as that for `wireframe`. We have used three additional arguments to specify scale labels; such labeling will be discussed in the section *Scales and Labels* (page 374).



*Figure 9.20: 3D scatterplot, or cloud.*

## The Display Functions and Their Formulas

The following listing of the general display functions and their formulas is instructive because it shows certain conventions and consistencies in the formula mechanism:

### Graph One Numerical Variable Against Another

```
xyplot(numeric1 ~ numeric2)
```

**Compare the Sample Distributions of Two or More Sets of Data**

```
bwplot(factor ~ numeric)
stripplot(factor ~ numeric)
qq(factor ~ numeric)
```

**Graph Measurements with Labels**

```
dotplot(character ~ numeric)
barchart(character ~ numeric)
piechart(character ~ numeric)
```

**Graph the Sample Distribution of One Set of Data**

```
qqmath(~ numeric)
histogram(~ numeric)
densityplot(~ numeric)
```

**Graph Multivariate Data**

```
splom(~ data.frame)
parallel(~ data.frame)
```

**Graph a Function of Two Variables Evaluated on a Grid**

```
contourplot(numeric1 ~ numeric2 * numeric3)
levelplot(numeric1 ~ numeric2 * numeric3)
wireframe(numeric1 ~ numeric2 * numeric3)
```

**Graph Three Numerical Variables**

```
cloud(numeric1 ~ numeric2 * numeric3)
```

## ARRANGING SEVERAL GRAPHS ON ONE PAGE

Several graphs, made separately by Trellis display functions, can be displayed on a single page. There is one restriction. None of the individual graphs may be a multipanel conditioning display with more than one page.

**print**

Figure 9.21 shows two graphs arranged on one page:

```
> attach(fuel.frame)
> box.plot <- bwplot(Type ~ Mileage)
> scatter.plot <- xyplot(Mileage ~ Weight)
> detach()
> print(box.plot, position = c(0, 0, 1, .4), more = T)
> print(scatter.plot, position = c(0, .35, 1, 1))
```

The argument `position` specifies the position of each graph on the page using a page coordinate system in which the lower left corner of the page is (0, 0) and the upper right corner is (1, 1). The *graph rectangle* is the portion of the page allocated to a graph. `position` takes a vector of four numbers; the first two numbers are the coordinates of the lower left corner of the graph rectangle, and the second two numbers are the coordinates of the upper right corner. The argument `more` has been give a value of `T`, which says that more drawing is coming.

Notice that in the above example the graph rectangles overlap somewhat. Here is the reason. The graph contains margins (empty space) around the edges of the graph. But in arranging graphs on a page, we might well want to overlap margin space to use the page space as efficiently as possible.

The following code illustrates another argument, `split`, that provides a different method for arranging the plots on the page:

```
> attach(fuel.frame)
> scatter.plot <- xyplot(Mileage ~ Weight)
> other.plot <- xyplot(Mileage ~ Disp.)
> detach()
> print(scatter.plot, split = c(1, 1, 1, 2), more = T)
> print(other.plot, split = c(1, 2, 1, 2))
```

`split` takes a vector of four values. The last two define an array of subregions in the graphics region. In our example, the array has one column and two rows for both plots. The first two values of `split` prescribe the subregion in which the current plot is to be drawn.

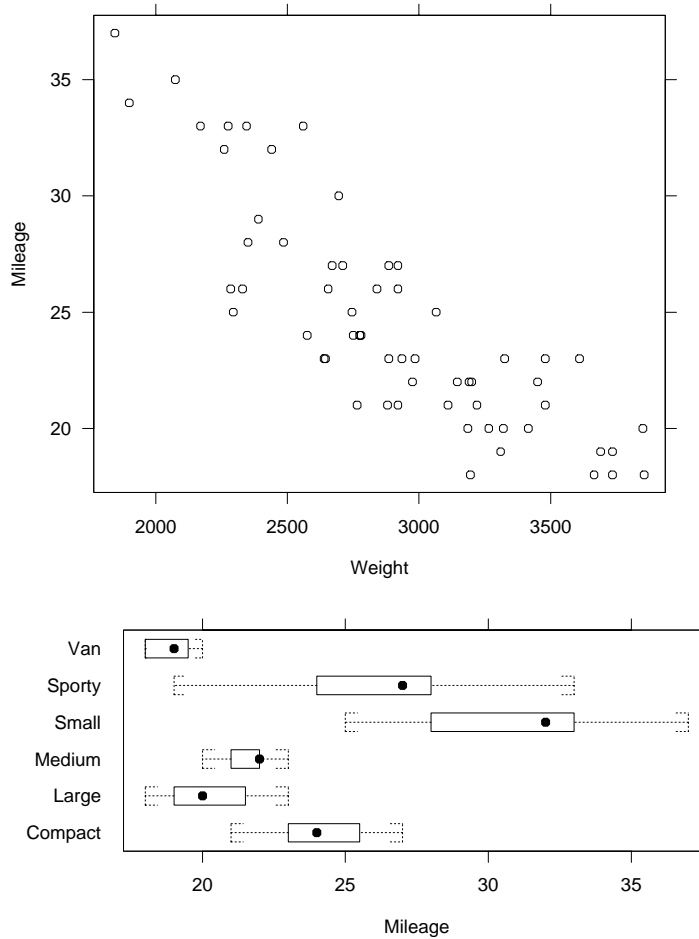


Figure 9.21: Multiple graphs on a page.



# MULTIPANEL CONDITIONING

## A Data Set: barley

The data frame `barley` contains data from an experiment carried out in Minnesota in the 1930s. At six sites, ten varieties of barley were grown in each of two years. The data collected for the experiment are the yields in bushels/acre for all combinations of site, variety, and year, so there are  $6 \times 10 \times 2 = 120$  observations (yield is numeric, the others are factors).

```
> names(barley)

[1] "yield" "variety" "year" "site"
```

## About Multipanel Display

Figure 9.22 uses multipanel conditioning to display the barley data. Each panel displays the yields of the ten varieties for one year at one site; variety is graphed along the vertical scale and yield is graphed along the horizontal scale. For example, the lower left panel displays values of variety and yield for Grand Rapids in 1932. The *panel variables* are `yield` and `variety` and the *conditioning variables* are `year` and `site`.

## formula argument

Figure 9.22 was made by the following command:

```
> dotplot(variety ~ yield | year * site, data = barley)
```

The `|` is read as “given”. Thus the formula is read as `variety` “is graphed against” `yield` “given” `year` and `site`. Thus a simple use of formula creates a complex multipanel display.

## Columns, Rows, and Pages

A multipanel conditioning display is a three-way rectangular array laid out into columns, rows, and pages. In Figure 9.22, there are two columns, six rows and one page. The numbers of columns, rows, and pages are selected by an algorithm that attempts to fill up as much of the graphics region as

possible subject to certain constraints. As we will see in the section Summary: The Layout of a Multipanel Display (page 368), there is an argument layout that allows you to choose the numbers.

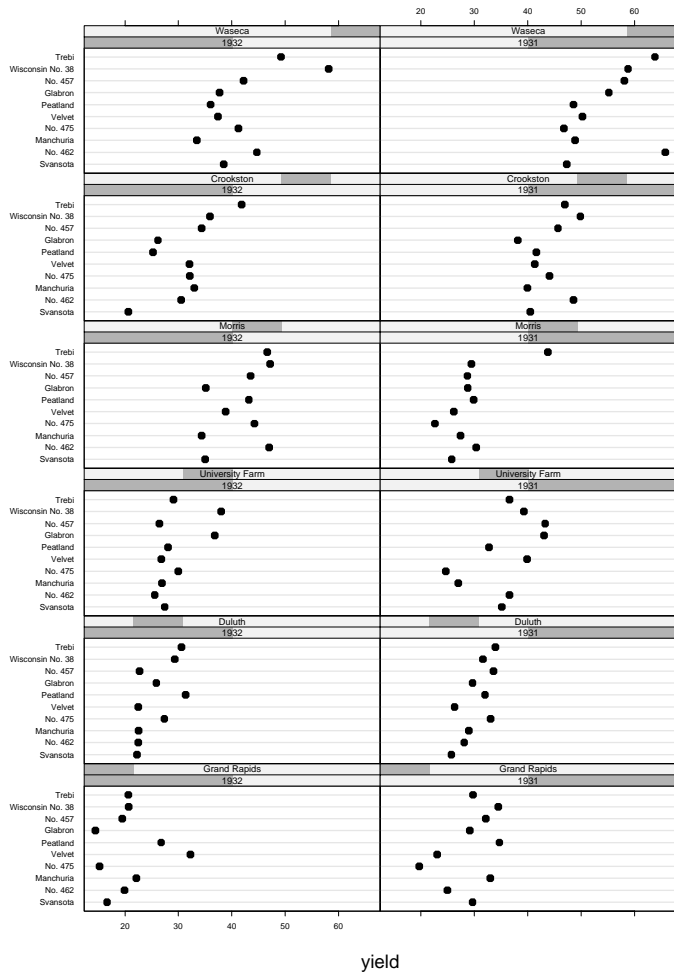


Figure 9.22: Multipanel conditioning on the barley data.

# Packet Order and Panel Order

In the above formula, the conditioning variable year appeared first and site appeared second. This gives an explicit ordering to the conditioning variables. Each of these variables is a factor with levels:

```
> levels(barley$year)
```

```
[1] "1932" "1931"
```

```
> levels (barley$site)
```

```
[1] "Grand Rapids" "Duluth" "University Farm"
```

```
[4] "Morris" "Crookston" "Waseca"
```

The levels of each factor are ordered by their order of appearance in the `levels` attribute. As we will discuss shortly, we can control the order by making the factor an *ordered factor*. A *packet* is information sent to a panel for display. For Figure 9.22, each packet includes the values of `variety` and `yield` for a particular combination of `year` and `site`. Packets are ordered by the orderings of the conditioning variables and their levels; the levels of the first conditioning variable vary the fastest, the levels of the second conditioning variable vary the next fastest, and so forth. For Figure 9.22, the order of the packets is

```
1932 Grand Rapids
1931 Grand Rapids
1932 Duluth
1931 Duluth
1932 University Farm
1931 University Farm
1932 Morris
1931 Morris
1932 Crookston
1931 Crookston
1932 Waseca
1931 Waseca
```

The panels of a multipanel display are also ordered. The bottom left panel is panel one. From there we move fastest through the columns, next fastest through the rows, and the slowest through the pages. The panel ordering rule is like a graph, not like a table; the origin is at the lower left and as we move either from left to right or from bottom to top, the panel order increases. The following shows the panel order for Figure 9.22, which has two columns, six rows, and one page:

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

In Trellis Graphics, packets are assigned to panels according to the packet order and the panel order. Packet 1 goes in panel 1, packet 2 goes into panel 2 and so forth. In Figure 9.22, the two orderings result in the `year` variable changing along the columns and the `site` variable changing along the rows. Note that as the levels for one of these factors increase, the darkened bars in the strip label for the factor move from left to right.

## layout Argument

Multipanel conditioning is a powerful tool for understanding how a response depends on two or more explanatory variables. In such an analysis, it is typically important to make as many displays as necessary to have each explanatory variable appear at least once as a panel variable. In Figure 9.22, `variety`, an explanatory variable, appears as a panel variable.

We will make a new display with `site` as a panel variable. The argument `layout` specifies the numbers of columns, rows, and pages:

```
> dotplot(site ~ yield | year * variety, data = barley,  
+ layout = c(2, 5, 2))
```

The result is shown in Figure 9.23, the first page, and in Figure 9.24, the second page.

If we do not specify `layout`, Trellis Graphics chooses the numbers of columns, rows, and pages by a layout algorithm. The algorithm takes into account the aspect ratio, the number of packets, the number of conditioning variables, and the number of levels of each conditioning variable. It chooses the numbers to maximize the size of the graph within the graphics region.

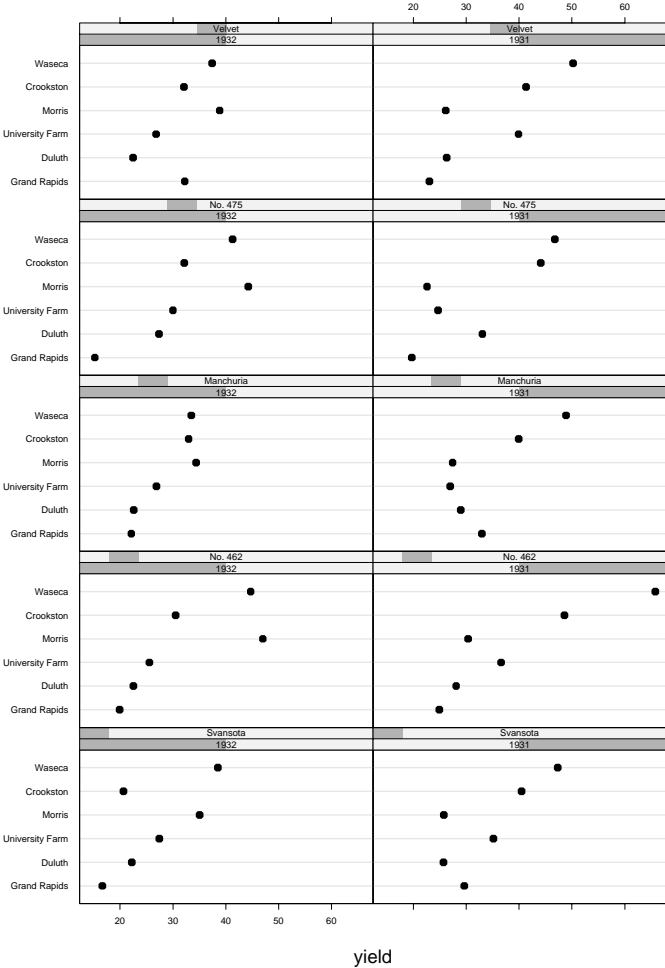


Figure 9.23: The first page of the multipage plot of the barley data.

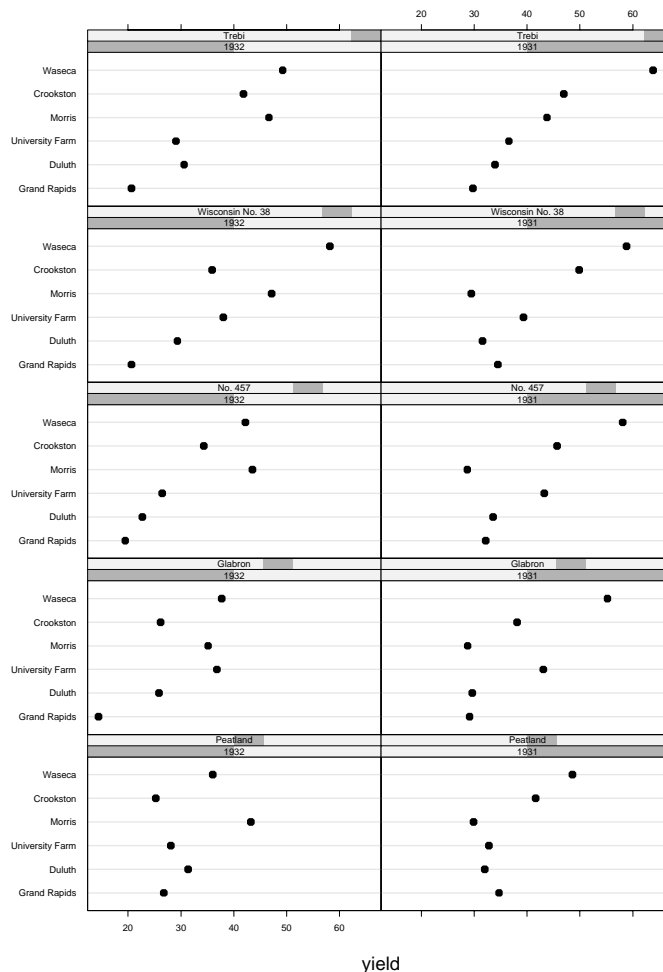


Figure 9.24: The second page of the multipage plot of the barley data.

## Main-Effects Ordering

For the barley data, the explanatory variables are categorical. The data set for each is a factor. (Since there are only two years, the `year` variable is treated as a factor rather than a numeric vector.) For each factor, consider the median yield for each level. For example, for `variety`, the level medians are

```
> variety.medians <- tapply (barley$yield, barley$variety,
+ median)
```

```
> variety.medians
```

Svansota	No. 462	Manchuria	No. 475	Velvet	Peatland
28.55	30.45	30.96667	31.06667	32.15	32.38334
Glaron	No. 457	Wisconsin	No. 38	Trebi	
32.4	33.96666		36.95	39.2	

The barley displays in Figure 9.22 to Figure 9.24 use an important display method: *main-effects ordering of levels*. This greatly enhances our ability to perceive effects. Consider Figure 9.22. On each panel, the varieties are ordered from bottom to top by the variety medians; Svansota has the smallest median and Trebi has the largest. The site panels have been ordered from bottom to top by the site medians; Grand Rapids has the smallest median and Waseca has the largest. Finally, the year panels are ordered from left to right by the year medians; 1932 has the smaller median and 1931 has the larger.

This median ordering is achieved by making the data set for each explanatory variable an ordered factor, where the levels are ordered by the medians. For example, suppose `variety` started out as a factor without the median ordering. We get the ordered factor through the following:

```
> barley$variety <-ordered (barley$variety,
+ levels = names(sort(variety.medians)))
```

## reorder.factor

Main effects ordering is so important and is carried out so often that Trellis Graphics includes a function `reorder.factor` to carry it out. Here, it is used to reorder `variety`:

```
> barley$variety <-reorder.factor (barley$variety,
+ barley$yield, median)
```

The first argument is the factor to be reordered, the second is the data on whose main effects the reordering is based, and the third argument is the function to be applied to the second argument to compute main effects.

## Controlling the Pages of a Multipage Display

If a multipage display is sent to a screen device, the default behavior is that all pages will be drawn, and stepped through using the tabs at the bottom of the pages.

## Summary: The Layout of a Multipanel Display

To layout a multipanel display in a certain way you specify the following:

- An ordering of the conditioning variables by the order you enter them in the argument `formula`.
- An ordering of the levels of each factor, possibly by creating an ordered factor.
- The number of columns, rows, and pages through the argument `layout`.

## A Data Set: ethanol

The data frame `ethanol` contains three variables from an industrial experiment with 88 runs:

```
> names (ethanol)
```

```
[1] "NOx" "C" "E"
```

```
> dim (ethanol)
```

```
[1] 88 3
```

The concentrations of oxides of nitrogen (NO<sub>x</sub>) in the exhaust of an engine were measured for different settings of compression ratio (C) and equivalence ratio (E). These measurements were part of the same experiment that produced the measurements in the data frame `gas` introduced in the section A Data Set: `gas` (page 335).

## Conditioning on Discrete Values of a Numeric Variable

For the barley data, the explanatory variables are factors, so it is natural to condition on the levels of each factor. This is not the case for the ethanol data; both explanatory variables, C and E, are numeric. Suppose for the ethanol data, that we want to graph NO<sub>x</sub> against E given C. The variable C has five unique values; in other words, the variable, while numeric, is discrete:

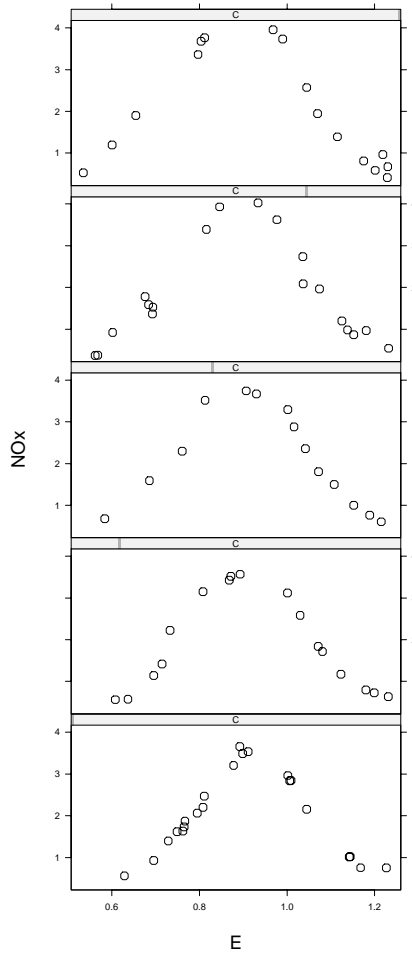
```
> table (ethanol$C)
```

```
7.5 9 12 15 18  
22 17 14 19 16
```



It makes sense then to condition on the unique values of  $C$ . Figure 9.25 does this:

```
> xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2)
```



*Figure 9.25: Multipanel conditioning.*

When a numeric variable is used as a conditioning variable in the argument formula, then conditioning is automatically carried out on the sorted unique values. In other words, the levels of the variable in such a case are the unique values. The order of the levels is from smallest to largest. For  $C$ , the first level is 7.5, the second 9, and so forth. Thus the first packet includes values of  $\text{NOx}$  and  $E$  for  $C = 7.5$ , the second packet includes the values for  $C$

= 9, and so on. The packets fill the panels according to the packet order and the panel order. In Figure 9.25, the values of  $C$ , which are indicated by the darkened bars in the strip labels, increase from bottom to top.

## Conditioning on Intervals of a Numeric Variable

For the ethanol data we graphed  $NOx$  against  $E$  given  $C$  in Figure 9.25. We would like to see  $NOx$  against  $C$  given  $E$  as well. But  $E$  varies in a nearly continuous way; there are 83 unique values out of total of 88 values. Clearly we cannot condition on single values. Instead, we condition on intervals. This is done in Figure 9.26. On each panel,  $NOx$  is graphed against  $C$  for  $E$  in an interval. The intervals, which are portrayed by the darkened bars in the strip, are ordered from low to high, so as we go left to right and bottom to top through the panels, the intervals go from low to high. The intervals overlap. The next section describes how they were created and the expression that produced the graph.

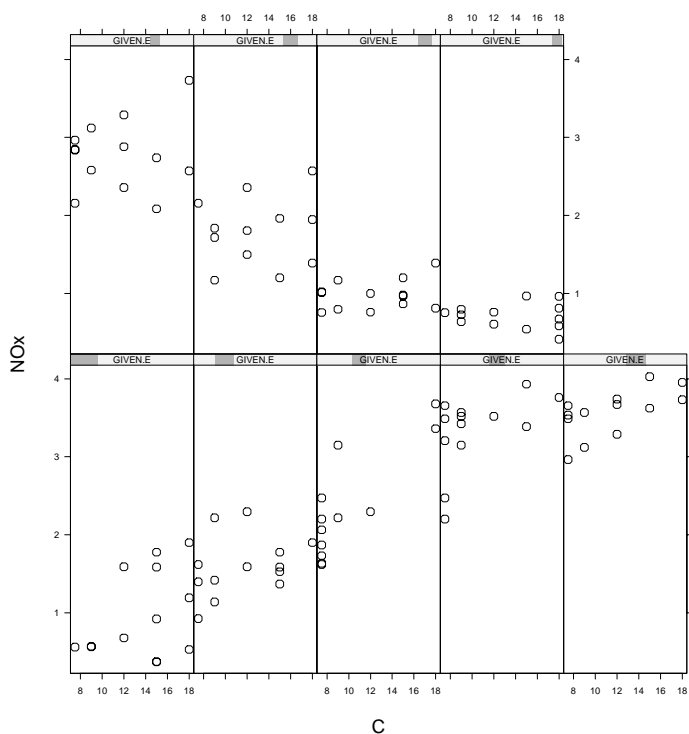


Figure 9.26: *Conditioning intervals.*

**equal.count**

The nine intervals in Figure 9.26 were produced by the *equal count algorithm*.

```
> GIVEN.E <- equal.count (ethanol$E, number= 9,
+ overlap= 1/4)
```

There are two inputs to the algorithm, the number of intervals and a target fraction of points to be shared by each pair of successive intervals. In Figure 9.26, the inputs are 9 and 1/4. The algorithm picks interval endpoints that are values of the data; the left endpoint of the lowest interval is the minimum of the data, and the right endpoint of the highest interval is the maximum of the data. The endpoints are chosen to make the counts of points in the intervals as nearly equal as possible, and the fractions of points shared by successive intervals as close to the target fraction as possible.

The command that produced Figure 9.26 is

```
> xyplot(NOx ~ C | GIVEN.E, data = ethanol, aspect = 2.5)
```

The aspect ratio was chosen to be 2.5 to approximately bank the underlying pattern of the points to 45 degrees. Notice that the automatic layout algorithm chose five columns and two rows.

**shingle**

The result of `equal.count` is an object of class `shingle`. The class is named “shingle” because of the overlap, like shingles on a roof. First, a shingle contains the numerical values of the variable and can be treated as an ordinary numeric variable:

```
> range (GIVEN.E)

[1] 0.535 1.232
```

Second, a shingle has the intervals attached as an attribute. There is a plot method, a special Trellis function, that displays the intervals. Figure 9.27 shows the intervals of `GIVEN.E`:

```
> plot (GIVEN.E)
```

You can use the function `levels` to extract the intervals from the shingle:

```
> levels (GIVEN.E)

      min      max
0.535 0.686
0.655 0.761
0.733 0.811
0.808 0.899
0.892 1.002
0.990 1.045
```

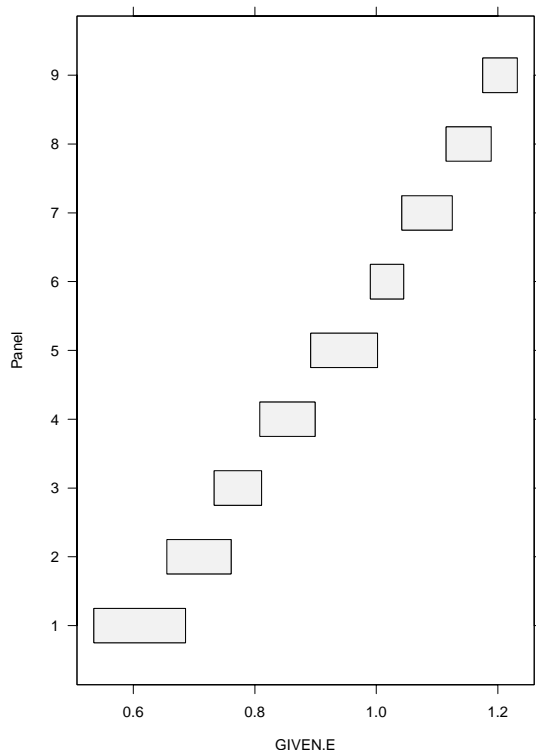
```
1.042 1.125  
1.115 1.189  
1.175 1.232
```

A shingle can be specified directly by the function `shingle`. For example, the following creates five intervals of equal width and no overlap for the variable `ethanol $E`:

```
> endpoints <- seq(min(ethanol $E), max(ethanol $E),  
+ length = 6)  
> GIVEN.E <- shingle(ethanol $E,  
+ intervals = cbind(endpoints[-6], endpoints[-1]))  
> levels(GIVEN.E)
```

```
      min      max  
0.5350 0.6744  
0.6744 0.8138  
0.8138 0.9532  
0.9532 1.0926  
1.0926 1.2320
```

The argument `intervals` is a two-column matrix; holding the left endpoints and the right endpoints of the intervals, respectively.



*Figure 9.27: Plotting intervals using shingles.*

## SCALES AND LABELS

The functions presented in the section General Display Functions (page 341) have arguments that specify the scales and labels of graphs. These arguments are discussed in this section.

### **xlab, ylab, main and sub arguments**

To produce a scatterplot of NO<sub>x</sub> against E for the gas data, which were introduced in the section A Data Set: gas (page 335):

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2)
```

The plot will show the label for the horizontal, or x, scale, and the label for the vertical, or y, scale are taken from the names used in the argument formula. We can specify these scale labels as well as a main title at the top and a subtitle at the bottom, using the following code:

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2,
+ xlab = "Equivalence Ratio", ylab = "Oxides of Nitrogen",
+ main = "Air Pollution", sub = "Single-Cylinder Engine")
```

Each of these four label arguments can also be a list. The first component of the list is a new character string for the text of the label. The other components specify the size, font, and color of the text. The component `cex` specifies the size; `font`, a positive integer, specifies the font; and `col`, a positive integer, specifies the color. The following code changes the sizes of the title and subtitle:

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2,
+ xlab = "Equivalence Ratio", ylab = "Oxides of Nitrogen",
+ main = list("Air Pollution", cex = 2),
+ sub = list("Single-Cylinder Engine", cex = 1.25))
```

### **xlim and ylim arguments**

In Trellis, the upper value of the scale line for a numeric variable is the maximum of the data to be plotted plus 4% of the range of the data. Similarly, the lower value of the scale line for a numeric variable is the minimum of the data to be plotted minus 4% of the range of the data. The 4% helps prevent the data values from running into the edge of the plot.

We can alter the extremes of the horizontal scale line by the argument `xlim`, a vector of two values. The first value replaces the minimum of the data in the above procedure, and the second value replaces the maximum. Similarly, we can alter the vertical scale by the `ylim` argument.

In plots created with the code listed above, NO<sub>x</sub> is graphed along the vertical scale. The limits of this variable are:

```
> range(gas$NOx)
```

```
[1] 0.537 5.344
```

To include the values 0 and 6 in the vertical scale:

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2, ylim = c(0, 6))
```

## scales and pscales arguments

The argument `scales` affects tick marks and tick mark labels. In the plot produced by the code above, there would be seven tick marks and tick mark labels along the vertical scale and six along the horizontal. The function `scales` is used to reduce the number of ticks and increase the size of the tick labels:

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2, ylim = c(0, 6),  
+ scales = list(cex = 2, tick.number = 4))
```

The argument `scales` is a list. The list component `cex` affects the size. The list component `tick.number` affects the number, but it is just a suggestion; an algorithm tries to find tick values that are pretty, while trying to come as close as possible to the specified number.

We can also specify the tick marks and labels separately for each scale. The specification

```
> scales = list(cex = 2, x = list(tick.number = 4),  
+ y = list(tick.number = 10))
```

changes `cex` on both scales, but `tick.number` has been set to 4 for the horizontal, or `x`, scale, and has been set to 10 for the vertical, or `y`, scale. Thus the rule is this: specifications for the horizontal scale appear in the argument `scales` as a component `x` that is itself a list, specifications for the vertical scale appear in `scales` as a component `y` that is a list, and specifications for both scales appear as remaining components of the argument `scales`.

There is an exception to the behavior of the `scales` argument. The two 3-D general display functions `wireframe` and `cloud` currently do not accept changes to each scale separately; in other words, components `x`, `y`, and `z` cannot be used. The general display function `piechart` has no tick marks and labels, so the argument `scales` does not apply at all. The general display function `splom` has many scales, so the same delicate control is not available, but more limited control is available through the argument `pscales`. See the on-line help for `pscales` for more details.

### 3-D Display: aspect Argument

The aspect ratio, the height of a panel data region divided by the width, is controlled by the `aspect` argument. This argument was introduced in the section Aspect Ratio (page 339) for 2-D displays. The behavior of the `aspect` argument for the two 3-D general display functions, `wireframe` and `cloud`, is somewhat different. Since there are three axes, we must specify two aspect ratios to specify the shape of the 3-D box around the data. Suppose the formula and the `aspect` arguments are

```
formula = z ~ x * y, aspect = c(1, 2)
```

Then the ratio of the length of the y-axis to the length of the x-axis is 1, and the ratio of the length of the z-axis to the length of the x-axis is 2.

### Changing the Text in Strip Labels

The default text in the strip label for a numeric conditioning variable is the name of the variable. This can be illustrated with the code below, which displays the ethanol data introduced in the section A Data Set: ethanol (page 368):

```
> xyplot(NOx ~ E | C, data = ethanol)
```

The default text in the strip label of a factor conditioning variable is the name of the factor level for the panel. The barley data introduced in the section A Data Set: barley (page 361) illustrates this:

```
> dotplot(variety ~ yield | year * site, data = barley)
```

The name of the factor, for example, `site`, does not appear because seeing the names of the levels is typically enough to convey the name of the factor.

Thus the text comes from the names given to variables and factor levels in the data sets that are plotted. If we want to change the text we can change the names. For example, if we want to change the long label “University Farm” to “U. Farm” then we can change the names of the levels of the factor `site`:

```
> levels(barley$site)

[1] "Grand Rapids" "Duluth" "University Farm"
[4] "Morris" "Crookston" "Waseca"

> levels(barley-site)[3] <- "U. Farm"
```



**par.strip.text  
argument**

The size, font, and color of the text in the strip labels can be changed by the argument `par.strip.text`, a list whose components are the parameters `cex` for size, `font` for the font, and `col` for the color. For example, we can make huge strip labels by

```
> par.strip.text = list(cex = 2)
```

**strip argument**

The argument `strip` allows very delicate control of what is put in the strip labels. One usage is to remove the strip labels altogether:

```
strip = F
```

Another is to control the inclusion of names of conditioning variables in strip labels.

```
> dotplot(variety ~ yield | year * site, data = barley,
+ strip = function(...) strip.default(..., strip.names =
+ c(T, T)))
```

The argument `strip.names` takes a logical vector of length two. The first element tells whether or not the names of factors should be included along with the names of the levels of the factor, and the second element tells whether or not the names of shingles should be included. The default is `c(F, T)`.

## PANEL FUNCTIONS

The data region of a panel on a Trellis display is the rectangular region where the data are plotted. A *panel function* has the sole responsibility for drawing in the data regions produced by a general display function. The panel function is given as an argument of the general display function. The other arguments of the general display function manage the superstructure of the graph—scales, labels, boxes around the data region, and keys. The panel function manages the symbols, lines and so forth that encode the data in the data region.

Every general display function has a default panel function. In all examples given so far in this chapter, the default panel function has been doing the drawing.

### How to Change the Rendering in the Data Region

You can change what is drawn in the data region by one of two mechanisms. First, a default panel function has arguments. You can change the rendering by using these arguments; in fact, you can give them to the general display function, which will pass them along to the panel function. Second, you can write your own panel function.

### Passing Arguments to a Default Panel Function

The name of the default panel function for a general display function is “panel.” followed by the name of the general function. For example the default panel function for `xyplot` is `panel.xyplot`. You can use S-PLUS online help to see the arguments of a default panel function. For example, `?panel.xyplot` tells you about the panel function for `xyplot`.

You can give an argument to a panel function by giving it to the general display function; the general display function passes it on to the panel function. For example, `xyplot` can pass `pch` to `panel.xyplot` to specify a “+” as the plotting symbol:

```
> xyplot(N0x ~ E, data = gas, aspect = 1/2, pch = "+")
```

### Writing A Panel Function: panel argument

If you write your own panel function, you give it to the general display function as the argument `panel`. For example, if you have your own panel function `mypanel`, you specify

```
panel = mypanel
```

A panel function is always a function of at least two arguments; the first two are named `x` and `y`. Suppose, for the gas data, that you want to use `xyplot` to graph NOx against E and use a “+” as the plotting symbol for all observations except that for which NOx is a maximum, in which case you want to use “M”. There is no provision for `xyplot` to do this, so you must write your own. First, let us write the panel function:

```
> panel.special <- function(x, y)
+ {
+   biggest <- y == max(y)
+   points(x[!biggest], y[!biggest], pch = "+")
+   points(x[biggest], y[biggest], pch = "M")
+ }
```

The function `points` is a core graphics function. It graphs individual points on a graph. Its first argument `x` contains the coordinates of the points along the horizontal scale, and its second argument `y` contains the coordinates of the points along the vertical scale. The third argument `pch` gives the symbol used to display the points. To show the result of giving `panel.special` to `xyplot`, try:

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2,
+ panel = panel.special)
```

The panel function for this could also have been defined as part of the `xyplot` command:

```
> xyplot(NOx ~ E, data = gas, aspect = 1/2,
+ panel = function(x, y) { biggest <- y == max(y)
+   points(x[!biggest], y[!biggest], pch = "+")
+   points(x[biggest], y[biggest], pch = "M") } )
```

## A Panel Function for a Multipanel Display

In most cases, a panel function that is used for a single panel display can be used for a multipanel display as well. The panel function `panel.special`, could be used to show the maximum value of NOx on each panel of a multipanel display of the ethanol data:

```
> xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2,
+ panel = panel.special)
```

## Special Panel Functions

Even if you write your own panel function you might want to use the default panel function as part of it. This is often true when you want to augment a standard Trellis panel. Also, Trellis Graphics provides some special purpose panel functions. One of them is `panel.loess`. It adds smooth curves to scatterplots.

To add smooth curves to a multipanel display of the ethanol data:

```
> GIVEN.E <- equal.count(ethanol$E, number = 9,
+ overlap = 1/4)
> xyplot(NOx~C | GIVEN.E, data = ethanol, aspect = 2.5,
+ panel = function(x, y) { panel.xyplot(x, y)
+ panel.loess(x, y, span=1) } )
```

The default panel function `panel.xyplot` draws the points of the scatterplot on each panel. The special panel function `panel.loess` computes and draws the smooth curves; the argument `span`, the smoothing parameter, has been specified.

## subscripts argument

If you request it, another component of the packet sent to each panel is the subscripts that tell which original observations make up the the packet. Knowing these subscripts is helpful for getting the values of other variables that might be needed for rendering on the panel. In such a case the panel function argument `subscripts` contains the subscripts. To see the observation numbers added to the graph of NOx against E given C:

```
> xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2,
+ panel = function(x, y, subscripts)
+ text(x, y, subscripts, cex=.75) )
```

## Commonly-Used S-PLUS Graphics Functions and Parameters

The core graphics functions commonly used in writing panel functions are:

`points`, `lines`, `text`, `segments`, and `polygon`.

You can use the S-PLUS online help to see what they do. The core parameters commonly used in writing panel functions are:

`col`, `lty`, `pch`, `lwd`, and `cex`.

Use `?par` for their definitions.

## PANEL FUNCTIONS AND THE TRELLIS SETTINGS

Trellis Graphics, as we have discussed, is implemented using traditional S-PLUS core graphics, which has controllable graphical parameters that determine the characteristics of plotted objects. For example, if we want to use a symbol to show points on a scatterplot, graphical parameters determine the type, size, font, and color of the symbol.

In Trellis Graphics, the default panel functions for the general display functions select graphical parameters to render plotted elements as effectively as possible. But because the most desirable choices for one graphics device can be different from those for another device, the default graphical parameters are device dependent. These parameters are contained in lists which we will refer to as the *Trellis settings*. When `trellis.device` sets up a graphics device, the Trellis settings are established for that device and are saved on a special data structure.

When you write your own panel functions, you may want to make use of the Trellis settings to provide good performance across different devices. Three functions enable you to access, display, and change the settings for the current device. `trellis.par.get` lets you get settings for use in a panel function. `show.settings` shows graphically the values of the settings. `trellis.par.set` lets you change the settings for the current device.

### `trellis.par.get`

Here is the panel function `panel.xyplot`:

```
function(x, y, type = "p", cex = plot.symbol$cex, pch =
plot.symbol$pch, font = plot.symbol$font, lwd =
plot.line$lwd, lty = plot.line$lty, col = if(type == "l")
plot.line$col else plot.symbol$col, ...) {
  if (type == "l")
  {
    plot.line <-
      trellis.par.get("plot.line") lines(x, y,
      lwd=lwd, lty=lty, col=col, type=type, ...) }
  else
  {
    plot.symbol <- trellis.par.get("plot.symbol")
    points(x, y, pch = pch, font = font, cex = cex,
    col = col, type = type, ...) }
}
```

If the argument type is "p", which means that point symbols are used to plot the data, then the plotting symbol is defined by the settings list `plot.symbol`; the components of this list are given to the function `points` that draws the symbols. The list is accessed by `trellis.par.get`. Here is the list `plot.symbol` for the `graphsheets` device:

```
> trellis.device(graphsheet)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol

$pch:
[1] 1
$col:
[1] 2
$cex:
[1] 0.8
$font:
[1] 1
```

The `pch` of 1 and `col` of 2 produces a cyan circle.

If type is "l", which means that `lines` is used to plot the data, then the graphical parameters for the lines are in the settings list `plot.line`:

```
> trellis.device(graphsheet)
> plot.line <- trellis.par.get("plot.line")
> plot.line

$col:
[1] 2
$lwd:
[1] 1
$lty:
[1] 1
```

This is a cyan-colored solid line.

## **show.settings**

`show.settings` displays the graphical parameters in the Trellis settings for the current device. To see the result for black and white postscript:

```
> trellis.device(graphsheet)
> show.settings()
```

Each panel displays one or more settings lists. The names of the settings appear below the panels. For example, the panel in the third row (from the top) and first column shows plotting symbols with graphical parameters

`plot.symbol` and lines with graphical parameters `plot.line`, and the panel in the third row and third column shows that the panel function of the general display function `histogram` uses the graphical parameters in `bar.fill` for the color that shades the bars of a histogram.

## **trellis.par.set**

The Trellis settings for the current device can be changed:

```
> trellis.device(graphsheet)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol$col
```

```
[1] 2
```

```
> plot.symbol$col <- 3
> trellis.par.set("plot.symbol", plot.symbol)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol$col
```

```
[1] 3
```

`trellis.par.set` sets an entire Trellis setting list, not just some of the components. Thus the simplest way to make a change is to get the current list, alter it, and then save the altered list. The change lasts only as long as the device continues. If the S-PLUS session is ended the altered settings are removed.

## SUPERPOSING TWO OR MORE GROUPS OF VALUES ON A PANEL

One common visualization task is superposing two or more groups of values in the same data region, encoding the different groups in different ways to show the grouping. For example, we might graph leaf width against leaf length for two samples of leaves, one from maple trees and one from oaks, and use a circle as the plotting symbol for the maples and a plus for the oaks.

Superposition is achieved by the panel function `panel.superpose`. In addition, the `key` argument of the general display functions can be used to show the group encoding.

### `panel.superpose`

Superposition is illustrated by using the data frame `fuel.frame`. For 60 automobiles, `Mileage` is graphed against `Weight` for six types of vehicles described by the factor `Type`:

```
> table(fuel.frame$Type)
```

```
Compact Large Medium Small Sporty Van
      15      3      13      13      9      7
```

The vehicle types are encoded by using different plotting symbols. (Nothing on the graph indicates which symbol is for which type, but the next section contains information about drawing a legend, or `key`.)

The panel function `panel.superpose` carries out such a superposition:

```
> xyplot(Mileage ~ Weight, data = fuel.frame, aspect = 1,
+ groups=Type, panel = panel.superpose)
```

The factor `Type` is given to the argument `groups` of `xyplot`. But `groups` is also an argument of `panel.superpose`, so `Type` is passed along to the panel function to be used to determine the plotting symbols.

The plotting symbols are the defaults that are set up by the trellis device function `trellis.device`; such trellis settings were discussed in the section *Panel Functions and the Trellis Settings* (page 381). The specific settings used by `panel.superpose` are discussed later in this section. The default symbols have been chosen to enhance the visual assembly of each group of points; that is, we want to effortlessly assemble the plotting symbols of a given type to form a visual gestalt or whole. If assembly can be performed efficiently then we can compare the characteristics of the data for different automobile types.



You can choose your own plotting symbols. For example, suppose that we want to use the first letters of the vehicle types, but with “S” (for “Small”) replaced by “P” (for “Peewee”) to avoid duplication with “Sporty”:

```
> mysymbols <- c("C", "L", "M", "P", "S", "V")
```

`panel.superpose` has an argument `pch` that can be used to specify the symbols:

```
> xyplot(Mileage ~ Weight, data = fuel.frame, aspect = 1,
+ groups=Type, pch = mysymbols, panel=panel.superpose)
```

Notice that, again, we specify an argument of the panel function — in this case `pch` — by giving it as an argument to `xyplot`, which passes it along to the panel function.

`panel.superpose` will also superpose curves. To superpose a line and a quadratic :

```
x <- seq(0, 1, length=50)
linquad <-c(x, x^2)
x <- rep(x, 2)
which <- rep(c("linear", "quadratic"), c(50, 50))
xyplot(linquad~x, xlab = "Argument", ylab = "Functions",
       aspect = 1, groups = which, type="l",
       panel = panel.superpose)
```

The argument `type` controls the method of plotting. For the argument `type="p"`, the default, the data are rendered by plotting symbols. For `type="l"`, the data are rendered by lines.

The function `panel.superpose` uses the graphical parameters in the Trellis setting `superpose.symbol` for the default plotting symbols. For black and white postscript, the setting results in different symbol types:

```
> trellis.device(postscript)
> trellis.par.get("superpose.symbol")

$pch:
[1] "\001" "+" ">" "s" "w" "#" "{"
$col:
[1] 1 1 1 1 1 1 1
$cex:
[1] 0.85 0.85 0.85 0.85 0.85 0.85 0.85
$font:
[1] 1 1 1 1 1 1 1
```

There are seven symbols, providing for up to seven groups. If there are two groups, the first two symbols are used; if there are three groups, the first three symbols are used; and so forth. The setting for the default line types is `superpose.line`:

```
> trellis.par.get("superpose.line")

$lwd:
[1] 1 1 1 1 1 1 1
$lt:
[1] 1 2 3 4 5 6 7
$col:
[1] 1 1 1 1 1 1 1
```

There are seven line types.

A call to `trellis.settings` will show the seven symbols in the first panel, and the seven line types in the second panel of the top row.

The function `panel.superpose` can be used with any general display function where superposing different groups of values makes sense. For example, we can superpose data sets with `xyplot`, or with `dotplot`, or with many of the other general display functions. By achieving superposition through the `panel` function, we do not need a special superposition general display function for each type of graphical method, which makes things much simpler.

To illustrate this, the following code produces a dot plot of the barley data discussed earlier:

```
> barley.plot <- dotplot(variety ~ yield | site,
+ data = barley, groups = year, layout = c(1, 6),
+ aspect = .5, xlab = "Barley Yield (bushels/acre)")
+ panel = function(x, y, ...) {
+ dot.line <- trellis.par.get("dot.line")
+ abline(h = unique(y), lwd = dot.line$lwd,
+ lt = dot.line$lt, col = dot.line$col)
+ panel.superpose(x, y, ...) }
> print(barley.plot)
```

On each panel, data for two years are displayed, and the years, 1931 and 1932, are distinguished by different plotting symbols. The plot has been saved in the Trellis object `barley.plot` for use later on.

The general display function `dotplot` has not sent the factor `variety` to the `panel` function to be the `y` vector for the function, but rather has sent a numeric vector of values 1 to 10 with 1 corresponding to the first of the 10

levels of the factor, with 2 corresponding to the second level, and so forth. And the display function has sent the values of `yield` as the vector `x`. The conditioning vector is `site`; thus on each panel there are 20 values of `x` and 20 values of `y`; for each level of variety, there are two values of `x` (one for 1931 and one for 1932) and two values of `y`, and there are 10 levels of variety. The plotting symbols are drawn by `panel.superpose` at the 20 values of `x` and `y` on each panel.

The panel function for this `dotplot` example is more complicated than that for the `xyplot` examples because, along with superposing the plotting symbols by `panel.superpose`, the horizontal lines of the dot plot must be drawn. `abline` draws the lines at the unique values of `y`. The characteristics of the line are specified by the Trellis setting `dot.line`.

## key argument

A key can be added to a Trellis display through the argument `key` of the general display functions. The argument is a list. With one exception, the component names are the names of the arguments of the function `key`, which actually does the drawing of the key, so the values of these components are given to the corresponding arguments of `key`. The exception is the component, `argument.space` which can leave extra space for a key in the margins of the display.

The `key` argument is easy to use yet is quite powerful; it has the capability to draw most keys used in practice and many yet to be invented:

```
update(barley.plot,  
key = list(  
  points = Rows(trellis.par.get("superpose.symbol"), 1:2),  
  text = list(levels(barley$year)))
```

The plot would be drawn using `update` to alter `barley.plot`. The component `text` of the `key` argument is a list with the year names. The component `points` is a list with the graphical parameters of the two symbols used by `panel.superpose` to plot the data. These parameters are from the Trellis setting `superpose.symbol`, which `panel.superpose` uses to draw the plotting symbols.

We want to give the component `points` only the parameters of the symbols used, so the function `Rows` extracts the first two elements of each component of `superpose.symbol`:

```
> trellis.device(postscript)
```

```
> Rows(trellis.par.get("superpose.symbol"), 1:2)

$pch:
[1] "o" "+"
$col:
[1] 1 1
$cex:
[1] 1 1
$font:
[1] 1 1
```

The key has two entries, one for each year. If there had been four years there would have been four entries. Each entry has two items; as we shall see, we can specify more items if we choose. The order of the items is the order of specification in the argument `key`; in the above expression, `points` is first and `text` is second, so in the key, the symbol is the first item and then the text is the second item. Had we specified `text` first, the symbol would have followed the text in each entry.

The two entries, by default, are drawn as an array with one column and two rows. We can change this by the argument `columns`. Also, we can switch the order of the symbols and the text:

```
update(barley.plot,
  key = list(
    text = list(levels(barley$year)),
    points = Rows(trellis.par.get("superpose.symbol"), 1:2),
    columns = 2))
```

The argument `space` allocates space for the key in the margins. It takes one of four values — "top", "bottom", "right", "left" — allocating the space on the side of the graph described by the value. So far, it has been allocating space at the top, which is the default, and placing the key in the allocated space. More will be said about the `space` argument later.

If the default location of the key seems a bit too far from the rest of the graph, the key can be repositioned, and a border can be drawn around it:

```
update(barley.plot,
  key = list(
    points = Rows(trellis.par.get("superpose.symbol"), 1:2),
    text = list(levels(barley$year)),
    columns = 2,
    border = 1,
    space = "top",
    x = .5,
```

```
y = 1.02,  
corner = c(.5, 0))
```

The argument `border` draws a border; it takes a number that specifies the color in which the border should be drawn.

The repositioning uses two coordinate systems. The first describes locations in the rectangle that just encloses the panels of the display, but not including the tick marks; the lower left corner of this panel rectangle has coordinates (0,0), and the upper right corner has coordinates (1,1). A location in the panel rectangle is specified by the components `x` and `y`. The second coordinate system describes locations in the border rectangle of the key, which is shown when the border is drawn; the lower left corner of the key rectangle has coordinates (0,0), and the upper right corner has coordinates (1,1). A location in the border rectangle is specified by the component `corner`, a vector with two elements, the horizontal and vertical coordinates. The key is positioned so that the locations specified by the two coordinate systems are at the same place on the graph.

Having two coordinate systems makes it far easier to get the key to a desired location quickly, often on the first try.

Notice that we specified the `space` argument to be "top". The reason is that as soon as we specify a value for any of the coordinate arguments `x`, `y`, or `corner`, no default space is allocated in any margin location unless we explicitly use the argument `space`. If we do not use the coordinate arguments, the `space` argument defaults to "top". To allocate space to the right:

```
update(barley.plot,  
  key = list(  
    points = Rows(trellis.par.get("superpose.symbol"), 1:2),  
    text = list(levels(barley$year)),  
    space = "right"))
```

To draw a border, and to position the key by putting the upper left corner of the border rectangle at the same vertical position as the top of the panel rectangle and at a horizontal position slightly to the right of the right side of the panel rectangle:

```
update(barley.plot,  
  key = list(  
    points = Rows(trellis.par.get("superpose.symbol"), 1:2),  
    text = list(levels(barley$year)),  
    space = "right",  
    border = 1  
    corner = c(0, 1),
```

```
x = 1.05,  
y = 1))
```

So far we have seen that components points and text can be used to create items in key entries. A third component, `lines`, draws line items. To illustrate this, let us return to graphing `Mileage` against `Weight` for six types of vehicles. The following code makes the plot, and adds two loess smooths with two different values of the smoothing parameter span:

```
superpose.line <- trellis.par.get("superpose.line")  
superpose.line$col[3:6] <- 0  
superpose.symbol <- trellis.par.get("superpose.symbol")  
xyplot(Mileage ~ Weight,  
  data = fuel.frame,  
  groups=Type,  
  aspect = 1,  
  panel = function(x, y, ...) {  
    panel.superpose(x, y, ...)  
    panel.loess(x, y, span=1/2,  
      lwd = superpose.line$lwd[1],  
      lty = superpose.line$lty[1],  
      col = superpose.line$col[1])  
    panel.loess(x, y, span=1,  
      lwd = superpose.line$lwd[2],  
      lty = superpose.line$lty[2],  
      col = superpose.line$col[2])  
  },  
  key = list(  
    transparent = T,  
    x = .95,  
    y = .95,  
    corner = c(1, 1),  
    lines=c(Rows(superpose.line, 1:6),  
      list(size=c(3, 3, 0, 0, 0, 0)))  
    text = list(c("Span = 0.5", "Span = 1.0", rep("", 4))),  
    points = Rows(superpose.symbol, 1:6),  
    text = list(levels(fuel.frame$Type))))
```

## DATA STRUCTURES

Trellis Graphics uses the S-PLUS formula language to specify the data for plotting. This requires the data to be stored in data sets that work with formulas. Roughly speaking, this means the data variables must be either from a data frame or be vectors of the same length. (This is also true of the S-PLUS modeling functions such as `lm`.) But in S-PLUS there are many other data structures. So that Trellis functions will be easy to use, three functions convert data structures of different kinds into data frames — `make.groups`, `as.data.frame.array`, and `as.data.frame.ts`.

### `make.groups`

The function `make.groups` takes several vectors and constructs a data frame with two components: `data` and `whi ch`. For example, consider payoffs of the New Jersey Pick-It lottery from three time periods. The data are stored as three vectors of values. Suppose we want to make box plots to compare the three distributions: We first convert the three vectors to a data frame:

```
> lottery <- make.groups(lottery.payoff, lottery2.payoff,
+ lottery3.payoff)
> names(lottery)

[1] "data" "whi ch"

> levels(lottery$whi ch)

[1] "lottery.payoff" "lottery2.payoff" "lottery3.payoff"
```

The `data` component is simply the combined numbers from all the `make.groups` arguments. The `whi ch` component is a factor with three levels, giving the names of the original data vectors. Now we can make the box plots:

```
> bwplot(whi ch ~ data, data = lottery)
```

### `as.data.frame.array`

The function `as.data.frame.array` converts arrays into data frames. Consider the object `iris`, a three-way array of 50 measurements of four variables for each of three varieties of irises:

```
> dim(iris)

[1] 50 4 3
```

To turn `iris` into a data frame in preparation for Trellis plotting use:

```
iris.df <- as.data.frame.array(iris, col.dims = 2)

names(iris.df)[5:6] <- c("flower", "variety")
```

The resulting data frame has what used to be its second dimension turned into four columns:

```
> iris.df[1:5, ]
```

	Sepal L.	Sepal W.	Petal L.	Petal W.	flower	variety
1	5.1	3.5	1.4	0.2	1	Setosa
2	4.9	3.0	1.4	0.2	2	Setosa
3	4.7	3.2	1.3	0.2	3	Setosa
4	4.6	3.1	1.5	0.2	4	Setosa
5	5.0	3.6	1.4	0.2	5	Setosa

To produce a scatterplot matrix of the data:

```
superpose.symbol <- trellis.par.get("superpose.symbol")
for (i in 1:4)
  iris.df[, i] <- jitter(iris.df[, i])
splom(~iris.df[, 1:4],
      key = list(
        space="top", columns = 3,
        text = list(levels(iris.df$variety)),
        points = Rows(superpose.symbol, 1:3)),
      varnames = c("Sepal Length\n (cm)",
                   "Sepal Width\n (cm)",
                   "Petal Length\n (cm)", "Petal Width\n (cm)"),
      groups=iris.df$variety,
      panel = panel.superpose)
```

To prevent exact overlap of many of the plotting symbols, the data have been jittered before plotting.

### **as.data.frame.ts**

The function `as.data.frame.ts` takes one or more time series as arguments and produces a data frame with components named `series`, `which`, `time`, and `cycle`. The `series` component is the data from all of the time series combined into one long vector. The `time` component gives the time associated with each of the points (measured in the same units as the original series, for example, years), and `cycle` gives the periodic component of the time (for example 1=Jan, 2=Feb, ...). Finally, the `which` component is a factor



that tells which of the time series the measurement came from. In the following example there is only one series, hstart, but in general as.data.frame.ts can take many arguments:

```
> as.data.frame.ts(hstart)[1:5, ]
```

	series	which	time	cycle
1	81.9	hstart	1966.000	Jan
2	79.0	hstart	1966.083	Feb
3	122.4	hstart	1966.167	Mar
4	143.0	hstart	1966.250	Apr
5	133.9	hstart	1966.333	May

To graph housing starts for each month separately from 1966 to 1974:

```
> xyplot(series ~ time|cycle,  
+ data = as.data.frame.ts(hstart), type = "b",  
+ xlab="Year", ylab="Housing Starts by Month")
```

## MORE ON ASPECT RATIO AND SCALES: PREPANEL FUNCTIONS

Banking to 45 degrees is an important display method built into Trellis Graphics through the argument `aspect`. And the ranges of scales on the panels can be controlled by the arguments `xlim` and `ylim`, or by the argument `scales`. Another argument, `prepanel`, is a function that supplies information for the banking and range calculations.

### **prepanel argument**

The code below will plot the ethanol data; NOx is graphed against E given C and loess curves have been superposed.

```
> xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2,
+ panel = function(x, y) {
+ panel.xyplot(x, y) panel.loess(x, y, span=1/2, degree=2) } )
```

There are now two things we would like to do with this plot, one involving the aspect ratio and the other involving the ranges of the scales.

First, we have set the aspect ratio to 1/2 using the `aspect` argument. We could have set the argument `aspect` to "xy" to carry out 45 degrees banking of the line segments that connect the points of the plot, that is, the graphed values of E and NOx. But normally we do want to carry out banking of the raw data if they are noisy; rather we want to bank an underlying smooth pattern. In this example, we want to bank using the line segments of the loess curves.

Second, in the top panel, the loess curve exceeds the maximum value along the vertical scale and so is chopped off. It is important to understand why this happened. The scales were chosen based on the values of E and NOx. The loess curves were computed by the `panel` function after all of the scaling had been carried out. We would like a way for the scaling to take account of the values of the loess curve.

The argument `prepanel` allows us to bank to 45 degrees based on the loess curves and to take the curves into account in computing the ranges of the scales:

```
> xyplot(NOx ~ E | C, data = ethanol,
+ prepanel = function(x, y)
+ prepanel.loess(x, y, span=1/2, degree=2), layout = c(1,6),
+ panel = function(x, y) {
+ panel.xyplot(x, y)
+ panel.loess(x, y, span=1/2, degree=2) })
```

The `prepanel` argument takes a function and does panel-by-panel computations, just like the argument `panel`, but these computations are carried out before the scales and aspect ratio are determined and so, can be used in their determination. The returned value of a `prepanel` function is a list with prescribed component names. These names are shown in the `prepanel` function `prepanel.loess`:

```
> prepanel.loess

function(x, y, ...)
{
  xlim <- range(x)
  ylim <- range(y)
  out <- loess.smooth(x, y, ...)
  x <- out$x
  y <- out$y
  list(xlim = range(x, xlim), ylim = range(y, ylim),
       dx = diff(x), dy = diff(y))
}
```

The component values `xlim` and `ylim` determine ranges for the scales just as they do when they are given as arguments of a general display function. The values of `dx` and `dy` are the horizontal and vertical changes of the line segments that are to be banked to 45 degrees.

The function `prepanel.loess` computes the smooths for all panels, computes values of `xlim` and `ylim` that ensure the curve will be included in the ranges of the scales, and then passes along the changes of the line segments that will make up the plotted curve. Any of the component names can be missing from the list; if either `dx` or `dy` is missing, the other must be as well. When `dx` and `dy` are present, they give the information needed for banking to 45 degrees as well as the instruction to do so; thus the `aspect` argument should not be used as an argument when `dx` and `dy` are present.

## More on Multipanel Conditioning

The multipanel conditioning of Trellis Graphics has three more arguments that assist in the control of the layout, visual design, and labeling. The argument `between` puts space between adjacent columns or adjacent rows. The argument `skip` allows a panel position to be skipped when packets are sent to the panels for drawing. The `page` argument can add page numbers, text, or even graphics to each page of a multipage Trellis display.

**between  
argument**

To graph the barley data:

```
> barley.plot <- dotplot(site ~ yield | variety * year,  
+ data = barley, aspect = "xy", layout = c(2, 5, 2))  
> barley.plot
```

In the resulting two-page Trellis display, `yield` is plotted against `site` given `variety` and `year`.

The layout — 2 columns, 5 rows, and 2 pages — has put the measurements for 1931 on the first page and for 1932 on the second page. The display will be saved in `barley.plot` for future editing. The panels can be squeezed into one page by changing `layout` from (2,5,2) to (2,10,1):

```
> barley.plot <- update(barley.plot, layout = c(2, 10, 1))  
> barley.plot
```

Rows 1 to 5 (starting from the bottom) have the 1932 data and rows 6 to 10 have the 1931 data. The change in the value of the year variable from rows 5 to 6 is indicated by the text of the strip label, but a stronger indication of a change would occur if there was a break in the display between rows 5 and 6.

The argument `between` can be used to insert space between adjacent rows or adjacent columns of a Trellis display. To illustrate this, try the following, which puts space between rows 5 and 6 of the barley display:

```
> barley.plot <- update(barley.plot,  
+ between = list(y=c(0, 0, 0, 0, 1, 0, 0, 0, 0)))  
> barley.plot
```

The argument `between` is a list with components `x` and `y`; either can be missing. `x` is a vector whose length is equal to the number of columns minus one; the values are the amounts of space, measured in character heights, to be inserted between columns. Similarly, `y` specifies the amounts of space between rows.

**skip argument**

The argument `skip`, which takes a logical vector, controls skipping. Each element says whether or not to skip a panel. For example:

```
> market.plot <-  
+ bwplot(age~log(1+usage) | income*pick,  
+ strip = function(...)  
+ strip.default(..., strip.names = T),  
+ skip=c(F, F, F, F, F, F, F, T),  
+ layout=c(2, 4, 2),  
+ data=market.survey)  
> market.plot
```

The layout will have eight panels per page but there are seven plots. On both pages, the last panel is skipped. The skipping has been done because the conditioning variable `income` has seven levels.

### **page argument**

The argument `page` can add page numbers, text, or graphics to each page of a multipage Trellis display. `page` should be a function of a single argument `n`, the page number; the function tells what to draw on page `n`. For example:

```
> update(market.plot, page = function(n)
> text(x=.75, y=.95, paste(" page", n), adj = .5))
```

`text`, an S-PLUS core graphics function, uses a coordinate system that is the same as the panel rectangle coordinate system for the argument `key`; (0,0) is the lower left corner and (1,1) is the upper left corner.

## SUMMARY OF TRELLIS FUNCTIONS AND ARGUMENTS

*Table 9.1: An Alphabetical Guide to Trellis Graphics.*

Statement	Purpose	Example
<code>as.data.frame.array</code>	function	<code>iris.df &lt;- as.data.frame.array(iris, col.dims = 2)</code>
<code>as.data.frame.ts</code>	function	<code>data.frame.ts(hstart) [1:5,]</code>
<code>aspect</code>	argument	<code>xyplot(NOx ~ E, data = gas, aspect = 1/2, xlab = "Equivalence Ratio", ylab = "Oxides of Nitrogen", main = "Air Pollution", sub = "Single-Cylinder Engine")</code>
<code>barchart</code>	function	<code>barchart(names(mileage.means) ~ mileage.means, aspect = 1)</code>
<code>between</code>	argument	<code>barley.plot &lt;- update(barley.plot, between = list(y=c(0,0,0,0,1,0,0,0)))</code>
<code>bwplot</code>	function	<code>bwplot(Type ~ Mileage, data = fuel.frame, aspect = 1)</code>
<code>cloud</code>	function	<code>cloud(Mileage ~ Weight * Disp., data = fuel.frame, screen = list(z=-30, x=-60, y=0), xlab = "W", ylab = "D", zlab = "M")</code>
<code>contourplot</code>	function	<code>contourplot(dataz~datax * datay, data = gauss, aspect=1, at = seq(.1, .9 by = .2))</code>
<code>data</code>	argument	<i>see aspect example</i>
<code>densi.tplot</code>	function	<code>densi.tplot(~Mileage, data = fuel.frame, aspect = 1/2, width = 5)</code>
<code>dev.cur</code>	function	<code>dev.cur()</code>
<code>dev.list</code>	function	<code>dev.list()</code>
<code>dev.off</code>	function	<code>dev.off()</code>
<code>dev.set</code>	function	<code>dev.set(which = 2)</code>

Table 9.1: An Alphabetical Guide to Trellis Graphics.

Statement	Purpose	Example
<code>dotplot</code>	function	<code>dotplot(names(mileage.means) ~ log(mileage.means, base=2), aspect=1, cex=1.25)</code>
<code>equal.count</code>	function	<code>GIVEN.E &lt;- equal.count(ethanol\$E, number = 9, overlap = 1/4)</code>
<code>formula</code>	argument	<code>xyplot(formula = gas\$NOx ~ gas\$E)</code>
<code>histogram</code>	function	<code>histogram(Mileage, data = fuel.frame, aspect = 1, nint = 10)</code>
<code>intervals</code>	argument	<code>GIVEN.E &lt;- shingle(ethanol\$E, intervals = cbind(endpoints[-6], endpoints[-1]))</code>
<code>jitter</code>	argument	<code>stripplot(Type ~ Mileage, data = fuel.frame, jitter = TRUE, aspect = 1)</code>
<code>key</code>	argument	<code>update(barley.plot, key = list(points = Rows(trellis.par.get("superpose.symbol"), 1:2), text = list(levels(barley\$year)))</code>
<code>layout</code>	argument	<code>dotplot(site ~ yield   year * variety, data = barley, layout = c(2, 5, 2))</code>
<code>levelplot</code>	function	<code>levelplot(dataz ~ datax * datay, data = gauss, aspect = 1, cuts = 6)</code>
<code>levels</code>	function	<code>levels(barley\$year)</code>
<code>main</code>	argument	<i>see</i> aspect example
<code>make.groups</code>	function	<code>lottery &lt;- makegroups(lottery.payoff, lottery2.payoff, lottery3.payoff)</code>
<code>market.plot</code>	function	<code>update(market.plot, page = function(n) text(x=.75, y=.95, paste("page", n), adj = .5))</code>
<code>page</code>	argument	<i>see</i> market.plot example

*Table 9.1: An Alphabetical Guide to Trellis Graphics.*

Statement	Purpose	Example
<code>panel</code>	argument	<code>panel.special &lt;- function(x, y){   biggest &lt;- y == max(y)   points(x[!biggest], y[!biggest], pch = "+")   points(x[biggest], y[biggest], pch = "M")}</code>
<code>panel.superpose</code>	function	<code>xyplot(Mileage ~ Weight, data = fuel.frame,   aspect=1, groups=Type,   panel=panel.superpose)</code>
<code>panel.loess</code>	function	<code>xyplot(NOx ~ C   GIVEN.E, data = ethanol,   aspect = 2.5, panel = function(x, y) {     panel.xyplot(x, y)     panel.loess(x, y, span=1)})}</code>
<code>panel.xyplot</code>	function	<i>see panel.loess example</i>
<code>parallel</code>	function	<code>parallel(~fuel.frame)</code>
<code>par</code>	function	<code>par(ask = TRUE)</code>
<code>par.strip.test</code>	argument	<code>par.strip.test = list(cex = 2)</code>
<code>piechart</code>	function	<code>piechart(names(mileage.means) ~ mileage.means)</code>
<code>prepanel</code>	argument	<code>xyplot(NOx ~ E   C, data = ethanol,   prepanel = function(x, y) prepanel.loess(x,     y, span=1/2, degree=2), layout = c(1, 6),   panel = function(x, y) {panel.xyplot(x, y)     panel.loess(x, y, span=1/2, degree=2)})}</code>
<code>prepanel.loess</code>	function	<i>see prepanel example</i>
<code>print</code>	function	<code>print(box.plot, position = c(0, 0, 1, .4),   more = T)</code>
<code>print.trellis</code>	function	<code>print.trellis()</code>
<code>pcales</code>	argument	<code>pcales = 1</code>
<code>qq</code>	function	<code>qq(Type ~ Mileage, data = fuel.frame,   aspect = 1, subset =(Type == "Compact")     (Type == "Small"))</code>
<code>qqmath</code>	function	<code>qqmath(~ Mileage, data = fuel.frame,   subset = (Type == "Small"))</code>



Table 9.1: An Alphabetical Guide to Trellis Graphics.

Statement	Purpose	Example
<code>reorder.factor</code>	function	<code>barley\$variety &lt;- reorder.factor(barley\$variety, barley\$yield, median)</code>
<code>Rows</code>	function	<code>Rows(trellis.par.get("superpose.symbol"), 1:2)</code>
<code>scales</code>	argument	<code>xyplot(NOx ~ E, data = gas, aspect = 1/2, ylim = c(0, 6), scales = list(cex = 2, tick.number = 4))</code>
<code>screen</code>	argument	<code>wireframe(dataz~datax * datay, data=gauss, drape = F, screen = list(z=45, x=-60, y=0))</code>
<code>shingles</code>	function	<code>GIVEN.E &lt;- shingle(ethanol\$E, intervals = cbind(endpoints[-6], endpoints[-1]))</code>
<code>show.settings</code>	function	<code>show.settings()</code>
<code>skip</code>	argument	<code>bwplot(age~log(1+usage)   income*pick, strip = function(...) strip.default(..., strip.names = T), skip = c(F, F, F, F, F, F, F, T), layout=c(2, 4, 2), data=market.survey)</code>
<code>span</code>	argument	<i>see</i> <code>prepanel.loess</code> example
<code>space</code>	argument	<code>update(barley.plot, key = list(points = Rows(trellis.par.get("superpose.symbol"), 1:2), text = list(levels(barley\$year)), space = "right"))</code>
<code>splom</code>	function	<code>splom(~fuel.frame)</code>
<code>strip</code>	argument	<i>see</i> <code>skip</code> example
<code>striplot</code>	function	<i>see</i> <code>jitter</code> example
<code>sub</code>	argument	<i>see</i> <code>aspect</code> example
<code>subscripts</code>	argument	<code>xyplot(NOx ~ E   C, data = ethanol, aspect = 1/2, panel = function(x, y, subscripts) text(x, y, subscripts, cex=.75))</code>

*Table 9.1: An Alphabetical Guide to Trellis Graphics.*

Statement	Purpose	Example
<code>subset</code>	argument	<code>xyplot(N0x~E, data = gas, subset = E &lt; 1.1)</code>
<code>superpose.symbol</code>	argument	<code>trellis.par.get("superpose.symbol")</code>
<code>trellis.args</code>	function	<code>?trellis.args</code>
<code>trellis.device</code>	function	<code>trellis.device(postscript, onefile = FALSE)</code>
<code>trellis.par.get</code>	function	<code>plot.line &lt;- trellis.par.get("plot.line")</code>
<code>trellis.par.set</code>	function	<code>trellis.par.set("plot.symbol", plot.symbol)</code>
<code>update</code>	function	<code>foo &lt;- update(foo, main = "Dependence of N0x on E")</code>
<code>width</code>	argument	<i>see</i> densityplot example
<code>widthreframe</code>	function	<i>see</i> screen example
<code>xlab</code>	argument	<i>see</i> aspect example
<code>xlim</code>	argument	<code>xlim &lt;- range(x)</code>
<code>xyplot</code>	function	<code>xyplot(Mileage ~ Weight, data = fuel.frame, aspect = 1)</code>
<code>ylab</code>	argument	<i>see</i> aspect example
<code>ylim</code>	argument	<i>see</i> scales example

# OBJECT-ORIENTED PROGRAMMING IN S-PLUS

# 10

---

<b>Fundamentals of Object-Oriented Programming</b>	<b>405</b>
Generic Functions in S-PLUS	406
Classes and Methods in S-PLUS	407
Public and Private Views of Methods	410
<b>Defining New Classes in S-PLUS</b>	<b>411</b>
<b>Group Methods</b>	<b>415</b>
<b>Replacement Methods</b>	<b>422</b>

Throughout the first chapters, almost no mention has been made of object-oriented programming. Yet one of the very first statements in the book was that S-PLUS is an object-oriented programming language, and that it takes full advantage of the powerful concepts of classes and methods.

The advantages of object-oriented programming do not evidence themselves when you are writing a single function for a particular purpose. Instead, the advantages arise when you are designing a large system that will do *similar*, but not identical, things to a variety of data objects. By specifying *classes* of data objects for which identical effects will occur, you can define a single *generic* function that embraces the similarities across object types, but permits individual implementations or *methods* for each defined class. For example, if you type an expression of the form `print(object)`, you expect S-PLUS to print the object in a suitable format. All the various predefined printing routines *could* be combined into a single function; in such a case the `print` function would need to be modified every time a new class of objects was created. In object-oriented programming, however, the `print` function is truly generic; it should not have to be modified to accommodate new classes of objects. Instead, the objects carry their own methods with them. Thus, when you create a class of objects, you can also create a set of methods to specify how those objects will behave with respect to certain generic operations.

As a concrete example, consider the way S-PLUS prints character vectors and factors. Both are created originally from vectors of character strings, and when printed, both give essentially the same information:

```
> xxx <- c("Whi te", "Bl ack", "Gray", "Gray", "Whi te", "Whi te")
> yyy <- factor(xxx)
> print(xxx)

[1] "Whi te" "Bl ack" "Gray" "Gray" "Whi te" "Whi te"

> print(yyy)

[1] Whi te Bl ack Gray Gray Whi te Whi te
```

The distinct look of the printed factor arises because factors are a distinct class of object, with their own printing method, the function `print.factor`.

Note also that Chapter 12, *Customized Analytics: A Detailed Example*, has additional examples of object-oriented programming.

---

# FUNDAMENTALS OF OBJECT-ORIENTED PROGRAMMING

Object-oriented programming uses the data being acted upon to determine what actions take place. Thus, a common synonym for *object-oriented* is *data-driven*. Because the actual actions are determined by the data, the commands or function calls are, in effect, simply messages or requests from the user to the data: *print yourself, summarize yourself*.

The requests are generally expressed as calls to *generic* functions. A generic function, such as `print` or `plot`, takes an arbitrary object as its argument. The nature of the object then determines how the action specified by the generic function is carried out. The actual actions are performed by *methods* which implement the action called for by the generic function for a particular type of data. Most generic functions have default methods which are used if no more specific method can be found. For example, if you type the expression `print(myobject)` with *myobject* a factor, S-PLUS will print *myobject* using the method `print.factor`. If *myobject* is a vector, the printing is performed by `print.default`.

As an S-PLUS user, you should never need to explicitly call a method; generic functions provide all the interface you need for most purposes. The importance of the object-oriented programming paradigm is in *extending* S-PLUS's capabilities. To see why this is so, imagine you are writing a program to draw various shapes. You envision a hierarchy of shapes, some open, some closed, some with straight sides, some with curved sides. You want the user interface to be simple, so that a call such as `draw(circle)` will draw a circle. In traditional programming, the complexity will be built into the `draw` function, which would likely be driven by a large switch-type statement, or series of if-else statements. How each shape is to be drawn is specified by one case of the switch or one clause in the if-else. But what happens when you define a new shape? You must modify the `draw` function to define a new case. (If `draw` were written completely using S-PLUS expressions, this would not be an impossible task. But suppose `draw` was implemented as a piece of C code, and you don't have the source!)

If `draw` is generic, however, you need not modify it when you want to add new cases. You simply write a method specific to the new case.

## Generic Functions in S-PLUS

Generic functions in S-PLUS tend to be extremely simple, thanks to the utility function `UseMethod`. `UseMethod` is an internally implemented function that finds the appropriate method and evaluates a call to it. The typical generic function consists of a single call to `UseMethod`:

```
> plot

function(x, ...)
  UseMethod("plot")

> print

function(x, ...)
  UseMethod("print")
```

When the generic function is called, `UseMethod` determines the class of the argument `x`, finds the appropriate method, then constructs and evaluates a call of the form *method*(`x`, ...), where ... represent additional arguments to the method.

Although virtually all generic functions have the simple structure shown above for `print` and `plot`, there can be a need for a slightly more complicated definition. For example, the `assign` generic function is designed to store objects in different classes of databases, so that what is important to `assign` is not the class of the assigned *object*, but the class of the assigned *database*. To tell `UseMethod` that this is the case, the call to `UseMethod` has a second argument to specify which argument to `assign` is to be searched for its class attribute:

```
> assign

function(x, value, frame, where = NULL)
  UseMethod("assign", where)
```

The `browser` function acts generically when called with an argument, but has a specific action when called with no arguments (in part because you need an argument to find a method). This is embodied in its definition:

```
> browser
function(object, ...)
if(nargs()) UseMethod("browser") else
{
  nframe <- sys.parent()
  msg <- paste(deparse(sys.call(nframe)), collapse = " ")
  if(nchar(msg) > 30)
    msg <- paste(substring(msg, 1, 30), "...")
  browser.default(nframe,
                  message = paste("Called from:",
                                  msg))
}
```

## Classes and Methods in S-PLUS

S-PLUS determines which method to use for a given object by looking at the `class` attribute of the object, if it exists. If the `class` attribute is missing, the default class is assumed. For example, factors are identified by class "factor", while vectors have no class attribute, so are of class default.

Methods are named using the convention *action.class*, where *action* is the name of the generic function, and *class* is the class to which the method is specific. Thus `plot.factor` is the plot method for factors, and `is.na.data.frame` is the missing value test method for data frames.

A class attribute is just a character vector, and it can be of any length. The first element in the class attribute is the most specific class of the object. Thus, for example, an ordered factor has class attribute `c("ordered", "factor")`, and is said to have class `ordered`. Subsequent elements in the class attribute specify classes from which the specific class *inherits*. Inheritance is a powerful concept in object-oriented programming, because it lets you define a new class with only the features needed to distinguish it from classes from which it inherits. Thus, ordered factors are simply factors for which the levels have a specific ordering.

If S-PLUS finds no method for the most specific class of an object, it looks in turn at each of the classes from which the object inherits. As soon as S-PLUS finds an appropriate method, it uses it. Every class inherits from class default, so the default method is used if no more specific method exists.

To take full advantage of inheritance you need to be able to define methods incrementally, so that a specific method can act something like a pre- or post-processor to a more general method. Thus, for example, a method for

printing ordered factors should be able to draw on an existing method for printing factors. S-PLUS satisfies this requirement by providing the `NextMethod` function. Like `UseMethod`, `NextMethod` is internally implemented. `NextMethod` finds the next most specific method after the current method, and creates and evaluates a call to that method. For example, here is the definition of `print.ordered`:

```
> print.ordered
function(x, ...)
{
  NextMethod("print")
  cat("\n ", paste(levels(x), collapse = " < "), "\n ")
  invisible(x)
}
```

Like all print methods, `print.ordered` returns its first argument. Values for all methods should be compatible. The call to `NextMethod` in this case finds the function `print.factor`. To the output of `print.factor`, `print.ordered` appends the ordered levels. In this case, then, the specific method for ordered factors is just a post-processor to the method for factors in general. On the other hand, most of `print.factor` is simply pre-processing for `print.default`:

```
> print.factor
function(x, quote = F, abbreviate.arg = F, ...)
{
  if(length(xx <- check.factor(x)))
    stop(paste(
      "cannot be interpreted as a factor:\n\t", xx))
  xx <- x
  l <- levels(x)
  class(x) <- NULL
  if(abbreviate.arg)
    l <- abbreviate(l)
  if(any(is.na(x)))
  {
    l <- c(l, "NA")
    x[is.na(x)] <- length(l)
  } else x <- l[x]
  NextMethod("print", quote = quote)
  if(any(is.na(match(l, unique(x)))))
  {
    cat("Level s: \n ")
    print.atomics(l)
  }
  invisible(xx)
}
```



To build objects of a specific class, you need to define a *constructor*, or *generator* function. Typically, generator functions have the name of the object they create—vector, factor, and so on. Here is the definition of the factor generator function:

```
> factor
function(x, levels = sort(unique(x)),
  labels = as.character(levels), exclude = NA) {
  if(length(exclude) > 0)
  {
    storage.mode(exclude) <- storage.mode(levels)
    # levels <- complement(levels, exclude)
    levels <- levels[is.na(match(levels, exclude))]
  }
  y <- match(x, levels)
  names(y) <- names(x)
  levels(y) <- if(length(labels) == length(levels))
  {
    labels
  } else
  if(length(labels) == 1 )
  {
    paste(labels, seq(along = levels), sep = "" )
  } else
    stop(paste("invalid labels argument, length",
      length(labels), "should be", length(levels),
      "or 1"))
  class(y) <- "factor"
  y
}
```

Here, the generator function explicitly sets the class attribute. Not all generator functions produce objects with a non-null class attribute. For example, `numeric` generates numeric vectors, which have no class attribute.

You can view the class of any object with the class function:

```
> class(kyphosis)

[1] "data.frame"
```

You can modify the class of an object by using `class` on the left-hand-side of an assignment, as in ordered:

```
> class(myobject) <- "myclass"
```

However, modifying the class attribute should not be done lightly. Assigning a class requires that the object is a compatible value object of that class.

## Public and Private Views of Methods

An important distinction is often made in object-oriented programming between the *public* (or *external*) view and the *private* (or *internal*) view of the implementation of a class. The private view is the view of the implementor and the S-PLUS software; any time you use the S-PLUS structure of an object in defining a method, you are using this private view. The public view is the conceptual view of the object and the functions that operate on it—a matrix is an  $m \times n$  array, created by a function `matrix`. Ideally, the casual user of a function should not be concerned with the private view—the public view should be adequate for most situations.

When you are developing new methods, you must be clear at all times about which view you are using, because the private view, unlike the public view, is *implementation dependent*. If the implementation of a class changes, methods defined using the private view need to be examined to see if they are still valid. Using the private view of objects in defining new methods is generally more efficient, particularly for the most commonly used methods. Public methods, on the other hand, are easier to maintain.

## DEFINING NEW CLASSES IN S-PLUS

Defining new classes in S-PLUS can be done either informally, through defining attributes or components (as with older classes such as vectors and matrices), or more formally, through the use of the specific `class` attribute. The latter method is recommended, because it greatly simplifies programming and gives your new class the power of S-PLUS's generic dispatch. However, defining the attributes and components that characterize the new class remains an important part of defining the class.

As with many programming tasks, the key to successfully defining new classes is to *abstract* the identifying features of a given data object, sharply contrasting objects within the class from those outside the class. For example, a data object with the attribute `dim` is necessarily an array. Testing for this attribute is equivalent to testing for membership in the class.

As an example of new classes in S-PLUS, we will define a class of graphical *shapes*. In our simple model, shapes will be specified as a sequence of points. *Open* shapes, such as line segments and arcs, are specified by their endpoints. *Closed* shapes, such as circles and squares, are specified by *starting points* and points that uniquely determine the shape. For example, a circle is specified as a center and a point on the circle. A square is specified by one corner and a side length, while a rectangle is specified by two diagonal corners.

Our motivation for defining these shapes is to create a rudimentary freehand drawing using a graphics windows. For this reason, we define our classes so that objects can be created easily using a sequence of mouse clicks via the `locator` function. For example, here is a *generator* function for circles:

```
circle <-
function(center, radius, point.on.edge)
{
  center <- as.point(center)
  val <- NULL
  if(length(center$x) == 2)
  {
    val <- list(center = list(x = center$x[1],
                              y = center$y[1]),
                radius = sqrt(
                  diff(center$x)^2 +
                  diff(center$y)^2))
  } else
  if(length(centerx) == 1)
  {
    if(missing(radius))
    {
      point.on.edge <- as.point(point.on.edge)
    } else

```

```
      if(is.atomic(radius))
      {
        val <- list(center = center,
                     radius = abs(radius))
      } else
      {
        point.on.edge <- as.point(radius)
      }
      if(is.null(val))
      {
        val <- list(center = list(x =
                                   center$x[1],
                                   y = center$y[1]),
                     radius = sqrt((
                                   point.on.edge$x-center$x)^2 +
                                   (point.on.edge$y-center$y)^2))
      }
    }
    class(val) <- "circle"
    val
  }
}
```

The `circle` function lets you express the circle in several natural ways. You can give the center as either a list containing  $x,y$  components, as you might get from the `locator` function, or you can give it as an  $xy$ -vector. You can give the radius as a scalar, or a second point from which the radius can be calculated. For example, here is how you might define a simple circle from the S-PLUS command line:

```
> simple.circle <- circle(center = c(0.5, 0.5), radius=0.25)
> simple.circle

$center:
$center$x:
[1] 0.5

$center$y:
[1] 0.5

$radius:
[1] 0.25

attr(,"class"):
[1] "circle" "closed"
```

We stored the circle as a list for ease of access to individual elements; however, the default printing for lists seems rather too formal for a circle, where all we really need to see is a center and radius. Thus, it makes sense to define a method for use with the `print` generic function:

```
print.circle <-
function(x, ...)
{
  cat(" Center: x =", x$center$x, "\n ",
      " y =", x$center$y, "\n ",
      "Radius: ", x$radius, "\n ")
}
```

This is a simple method, but it provides the result we desire:

```
> simple.circle

Center: x = 0.5
       y = 0.5
Radius: 0.25
```

When defining a method, you should ensure that its arguments match those of the generic. It may have extra arguments (that is the point of the `...` built into every generic), but it should also have all the generic's arguments.

We define the `draw` function as a generic function; we can draw shapes with `draw`, and so long as we define appropriate methods for all classes of shapes, we can expect it to do the right thing:

```
draw <-
function(x, ...)
  UseMethod("draw")
```

The call to `UseMethod` signals the evaluator that `draw` is a generic function, and thus the evaluator should first look for a specific method based on the class of the object, starting with the most-specific class, and moving up through less specific classes until the most general class is reached. All S-PLUS objects share the same general class, `class default`. Here, for example, is a version of the method `draw.circle`:

```
draw.circle <-
function(x, ...)
{
  center <- x$center
  radius <- x$radius
  symbols(center, circles = radius, add = T, inches = F,
  ...)
```

If you call `draw` with an object of class `circle` as its argument, the S-PLUS evaluator finds the appropriate method and draws a circle on the current graphics device.

## GROUP METHODS

Three groups of S-PLUS functions, all defined as calls to `.Internal`, are treated specially by the methods mechanism: the *Ops* group, containing standard operators for arithmetic, comparison, and logic; the *Math* group, containing the elementary vectorized mathematics functions (e.g. `sin`, `exp`); and the *Summary* group, containing functions (such as `max` and `sum`) that take a vector and return a single summary value. The table below lists the functions in each of the three groups.

*Table 10.1: Functions affected by group methods*

Group	Functions in Group
Ops	"+" (unary and infix), "-" (unary and infix), "*", "/", "!" (unary not), sign, "^", "%%", "%/%", "<", ">", "<=", ">=", "==", "!=", " ", "&"
Math	abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cummax, cumsum, cumprod, exp, floor, gamma, lgamma, log, log10, round, signif, sin, sinh, tan, tanh, trunc
Summary	all, any, max, min, prod, range, sum

Rather than writing individual methods for each function in a group, you can define a single method for the group as a whole. There are 17 functions in the Ops group (19 if you count both the unary and infix forms of "+" and "-") and 26 in the Math group, so the savings in programming can be significant. Of course, in writing a group method, you are responsible for ensuring that it gives the appropriate answer for all functions in the group.

Group methods have names of the form *group.class*. Thus `Math.factor` is the Math group method for objects of class `factor`, `Ops.ordered` is the Ops group method for objects of class `ordered`, and `Summary.data.frame` is the Summary group method for objects of class `data.frame`. If the method handles all the functions in the group in the same way, it can be quite simple, as for example `Summary.data.frame`:

```
> Summary.data.frame
function(x, ...)
{
  x <- as.matrix(x)
```

```

    if(!is.numeric(x))
      stop("not defined on a data frame with non-numeric
           variables")
    NextMethod(.Generic)
  }

```

**Caution**

One caution about the Summary group—it does not include either mean or median, both of which are implemented as S-PLUS code.

However, the economy of the group method is still significant even if a few of the functions need to be handled separately. As an example of a non-trivial group method, we will define a group of operators for the finite field  $Z_7$ .  $Z_7$  consists of the elements  $\{a_7 = 0, b_7 = 1, c_7 = 2, d_7 = 3, e_7 = 4, f_7 = 5, g_7 = 6\}$  (usually just called 0 to 6) with the usual operations defined so that any operation on two elements of the set yields an element of the set. Thus, for example,  $c_7 * e_7 = b_7 = 1$ ,  $d_7 / f_7 = c_7 = 2$ , and so on. Addition, subtraction, and multiplication are simply the usual arithmetic operations performed modulo 7, but division requires some extra work to determine each element's *multiplicative inverse*. Also, while elements of the field can be meaningfully combined with integers, they cannot be meaningfully combined with other real numbers or complex numbers.

We define a new class, `zseven`, to represent the finite field  $Z_7$ . The generator function is simple:

```

zseven <-
function(x) {
  if(any(x %% 1 != 0))
  {
    x <- as.integer(x)
    warning("Non-integral values coerced to int")
  }
  x <- x %% 7
  class(x) <- "zseven"
  x
}

```

The following example shows the value returned by a typical input vector:

```

> zseven(c(5, 10, 15))

[1] 5 3 1

```



We suppressed the printing of the class attribute by defining a method for `print`:

```
print.zseven <-
function(x, ...)
{
  x <- unclass(x)
  NextMethod("print")
}
```

But the significant part of our work is to define a group method `Ops.zseven` that will behave correctly for all 17 functions in the `Ops` group. Most of these are binary operations, so we begin by defining our method to have two arguments, `e1` and `e2`:

```
> Ops.zseven <- function(e1, e2) {}
```

While performing calculations, we want to ignore the class of our operands, so we begin with the following assignment:

```
e1 <- unclass(e1)
```

We do not unclass `e2` immediately, because we have to allow for the possibility that the operation is one of the unary operators ("`+`", "`-`", and "`!`"). We also want to test that `e1` is a value that makes sense in  $Z_7$  arithmetic:

```
# Test that e1 is a whole number
if(is.complex(e1) || any(e1 %% 1 != 0))
  stop("Operation not defined for e1")
# Allow for unary operators
if(missing(e2))
{
  if(.Generic == "+")
    value <- e1
  else if(.Generic == "-")
    value <- - e1
  else if(.Generic == "sign")
    value <- sign(e1)
  else value <- !e1
}
```

(The object `.Generic` is created in the evaluation frame, and contains the name of the function actually being called.)

Now we are ready to include `e2` in our calculations; we need to treat division specially, but everything else passes on to the generic methods incorporated in S-PLUS's internal code:

```
    else e2 <- unclass(e2)
# Test that e2 is a whole number
    if(is.complex(e2) || any(e2 %% 1 != 0))
        stop("Operation not defined for e2")
# Treat division as special case
    if(.Generic == "/")
        value <- e1 * inverse(e2, base = 7)
    else value <- NextMethod(.Generic)
```

Finally, we need to insure that numeric results are of class `zseven`, while logical results are passed back unchanged:

```
switch(mode(value), numeric = zseven(value), logical =
value)
```

Put together, the complete method looks like this:

```
Ops.zseven <-
function(e1, e2) {
    e1 <- unclass(e1)
# Test that e1 is a whole number
    if(is.complex(e1) || any(e1 %% 1 != 0))
        stop("Operation not defined for e1")
# Allow for unary operators
    if(missing(e2))
    {
        if(.Generic == "+")
            value <- e1
        else if(.Generic == "-")
            value <- - e1
        else if(.Generic == "sign")
            value <- sign(e1)
        else value <- !e1
    } else
    {
        e2 <- unclass(e2)
# Test that e2 is a whole number
        if(is.complex(e2) || any(e2 %% 1 != 0))
            stop("Operation not defined for e2")
```

---

```

# Treat division as special case
if(.Generic == "/")
  value <- e1 * inverse(e2, base = 7)
else value <- NextMethod(.Generic)
}
switch(mode(value), numeric = zseven(value),
       logical = value)
}

```

An alternative approach, which also works, is to ignore the special case of division in the group method, and write an individual method for division:

```

"/.zseven" <-
function(e1, e2)
{
  e1 <- unclass(e1)
  e2 <- unclass(e2)
  # Test that e1 is a whole number
  if(is.complex(e1) || any(e1 %% 1 != 0))
    stop("Operation not defined for e1")
  # Test that e2 is a whole number
  if(is.complex(e2) || any(e2 %% 1 != 0))
    stop("Operation not defined for e2")
  zseven(e1 * inverse(e2, base = 7))
}

```

Individual methods, if they exist, override group methods. In this example, the overhead of testing makes it simpler to incorporate the special case within the group method.

The special case of division required us to specify an `inverse` function to find multiplicative inverses. A working version can be defined as follows:

```
inverse <-  
function(x, base = 7)  
{  
  set <- 1:base  
  # Find the element e2 of the set such that e2*x=1  
  n <- length(x)  
  set <- outer(x, set) %% base  
  return.val <- numeric(n)  
  for(i in 1:n)  
  {  
    return.val[i] <- min(match(1, set[i, ]))  
  }  
  return.val  
}
```

Now that we've done all the work, let's try a few examples:

```
> x7 <- zseven(c(3, 4, 5))  
> y7 <- zseven(c(2, 5, 6))  
> x7 * y7
```

```
[1] 6 6 2
```

```
> x7 / y7
```

```
[1] 5 5 2
```

```
> x7 + y7
```

```
[1] 5 2 4
```

```
> x7 - y7
```

```
[1] 1 6 6
```

```
> x7 == y7
```

```
[1] F F F
```

```
> x7 >= y7
```

```
[1] T F F
```

```
> -x7
```

```
[1] 4 3 2
```

Just to be sure our last answer is what we expect it to be, we try one final example:

```
> -x7 + x7
```

```
[1] 0 0 0
```

We get the expected answer.

## REPLACEMENT METHODS

*Replacement functions* are described in the online help. Replacements are functions that can appear on the left side of an assignment arrow, typically replacing either an element or attribute of their arguments. Recall that S-PLUS interprets the expression `f(x) <- value` as `x <- "f<-"(x, value)`, so that the replacement function to be defined has a name of the form "f<-. All replacement functions act generically, that is, methods can be written for them.

As an example, consider again our class `zseven`. We want to define replacement to ensure that any new value remains in the class—that is, we want to ensure that all the elements in an object of class `zseven` are from the set  $\{0, 1, 2, 3, 4, 5, 6\}$ . To do this, we write the following method:

```
"[<-.zseven"
<- function(x, ..., value)
{
  if (is.complex(value) || value %% 1 != 0)
    stop("Replacement not meaningful for this value")
  x <- NextMethod("[<-")
  x <- x %% 7
  x
}
```

This method is an example of a *public* method; it does not use any special knowledge of the implementation of the class `zseven`, but simply the `public` view that `zseven` is essentially just the integers mod seven.

# PROGRAMMING THE USER INTERFACE USING S-PLUS

# 11

---

<b>The GUI Toolkit</b>	<b>425</b>
GUI Objects	427
GUI Toolkit Functions	427
<b>General Object Manipulation</b>	<b>428</b>
guiCreate(classname, ... )	428
guiCopy(classname, Name , NewName ,... )	431
guiModify( classname, Name ,...)	432
guiMove( classname, Name , NewName ,... )	432
guiOpen(docClassname, FileName ,...)	434
guiOpenView(docClassname, Name ,...)	435
guiRemove(classname, Name )	435
guiSave(docClassname, Name , FileName )	436
guiRemoveContents	437
<b>Information On Classes</b>	<b>438</b>
guiGetClassNames()	438
guiPrintClass	438
guiGetArgumentNames(classname)	440
<b>Information on Properties</b>	<b>441</b>
guiGetPropertyValue(classname, Name , PropName )	441
guiGetPropertyOptions	442
guiGetPropertyPrompt	442
<b>Object Dialogs</b>	<b>444</b>
guiDisplayDialog(classname, Name , bModal = F)	444
guiModifyDialog(winId, PropName , PropValue )	446
<b>Selections</b>	<b>448</b>
guiGetSelectionNames(classname, ContainerName , StartSelection = -1, EndSelection = -1)	448
guiSetRowSelections	449
guiGetRowSelections	449
guiGetRowSelectionExpr	449

<b>Options</b>	<b>451</b>
guiSetOption	451
guiGetOption	451
<b>Graphics Functions</b>	<b>452</b>
guiPlot	452
Identifying Specific Graphics Objects	452
guiGetPlotClass	453
guiUpdatePlots	454
<b>Utilities</b>	<b>455</b>
guiRefreshMemory	455
guiExecuteBuiltIn	455
<b>Summary of GUI Toolkit Functions</b>	<b>456</b>



---

## THE GUI TOOLKIT

S-PLUS is equipped with a graphical user interface (GUI) toolkit for programmers to create and manipulate GUI components: menus, toolbars, dialogs and graphics. The S-PLUS GUI toolkit is a set of S-PLUS functions that enables communications between S-PLUS applications and Windows. It provides facilities for the following applications:

1. Automating an interactive S-PLUS session.
2. Extending or customizing the existing S-PLUS GUI.
3. Developing a new GUI on top of S-PLUS.

This toolkit is object-oriented and can operate on virtually any S-PLUS GUI object. The functionality of the GUI toolkit is also available using the point-and-click operations in dialogs. The toolkit approach is for S-PLUS programmers who deal with GUI applications that are too complex for simple point-and-click operations. The general user may find it more convenient to use the dialog based tools.

GUI programs, called scripts, can either be run in the Commands window, just like any other S-PLUS program, or from Script windows, which open when a script file is opened.

A set of sample script files, shipped with S-PLUS, illustrate various uses of the toolkit (these are located in the `splus2000\samples` directory). An example showing how to create and display a simple function dialog is listed in Table 11.1.

*Table 11.1: This script to display a dialog is in the file **simple1.ssc**.*

```
#-----
# simple1.ssc: creates and displays a simple function dialog.
# This is the simplest function dialog for a function with one argument.
#-----
#-----
# Step 1: define the function to be executed
#         when the OK or Apply button is pushed
#-----
simple1 <- function(arg1){ return("Ok or Apply button is pushed!") }
#-----
# Step 2: create individual properties for arguments in the function
#-----
gui Create("Property", Name = "simple1Prop0", DialogControl = "String",
          DialogPrompt = "MyReturn", DefaultVal ue = "w");
gui Create("Property", Name = "simple1Prop1", DialogControl = "String",
          DialogPrompt = "&Y Val ue", DefaultVal ue = "30");
#-----
# Step 3: create the function info object
#-----
gui Create("FunctionInfo", Function = "simple1",
          PropertyList = c("simple1Prop0", "simple1Prop1"))
#-----
# Step 4: display the dialog
# This step must be preceded by all previous steps that created
# all required GUI objects. The statement below is equivalent to
# double click on the function name in the object explorer.
# It can be embedded in an S function to display the dialog from anywhere.
#-----

gui DisplayDialog("Function", Name= "simple1");
```

Running this script file will display the dialog in Figure 11.1. There are two ways to run a script, either from the menus by opening the script file, then clicking on the Run toolbar button, or from entering the command:

```
> source("simple1.ssc")
```

into the Commands window.

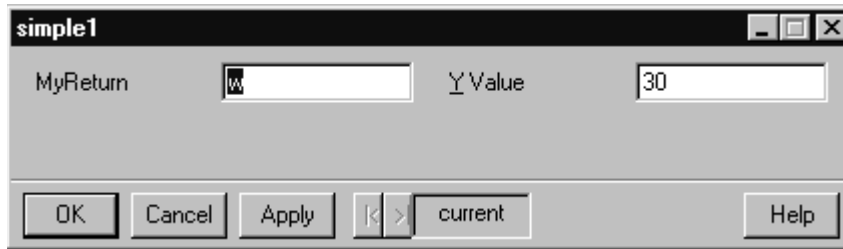


Figure 11.1: The dialog created by *simple1.ssc*.

## GUI Objects

GUI objects are the building blocks of the graphical user interface subsystem of S-PLUS. They are created, modified, and manipulated according to the events driven by user-interaction with the GUI. These events are recorded in the History log as sequences of S-PLUS commands. Note that actions from scripts loaded or created, and run, in the Script window are not then stored in the History log. To get a complete list of all these building blocks type:

```
> gui GetClassNames()
```

GUI objects created are listed in the Object Explorer, but are not stored in the standard S-PLUS databases. Instead they are kept in special binary files, which are loaded on start-up and saved on exit.

## GUI Toolkit Functions

S-PLUS functions in the GUI toolkit operate on GUI objects and generally have `gui_` as prefix, for example: `gui Create`, `gui Modify`.

The history log records all GUI operations in a session, using these S-PLUS functions. The user is encouraged to look at the History log for examples of how these functions are used.

Individual S-PLUS functions in the GUI toolkit are described below. The functions do not return anything, unless a return value is described.

## GENERAL OBJECT MANIPULATION

The S-PLUS graphical user interface is an object-oriented system. Graph elements, menus, and dialog components are all objects which may be manipulated.

The two most common actions to perform on an object are *creating* the object and *modifying* the object. Objects may also be copied, moved, and removed.

Documents such as graph sheets and scripts may be created, opened, viewed, saved, and removed.

Graphical user interface objects persist in memory for the duration of an S-PLUS session. Interface elements such as menus and dialogs are automatically saved to disk at the end of a session. The user is prompted as to whether graph objects are to be saved at the end of a session.

### `guiCreate(classname, ... )`

The function `guiCreate` creates a new GUI object of the type specified by its first argument. The object name is actually specified by an optional argument, although to be useful this is in practice a required argument. In most cases this is `Name`, but in some cases it is different, such as `NewName`.

This function is referred to as a *property command*, which means that the arguments are the properties of the object or class that the function is working on. Because this function works on objects or classes, the number of arguments depends on the number of properties for the object or class. Use the `guiGetArgumentNames(classname)` to return a list containing the valid argument names for the class in question.

<b>Note</b>
In the current version of <code>guiCreate</code> , creating an object with the same class and object name as an existing one would modify the existing object. You must use <code>guiRemove</code> before <code>guiCreate</code> to ensure the clean creation of a new object.

Table 11.2: Arguments to *guiCreate*.

Argument	Required	Description
<code>cl assname</code>	Required	A character string specifying the class of the object to be created. Use <code>gui GetCl assNames()</code> to get a list of all class names which can be used here. There are many options, including "Property", "BoxPl ot", "XAx i sTi tle", and so on.
<code>Name</code>	Usually Required	A character string specifying the name attached to the created argument. See the discussion on Object Name below.
There is a whole range of optional arguments, which vary depending on which classname is specified.	Optional	Use <code>gui GetArgumentNames(cl assname)</code> to get a list of all the argument names which can be used here.

## Object Name

For most of the functions in the GUI Toolkit the name of the object must be specified. This argument (usually called `Name`) is a character string containing the *object path name*. The syntax for this object path name is the same as the file path name but the delimiter is "\$" rather than "\" or "/", and the leading "\$\$" is used to specify the root object. For example, if a box plot name "BOX1" is created in a graphsheets name "GS1", then the function to create this box plot would be

```
> gui Create(" Box", Name=" $$GS1$BOX1")
```

All objects can have names assigned to them through scripts. Objects can also have a number assigned to them, but in all cases the `Name` is passed as a character string. The number indicates the object's position in the list of objects of that type for the document the object is located in. For example, if you want to refer to the name of the main title that is in the first plot of the graph called GRAPH1, you would specify:

```
> gui Modi fy("Mai nTi tle", Name = "$$GS1$1$1", Ti tle ="Ti tle")
```

The first part of the name path "\$\$GS1" represents the name of the graph sheet that the title is in. The next part "\$1" represents the first graph in the graph sheet that the title is in. The third and last "\$1" is the number of the title you want to modify, in this case the first main title in the graph.

The name immediately following a double delimiter "\$\$" is always treated as the name of a graph sheet or other document that the object is in. Names immediately following a single delimiter "\$" can be either the name of the graph or other container for the object or the object name itself.

For commands that work with objects, it is not necessary to specify the complete path to the object name. You can specify just the name of the object and the path will be determined based on which document is current. For example,

```
> gui Create("Arrow", Name = "ARROW1")
```

ARROW1 will be searched for in the current graph sheet and in the current graph in that graph sheet document. If not found, it will be created along with the necessary container objects. In this case a graph sheet would be created to contain the arrow.

If the path has only two parts, you can use "\$\$" and "\$" to distinguish between graphs and graph sheets. For example,

```
> gui Create("Arrow", Name = "$$GRAPH1$ARROW1")
```

This command will create the arrow named ARROW1 in the graph GRAPH1 in the current graph sheet.

```
> gui Create("Arrow", Name = "$$GS1$ARROW1")
```

This command will create the arrow ARROW1 in the graph sheet GS1. This implies that the arrow is not inside a graph but inside the graph sheet.

### Example

```
> gui Create ("Property", Name = "si mpl e1Prop1",  
+ Di al ogControl = "I nteger")
```

This will create a property object called si mpl e1Prop1. The Object Explorer can be used to examine the contents of this object.

### See Also

```
gui Copy, gui Modi fy, gui Open, gui OpenVi ew, gui Save,  
gui GetArgumentNames, gui GetCl assNames, gui GetPropertyVal ue,  
gui Di spl ayDi al og
```

## guiCopy(classname, Name , NewName ,... )

Copies the object identified by Name and classname, to a new object called NewName. The list of other arguments varies by classname, and can be used to change any properties of the copied object.

This function is also a property command, see gui Create for more details on the implications of this

*Table 11.3: Arguments to guiCopy.*

Argument	Required	Description
cl assname	Required	A character string specifying the class of the object to be copied.
Name	Required	A character string specifying the source object path name.
NewName	Required	A character string specifying the destination object path name.
Optional arguments can be used to change object properties during copying.		

### Examples

```
> gui Copy ("Property", Name="si mpl e1Prop1",
+ NewName="si mpl e1Prop2")

> gui Copy ("Property", Name="si mpl e1Prop1",
+ NewName="si mpl e1Prop2", Di al ogControl ="Fl oat",
+ Defaul tVal ue="2. 2")
```

The first example just copies an object, the second modifies two properties in the copied object only. Source objects are not modified.

The Object Explorer can be used to examine the contents of new objects, or use the gui GetPropertyVal ue function.

### See Also

```
gui Create, gui Modi fy, gui Open, gui OpenVi ew, gui Save,
gui GetArgumentNames, gui GetCl assNames, gui GetPropertyVal ue,
gui Di spl ayDi al og
```

## guiModify( classname, Name ,...)

Modify a GUI object of the type identified by Name and classname. This function is a property command, see guiCreate for more details. The optional arguments are used to modify the object properties.

Table 11.4: Arguments to guiModify.

Argument	Required	Description
cl assname	Required	A character string specifying the class of the object to be modified.
Name	Required	A character string specifying the object path name.
Optional arguments are used to modify the objects properties.		

### Example

```
> guiModi fy("Property", Name = "si mpl e1Prop1",  
+ Di al ogControl = "Stri ng", Defaul tVal ue = "OK")
```

This will modify the si mpl e1Prop1 property object to use String as its dialog control type, with the value “OK”.

### See Also

gui Create, gui Copy, gui Modi fy, gui Open, gui OpenVi ew,  
gui Save, gui GetArgumentNames, gui GetCl assNames,  
gui GetPropertyVal ue, gui Di spl ayDi al og

## guiMove( classname, Name , NewName ,... )

Moves the object to a new location, with the option of a new name specified by NewName. The list of other arguments varies by classname, and is used to change any properties of the moved object.



This function is a property command, see `gui Create` for more details.

*Table 11.5: Arguments to `guiMove`.*

Argument	Required	Description
<code>cl assname</code>	Required	A character string specifying the class of an object to be moved.
<code>Name</code>	Required	A character string specifying the object path name.
<code>NewItem</code>	Required	A character string specifying the destination object path name.
Optional arguments allow an object to be modified while being moved.		

### Example

```
> gui Move ("Property", Name="si mpl e1Prop1",
+ NewName="si mpl e1Prop3", Defaul tVal ue = "good")
```

This will move the `si mpl e1Prop1` property object to `si mpl e1Prop3`. Also, the `Defaul tVal ue` property is modified to “good”. The Object Explorer can be used to examine the contents of this object.

### See Also

`gui Create`, `gui Copy`, `gui Modi fy`, `gui Open`, `gui OpenVi ew`,  
`gui Save`, `gui GetArgumentNames`, `gui GetCl assNames`,  
`gui GetPropertyVal ue`, `gui Di spl ayDi al og`

# guiOpen(docClassname, FileName ,...)

Opens a document identified by `FileName` and `docClassname`. Using the optional arguments: `Hide`, `Show`, `Top`, `Left`, `Width`, and `Height` you can control the display location and size of the document window. You can open a graph into a full screen by specifying `Show = Full Screen`. This function is a property command, see `gui Create` for more details.

Table 11.6: Arguments to `guiOpen`.

Argument	Required	Description
<code>docClassname</code>	Required	A character string specifying the class of a document object to be opened: <code>Script</code> , <code>GraphSheet</code> , <code>Report</code> , or <code>ObjectBrowser</code> .
<code>FileName</code>	Required	A character string giving the name of the file including the file's title plus the entire directory path.
<code>Hide</code>	Optional	<code>FALSE</code> will open and display the object, <code>TRUE</code> will open the window, but it will not become the active window.
<code>Show</code>	Optional	One of: "Normal ", "Mi ni mi zed", "Maxi mi zed", "Full Screen". <code>FullScreen</code> only applies to <code>GraphSheets</code> .
<code>Durati on</code>	Optional	Amount of time in seconds that a graphsheet is displayed on a full screen. If set to 0, the graph is displayed until a key or mouse button is clicked.
<code>Top</code> <code>Left</code>	Optional	"Auto" lets the system decide on the window size, or specific coordintates can be specified for the top-left corner of the window.
<code>Width</code>	Optional	"Auto" or a specific width
<code>Height</code>	Optional	"Auto" or a specific height

## Example

```
> guiOpen( "Script",
+ FileName = "C: \\work\\exampl es\\di al ogs\\si mpl e1. ssc")
```

This will open the script file specified by `FileName` and display it in a script window.

**See Also**

[gui Create](#), [gui Copy](#), [gui Modify](#), [gui Open](#), [gui OpenView](#),  
[gui Save](#), [gui GetArgumentNames](#), [gui GetClassNames](#),  
[gui GetPropertyValue](#), [gui DisplayDialog](#)

**guiOpenView(docClassname, Name ,...)**

Opens a new view on a document identified by `Name` and `docClassname`. The document class is one of `Script`, `GraphSheet`, `Report`, `ObjectBrowser`, `data.frame`, `vector`, and `matrix`. The objects must exist and be seen within the `Object Browser`.

The required arguments, and optional arguments: `Hide`, `Show`, `Top`, `Left`, `Width`, and `Height` are identical to those described for `gui Open`.

**Example**

```
> guiOpenView("data.frame", Name = "beer")
```

This will open a grid view for a data frame called `beer`.

**See Also**

[gui Create](#), [gui Copy](#), [gui Modify](#), [gui Open](#), [gui OpenView](#),  
[gui Save](#), [gui GetArgumentNames](#), [gui GetClassNames](#),  
[gui GetPropertyValue](#), [gui DisplayDialog](#)

**guiRemove(classname, Name )**

Remove the object identified by `Name` and `classname`.

*Table 11.7: Arguments to `guiRemove`.*

Argument	Required	Description
<a href="#">classname</a>	Required	A character string specifying the class of an object to be removed.
<a href="#">Name</a>	Required	A character string specifying the object path name.

### Example

```
> gui Remove("Property", Name = "simple1Prop3")
```

This will delete the property object `simple1Prop3`. This object should disappear from the Object Explorer listing.

### See Also

`gui Create`, `gui Copy`, `gui Modify`, `gui Open`, `gui OpenView`,  
`gui Save`, `gui GetArgumentNames`, `gui GetClassNames`,  
`gui GetPropertyValue`, `gui DisplayDialog`

## guiSave(docClassname, Name , FileName )

Saves the document identified by `Name` and `docClassname` to the file specified by `FileName`.

*Table 11.8: Arguments to guiSave.*

Argument	Required	Description
<code>docClassname</code>	Required	A character string specifying the class of an document object to be saved, from the same range of options as <code>guiOpen</code> .
<code>Name</code>	Required	A character string specifying the source object path name.
<code>FileName</code>	Required	A character string giving the name of the destination path and file.

### Example

```
> gui Save( "Script", Name = "$simple1",  
+ FileName = "C:\\work\\examples\\dialogs\\simple1.ssc");
```

This will save the script document `simple1` as:

```
"C:\\work\\gui local\\examples\\dialogs\\simple1.ssc"
```

on your hard drive.

**See Also**

[gui Create](#), [gui Copy](#), [gui Modify](#), [gui Open](#), [gui OpenView](#),  
[gui GetArgumentNames](#), [gui GetClassNames](#), [gui GetPropertyValue](#),  
[gui DisplayDialog](#)

**guiRemoveContents**

Use `gui RemoveContents` to remove the objects contained by the specified container.

For example,

```
> gui RemoveContents("GraphSheet", Name=gui GetGSName)
```

will clear the contents of the current graph sheet, leaving it blank.

## INFORMATION ON CLASSES

The GUI contains a wide variety of object classes. Functions are available which provide information on the classes available, and the properties of each class.

### guiGetClassNames()

This function provides information about GUI classes of objects. It lists all the possible GUI class names. There are no required or optional arguments.

#### Return Value

It returns a list of all GUI class names, in ascending alphabetical order.

#### Example

```
guiGetClassNames()
```

#### See Also

```
guiCreate, guiCopy, guiModify, guiOpen, guiOpenView,  
guiSave, guiGetArgumentNames, guiGetClassNames,  
guiGetPropertyValues, guiDisplayDialog
```

### guiPrintClass

Use the `guiPrintClass` function to obtain a list of properties for any GUI class, and for each property, a list of acceptable values. You can use the results of this function to help construct calls to `guiCreate` and `guiModify`. For example, suppose you wanted to make a line plot. You could call `guiPrintClass` on the class "LinePlot" and see what properties such a plot contains, then construct a call to `guiCreate` to build the plot you wanted, as follows:

```
> guiPrintClass("LinePlot")  
CLASS:   LinePlot  
ARGUMENTS:  
  Name  
  Prompt:  
  Default:  
  DataSet  
  Prompt:      Data Set
```

```

        Default t:      ""
xCol umn
    Prompt:           x Col umn(s)
    Default t:       ""
yCol umn
    Prompt:           y Col umn(s)
    Default t:       ""
...
Li neStyl e
    Prompt:           Styl e
    Default t:       "Sol id"
    Option List: [ None, Sol id, Dots, Dot Dash, Short Dash,
Long Dash, Dot Dot Dash, Al t Dash, Med Dash, Tiny Dash ]
Li neCol or
    Prompt:           Col or
    Default t:       "Cyan"
    Option List: [ Black, Blue, Green, Cyan, Red, Magenta,
Brown, Lt Gray, Dark Gray, Lt Blue, Lt Green, Lt Cyan,
Lt Red, Lt Magenta, Yell ow, Bright Whi te, Transparent,
User1, User2, User3, User4, User5, User6, User7, User8,
User9, User10, User11, User12, User13, User14, User15,
User16 ]
Li neWei ght
    Prompt:           Wei ght
    Default t:       "1"
    Option List: [ Hai rline, 1/4, 1/3, 1/2, 1, 2, 3, 4, 5,
6, 8, 10, 12 ]
...
> gui Create("Li nePl ot", DataSet="fuel . frame", xCol umn="Wei ght",
+   yCol umn="Mi l eage", Li neStyl e="Al t Dash", Li neCol or="Magenta")

```

S-PLUS provides default values for most unspecified properties; thus, the plot produced by the above command shows cyan open circles at each data point. The default values for plot colors, line styles, and other basic characteristics are set in Options/Graph Styles. Other defaults can be modified by saving the object as a default.

# guiGetArgumentNames(classname)

This function returns a character string vector containing the argument names relevant to the classname in the same order as the argument names would appear using `guiGetPropertyValue(classname)`.

Table 11.9: Arguments to `guiGetArgumentNames`.

Argument	Required	Description
<code>classname</code>	Required	A character string specifying the class of the object in question.

## Return Value

A character string vector containing the list of all argument names for the specified classname.

## Example

```
> guiGetArgumentNames("Property")  
  
[1] "NewName"           "NewIndex"  
[3] "GroupItems"        "Type"  
[5] "DefaultValue"      "ParentProperty"  
[7] "DialogPrompt"      "DialogControl "  
[9] "Range"             "OptionList"  
[11] "PropertyList"      "CopyFrom"  
[13] "IsRequired"        "UseQuotes"  
[15] "NoQuotes"          "IsList"  
[17] "NoFunctionArg"     "Disable"  
[19] "HelpString"        "IsReadOnly"  
[21] "OptionListDelimiter"
```

## See Also

```
guiCreate, guiCopy, guiModify, guiOpen, guiOpenView,  
guiSave, guiGetArgumentNames, guiGetClassNames,  
guiGetPropertyValue, guiDisplayDialog
```



# INFORMATION ON PROPERTIES

When working with a GUI object, you may be interested in information regarding the properties for that object or object class. Functions are available which provide property names for a class, acceptable values for a property, prompts for a property, and values for a particular object.

## guiGetPropertyValue(classname, Name , PropName )

This function will return a character vector with the values of all the properties of the identified object, in the same order as the argument names listed by guiGetArgumentNames(classname).

Table 11.10: Arguments to guiGetPropertyValue.

Argument	Required	Description
classname	Required	A character string specifying the class of the object in question.
Name	Required	A character string specifying the object path name.
PropName	Optional	See Return Value below.

### Return Value

If PropName is specified, the return value is a character string containing the value of just that specified property, otherwise the return value is a character string vector containing the property values of all the properties of the object.

### Examples

```
> guiGetPropertyValue("Property", "simpleProp1")

[1] ""      "-1"     ""      "Normal " "30"
[6] ""      "&Y Value" "String" ""      ""
[11] ""      ""      "T"     "T"     "T"
[16] "T"     "T"     "T"     ""      "T"
[21] ""
```

```
> guiGetPropertyVal ue("Property", "si mpl e1Prop1",  
  PropName="Type")
```

```
[1] "Normal "
```

### See Also

```
gui Create, gui Copy, gui Modi fy, gui Open, gui OpenVi ew,  
gui Save, gui GetArgumentNames, gui GetCl assNames,  
gui GetPropertyVal ue, gui Di spl ayDi al og
```

## guiGetPropertyOptions

Use the `guiGetPropertyOptions` function to see a list of acceptable values for a given GUI property. For example, you can determine the available border styles for objects of GUI class "Box" as follows:

```
> guiGetPropertyOptions("Box", "BorderStyl e")  
[1] "None"           "Sol i d"         "Dots"           "Dot Dash"  
[5] "Short Dash"    "Long Dash"      "Dot Dot Dash"  "Al t Dash"  
[9] "Med Dash"      "Ti ny Dash"
```

## guiGetPropertyPrompt

Use the `guiGetPropertyPrompt` to see basic information about the property, such as its GUI prompt, its default value, and whether it is a required property. For example, for the GUI class "Box", the Border Style property information is as follows:

```
> guiGetPropertyPrompt("Box", "BorderStyl e")  
$PropName:  
[1] "BorderStyl e"  
  
$prompt:  
[1] "Styl e"  
  
$defaul t:  
[1] "Sol i d"
```

---

\$optional :

[1] T

\$data.mode:

[1] "character"

# OBJECT DIALOGS

Every GUI object has a corresponding dialog. This dialog may be displayed and its control values modified.

## guiDisplayDialog(classname, Name , bModal = F)

This function displays a dialog associated with a GUI object. The dialog can be optionally displayed as *modal* or *modeless*, using the `bModal` argument.

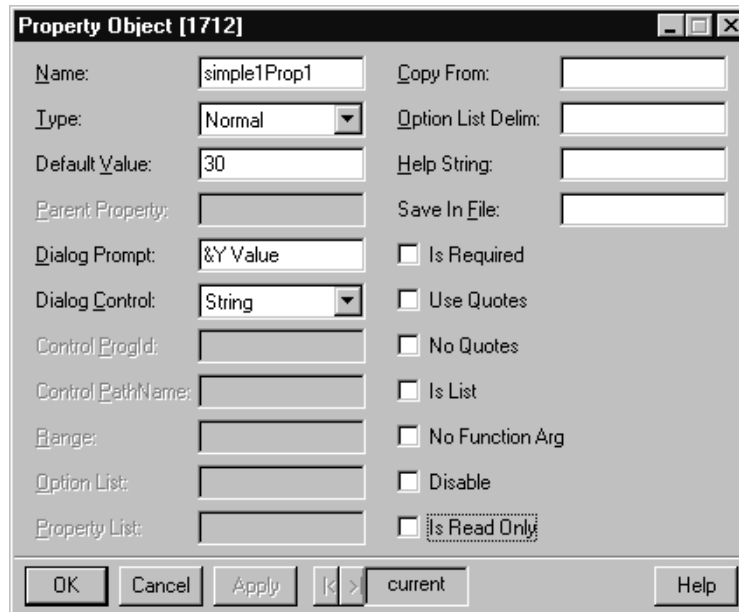
Table 11.11: Arguments to *guiDisplayDialog*.

Argument	Required	Description
<code>classname</code>	Required	A character string specifying the class of the object in question.
<code>Name</code>	Required	A character string specifying the object path name.
<code>bModal</code>	Optional	Set to TRUE for a modal dialog, otherwise the dialog will be modeless.

### Example

```
guiDisplayDialog("Property", "simple1Prop1")
```

This will result in the dialog shown in Figure 11.2.



*Figure 11.2: The simple1Prop1 property object is created after running the simple1.ssc script.*

### See Also

[gui Create](#), [gui Copy](#), [gui Modify](#), [gui Open](#), [gui OpenView](#),  
[gui Save](#), [gui GetArgumentNames](#), [gui GetClassNames](#),  
[gui GetPropertyValue](#)

## guiModifyDialog(winId, PropName , PropValue )

This function is used to modify the current value of a live active dialog. It enables communications between two or more active dialogs.

Table 11.12: Arguments to guiModifyDialog.

Argument	Required	Description
winId	Required	The Window (Dialog) ID of a live active dialog, obtained through an S-PLUS callback function.
PropName	Required	The name of the property object
PropValue	Required	The new value to place in the property specified by PropName

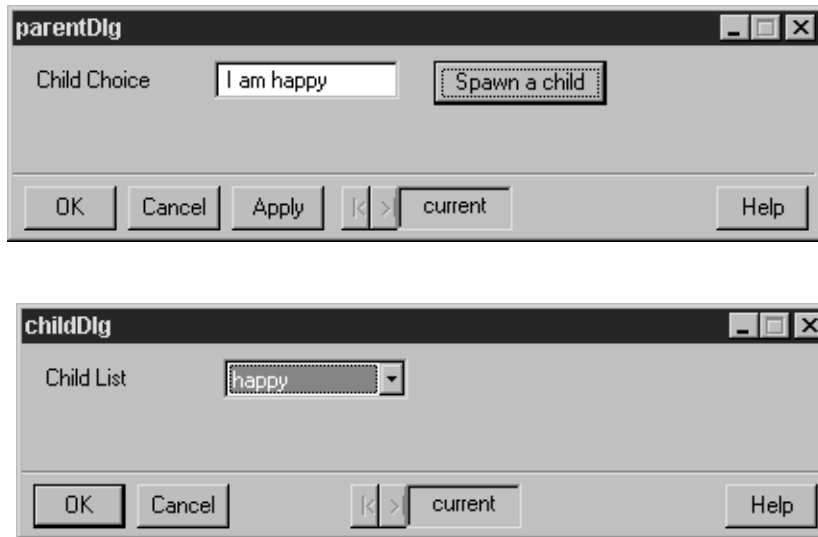
### Example

The file **dlgcomm.ssc** in the **Samples\Dialogs** directory contains a complete script for creating and displaying two dialogs that can communicate with each other through the function `guiModifyDialog`.

Table 11.13: The opening comments to the script file `dlgcomm.ssc`.

```
# This file contains an example of a dialog (parentDlg) that can spawn
# another dialog (childDlg). The childDlg can then modify a current property
# value of the parentDlg.
#
# 1. Run this script to define all components for the two dialogs,
#    and to display parentDlg.
#
# 2. Click on the "Spawn a child" button in parentDlg, so the dialog
#    childDlg appears.
#
# 3. In childDlg dialog, select an item in the "Child List" list box.
#
# 4. Look in the "Child Choice" string box in the parentDlg dialog,
#    it should now have changed.
```

Following the four steps in Table 11.13 dialogs should be displayed as shown in Figure 11.3.



*Figure 11.3: Click on Spawn a child, then select happy child, to create the child dialog shown.*

### See Also

[gui DisplayDialog](#), [gui Create](#), [gui Copy](#), [gui Modify](#), [gui Open](#),  
[gui OpenView](#), [gui Save](#), [gui GetArgumentNames](#),  
[gui GetClassNames](#), [gui GetPropertyValue](#)

## SELECTIONS

The standard approach to working with objects in a graphical user interface is to select objects and then perform some action based on the selection. S-PLUS provides programatic access to determine what objects are selected.

**guiGetSelectionNames(classname, ContainerName ,  
StartSelection = -1, EndSelection = -1)**

This function returns a character string vector containing the names of objects, in the order in which they have been selected by the user.

*Table 11.14: Arguments to guiGetSelectionNames.*

Argument	Required	Description
<code>cl assname</code>	Required	A character string specifying the class of the object in question.
<code>Contai nerName</code>	Optional	The container object name, for example: "\$SDS1" (datasheet 1) or "\$SGS1" (graphsheets 1). If specified, the selection is restricted to a specific container in the object hierarchy. If the container name is omitted, the active object for the default container would be used. For data objects, such as data frames, this would be the current Data window. For plots, it would be Graph #1 in the current graphsheets.
<code>StartSel ecti on</code>	Optional	The first object of those selected to be included in the prepared list. The default, -1, indicates all relevant object names will be returned.
<code>EndSel ecti on</code>	Optional	The last object of those selected to be included in the prepared list. If EndSelection is not specified, or is -1, then only one object name is returned, that specified by StartSelection.

### Examples

```
> guiGetSel ecti onNames(" factor")
> guiGetSel ecti onNames(" factor", " beer")
```



The first example will return a character vector containing names for selected factor data, from the currently active data frame. Note that objects can be selected and seen in the Object Explorer.

The second will return a character vector containing names for selected factor data, of the beer data frame.

## **guiSetRowSelections**

Use the `guiSetRowSelections` function to specify one or more rows of a data set as "selected"; that is, they appear highlighted in a Data window view, and plotted symbols appear highlighted in a Graphsheet window. This selection can be done interactively in the GUI; this function permits the same behavior programmatically. This is useful, for example, if you want to highlight known outliers in a data set.

## **guiGetRowSelections**

Use the `guiGetRowSelections` function to obtain a list of rows in the current data set that are selected.

## **guiGetRowSelectionExpr**

Use the `guiGetRowSelectionExpr` function to obtain an S-PLUS expression for the set of rows currently selected in a GraphSheet or Data Window. For example, consider the Data Window shown in Figure 11.4.

fuel.frame		
		1
		Weight
46	Mazda 929 V6	3480.00
47	Nissan Maxima V6	3200.00
48	Oldsmobile Cutlass Ciera 4	2765.00
49	Oldsmobile Cutlass Supreme V6	3220.00
50	Toyota Cressida 6	3480.00
51	Buick Le Sabre V6	3325.00
52	Chevrolet Caprice V8	3855.00
53	Ford LTD Crown Victoria V8	3850.00

*Figure 11.4: Data Window with two rows highlighted in fuel.frame.*

Rows 46 and 51 of the fuel.frame data set are selected. To store this information for future use, you can use `guiGetRowSelecti onExpr` as follows:

```
> guiGetRowSel ecti onExpr("fuel.frame")  
[1] "46, 51"
```

You can select those same rows in a later session using `guiSetRowSel ecti on`:

```
> guiSetRowSel ecti on("fuel.frame", "46, 51")
```

---

## OPTIONS

All elements of the S-PLUS interface are under programation control, including options.

### **guiSetOption**

Use the `guiSetOption` function to set options available in the GUI under the Options menu. For example, to disable Tool Tips in dialogs, you would use `guiSetOption` as follows:

```
> guiSetOption("ToolTipsForDialogs", "F")
```

### **guiGetOption**

Use the `guiGetOption` function to obtain the current value of any option available in the GUI under the Options menu. For example, to get the current Trellis background color, use `guiGetOption` as follows:

```
> guiGetOption("BackColorTrellis")  
[1] "Lt Gray"
```

## GRAPHICS FUNCTIONS

Graphics objects can be created and manipulated using `guiCreate` and `guiModify`. In addition, functions are available which are specifically designed for use with graphics objects.

### **guiPlot**

Use the `guiPlot` function as a convenient way to create editable graphics from S-PLUS functions. Unlike `guiCreate` and `guiModify`, which can be used to create graphics but are also used to create other GUI objects, `guiPlot` is used exclusively to create graphics. It therefore has a simpler and more intuitive syntax.

For example, suppose you want to create two line plots on the same graph in a new graph sheet, and store the data within the graph sheet. The following calls do exactly that:

```
> x <- 1:30
> guiPlot("Line", DataSetValues=data.frame(x, cos(x),
      sin(x)))
[1] "GS2"
```

Suppose you want to create a Trellis graph with two conditional variables. You can do this with `guiPlot` as follows:

```
> guiPlot("Loess", DataSetValues=environmental,
      NumConditioningVars=2)
[1] "GS3"
```

### Identifying Specific Graphics Objects

To modify specific pieces of editable graphics using `guiModify`, you must specify the object name, showing its path in the object hierarchy. You can use the following functions to get the object name for a specific object type. Most of them take a `GraphSheet` name and a `GraphNum` argument; you can use `guiGetGSName` to obtain the name of the current `GraphSheet`:

- `guiGetAxisLabelName`: returns the name of the `AxisLabels` for a specified axis (axis 1 by default).

- `gui GetAxisName`: returns the name of the axis for a specified axis (axis 1 by default).
- `gui GetAxisTitleName`: returns the name of the axis title for a specified axis (axis 1 by default).
- `gui GetGSName`: returns the name of the current GraphSheet. (This function takes no arguments.)
- `gui GetGraphName`: returns the GraphName of the graph with the specified GraphNum in the specified GraphSheet.

`gui GetPlotName`: returns the Name of the plot with the specified PlotNum in the specified GraphNum in the specified GraphSheet.

For example,

```
> gui Plot("Line"DataSetValues=data.frame(1:20, sin(1:20)))
> gui Modify("YAxisTitle", Name=gui GetAxisTitleName(),
  Title="sin(x)")
```

## guiGetPlotClass

Use the `gui GetPlotClass` function to do one of the following:

1. For a specified plot type, return the GUI class to which the plot type belongs. The class name is a required argument in `gui Modify`.
2. If no plot type is specified, return a list of valid plot types. These are the valid plot types for `gui Plot`.

For example,

```
> gui GetPlotClass("Scatter")
[1] "LinePlot"
```

```
> gui GetPlotClass()
[1] "Scatter"          "Line"            "LineScatter"
[4] "IsolatedPoints"  "HighDensity"     "Text"
[7] "Bubble"          "Color"           "BubbleColor"
[10] "Loess"           "Spline"          "Robust"
[13] "Dot"            "TimeSeries"      "Step"
[16] "Vertical Step"   "HorizontalDensity" "YZeroDensity"
[19] "Super"          "Kernel"          "LinearCF"
[22] "Polynomial"     "ExpCF"           "LnCF"

...
> gui Plot("Loess", DataSetValues=environmental[, 1:2])
> gui Modify(gui GetPlotClass("Loess"), Name=
  gui GetPlotName(), LineColor="Red")
```

## guiUpdatePlots

To update the plots created by `gui Plot(DataSetValues=...)` with new data set values, use `gui UpdatePlots`.

For example,

```
> gsName <- gui Plot("Scatter", DataSetValues=fuel.frame
[, 1:2])
> gui UpdatePlots(GraphSheet=gsName, DataSetValues=
  environmental[, 1:2])
```

The number of columns in the data set used in `gui UpdatePlots` should be the same as the number of columns in the original data set used in `gui Plot`.

## UTILITIES

Utility functions are available to perform GUI actions not related to specific objects.

### guiRefreshMemory

Use the `gui RefreshMemory` to remove unneeded objects from memory; you can optionally restore the object's summary data after clearing the entire object from memory.

### guiExecuteBuiltIn

Use the `gui ExecuteBuiltIn` function to launch dialogs or perform other operations that are "built-in" to the GUI. Built-in operations are stored for each GUI property, and can be viewed for any particular object using the `gui GetPropertyValue` function. For example, suppose we wanted to view the "About S-PLUS" dialog at some point in our function. Open the Object Explorer and create a new page containing the Interface Class Menu Item. Expand the `SPlusMenuBar` node and highlight the menu of interest in the left pane. Right-click on the desired menu item in the right pane and select Command from the right-click menu. The built-in operation is shown at the top of the page:

```
> gui ExecuteBuiltIn(" $$SPUsMenuBar$Object_Browser$Window$
  Title-Vertical ")
```

We can then use this command in a call to `gui ExecuteBuiltIn`:

```
> gui ExecuteBuiltIn(
  "$$SPUsMenuBar$Object_Browser$Help$About_S_PLUS")
```

# SUMMARY OF GUI TOOLKIT FUNCTIONS

*Table 11.15: Summary of graphics interface functions.*

Function	Description
<code>gui Create</code>	Creates all interface objects.
<code>gui Copy</code>	Copies one object to another, possibly with modifications.
<code>gui Modify</code>	Modifies an object.
<code>gui Move</code>	Moves an object, possibly with modifications.
<code>gui Open</code>	Opens an object of type Script, GraphSheet, ObjectBrowser or Report.
<code>gui OpenView</code>	Opens an object that is viewable by the Object Explorer.
<code>gui Remove</code>	Deletes an object created by <code>guiCreate</code> .
<code>gui Save</code>	Saves an object to a file.
<code>gui GetClassNames</code>	Lists all available object types.
<code>gui GetPropertyValue</code>	Lists details of property values for a given object.
<code>gui GetArgumentNames</code>	Lists all relevant arguments for a given object.
<code>gui DisplayDialog</code>	Displays the identified dialog.
<code>gui ModifyDialog</code>	Modifies a currently active dialog, and can be used in conjunction with callback functions.
<code>gui GetSelectionNames</code>	Reports the objects currently selected by the user.



# CUSTOMIZED ANALYTICS: A DETAILED EXAMPLE

# 12

---

<b>Overview of the Case Study</b>	<b>458</b>
<b>The Basic Function</b>	<b>460</b>
<b>Enhancing the Function</b>	<b>461</b>
Type Checking	461
Adding Information on an Object	462
<b>Constructing Methods</b>	<b>465</b>
Adding a Class	465
Print Method	466
Summary Method	467
Plot Method	470
<b>Customized Graphical User Interface</b>	<b>474</b>
Adding Menu Items	475
Creating Toolbars	476
Removing Menu Items and Toolbars	477
Customizing the Dialog	477
Creating a Sophisticated Tabbed Dialog	479
The Menu Function	479
The GUI Function	480
Customizing the Context Menu	483
<b>Writing Help Files</b>	<b>487</b>
Creating the Help File	487
<b>Distributing Functions</b>	<b>489</b>
Using Text Files	489
Using Libraries	490

## OVERVIEW OF THE CASE STUDY

One of the major strengths of S-PLUS is that it is not just a statistics and graphics package but rather is a fully extensible programming environment. It is an ideal tool for the development and deployment of customized analytics. Programmers can develop their own analytic techniques for both their own use and use by others, and make these techniques available with their own summaries, plots, menus, and dialogs.

In this chapter we walk through a detailed example of creating a new modeling technique along with relevant methods for exploring and using the model. We then proceed to develop a custom graphical user interface to make this modeling technique easily available through menus, toolbars, and dialogs.

The code for these examples is available in the file **`samples/gaussfit.ssc`**, under the top-level directory where S-PLUS was installed.

### The Model

In statistics it is common to assume that a sample comes from a Gaussian distribution, also known as a Normal distribution, and to make Normal-theory based inferences given that assumption. When we say that a sample is Gaussian, we are saying that the values in the sample arise with a certain distribution parameterized by a mean and standard deviation. We will often use the sample mean and standard deviation to estimate these values and then proceed to develop confidence intervals for the value of the true mean given these sample values.

When performing such procedures it is useful to perform diagnostics to determine if the sample is consistent with a Gaussian distribution in order to determine whether Normal-theory based inference is appropriate. Both analytic and graphical techniques are available for assessing Normality.

In this example we will develop a function which fits a Gaussian distribution to a sample and make diagnostics available to assess Normality. This will be made available first as a set of command line functions, and then through a menu and dialog interface.

### The Functions

First we will write a simple function which will find estimates of location and scale for a sample which we will suppose to be Gaussian, that is to have a Normal distribution. The default will be to calculate the maximum likelihood estimates, although we will add an option to calculate robust estimates instead.

Next we will enhance the function to keep track of the data used and to do some error checking.

As a follow-up step, we will create `print`, `plot`, and `summary` methods to display the data and explore whether the supposed normality of the sample is justified.

Finally, we will develop menus, toolbars, and dialogs to call this function from the graphical user interface.

## Test Data

We will look at two sets of simulated data to demonstrate these functions. The first set will consist of a sample from a Gaussian distribution:

```
> set.seed(716)
> gauss.dat <- rnorm(200, mean=5, sd=2)
```

The second set will consist of data from a Chi-square distribution:

```
> chi sq.dat <- rchi sq(200, df=10)
```

## THE BASIC FUNCTION

To start out we will write a function which simply calculates estimates of the mean and standard deviation for the sample. The familiar sample estimates of the mean and standard deviation are the Maximum Likelihood Estimates (MLE's) for the Gaussian distribution. These are the estimates suggested by likelihood theory and commonly used in basic statistics. Actually, basic statistics typically uses an unbiased estimate of the variance, and its square root as an estimate of the standard deviation. This unbiased estimate divides by  $(\text{length}(x) - 1)$  instead of just  $\text{length}(x)$ .

One disadvantage in using the MLE's is that they are not robust to outliers in the sample. That is, they may be unduly influenced by a few abnormal points. We can use robust estimates of mean and standard deviation to get estimates which are more resistant to such influences. The function `gaussfit1` will take a vector `x` and construct either MLE or Robust estimates of the mean and standard deviation of the sample.

```
gaussfit1<-function(x, method="mle"){
  if (method=="mle") {
    mu<-mean(x)
    sigma<-sqrt(sum((x-mu)^2)/length(x))
  }
  else {
    mu<-median(x)
    sigma<-mad(x)
  }
  c(Mean=mu, SD=sigma)
}
```

In this function we simply take the sample vector and a character string specifying the type of estimates to use, calculate the appropriate set of estimates, and return a vector containing the estimates with the elements of the vector named appropriately. We may apply this function to our Gaussian sample

```
> gaussfit1(gauss.dat)
      Mean      SD
4.836349 1.810814
```

and to our Chi-Square sample

```
> gaussfit1(chi sq.dat)
      Mean      SD
10.07693 4.791151
```

## ENHANCING THE FUNCTION

Once we have a basic function we can enhance it to make it more flexible and descriptive. We will begin by adding some type checking to the function, and then incorporate more descriptive information into the returned results.

### Type Checking

Type checking is useful to make sure appropriate values are specified by the user, and to warn the user if inappropriate values are supplied. We will add some type checking to the function:

- First we will make sure `x` is a simple vector.
- We will provide an option to remove any missing values.
- We will make sure the method specified is either `"mle"` or `"robust"`. Also we can let user abbreviate this argument.

```
gaussfit2<-function(x, method="mle", na.rm=F){

# Make sure x is a vector.
x<-as.vector(x)
if (na.rm)
  x<-x[!is.na(x)]

# Match method with legal choices, exit if
# no match.
which.method<-pmatch(method,c("mle", "robust"))
if (is.na(which.method))
  stop('method must be "mle" or "robust".')

# Calculate statistics.
if (which.method==1) {
  mu<-mean(x)
  sigma<-sqrt(sum((x-mu)^2)/length(x))
}
else {
  mu<-median(x)
  sigma<-mad(x)
}
```

```
c(Mean=mu, SD=sigma)
}
```

We use some of these new options below:

- Use the "robust" method.

```
> gaussfi t2(gauss.dat, method="robust")
      Mean      SD
4. 901699 1. 737397
```

- Abbreviate the method argument.

```
> gaussfi t2(gauss.dat, method="r")
      Mean      SD
4. 901699 1. 737397
```

- Exit with an error message if illegal method argument specified.

```
> gaussfi t2(gauss.dat, method="mre")
Error in gaussfi t2(gauss.dat, method = "mre"...) : method must
be "mle" or "robust".
Dumped
```

- Omit missing values.

```
> gaussfi t2(c(1, 2, 3, NA))
      Mean      SD
      NA      NA

> gaussfi t2(c(1, 2, 3, NA), na.rm=T)
      Mean      SD
      2  0. 8164966
```

## Adding Information on an Object

The functions developed so far return the estimated parameter values, but no information on where the numbers come from or on what data was used to get the estimates. We can enhance the returned object by adding:

- The call used to construct the object.
- Optionally we may save the data as part of the object.

Since we are interested in returning an object with multiple components we will return a list rather than a vector.

```
gaussfit3<-function(x, method="mle", na.rm=F, save.x=F){

  # Make sure x is a vector.
  x<-as.vector(x)

  if (na.rm) x<-x[!is.na(x)]

  # Match method with legal choices, exit if no match.
  which.method<-pmatch(method, c("mle", "robust"))

  if (is.na(which.method))
    stop('method must be "mle" or "robust".')

  # Calculate statistics.
  if (which.method==1) {
    mu<-mean(x)
    sigma<-sqrt(sum((x-mu)^2)/length(x))
  }

  else {
    mu<-median(x)
    sigma<-mad(x)
  }

  # Save call.
  obj.call<-match.call()

  # Organize results.
  result<-list(call=obj.call, estimate=c(Mean=mu, SD=sigma))

  if (save.x) result$x<-x

  result
}
```

We may apply this new function to our sample sets of data. The call now tells us what data was used in each case.

- Gaussian data:

```
> gaussfi t3(gauss. dat)
$call:
gaussfi t3(x = gauss. dat)

$estimate:
  Mean  SD
4.836349 1.810814
```

- Chi-Square Data:

```
> gaussfi t3(chi sq. dat)
$call:
gaussfi t3(x = chi sq. dat)

$estimate:
  Mean  SD
10.07693 4.791151
```

- Saving the data as part of the object:

```
> gaussfi t3(c(1, 2, 3), save. x=T)
$call:
gaussfi t3(x = c(1, 2, 3), save. x = T)

$estimate:
  Mean  SD
2 0.8164966

$x:
[1] 1 2 3
```



## CONSTRUCTING METHODS

Now that we are returning more than a simple vector, the printed results are not particularly attractive. We can make them more attractive by specifying a special print routine specific to the results of this function. We do this by giving the object returned a class and writing a `print` method for the class.

We can also write `plot` and `summary` methods providing useful plots and summaries for this type of model.

We will construct `print`, `summary`, and `plot` methods for "gaussfit" objects. The `plot` and `summary` methods will be useful for assessing whether it is reasonable to assume that our data is Gaussian.

- The `print` method will print out the call and the estimates in a nicely formatted manner.
- The `summary` method will print out the same information as `print`, plus the results of two tests of whether the sample is consistent with a Gaussian distribution.
- The `plot` method will plot a histogram of the data with the fitted Gaussian density, plus three plots to assess the goodness of the fit.

### Adding a Class

In order to have specific `print`, `plot`, and `summary` methods for the object returned by the function, the returned object needs to have a class attribute indicating its class. To give our fitted object a class we simply assign a class to the result of the fitted object using the `class` function.

```
gaussfit <- function(x, method="mle", na.rm=F, save.x=F) {

  # Make sure x is a vector.
  x <- as.vector(x)

  if (na.rm) x <- x[!is.na(x)]

  # Match method with legal choices, exit if no match.
  which.method <- pmatch(method, c("mle", "robust"))

  if (is.na(which.method))
    stop('method must be "mle" or "robust".')
```

```
# Calculate statistics.
if (which.method==1) {
  mu<-mean(x)
  sigma<-sqrt(sum((x-mu)^2)/length(x))
}
else {
  mu<-median(x)
  sigma<-mad(x)
}

# Save call.
obj.call<-match.call()

# Organize results.
result<-list(call=obj.call, estimate=c(Mean=mu, SD=sigma))

if (save.x) result$x<-x

class(result) <- "gaussfit"

result
}
```

## Print Method

The `print` method for an object is used when we explicitly say `print(object)` or when we type the name of the object at the command line and press return. This method generally prints a concise description of the most important information in the object.

We can write a `print` method for a `gaussfit` object which displays the call and parameter estimates with descriptive labels:

- Gaussian data:

```
> gaussfit(gauss.dat)
Call:
gaussfit(x = gauss.dat)

Parameter Estimates:
      Mean      SD
4.836349 1.810814
```

- Chi-square data:

```
> gaussfit(chi sq. dat)
Call:
gaussfit(x = chi sq. dat)

Parameter Estimates:
      Mean      SD
10.07693  4.791151
```

The `print` method for the `gaussfit` object uses `print` to print the components of the object and `cat` to print text labeling each component. The `invisible(object)` as the last line of the function indicates that the object will be returned invisibly. This allows us to return the value for assignment if the results are assigned, while avoiding the automatic printing of results if the results are not assigned.

```
print.gaussfit<-function(x,...){

  cat("Call:\n")
  print(x$call)

  cat("\nParameter Estimates:\n")
  print(x$estimate)

  invisible(x)
}
```

## Summary Method

The `summary` method generally provides more detailed information than does `print`. Often `summary` performs additional computations to construct its results. The summary for a "`gaussfit`" object will print:

- The same output as does `print`.
- Results of a Chi-square goodness of fit test comparing the sample to a Gaussian distribution with the fitted mean and standard deviation.
- Results of a Kolmogorov-Smirnov test comparing the sample to a Gaussian distribution with the fitted mean and standard deviation.

Our summary will print the desired information and return a NULL value. Often summary methods will instead construct a list containing calculated values with a class such as "summary.gaussfit". This will then be printed by a print method for the summary object e.g. print.summary.gaussfit.

Note that to perform these tests we need to have the sample used to construct the object. If the object was created using save.x=T we may get the sample from the object. Otherwise, we may look at the call to see what data was used, and then use eval to get this data.

Our summary function returns the following results for the Gaussian data:

```
> summary(gaussfit(gauss.dat))
```

```
Call:
```

```
gaussfit(x = gauss.dat)
```

```
Parameter Estimates:
```

```
      Mean      SD
```

```
4.836349 1.810814
```

```
Chi-square Goodness of Fit Test
```

```
data: x
```

```
Chi-square = 10.29, df = 16, p-value = 0.8511
```

```
alternative hypothesis: True cdf does not equal the normal  
Distn. for at least one sample point.
```

```
One-sample Kolmogorov-Smirnov Test;
```

```
hypothesized distribution = normal
```

```
data: x
```

```
ks = 0.0578, p-value = 0.5158
```

```
alternative hypothesis: True cdf does not equal the normal  
Distn. for at least one sample point.
```

The large p-values indicate that the data is consistent with a Gaussian (Normal) distribution.

The results for the Chi-square data differ markedly:

```
> summary(gaussfit(chisq.dat))
```

```
Call:
```

```
gaussfit(x = chisq.dat)
```

Parameter Estimates:

Mean	SD
10.07693	4.791151

Chi-square Goodness of Fit Test

data: x

Chi-square = 46.33, df = 16, p-value = 0.0001

alternative hypothesis: True cdf does not equal the normal  
Distn. for at least one sample point.

One-sample Kolmogorov-Smirnov Test;

hypothesized distribution = normal

data: x

ks = 0.1152, p-value = 0.0099

alternative hypothesis: True cdf does not equal the normal  
Distn. for at least one sample point.

With the Chi-square data we have small p-value indicating the data is not likely to be from a Gaussian distribution.

The `summary` method for a `gaussfit` object locates the data used to construct the `gaussfit` object using either the call or the saved `x` values, prints the object, and calls some built-in tests of goodness of fit to assess the Normality of the sample.

```
summary.gaussfit<-function(object,...) {
  # Get data used to construct object.

  if (!is.null(object$x))
    x<-object$x
  else
    x<-as.vector(eval (object$call$x))

  # Print object.
  print(object)

  # Pull off estimates.
  mu<-object$estimate[1]
  sigma<-object$estimate[2]
```

```
# Do Chi-square test.
print(chi sq.gof(x, m = mu, s = sigma))
# Do Kolmogorov-Smirnov test.
print(ks.gof(x, m = mu, s = sigma))

invisible()
}
```

Note that there are some dangers involved in using the call to identify the data used. Due to the way S-PLUS searches for objects, it will not find the data unless the data is in either one of the directories on the search list or what is called the *session frame*. In particular, if the data is defined in some function which then calls `gaussfit`, the data will not be found. If the data set stored under the specified name has changed since the `gaussfit` object was created, then the data retrieved will not correspond to the fitted model. We have added the `save.x` argument as one way to deal with such situations.

## Plot Method

The `plot` method for an object generally produces plots describing the object, often including diagnostic plots. The plot for a "gaussfit" object will produce:

- A histogram of the data with the fitted Gaussian distribution overlaid.
- A histogram of the percentiles of the fitted Gaussian distribution corresponding to the observed data. If the data is Gaussian these percentiles will be roughly uniform.
- A QQPlot comparing the quantiles of the data with those of a Gaussian.
- A CDF Plot comparing the empirical CDF for the data with the CDF of a Gaussian.

When a plot function produces multiple figures, we must provide some mechanism for controlling layout of the plots. Possibilities include:

- Explicitly breaking the graphics device into multiple regions using `par(mfrow=c(2, 2))`.

- Simply producing the plots and leaving it to the user to break the graphics device into multiple regions.
- Automatically pausing between plots by setting `par(ask=T)`. This is done in `plot.data.frame`.
- Using menu to prompt the user as to which plot is produced. This is done in `plot.lm` when `ask=T` in the call to `plot.lm`.

We will split the region into a 2 by 2 grid by default, and include a "grid" argument to allow the user to override this gridding by setting `grid=F`.

The plots for Gaussian data are given in Figure 12.1. Note that in the bottom two plots the points fall tightly about the reference line, indicating that the sample is consistent with a Gaussian distribution.

```
> plot(gaussfit(gauss.dat))
```

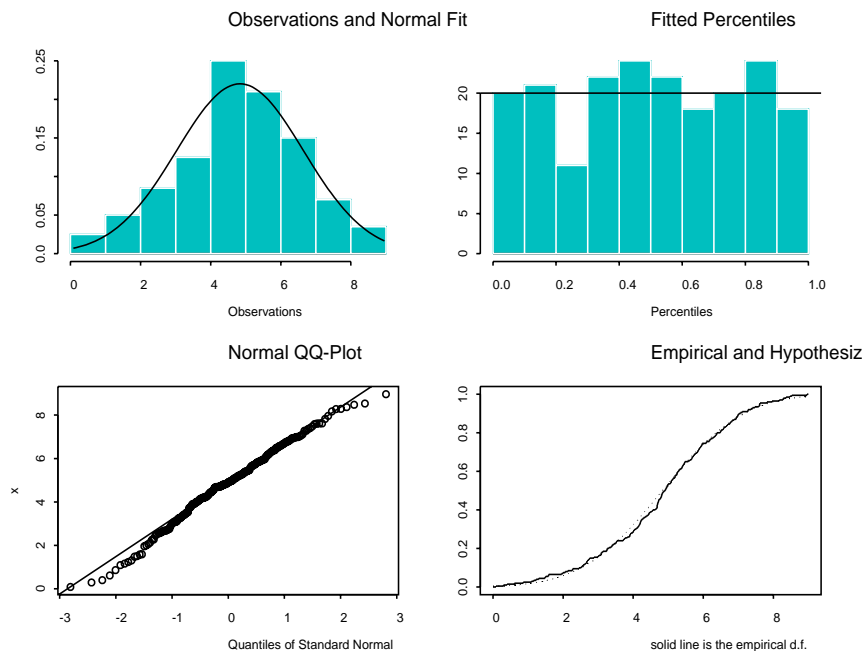


Figure 12.1: The points fall tightly around the reference line in the lower two plots.

In the plots for the Chi-square data the points in the bottom two figures depart from the reference line, indicating that the sample is not consistent with a Gaussian distribution.

```
> plot(gaussfit(chi sq.dat))
```

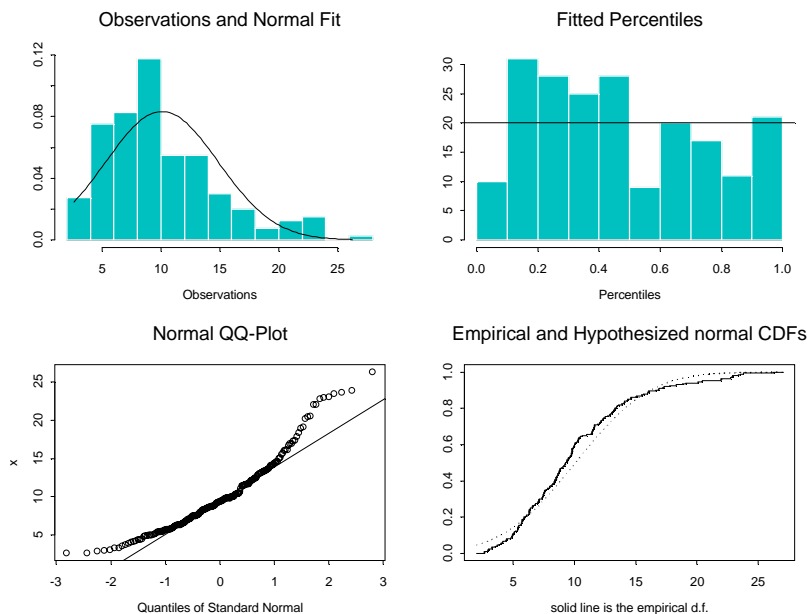


Figure 12.2: The points notably depart from the reference line in the lower two plots.

The plot method for `gaussfit` objects first resets the plot layout values and saves the current plot layout specification as `old.par`. The `on.exit` call specifies that the plot layout values will be reset to the old values on exit from the function, whether the exit is due to completion or to interruption by either the user or an error.

The function then extracts the estimates and data and uses them to produce the desired plots.



---

```

plot.gaussfit<-function(x, grid=T,...){
# Split graphics device if so desired.
if (grid){
  old.par <- par(mfrow = c(2, 2))
  on.exit(par(old.par))
}

# Get data used to construct object.
if (!is.null(x$x))
  data<-x$x
else
  data<-as.vector(eval (x$call$x))

# Pull off estimates.
mu <- object$estimate[1]
sigma<- object$estimate[2]

# Histogram
hist(data, prob = T, xlab="Observations")
new.x <- seq(min(data), max(data), length = 100)
new.dens <- dnorm(new.x, mean = mu, sd = sigma)
lines(new.x, new.dens)
title("Observations and Normal Fit")

# Fitted Percentiles
pct.fit<-pnorm(data,mean=mu, sd=sigma)
hist.centers<-hist(0.0001+0.999*pct.fit,
  nclass=10, xlab="Percentiles")
abline(h=length(pct.fit)/length(hist.centers))
title("Fitted Percentiles")

# QQ-Plot
qqnorm(data)
qqline(data)
title("Normal QQ-Plot")

# CDF
cdf.compare(data, mean=mu, sd=sigma)

invisible()
}

```

## CUSTOMIZED GRAPHICAL USER INTERFACE

We now have a well developed function for fitting a Gaussian density to a set of data, along with relevant plots and summaries for assessing whether the Gaussian fits the data well. A further step in developing this set of functions is to make them easily accessible from the Graphical User Interface (GUI).

We will augment our menus and toolbars to access the `gaussfit` function, as well as the corresponding plots and summaries. We will do this in steps, starting from a simple implementation and proceeding to a sophisticated customized dialog:

- We will first create a menu item to call `gaussfit`. The menu will launch an autogenerated dialog which prompts for the input to `gaussfit` and either save the `gaussfit` object or print the results.
- Next we will create a toolbar palette and button which launches the autogenerated dialog.
- We will then provide a function to remove the menu items and toolbars which were created. This is an important tool to allow the end user to undo changes to the interface.
- We then create a customized dialog which reorganizes the arguments in the dialog, adds custom controls, and provides descriptive prompt names. This customized dialog adds polish to the interface and ease of use in accessing the function.
- Next we create a sophisticated tabbed dialog which fits the `gaussfit` model and optionally calls the `print`, `summary`, and `plot` functions for `gaussfit`. This allows the user to specify model settings and detailed summary results in a single dialog.
- As a final touch we will customize the context menu for the `gaussfit` object so that a user can right-click on a `gaussfit` object in the Object Explorer and select `Print`, `Summary`, or `Plot` to perform that operation on the object.

The user need not perform all of these steps to make their functions available through the Graphical User Interface. Some programmers will find the first step satisfactory, while other will go far beyond what we show here.

In S-PLUS there is a set of functions for creating GUI objects such as graphs, menus, toolbars, and dialogs. The functions which are most useful are `gui Create`, `gui Modify`, and `gui Remove`. These functions will be used extensively in this section to generate GUI objects. Note that GUI objects are not stored as objects in the working directory. Rather they are kept in memory while S-PLUS is running and are written to disk with the rest of the interface options between sessions.

## Adding Menu Items

Menu items are represented by `MenuItem` objects. These objects specify the menu structure and actions to perform upon reaching terminal nodes in the hierarchical menus.

- Items of `Type="Menu"` can contain other menu items as child objects. They are used to build the parent nodes in hierarchical menus.
- Items of `Type="MenuItem"` perform actions when selected. They can be used to perform actions or to launch dialogs for functions.

We will use `gui Create` to create a new menu topic "My Methods" on the main S-PLUS menu which has an entry "Gaussian Fit" which calls our `gaussfit` function.

```
create.menu.gaussfit<-function() {

# Create new menu items
gui Create("MenuItem",
          Name = "SPR usMenuBar$MyMethods",
          Type = "Menu",
          MenuItemText = "&My Methods",
          Index = 11,
          OverWrite=F)

gui Create("MenuItem",
          Name="SPR usMenuBar$MyMethods$Gaussfit",
          Type="MenuItem",
          Action="Function",
          Command="gaussfit",
          MenuItemText="&Gaussi an Fi t")
invisible()
}
```

The `Index` argument specifies the location in the menu structure at which to place the `MyMethods` menu item. The `Action` and `Command` arguments specify that we wish to call the function `gaussfit` when “Gaussian Fit” is selected.

As we have not yet provided information describing a dialog for `gaussfit`, a simple autogenerated dialog is used with an edit field for the name of the `Returned Value` and for each argument to `gaussfit`. If a `Returned Value` is specified then the object returned will be saved under that name, otherwise the results will be printed.

## Creating Toolbars

In addition to the menu item, we can call our function from a toolbar. Toolbars contain buttons which perform actions when pressed.

- Toolbar palettes are represented by `ToolBar` objects.
- Toolbar buttons are represented by `ToolBarButton` objects. Toolbar buttons can perform the same actions as menu items.

We can create a toolbar palette with a button to call `gaussfit`:

```
create.toolbar.gaussfit<-function(){  
  # Create new toolbar with button  
  
  guiCreate("ToolBar", Name = "MyMethods")  
  
  guiCreate("ToolBarButton",  
    Name = "MyMethods$Gaussfit",  
    Type = "Button",  
    Action = "Function",  
    Command="gaussfit",  
    TipText = "Gaussian Fit", ShowDialogOnRun=T)  
  
  invisible()  
}
```

If a toolbar does not immediately appear upon running this function, go to `View/Toolbars` in the menu and select `MyMethods` to display the new toolbar.

Clicking on the toolbar button will launch an autocreated dialog for `gaussfit`.

To edit the toolbar image, right-click on the button and select Edit Image. This will launch Paint with the bitmap to be edited. Upon exiting Paint save the edited image and it will be used for the toolbar button image.

## Removing Menu Items and Toolbars

We may want to remove menus and toolbars when we are done working with them. It is a good idea to provide a function to remove interface modifications.

The `gui Remove` function removes interface objects. Removing the parent node of the menu or toolbar hierarchy will also remove any child objects.

```
remove.toolbar.gaussfit<-function(){
  gui Remove("Tool bar", Name = "MyMethods")
  invisible()
}

remove.menu.gaussfit<-function(){
  gui Remove( "Menu item",
    Name = "SPUsMenuBar$MyMethods")
  invisible()
}
```

## Customizing the Dialog

The simple autogenerated dialog allows the user to enter arguments to the function, but does not provide much structure or guidance. We can create a customized dialog for `gaussfit` which allows the reorganization of arguments, has more descriptive prompts, and uses custom controls.

Dialogs are constructed by combining `Property` objects. Each property specifies one control in the dialog and corresponds to one argument in the function.

A `FunctionInfo` object maps controls in the dialog to arguments in the function, and specifies the order of the controls in the dialog. For details on the properties of these objects see Chapter 17, Extending the User Interface.

We will use `gui Create` to create a customized dialog for `gaussfit`.

```
create.dialog.gaussfit<-function(){
  gui Create("Property",
    Name="GaussfitX",
    DialogPrompt="Sample Data", DialogControl="String")
  gui Create("Property",
```

```
        Name="GaussfitMethod",
        DialogPrompt="Fitting Method",
        DialogControl="ListBox",
        OptionList=c("ml e", "robust"),
        UseQuotes=T,
        DefaultValue="ml e")

gui Create("Property",
        Name="GaussfitNa",
        DialogPrompt= "Remove Missing Values",
        DialogControl="CheckBox", DefaultValue=T)

gui Create("Property",
        Name="GaussfitSaveX",
        DialogPrompt= "Save Data with Fit",
        DialogControl="CheckBox",
        DefaultValue=F)

gui Create("Property",
        Name="GaussfitSaveAs",
        DialogPrompt="Save Fit As", DialogControl="String",
        DefaultValue="last.gaussfit")

gui Create( "FunctionInfo",
        Name = "gaussfit",
        Function = "gaussfit",
        DialogHeader = "Fit Gaussian Density",
        StatusString =
            "Fits Gaussian density to a set of data",
        PropertyList = "GaussfitX, GaussfitMethod,
            GaussfitNa, GaussfitSaveX, GaussfitSaveAs",
        ArgumentList =
            "#0 = GaussfitSaveAs,
            #1 = GaussfitX,
            #2= GaussfitMethod,
            #3 = GaussfitNa,
            #4 = GaussfitSaveX")
invisible()
}
```

## Creating a Sophisticated Tabbed Dialog

In a dialog, the user is interested in performing more actions during a single invocation of the dialog than they perform in a single function call at the command line.

We can wrap the functionality of `gaussfit`, `print.gaussfit`, `summary.gaussfit` into a single function `menuGaussfit`, which calls these functions based upon arguments to `menuGaussfit`. We will then construct a sophisticated tabbed dialog for `menuGaussfit`.

## The Menu Function

The `menuGaussfit` function has two basic sections. The first section calls `gaussfit`. We do some tricks with `match.call` and `eval` to get the call in the `gaussfit` object to contain the names of the objects passed into `menuGaussfit`. The second section calls the various methods for `gaussfit`.

If we were not concerned about the call we would simply do

```
fit <- gaussfit(x, method, na.rm, save.x)
```

in the first part of the function.

Our function `menuGaussfit`:

```
menuGaussfit <- function(x, method = "mle",
  na.rm = F, save.x = F, print.short.p=F,
  print.long.p=T, plot.p=T, plot.grid.p=F) {

  # Trick to pull off arguments specific to
  # gaussfit() and construct call to gaussfit().
  # This is one way to get gaussfit() to return
  # an object with a proper call.

  fun.call <- match.call()
  fun.call[[1]] <- as.name("gaussfit")
  fun.args <- is.element(arg.names(fun.call),
    arg.names(gaussfit))
  fun.call <- fun.call[c(T, fun.args)]
  fit <- eval(fun.call)

  # Specifying summaries and plots.
  if(print.short.p || print.long.p)
    cat("\n\+*** Gaussian Fit ***\n\n")
  if(print.short.p)
    print(fit)
```

```
if (print.long.p)
  summary(fi t)

if (plot.p)
  plot(fi t, grid=plot.grid.p)

invisible(fi t)
}
```

## The GUI Function

We will create menus and toolbars to invoke the `menuGaussfi t` function in the same manner as we did for `gaussfi t`. This time we will use more advanced techniques to specify a complicated dialog.

- Properties of `Type="Group"` and `Type="Page"` may be used to group controls and place them on tabbed dialog pages.
- Required arguments may be flagged as `IsRequired=T`. If these arguments are not present when OK or Apply is pressed a warning message box will appear stating that the argument is required, and the dialog will not be closed.
- Rather than defining new properties for all controls we will reuse some of the properties used by the built-in statistics dialog: `SPropPrintShort` and `SPropPrintLong`.

Note that a specific property can be used only once within a dialog, but properties may be shared between dialogs. Sharing properties enforces consistency between dialogs and makes it easy to change a prompt which appears in multiple dialogs. For more details on properties see Chapter 17, *Extending the User Interface*.

- The following function will create a menu item, toolbar, and dialog for `menuGaussfi t`.

```
create.gui.menuGaussfi t<-function(){

# Create new menu item
guiCreate( "MenuItem",
  Name = "SPI usMenuBar$MyMethods",
  Type = "Menu",
```



```
MenuIteMText = "&My Methods",
Index = 11,
OverWrite=F)

gui Create("MenuIteM",
    Name="SPI usMenuBar$MyMethods$MenuGaussfi t",
    Type="MenuIteM",
    Acti on="Functi on",
    Command="menuGaussfi t",
    MenuIteMText="&Detai led Gaussi an Fi t")

# Create new tool bar button
gui Create("Tool bar", Name = "MyMethods")

gui Create("Tool barButton",
    Name = "MyMethods$menuGaussfi t",
    Type = "Button",
    Acti on = "Functi on",
    Command="menuGaussfi t",
    Ti pText = "Detai led Gaussi an Fi t", ShowDi al ogOnRun=T)

## Create di al og
# Indi vi dual properti es

gui Create("Property",
    Name="Gaussfi tX",
    Di al ogPrompt="Sampl e Data",
    Di al ogControl ="Stri ng", I sRequi red=T)

gui Create("Property", Name="Gaussfi tMethod",
    Di al ogPrompt="Fi tti ng Method",
    Di al ogControl ="Li st Box",
    Opti onLi st=c("ml e", "robust"), UseQuotes=T,
    Defaul tVal ue="ml e", I sRequi red=T)

gui Create("Property", Name="Gaussfi tSaveX",
    Di al ogPrompt="Save Data wi th Model ",
    Di al ogControl ="Check Box", Defaul tVal ue=F)
```

```
gui Create("Property", Name="Gaussfi tNa",
          DialogPrompt="Remove Mi ssi ng Val ues",
          DialogControl ="Check Box", Defaul tVal ue=T)

gui Create("Property", Name="Gaussfi tSaveAs",
          DialogPrompt="Save Model As",
          DialogControl ="Stri ng",
          Defaul tVal ue="l ast. gaussfi t")

gui Create("Property", Name="Gaussfi tPl ot",
          DialogPrompt="Show Di agnosti c Pl ots",
          DialogControl ="Check Box", Defaul tVal ue=T)

gui Create("Property", Name="Gaussfi tPl otGri d",
          DialogPrompt="Di spl ay Pl ots As Gri d",
          DialogControl ="Check Box", Defaul tVal ue=F)

# Groups
gui Create("Property", Name="Gaussfi tPl otGroup",
          Type="Group", Di al ogPrompt="Pl ots",
          PropertyLi st="Gaussfi tPl ot,
          Gaussfi tPl otGri d")

gui Create("Property", Name="Gaussfi tSaveGroup",
          Type="Group",
          DialogPrompt="Save Model Obj ect",
          PropertyLi st="Gaussfi tSaveAs,
          Gaussfi tSaveX")

gui Create("Property",
          Name="Gaussfi tResul tsGroup", Type="Group",
          DialogPrompt="Pri nted Resul ts",
          PropertyLi st="SPropPri ntShort,
          SPropPri ntLong")

# Pages
gui Create("Property",
          Name="Gaussfi tModel Page", Type="Page",
          DialogPrompt="Model ",
          PropertyLi st="Gaussfi tX, Gaussfi tMethod,
          Gaussfi tNa, Gaussfi tSaveGroup")
```

```

gui Create("Property",
    Name="GaussfitResul tsPage", Type="Page",
    Di al ogPrompt="Resul ts",
    PropertyLi st="Gaussfi tResul tsGroup,
    Gaussfi tPl otGroup")

# Functi on I nfo
gui Create( "Functi onI nfo",
    Name = "menuGaussfi t",
    Functi on = "menuGaussfi t",
    Di al ogHeader =
        "Fi t Gaussi an Densi ty wi th Detai led Resul ts",
    StatusString = "Fi ts Gaussi an densi ty to a set of data
and di spl ays resul ts.",
    PropertyLi st =
        "Gaussfi tModel Page, Gaussfi tResul tsPage",
    ArgumentLi st = "#0 = Gaussfi tSaveAs,
    #1 = Gaussfi tX, #2= Gaussfi tMethod,
    #3 = Gaussfi tNa, #4 = Gaussfi tSaveX,
    #5= SPropPri ntShort, #6= SPropPri ntLong,
    #7= Gaussfi tPl ot, #8= Gaussfi tPl otGrid ")
i nvi si bl e()
}

```

## Customizing the Context Menu

At the command line the user interacts with model objects by fitting the model and then investigating it using functions such as `print`, `summary`, and `plot`. In the GUI we combine the print, summary, and plot steps as options performed at the same time as fitting, based on selections in the dialog. This is convenient for obtaining detailed information when first fitting a model.

When a model has already been fit it is inefficient to refit the model just to get a plot or summary. The more efficient approach is to select the model object in the Object Explorer and right-click to get a selection of actions to perform on the object. This right-click menu is called the context menu.

Actions specific to the particular object type may be added to the context menu by creating some interface objects:

- A `ClassInfo` object specifies which `MenuItem` object lists the actions to add to the context menu. Right-clicking will display a context menu with basic actions such as Cut, Copy, and Paste plus the actions specified in the menu item.

- `MenuItem` objects specify the contents of the context menu.
- `FunctionInfo` objects for the functions to be called describe the arguments and properties needed to construct a dialog for each function.

The `MenuItem` argument `ShowDialogOnRun=T` specifies that a dialog should be launched when the menu item is selected. If this value is set to `F` the specified function will be run without launching a dialog.

In this example we will use two built-in properties in our `FunctionInfo` object. The property `SPropInvisibleReturnObject` specifies that we do not want to save the result, and don't want a control for the name under which to save the result. The property `SPropCurrentObject` contains the name of the selected object in a non-editable string field.

The following function will create a context menu for a `gaussfit` object with actions for each menu item. Note that the `ClassInfo` object specifies the menu item to use, with the menu then using the same menu and dialog tools discussed previously.

```
create.contextmenu.gaussfit<-function(){

# ClassInfo Object
guiCreate("ClassInfo", Name = "gaussfit",
          ContextMenu = "gaussfit")

# FunctionInfo objects for method functions

guiCreate("FunctionInfo", Name="print.gaussfit",
          Function="print.gaussfit",
          PropertyList = "SPropInvisibleReturnObject,
                          SPropCurrentObject",
          ArgumentList =
            "#0=SPropInvisibleReturnObject,
            #1=SPropCurrentObject")

guiCreate("FunctionInfo",
          Name="summary.gaussfit",
          Function="summary.gaussfit",
          PropertyList =
            "SPropInvisibleReturnObject,
            SPropCurrentObject",
          ArgumentList =
```

```

        "#0=SPropl nvi si bl eReturnObj ect,
        #1=SPropCurrentObj ect")

gui Create("Property", Name="Gaussfi tPl otGri d",
        Dial ogPrompt="Di spl ay Pl ots As Gri d",
        Dial ogControl ="Check Box", Defaul tVal ue=F)

gui Create("Functi onI nfo", Name="pl ot. gaussfi t",
        Functi on="pl ot. gaussfi t",
        PropertyLi st =
        "SPropl nvi si bl eReturnObj ect,
        SPropCurrentObj ect,
        Gaussfi tPl otGri d",
        ArgumentLi st =
        "#0=SPropl nvi si bl eReturnObj ect,
        #1=SPropCurrentObj ect,
        #2=Gaussfi tPl otGri d")

# Context Menu

gui Create("Menul tem", Name="gaussfi t",
        Type = "Menu", DocumentType="gaussfi t")

gui Create("Menul tem",
        Name="gaussfi t$pri nt. gaussfi t",
        Type="Menul tem", DocumentType="gaussfi t",
        Acti on=" Functi on",
        Command="pri nt. gaussfi t",
        ShowDi al ogOnRun = T,
        Menul temText="Pri nt")

gui Create("Menul tem",
        Name="gaussfi t$summary. gaussfi t",
        Type="Menul tem", DocumentType="gaussfi t",
        Acti on=" Functi on",
        Command="summary. gaussfi t",
        ShowDi al ogOnRun=T,
        Menul temText="Summary")

```

```
gui Create("Menu Item",  
    Name="gaussfit$plot.gaussfit",  
    Type="Menu Item", DocumentType="gaussfit",  
    Action="Function", Command="plot.gaussfit",  
    ShowDialogOnRun=T,  
    MenuItemText="Plot")  
invisible()  
}
```

# WRITING HELP FILES

Help for built-in objects is provided using WinHelp. This is a compiled help system and as such it is not currently possible to insert user help files within the built-in help system. Users may provide help files for their functions as text files stored in the `_Help` subdirectory within their `_Data` directory.

Each help file must be given the same name that is used for the object in `_Data`. The name space for S-PLUS objects is not the same as the name space for Windows files. (Windows file names are case sensitive and Windows 3.1x file names must meet the 8.3 rule—8 characters, a dot and 3 characters). Because of this, S-PLUS objects are mapped to names such as `__1`, `__2`, `__3`, etc. in `_Data`. These same mangled names must be used for the corresponding help file in `_Help`. The `prompt` function, described below, takes care of this naming for you. If you have already created a help file for an object you can use the function `true.file.name` to find the objects mangled file name and then copy the help file into `_Help` with that name.

Highly experienced users may use a WinHelp compiler to create their own set of WinHelp based help files. Doing so is beyond the scope of this manual.

## Creating the Help File

The `prompt` function will create a template help file for a specific function:

```
> prompt(gaussfi t)
```

The template help file will be created in `_Data\_Help`.

This template help file can be edited to provide detailed help.

The template help file for `gaussfi t` has the following entries:

```
Repl ace wi th functi on to do??? header
E. g. "Pri nt a data obj ect"
```

DESCRIPTION:

```
Repl ace thi s li ne wi th bri ef descri pti on
```

USAGE:

```
gaussfi t(x, method="ml e", na.rm=F, save.x=F)
```

REQUIRED ARGUMENTS:

**OPTIONAL ARGUMENTS:**

Move the above line to just above the first optional argument

Describe x here

method

Describe method here

na.rm

Describe na.rm here

save.x

Describe save.x here

**VALUE:**

Describe the value returned

**SIDE EFFECTS:**

Describe any side effects if they exist

**DETAILS:**

Explain details here.

**REFERENCES:**

Put references here, make other sections like NOTE and

**WARNING as needed****SEE ALSO:**

Put functions to SEE ALSO here

**EXAMPLES:**

# The function is currently defined as

```
function(x, method = "mle", na.rm = F, save.x = F)
```

```
[...]
```

Warning: Moving help files from one directory to another can be difficult because of the name mangling. One needs to create the S-PLUS object in the new directory (new\_Data), use `true.file.name` to find its new mangled name and then copy the help file to the new `_Data\_Help` with the new mangled name.



# DISTRIBUTING FUNCTIONS

Once we have written a set of functions we may want to distribute them to others. The simple way to distribute functions is to store them in a text file and send the text file to others. This is the approach we use to provide the example functions in this section.

A more sophisticated approach is to create a library containing the functions and related files such as help files and toolbar button images.

This section describes how to create menu and dialog objects on the fly with functions. It is generally quicker to create the interface objects and store them in \*.DFT files, which are then loaded when the module is attached. This approach is described in the preceding chapter.

## Using Text Files

The simple way to distribute functions is to store them in a text file and send the text file to others. The `dump` and `source` functions may be used to store and recover objects.

- The `dump` function places objects in a file.

```
> dump(c("gaussfi t", "pri nt. gaussfi t"), "myfi le")
[1] "myfi le"
```

- The `source` function reads objects from a file.

```
> source("myfi le")
```

If we have help files we can also include the help files as separate text files. We can use the function `true.file.name` to determine the name under which to save each help file.

```
> true.file.name("pri nt. gaussfi t")
[1] "__27"
```

So in this example the user would save the help file for `pri nt. gaussfi t` in the `_Help` directory as `__27`.

If we have bitmaps representing toolbar button images we can include the \*.bmp files found in `_Prefs`. These files should be placed in the `_Prefs` directory.

We can package up the objects, help files, and components such as toolbar button images in a more polished manner by creating libraries.

## Using Libraries

We can create libraries to distribute functions and related files in an organized manner. A library is a directory containing objects, help files, and related files such as compiled C or Fortran code. Attaching a library makes the objects in that library available to the user by adding the relevant `_Data` directory to the search list.

The library may contain a `a.First.lib` function indicating actions to perform upon attaching the library. Typically this might include dynamically loading C or Fortran code and creating GUI objects such as menus, toolbars, and dialogs. A `.Last.lib` function may specify actions to perform upon unloading the library or exiting.

Any directory containing an `_Data` directory may be used as a library. If the directory is a subdirectory of the library directory under the S-PLUS program directory then the library may be made available by using:

```
> library(libraryname)
```

For example, `library(cluster)` will attach the built-in cluster library. It is not necessary for the library to reside in the library directory under the S-PLUS program directory. If it resides elsewhere the path to the directory containing your library directory must be specified with the `lib.loc` argument to `library`. For example, if your library is called *mylib*, and it exists as a directory under `c:\stat` then the library can be made available with

```
> library(mylib, lib.loc = "c:\\stat")
```

If the programmer shares a file system with the users to which the programmer intends to distribute routines, the library can reside anywhere on the file system and the end users need only be told to use the `library` command with the proper `lib.loc` argument.

To distribute the library to users who do not share a file system we can create a batch script to install the library properly. The most polished way to distribute a library is to create a self-extracting archive which has knowledge of where to install the library. Details of creating batch scripts and self-extracting executables depend upon the tools available to the programmer and are beyond the scope of this manual.

## Steps in Creating a Library

The directory structure for the library may be created in Windows Explorer or from within using system calls. The steps involved are:

- Create a main directory for the library in the library subdirectory of the program directory.
- Create `_Data` and `_Prefs` directories in this main directory.
- Source the objects into the `_Data`.
- Copy interface files such as toolbar bitmaps (\*.BMP) into `_Prefs`.
- If help files are present, create a `_Help` directory under `_Data` and place the help files in this directory.
- If desired, write `First.lib` and `Last.lib` functions to specify actions to perform upon loading and unloading the library. For example, we might want to create GUI objects upon loading the library.

## Creating Directories

First we will create directories using the function `mkdir`. We will use `getenv("SHOME")` to find the location of the S-PLUS program directory.

```
> mkdir(paste(getenv("SHOME"),
+ "\\library\\gaussfit", sep=""))
> mkdir(paste(getenv("SHOME"),
+ "\\library\\gaussfit\\_Data", sep=""))
> mkdir(paste(getenv("SHOME"),
+ "\\library\\gaussfit\\_Data\\_Prefs", sep=""))
> mkdir(paste(getenv("SHOME"),
+ "\\library\\gaussfit\\_Data\\_Help", sep=""))
```

## Putting Functions in the Library

Next we will use `library` to attach the new `gaussfit` library. We will use `first=T` to specify that it should be attached as the second directory in the search list.

```
> library(gaussfit, first=T)
```

We can use `assign` to copy our `gaussfit` related objects into this directory. We will make a vector of names of the functions we want to copy and then loop over this vector.

```
> gaussfit.funcs<-c("gaussfit1", "gaussfit2",
+ "gaussfit", "print.gaussfit",
+ "summary.gaussfit", "plot.gaussfit",
+ "create.menu.gaussfit",
```

```
+ "create. toolbar.gaussfit",  
+ "remove.menu.gaussfit",  
+ "remove.toolbar.gaussfit",  
+ "create.dialog.gaussfit", "menuGaussfit",  
+ "create.gui.menuGaussfit",  
+ "create.contextmenu.gaussfit")  
  
> for (i in 1:length(gaussfit.funcs)){  
+   assign(gaussfit.funcs[i],  
+   get(gaussfit.funcs[i]), where=2)  
+ }
```

Note that this will produce warnings indicating that we are creating objects in the second directory on the search list with the same names as objects in other directories on the search list. These warnings are expected and in this case indicate that we will want to remove the objects from the first directory after making the copies to avoid duplicate names.

An alternate approach is to keep a text file `gaussfit.ssc` containing our functions. We can then attach the library directory as our working directory and use `source` to create the functions in this directory.

```
> attach(paste(getenv("SHOME"),  
+   "\\library\\gaussfit\\_Data", sep=""), pos=1)  
> source("gaussfit.ssc")
```

### Copying External GUI Files

If we have edited the bitmap for a toolbar button we would want to include this new bitmap in our library. The steps involved are:

- Look in the `_Prefs` directory under the user files in the directory. This will be at the same level as the `_Data` used as the working database.
- Copy any (\*.BMP) files to the `_Prefs` in the `gaussfit` library directory.
- Modify the function calls in `create.toolbar.gaussfit` or `create.gui.menuGaussfit` which create the toolbar buttons. To specify the location of the toolbar button image, specify an `ImageFileName` argument when creating the toolbar button.

For example, the following command would specify that the `Gaussfit.BMP` file contains the toolbar button image for the “Detailed Gaussian Fit” toolbar button:

```
gui Create("Tool barButton",
  Name = "MyMethods$menuGaussfi t",
  Type = "Button",
  Action = "Function",
  Command="menuGaussfi t",
  ShowDialogOnRun=T, TipText = "Detailed Gaussi an Fi t",
  ImageFileName= paste(getenv("SHOME"),
    "\\library\\gaussfi t\\_Prefs\\Gaussfi t. BMP",
    sep=" ") )
```

## Copying Help Files

If we have help files which we want to include in the library we need to copy the text files into `library\gaussfit\Data_Help`.

It is important that we give each help file the same name as the object to which it is related. The name is the mechanism used by the `help` function to determine what help file to display. The `true.file.name` function will give the name under which each function is stored on disk. This is the same name to use for the help file.

Specify the name of the function and the location in the search list of the library to get the file name under which to save the help file.

```
> true.file.name("summary.gaussfi t", where=2)
[1] "__3"
```

So we would save the help file for `summary.gaussfi t` as `__3` in `library\gaussfit\Data_Help`.

## Start-up and Exit Actions

Finally, we may want to automatically create menus and toolbars upon attaching the library, and remove these changes upon detaching the library.

The `.First.lib` function is run when a library is attached. This may be used to modify the GUI when a library is attached.

```
.First.lib<-function(library, section){
  create.gui.menuGaussfi t()
  create.contextmenu.gaussfi t()
}
```

The `.Last.lib` function is run when a library is detached or the program exits normally. This may be used to remove GUI changes.

```
.Last.lib<-function(library, section, .data, where){
  remove.menu.gaussfi t()
  remove.tool bar.gaussfi t()
}
```

We could extend these functions to also remove the `Property`, `FunctionInfo`, `ClassInfo`, and context menu related objects which we create.

## **Distributing the Library**

Once we have created the library we will want to package it up as a compressed archive or a self-extracting executable. One approach is to use a utility such as WinZip or pkzip to compress up the gaussfit library directory, and include a `readme.txt` file indicating how to unzip this as a subdirectory of the library directory. A more sophisticated approach is to use these tools to produce a self-extracting archive which unpacks to the proper location. Creating these archives is beyond the scope of this manual.

<b>Introduction</b>	<b>496</b>
<b>Using S-PLUS as an Automation Server</b>	<b>497</b>
A Simple Example	497
Exposing Objects to Other Applications	503
Exploring Properties and Methods	504
Programming With Object Methods	505
Programming With Object Properties	510
Passing Data to Functions	511
Automating Embedded S-PLUS Graphs	514
<b>Using S-PLUS as an Automation Client</b>	<b>515</b>
A Simple Example	515
High-Level Automation Functions	521
Reference Counting Issues	522
<b>Automation Examples</b>	<b>526</b>
Server Examples	526
Client Examples	528

## INTRODUCTION

*Automation*, formerly known as OLE Automation, makes it possible for one application, known as an Automation client, to directly access the objects and functionality of another application, the Automation server. The server application *exposes* its functionality through a type library of objects, properties, and methods, which can then be manipulated programmatically by a client application. Automation thus provides a handy way for programs and applications to share their functionality.

In this chapter, we explore the procedures for using S-PLUS as both an Automation server and an Automation client. We begin by showing you how to expose S-PLUS objects and functions and then how to use them as building blocks in the program code of client applications. Later in the chapter, we examine the functions provided in the S-PLUS programming language for accessing and manipulating the Automation objects exposed by server applications.



## USING S-PLUS AS AN AUTOMATION SERVER

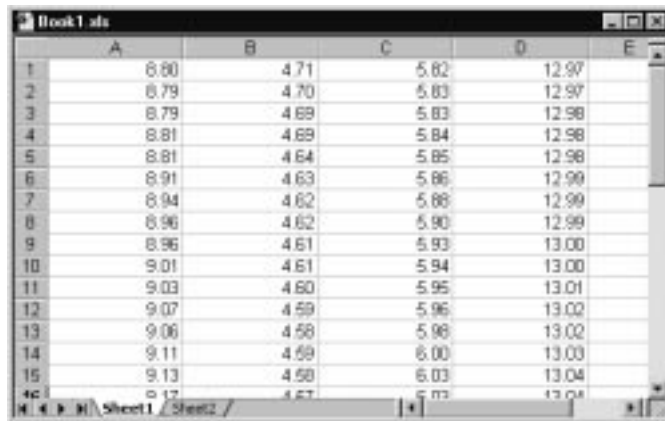
Programs and applications supporting Automation client features can access all the functionality of S-PLUS through the S-PLUS type library. The type library is a disk file of S-PLUS object and function information, along with properties and methods, that also provides help and syntax information. When you install S-PLUS on your system, you can elect to have setup register S-PLUS objects and functions for use in Automation and build the type library file.

Before explaining in detail how to program with S-PLUS Automation objects, let's first take a look at a simple example.

### A Simple Example

To demonstrate how to use S-PLUS as an Automation server, we present a simple example using Automation to pass data from an Excel worksheet to S-PLUS, which then performs a covariance estimation on the data and returns the resulting covariance matrix to Excel.

Consider the sample data shown in Figure 13.1. **Sheet1** of the Excel workbook **Book1.xls** contains data in 4 columns of 39 rows (not all rows are shown in Figure 13.1).



	A	B	C	D
1	8.80	4.71	5.82	12.97
2	8.79	4.70	5.83	12.97
3	8.79	4.69	5.83	12.98
4	8.81	4.69	5.84	12.98
5	8.81	4.64	5.85	12.98
6	8.91	4.63	5.86	12.99
7	8.94	4.62	5.88	12.99
8	8.96	4.62	5.90	12.99
9	8.96	4.61	5.93	13.00
10	9.01	4.61	5.94	13.00
11	9.03	4.60	5.95	13.01
12	9.07	4.59	5.96	13.02
13	9.06	4.58	5.96	13.02
14	9.11	4.59	6.00	13.03
15	9.13	4.58	6.03	13.04

Figure 13.1: Sample data in **Book1.xls**.

**Note**

The sample data in **Book1.xls** are taken from the **freeny.x** matrix included with S-PLUS. You can recreate this example by exporting the data into a new Excel worksheet and following the steps outlined below.

By writing a program in Visual Basic for Applications (Office 97), we can automate a conversation between Excel and S-PLUS to perform our task. The complete code for one such program is shown in Figure 13.2. (This example program is also included as **Book1.xls** in the **samples/oleauto/vba** folder in the S-PLUS program folder.)

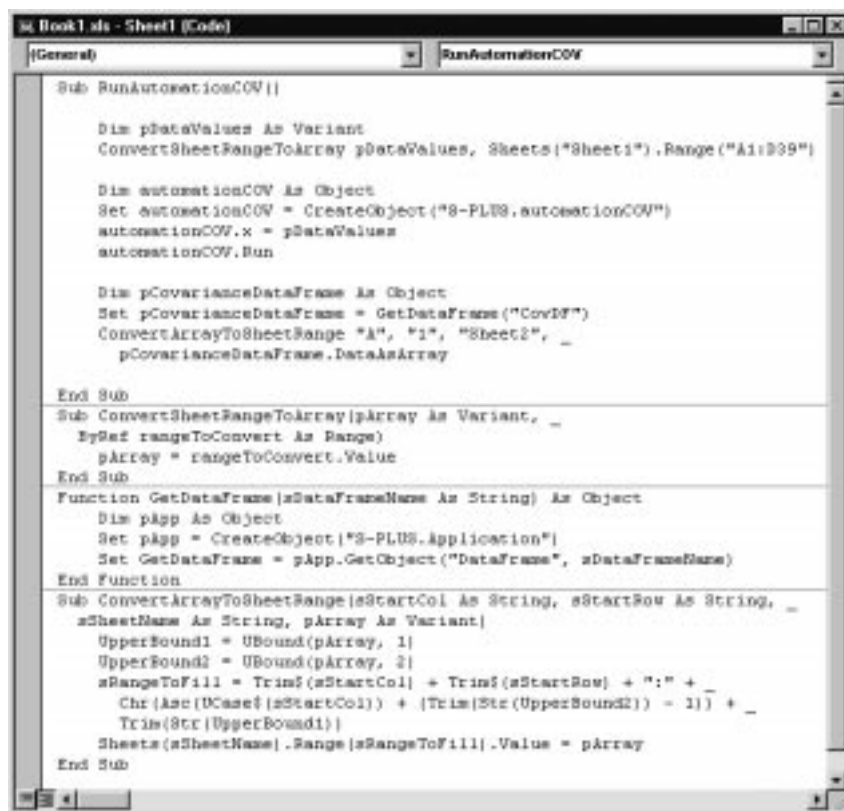
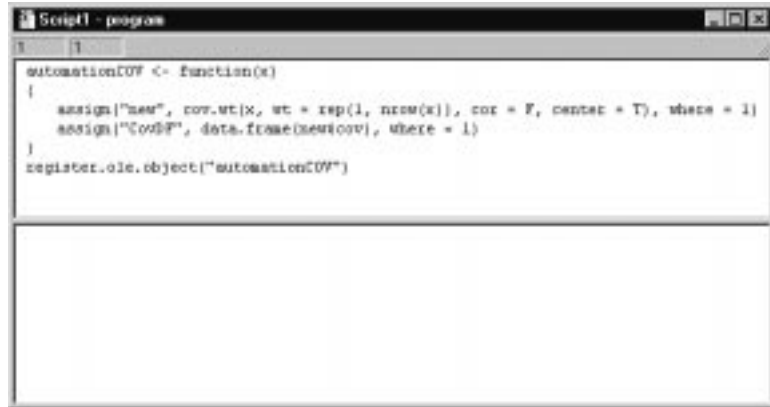


Figure 13.2: Complete code for our VBA program.


Before we examine the VBA code in detail, let's first define a new S-PLUS function and register it for use as an Automation object.

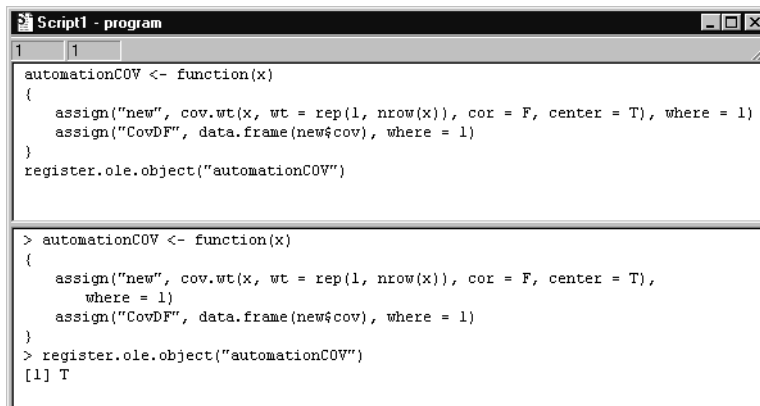
1. Open a new **Script** window in S-PLUS and enter the code shown in Figure 13.3.



*Figure 13.3: Defining and registering a new S-PLUS function.*

Our new function, `automationCOV()`, calls the built-in S-PLUS function `cov.wt()` to perform a weighted covariance estimation on the data received from Excel and extracts the `cov` component of the resulting list for return to Excel. After defining the new function, we use the `register.ole.object()` command to make it available to Excel.

- Click the **Run** button  on the **Script** window toolbar. As shown in Figure 13.4, `automationCOV()` is now defined and registered as an Automation object.



```

Script1 - program
1 1
automationCOV <- function(x)
{
  assign("new", cov.wt(x, wt = rep(1, nrow(x)), cor = F, center = T), where = 1)
  assign("CovDF", data.frame(new$cov), where = 1)
}
register.ole.object("automationCOV")

> automationCOV <- function(x)
{
  assign("new", cov.wt(x, wt = rep(1, nrow(x)), cor = F, center = T),
    where = 1)
  assign("CovDF", data.frame(new$cov), where = 1)
}
> register.ole.object("automationCOV")
[1] T

```

Figure 13.4: Running the script.

- Close the **Script** window. At the prompt to save the script in a file, click **No**.

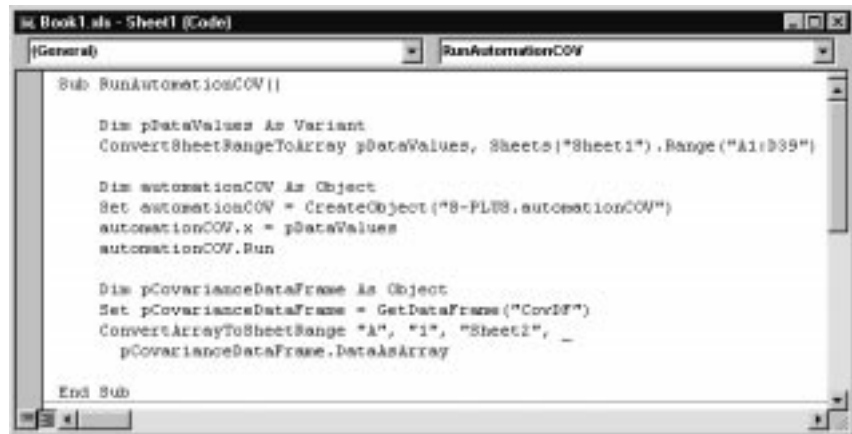
#### Note

If you prefer, you can define and register `automationCOV` directly from the **Commands** window.

Now that we have defined and registered our S-PLUS function, the next step is to write the module in Visual Basic.

- With **Book1.xls** open in Excel, choose **Tools ► Macro ► Visual Basic Editor** from the main menu.
- If the **Project Explorer** window is not open, open it by choosing **View ► Project Explorer**.
- Double-click **Sheet1** under the **Book1.xls** project to open the code window for **Sheet1**.

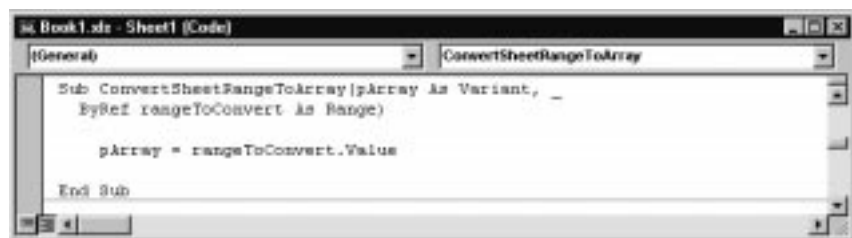
7. Enter the code for the first procedure in the module, `RunAutomationCOV`, as shown in Figure 13.5.



*Figure 13.5: The RunAutomationCOV procedure.*

`RunAutomationCOV` represents the central task we want to automate. In the first section of code, we declare a variable, `pDataValues`, in which to store the data on **Sheet1**. A call to the next procedure we will write, `ConvertSheetRangeToArray`, converts the range data into an array.

8. Enter the code for `ConvertSheetRangeToArray`, as shown in Figure 13.6.



*Figure 13.6: The ConvertSheetRangeToArray procedure.*

In the next section of code in `RunAutomationCOV` (see Figure 13.5), we declare a variable to capture our `automationCOV` function, pass the Excel data as a parameter to the function, and then run the function.

In the final section of code in `RunAutomationCOV` (see Figure 13.5), we declare a variable, `pCovarianceDataFrame`, in which to store our results and call the `GetDataFrame` function, the next procedure we will write, to return the results to Excel.

9. Enter the code for `GetDataFrame`, as shown in Figure 13.7.



Figure 13.7: The `GetDataFrame` procedure.

The last procedure we will write, `ConvertArrayToSheetRange`, is called in the last line of `RunAutomationCOV` and returns the covariance matrix to **Sheet2** in **Book1.xls**.

10. Enter the code for `ConvertArrayToSheetRange`, as shown in Figure 13.8.

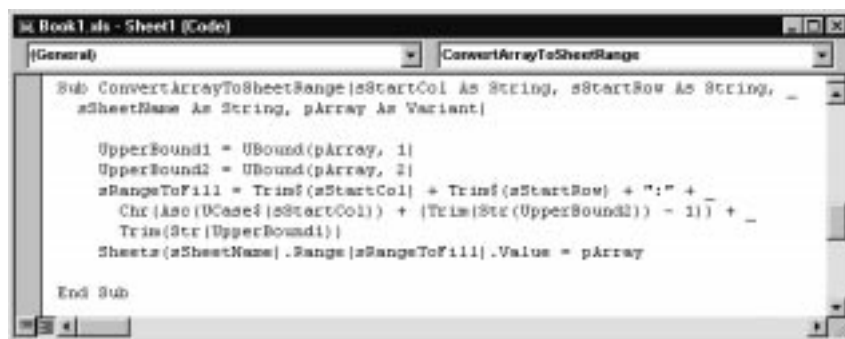



Figure 13.8: The `ConvertArrayToSheetRange` procedure.

With all the coding complete, it's time to run the module.

11. Click the **Run Sub/User Form** button  on the **Visual Basic** toolbar. The results are shown in Figure 13.9.

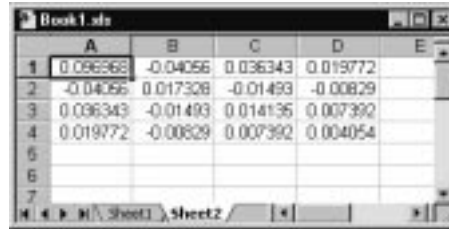


Figure 13.9: The covariance matrix returned to Excel.

## Exposing Objects to Other Applications

There are a number of ways in which S-PLUS Automation objects can be exposed to, or hidden from, client applications. One way is during the installation or uninstallation of S-PLUS. When installing S-PLUS, you can allow setup to register all S-PLUS Automation objects and build the type library. When uninstalling S-PLUS, all information concerning Automation objects is automatically removed from the registry, and the type library file is removed from disk.

In addition, S-PLUS itself provides the six functions listed in Table 13.1 that you can use at any time to register or unregister Automation objects and refresh or remove the type library.

Table 13.1: S-PLUS functions for exposing and hiding Automation objects.

Function	Description
<code>register.all.objects()</code>	This function registers all S-PLUS objects with the system registry and builds or rebuilds the type library file. Returns T for success or F for failure.
<code>unregister.all.objects()</code>	This function unregisters all S-PLUS objects and removes the type library file. Returns T for success or F for failure.
<code>register.object()</code>	This function registers one or more S-PLUS objects with the system registry. Returns T for success or F for failure.
<code>unregister.object()</code>	This function unregisters one or more S-PLUS objects. Returns T for success or F for failure.

*Table 13.1: S-PLUS functions for exposing and hiding Automation objects. (Continued)*

Function	Description
<code>rebuild.type.library()</code>	This function removes and then rebuilds the type library file with all currently registered S-PLUS objects. Returns <code>T</code> for success or <code>F</code> for failure.
<code>destroy.type.library()</code>	This function removes the type library file from disk. Note that executing this command does not unregister any objects but simply removes the type library. Returns <code>T</code> for success or <code>F</code> for failure.

Caution
Remember that unregistering your S-PLUS objects means that no Automation client will be able to access those objects and could potentially cause a client program to fail.

### Exposing S-PLUS Functions

Unlike other S-PLUS objects, functions are not automatically registered during setup. To expose built-in functions, or your own S-PLUS functions, for use in Automation clients, first define the function, if necessary, in S-PLUS and then call:

```
register.ole.object(names)
```

where *names* is a character vector of the function names you want to expose. This function returns `T` for successful registration or `F` otherwise.

To unregister your function, making it no longer available to Automation clients, call:

```
unregister.ole.object(names)
```

with the desired *names* argument.

### Exploring Properties and Methods

To start a conversation with S-PLUS from a client application, you must first create or get an S-PLUS object. Once an instance of the object has been created, it can be manipulated through its properties and methods.

While the type library is useful for determining which methods and properties are available for particular S-PLUS objects, it does not reveal the object hierarchy, how objects are related, or how to use object properties.



For more detailed programming information, S-PLUS provides a set of helpful HTML files that document the complete object hierarchy for all objects currently registered. The files also list, for each Automation object, not only its properties and methods, but also its possible containment paths, container objects, and child objects. Look in the **html.lib** folder within the S-PLUS program folder to find these HTML files.

You can also update this set of HTML files to reflect the S-PLUS functions you write and expose using `register.o.le.object()`. Simply call the following S-PLUS function:

```
rebuild.html.library(html.path, method.language.type =
"basic")
```

The first argument, `html.path`, specifies the path, including drive letter, where you want save the files. The optional second argument, `method.language.type`, can be used to indicate the language in which to write the example syntax for methods. The default value for this argument is "basic", but "c" or "c++" can also be used.

The function returns `T` if successful or `F` if not. If successful, `rebuild.html.library()` creates the `html.path` specified if it does not already exist and writes an **index.html** file showing the complete object hierarchy for the S-PLUS object system.

## Programming With Object Methods

S-PLUS Automation objects are owned by S-PLUS but can be created and manipulated remotely through their properties and methods. For example, to start S-PLUS from a client application, simply call the `CreateObject` method on the S-PLUS application object. In Automation terminology, S-PLUS is said to be *instantiated*.

The following example program in Visual Basic shows you how to create an instance of an S-PLUS graph sheet and then add an arrow to it using the `CreateObject` method:

```
Function CreateArrowInSPIus () As Integer
    Dim mySPIusGraphSheet As Object
    Dim mySPIusArrow As Object

    'Instantiate the graph sheet object
    Set mySPIusGraphSheet = CreateObject("S-PLUS.GraphSheet")

    'Add an arrow to this graph sheet
    Set mySPIusArrow = mySPIusGraphSheet.CreateObject("S-PLUS.Arrow")
End Function
```

Notice the form in which `CreateObject` is used in its second occurrence. Here, `CreateObject` is called as a method of the graph sheet object and so creates the arrow as a child object of the graph sheet container. Had we instead used `CreateObject("S-PLUS.Arrow")`, a new graph sheet would have been created with the arrow added to that one.

Another method common to most S-PLUS Automation objects is the `GetObject` function. You can use `GetObject` to get a reference to an object that already exists in S-PLUS. In the next section, we list the common methods available for most Automation objects.

### Common Object Methods

Except for function objects, all S-PLUS Automation objects have a set of common methods, listed in Table 13.2. Once an object has been created using `CreateObject` or `GetObject`, the other methods can be called. Consult the HTML files discussed on page 504 for detailed information concerning parameters for these methods.

*Table 13.2: Common object methods.*

Method	Description
<code>CreateObject</code>	Creates an object or child object of a particular type.
<code>RemoveObject</code>	Removes a child object from this container object.
<code>GetObject</code>	Gets an object or child object of the type specified, identified by an object path name.
<code>BeginTransaction</code>	Starts remembering property set calls so that all changes can be applied at once when <code>CommitTransaction</code> is called.
<code>CancelTransaction</code>	Cancels remembering property set calls made after the last call to <code>BeginTransaction</code> .
<code>CommitTransaction</code>	Commits all property changes made since the last call to <code>BeginTransaction</code> .
<code>ShowDialog</code>	Displays a modal property dialog for this object that allows you to change any or all of its properties, pausing the client program until <b>OK</b> or <b>Cancel</b> is pressed in the dialog.
<code>ShowDialogInParent</code>	Displays a modal property dialog for this object in the client program, pausing the program while the dialog is displayed. Returns TRUE if successful or FALSE if not.
<code>ShowDialogInParentModelless</code>	Displays a modeless property dialog for the object in the client program, which continues executing while the dialog is displayed. Returns TRUE if successful or FALSE if not.

*Table 13.2: Common object methods. (Continued)*

Method	Description
<a href="#">Objects</a>	Called with "Contai nees" parameter, returns a comma-delimited string listing all allowable child objects for this object. Called with "Contai ners" parameter, returns a list of objects that could contain this object.
<a href="#">Obj ectContai nees</a>	Returns an array of objects contained by this object. Returns an array of containee objects of the class name specified or an empty array if none are found.
<a href="#">Obj ectContai ner</a>	Returns the object that is the container of this object.
<a href="#">Cl assName</a>	Returns a string representing this object's class name.
<a href="#">PathName</a>	Returns a string representing this object's path name in S-PLUS.
<a href="#">Properti es</a>	Returns a comma-delimited string listing all the properties for this object.
<a href="#">GetPropertyI nformati on</a>	Returns a string array of information about the specified property for this object.
<a href="#">GetPropertyAl l owedVal ues</a>	Returns a string array of allowable values or allowable range of values for the specified property for this object.
<a href="#">Methods</a>	Returns a comma-delimited string listing all allowable methods for this object.
<a href="#">Obj ectsLi st</a>	Depending on the parameter specified, returns a string array of class names for objects that can be valid children or parents of this object.
<a href="#">MethodsLi st</a>	Returns a string array of method names that can be called on this object.
<a href="#">Properti esLi st</a>	Returns a string array of property names that can be used with this object to set or get values.
<a href="#">GetMethodHel pStri ng</a>	Returns a string containing a description of the method specified for this object.
<a href="#">GetMethodArgumentNames</a>	Returns a string array of argument names that can be used with the specified method for this object.
<a href="#">GetMethodArgumentTypes</a>	Returns a string array of argument data types that must be passed as parameters to the specified method for this object. Data types returned depend on the language type specified.

*Table 13.2: Common object methods. (Continued)*

Method	Description
<code>GetObjectRectangle</code>	Returns the rectangular coordinates (client or screen, depending on the input parameter) in a variant that contains this object. If unsuccessful, returns an empty variant.
<code>GetObjectPicture</code>	Returns an array of byte values in a variant representing the Windows metafile format picture of this object. If unsuccessful, returns an empty variant.
<code>SelectObject</code>	Selects this object in all views, returning <code>TRUE</code> if successful or <code>FALSE</code> if not.
<code>GetSelectedObjects</code>	Returns an array of currently selected objects that are contained in this object.
<code>GetSelectedText</code>	Returns a string containing the currently selected text in this object. If no selected text is found, returns an empty string.

#### Additional Methods for the Application Object

The S-PLUS “application” object is used to instantiate an S-PLUS session within a client application. All the common object methods can be applied to the application object, and in addition, it has a number of specific methods, listed in Table 13.3. For detailed information concerning parameters for these methods, consult the HTML files discussed on page 504.

*Table 13.3: Methods for the application object.*

Method	Description
<code>SetOptionValue</code>	Sets an option value in the program (as in the <b>Tools/Options</b> dialogs).
<code>GetOptionValue</code>	Gets the current setting for an option (as in the <b>Tools/Options</b> dialogs).
<code>ExecuteString</code>	Executes a string representing any valid S-PLUS syntax.
<code>ExecuteStringResult</code>	Returns a string representing the output from executing the string passed in. The format of the return string depends on the setting of the second parameter. If <code>TRUE</code> , the older S-PLUS 3.3 output formatting is applied. If <code>FALSE</code> , the new format is used.

*Table 13.3: Methods for the application object. (Continued)*

Method	Description
<a href="#">SetSAPI Object</a>	Sets a binary SAPI object created in the client program into S-PLUS, making it available to other S-PLUS operations. Returns TRUE if successful or FALSE if not.
<a href="#">GetSAPI Object</a>	Returns a binary SAPI object into a variant byte array given the name of the object in S-PLUS. If no object is found, returns an empty variant.

**Additional Methods for Graph Objects**

In addition to the common object methods listed in Table 13.2, Table 13.4 lists a number of methods available specifically for creating graphs and plots. Consult the HTML files discussed on page 504 for detailed information concerning parameters for these methods.

*Table 13.4: Methods for graph objects.*

Method	Description
<a href="#">CreatePlots</a>	Returns TRUE if successful or FALSE if not.
<a href="#">CreateConditionedPlots</a>	Returns TRUE if successful or FALSE if not.
<a href="#">CreateConditionedPlotsSeparateData</a>	Returns TRUE if successful or FALSE if not.
<a href="#">CreatePlotsGallery</a>	Returns TRUE if successful or FALSE if not.
<a href="#">CreateConditionedPlotsGallery</a>	Returns TRUE if successful or FALSE if not.
<a href="#">CreateConditionedPlotsSeparateDataGallery</a>	Returns TRUE if successful or FALSE if not.

**Methods for Function Objects**

Function objects differ from other S-PLUS objects in that they do not have all the same methods as other Automation objects. The methods available for functions are listed Table 13.5. For detailed information concerning parameters for these methods, consult the HTML files discussed on page 504.

*Table 13.5: Methods for function objects.*

Method	Description
<a href="#">Run</a>	Runs this function using the arguments (properties) most recently set.
<a href="#">ShowDialog</a>	Displays a dialog for this function that allows you to change any or all of the function's arguments, pausing the client program until <b>OK</b> or <b>Cancel</b> is pressed in the dialog.

Table 13.5: Methods for function objects. (Continued)

Method	Description
Properties	Returns a comma-delimited string listing all allowable arguments (parameters) for this function.
Methods	Returns a comma-delimited string listing all allowable methods for this function.
SetParameterClasses	Specifies the class of function parameters. Returns TRUE if successful or FALSE if not.
GetParameterClasses	Returns an array of strings representing the class names of the return value followed by each of the parameters of this function.

## Programming With Object Properties

You can get and set the properties of an Automation object to modify its appearance or behavior. For example, a property of the application object called `Visible` controls whether the S-PLUS main window will be visible in the client application.

When setting a series of properties for an object, you can use the `BeginTransaction` and `CommitTransaction` methods in a block to apply the changes all at once. The following example in Visual Basic illustrates how to use `BeginTransaction` and `CommitTransaction` to set color properties for an arrow on a graph sheet object:

```
Sub ChangeArrowPropertiesInSPIus()  
    Dim mySPIusGraphSheet As Object  
    Dim mySPIusArrow As Object  
  
    Set mySPIusGraphSheet = CreateObject("S-PLUS.GraphSheet")  
    Set mySPIusArrow = mySPIusGraphSheet.CreateObject("S-PLUS.Arrow")  
  
    ' Set properties for the arrow object  
    mySPIusArrow.BeginTransaction  
    mySPIusArrow.LineColor = "Green"  
    mySPIusArrow.HeadColor = "Green"  
    mySPIusArrow.CommitTransaction  
  
    sLineColor = mySPIusArrow.LineColor  
End Sub
```

Because an object updates itself whenever a property is set, using a `BeginTransaction/CommitTransaction` block can save you time and speed up your client program.

Unlike other S-PLUS objects, function objects only update when the `Run` method is called. Therefore, the `BeginTransaction` and `CommitTransaction` (and `CancelTransaction`) methods are not supported, or even needed, for function objects.

As an example, suppose the following function has been defined and registered in an S-PLUS script as follows:

```
MyFunctionInSplus <- function(a, b) {return(a)}
register.oled.object("MyFunctionInSplus")
```

The following Visual Basic example illustrates how to set a series of properties, or parameters, for the function object defined above:

```
Sub RunSplusFunction()
    Dim mySplusFunction As Object
    Set mySplusFunction = CreateObject("S-PLUS.MyFunctionInSplus")

    ' Set properties for the function object
    mySplusFunction.a = "1"
    mySplusFunction.b = "2"
    mySplusFunction.ReturnValue = "MyVariableInSplus"

    mySplusFunction.Run
End Sub
```

## Passing Data to Functions

The parameters, or arguments, of a function (and the function's return value) are properties of the function object and can be passed by value or by reference. When the data already exist in S-PLUS, passing by reference is faster because the data don't have to be copied into the client before they can be used. However, when the data to be passed are from a variable defined in the client, the data should be passed by value. Note that the return value must not be passed by reference.

By default, all parameter data are passed by value as a data frame. This default behavior could cause errors if the function expects a data type other than a data frame. You can control the data types used in a function object in one of two ways:

- By calling the `SetParameterClasses` method of the function with a comma-delimited string specifying the data types (or class names) for each of the parameters and the return value of the function.
- By setting the `ArgumentClassList` property of the `FunctionInfo` object with a comma-delimited string specifying the data types (or class names) for each of the parameters and the return value of the function.

For any parameter you want to pass by reference instead of by value, place an ampersand character (&) at the beginning of its class name in the string.

We can use the following S-PLUS script to define and register a function called `MyFunction`:

```
MyFunction <- function(aVector) {return(as.data.frame(aVector))}  
register.obj ect("MyFunction")
```

and then use `SetParameterClasses` to adjust how the data from Visual Basic are interpreted by `MyFunction`:

```
Dim pArray(1 to 3) as double  
pArray(1) = 1.0  
pArray(2) = 2.0  
pArray(3) = 3.0  
  
Dim pMyFunction as Object  
Set pMyFunction = CreateObject("S-PLUS. MyFunction")  
if (pMyFunction. SetParameterClasses("data. frame, vector")=TRUE) _  
then  
    pMyFunction. a = pArray  
    pMyFunction. Run  
  
Dim pReturnArray as Variant  
pReturnArray = pMyFunction. ReturnValue  
end if
```



The following example shows how a vector in S-PLUS can be passed by reference to `MyFunction` in S-PLUS, instead of passing data from variables in Visual Basic:

```
Dim pMyVectorInSPLUS as Object
Set pMyVectorInSPLUS = GetObject("vector", "MyVector")

Dim pMyFunction as Object
Set pMyFunction = CreateObject("S-PLUS.MyFunction")

if ( pMyFunction.SetParameterClasses (
    "data.frame, &vector" ) = TRUE ) then
    pMyFunction.a = pMyVectorInSPLUS
    pMyFunction.Run

    Dim pReturnArray as Variant
    pReturnArray = pMyFunction.ReturnValue
end if
```

Notice how the vector object `MyVector` is obtained from S-PLUS using `GetObject` and assigned directly to `pMyFunction.a` to avoid having to get the data from `MyVector` into a variant and then assign that variant data to `pMyFunction.a`. This is possible when you specify the `&` before a class name in `SetParameterClasses`.

As an alternative to using `SetParameterClasses` in the client, you can define the parameter classes using the `ArgumentClassList` property when you define the `FunctionInfo` object to represent the function in S-PLUS. This approach has the advantage of simplifying the Automation client program code but does require some additional steps in S-PLUS when defining the function.

Consider the following S-PLUS script to define the function `MyFunction` and a `FunctionInfo` object for this function:

```
MyFunction <- function(a)
{
    return(a)
}

gui Create(
    "FunctionInfo", Function = "MyFunction",
    ArgumentClassList = "vector, vector" );
```

The script sets `ArgumentClassList` to the string "vector, vector" indicating that data passed into and out of `MyFunction` via Automation will be done using S-PLUS vectors. When this approach is used, the corresponding client code becomes simpler because we no longer need to set the parameter classes for the function before it is used:

```
Dim pArray(1 to 3) as double
pArray(1) = 1.0
pArray(2) = 2.0
pArray(3) = 3.0

Dim pMyFunction as Object
Set pMyFunction = CreateObject("S-PLUS.MyFunction")
pMyFunction.a = pArray
pMyFunction.Run

Dim pReturnArray as Variant
pReturnArray = pMyFunction.ReturnValue
```

## Automating Embedded S-PLUS Graphs

With S-PLUS Automation support, it's easy to embed graph sheets in any Automation client, such as Visual Basic, Excel, Word, and others. You can create, modify, and save an embedded graph sheet with plotted data without ever leaving your client program.

The `vbembed` example shipped with S-PLUS demonstrates how to embed an S-PLUS graph sheet, add objects to it, modify these objects by displaying their property dialogs in the client program, delete objects from it, and save a document containing the embedded graph sheet. The `vcembed` example is a Visual C++/MFC application that shows how to embed and automate an S-PLUS graph in a C++ automation client. The `plotdata.xls` example illustrates how to embed an S-PLUS graph sheet, add a plot to it, send data from an Excel worksheet to be graphed in the plot, and modify the plot's properties using property dialogs. See Table 13.8 on page 526 for help in locating these example files.

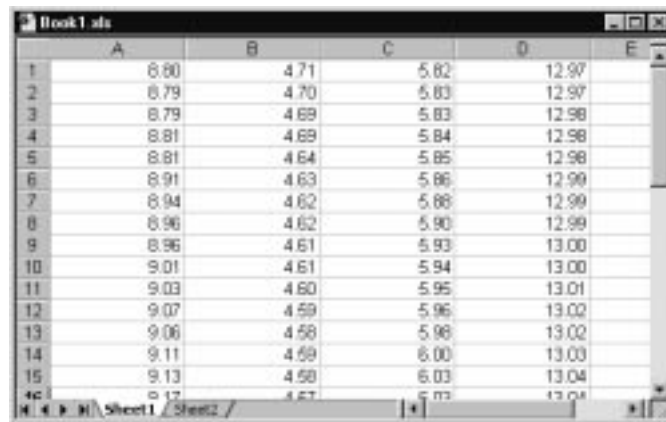
## USING S-PLUS AS AN AUTOMATION CLIENT

In addition to being used as an Automation server, S-PLUS can also function as an Automation client. A program in the S-PLUS programming language can create and manipulate the Automation objects exposed by other applications through their type libraries. S-PLUS provides a number of functions that allow you to create objects, set and get properties, call methods, and manage reference counting. Before discussing these functions in detail, let's take a look at a simple example.

### A Simple Example

To demonstrate how to use S-PLUS as an Automation client, we revisit the simple example presented on page 497, this time reversing the roles of S-PLUS and Excel. In this scenario, S-PLUS functions as the client application, retrieving data from an Excel worksheet, performing a covariance estimation on the data, and returning the resulting covariance matrix to Excel.

Consider again the sample data of Figure 13.1, reproduced in Figure 13.10. **Sheet1** of the Excel workbook **Book1.xls** contains data in 4 columns of 39 rows (not all rows are shown in Figure 13.10).



	A	B	C	D
1	8.80	4.71	5.82	12.97
2	8.79	4.70	5.83	12.97
3	8.79	4.69	5.83	12.98
4	8.81	4.69	5.84	12.98
5	8.81	4.64	5.85	12.98
6	8.91	4.63	5.86	12.99
7	8.94	4.62	5.88	12.99
8	8.96	4.62	5.90	12.99
9	8.96	4.61	5.93	13.00
10	9.01	4.61	5.94	13.00
11	9.03	4.60	5.95	13.01
12	9.07	4.59	5.96	13.02
13	9.06	4.58	5.96	13.02
14	9.11	4.59	6.00	13.03
15	9.13	4.58	6.03	13.04

Figure 13.10: Sample data in **Book1.xls**.

#### Note

The sample data in **Book1.xls** are taken from the **freeny.x** matrix included with S-PLUS. You can recreate this example by exporting the data into a new Excel worksheet and following the steps outlined below.

By writing a script in the S-PLUS programming language, we can automate a conversation between S-PLUS and Excel to perform our task. The complete code of one such script is shown in Figure 13.11. (This example script is also included as **Clitest1.ssc** in the **samples/oleauto/splus** folder in the S-PLUS program folder.)

```

pExcel <- create.ole.object("Excel.Application")
ExcelVisible <- get.ole.property(pExcel, "Visible")$Visible
if (!ExcelVisible) set.ole.property(pExcel, list(Visible=T))
pWorkbooks <- get.ole.property(pExcel, "Workbooks")[[1]]
pBook1 <- call.ole.method(pWorkbooks, "Open", "c:\\\\Excel\\Book1.xls", ReadOnly=F)
pSheets <- get.ole.property(pBook1, "Sheets")[[1]]
pSheet1 <- call.ole.method(pSheets, "Item", "Sheet1")
pRange1 <- call.ole.method(pSheet1, "Range", "A1:D39")
pData <- get.ole.property(pRange1, "Value")$Value
CovMatrix <- cov.wt(pData, wt=rep(1, nrow(pData)))$cov
CovDF <- data.frame(CovMatrix)
pSheet2 <- call.ole.method(pSheets, "Item", "Sheet2")
pRange2 <- call.ole.method(pSheet2, "Range", "A1:D4")
bResults <- set.ole.property(pRange2, list(Value=CovDF))
bNewBook1 <- call.ole.method(pBook1, "Save")
GameOver <- call.ole.method(pExcel, "Quit")

release.ole.object(pRange2)
release.ole.object(pSheet2)
release.ole.object(pRange1)
release.ole.object(pSheet1)
release.ole.object(pSheets)
release.ole.object(pBook1)
release.ole.object(pWorkbooks)
release.ole.object(pExcel)

rm(pExcel, ExcelVisible, pWorkbooks, pBook1, pSheets, pSheet1, pRange1,
  pData, CovMatrix, CovDF, pSheet2, pRange2, bResults, bNewBook1, GameOver)

```

Figure 13.11: Complete code for our S-PLUS script.

1. Open a new **Script** window in S-PLUS and enter the first block of code, as shown in Figure 13.12.

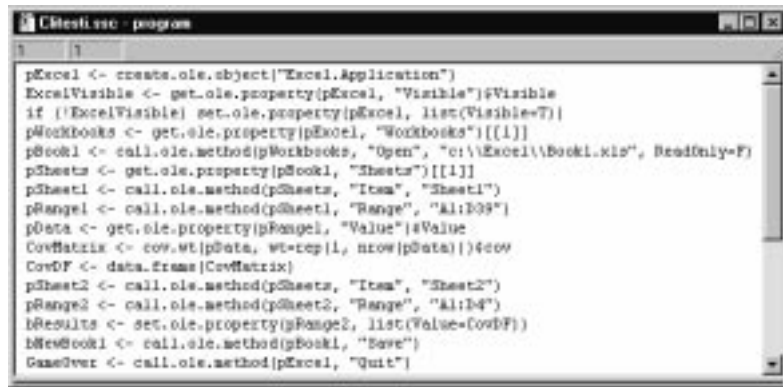


Figure 13.12: The core code for our S-PLUS script.

The code in Figure 13.12 represents the central task we want to automate. Let's examine each line in detail.

We start a conversation with Excel from S-PLUS by creating an instance of Excel using the S-PLUS function `create.ole.object`:

```
pExcel <- create.ole.object("Excel.Application")
```

To see what's happening in Excel as the script runs, we can set the `Visible` property of the Excel application object to `True`. To do so, we first capture the value of the `Visible` property using the `get.ole.property` function:

```
ExcelVisible <- get.ole.property(pExcel, "Visible")$Visible
```

#### Note

The `get.ole.property` function returns a list of properties. Use the `$` or `[[ ]]` operator to extract the value of an individual component of the list.

We then test the value of the `Visible` property and set it to `True` using the `set.ole.property` function:

```
if (!ExcelVisible) set.ole.property(pExcel, list(Visible=T))
```

**Note**

The `set.ole.property` function expects a list of properties to set.

To open the **Book1.xls** workbook, we first get the value of the `Workbooks` property of the Excel application object:

```
pWorkbooks <- get.ole.property(pExcel, "Workbooks")[[1]]
```

and then call the `Open` method on the `pWorkbooks` object using the S-PLUS function `call.ole.method`:

```
pBook1 <- call.ole.method(pWorkbooks, "Open",  
"c:\\Excel\\Book1.xls", ReadOnly=F)
```

**Note**

When using `call.ole.method` to call a method on an Automation object, consult the type library of the server application for a list of arguments relevant to the method you are calling.

Since, in this example, we are automating a conversation with Excel, we must follow the Excel object model and navigate through Excel's object hierarchy in order to access a range of cells:

```
pSheets <- get.ole.property(pBook1, "Sheets")[[1]]  
pSheet1 <- call.ole.method(pSheets, "Item", "Sheet1")  
pRange1 <- call.ole.method(pSheet1, "Range", "A1:D39")
```

Having arrived at the level of actual cell contents, we can now capture our data with the following statement:

```
pData <- get.ole.property(pRange1, "Value")$Value
```

In the next two statements, we use standard S-PLUS functions to perform the covariance estimation and convert the resulting matrix into a data frame:

```
CovMatrix <- cov.wt(pData, wt=rep(1, nrow(pData)))$cov  
CovDF <- data.frame(CovMatrix)
```

With the results now stored in an S-PLUS variable, we again navigate through the Excel object hierarchy to the target range of cells on **Sheet2**:

```
pSheet2 <- call.ole.method(pSheets, "Item", "Sheet2")  
pRange2 <- call.ole.method(pSheet2, "Range", "A1:D4")
```

and place the results in the target range:

```
bResults <- set.ole.property(pRange2, list(Value=CovDF))
```

Finally, the last two statements save the workbook and close the Excel application:

```
bNewBook1 <- call.ole.method(pBook1, "Save")
GameOver <- call.ole.method(pExcel, "Quit")
```

Before we proceed, let's pause for a moment and run the script as written so far.



2. Click the **Run** button  on the **Script** window toolbar.
3. After the script finishes running, click the **Object Explorer** button  on the **Standard** toolbar. See Figure 13.13.

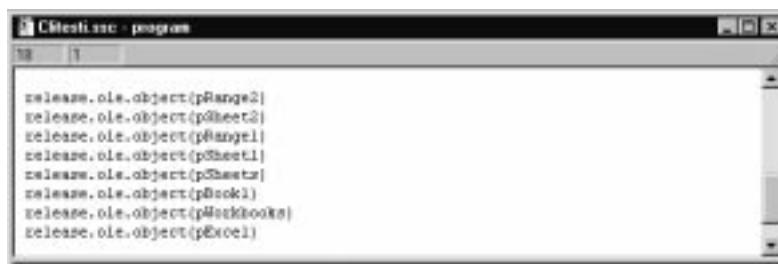


Figure 13.13: Objects created during execution of our S-PLUS script.

The right pane of the **Object Explorer** in Figure 13.13 displays a list of all the data objects created during execution of the script. Of particular interest are the eight objects of class `OLEClient`.

As we will see in the section Reference Counting Issues on page 522, objects of class `OLEClient` that are created directly or indirectly during program execution must be released at program end to allow the server application to close. This is accomplished in the next block of code we'll enter.

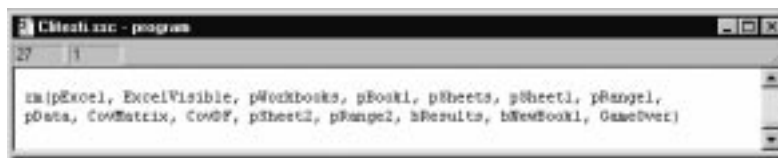
4. Add the second block of code to the script, as shown in Figure 13.14.



*Figure 13.14: Code for releasing all OLE objects.*

After releasing all the OLE objects, the last thing to do is to clean up our working data by deleting all the data objects created during execution of the script.

5. Add the last block of code to the script, as shown in Figure 13.15.



*Figure 13.15: Code for removing all data objects.*

With all the coding complete, it's time to run the script.

6. Click the **Run** button  on the **Script** window toolbar.



7. After the script finishes running, start Excel, open the **Book1.xls** workbook, and click the tab for **Sheet2**. The results are shown in Figure 13.16.



Figure 13.16: The covariance matrix sent to Excel.

## High-Level Automation Functions

As demonstrated in the last section, you can use S-PLUS as an Automation client by writing a program in the S-PLUS programming language to create and manipulate the Automation objects of other applications.

S-PLUS provides four functions, listed in Table 13.6, for creating objects, setting and getting properties, and calling methods. The Automation objects of the server application are passed as arguments to these functions. In addition, S-PLUS provides several other functions for managing reference counting; see page 522 for details.

For each of the functions listed in Table 13.6, two optional character vector attributes called “error” and “warning” may be returned with each return value. These attributes can be used to stop the program and display warnings, errors, etc.

Table 13.6: High-level Automation functions.

Function	Description
<code>create.ole.object(name)</code>	<p>The <i>name</i> argument is the name of the instance, a character vector of length 1.</p> <p>This function returns a character vector of class <code>OLEClient</code>, representing the particular instance. The first string in the vector is the instance ID of the object, stored in S-PLUS. The other strings are various pieces of information about the instance (such as server name). On error, <code>NULL</code> is returned.</p>

Table 13.6: *High-level Automation functions. (Continued)*

Function	Description
<code>get.ole.property(instance, property.names)</code>	<p>The <i>instance</i> argument is an object previously created within the same session. The <i>property.names</i> argument is a character vector of property names appropriate to the instance, as specified in the type library of the server application.</p> <p>This function returns a list of the values of the properties specified in the <i>property.names</i> argument. NULL is returned for a property that cannot be fetched.</p>
<code>set.ole.property(instance, properties)</code>	<p>The <i>instance</i> argument is an object previously created within the same session. The <i>properties</i> argument is a list of elements, each element being a property name set to a desired value, which must be an atomic type of length 1.</p> <p>This function returns a vector of logicals, with T for each property successfully set and F otherwise.</p>
<code>call.ole.method(instance, method.name, ...)</code>	<p>The <i>instance</i> argument is an object previously created within the same session. The <i>method.name</i> argument is the name of the method to be invoked on the instance object. “...” represents the arguments to be passed as arguments to the method. (Only supported types may be passed, that is, at this point, atomic types—character, single, integer, numeric vectors—of length 1.) Methods for particular objects and arguments to specific methods can be found in the type library of the server application.</p> <p>This function returns an S-PLUS object containing the result of calling the method on the particular instance. On error, NULL is returned.</p>

## Reference Counting Issues

When you create an object using `create.ole.object`, or when an object is created indirectly from the return of a property using `get.ole.property` or `call.ole.method`, the object is given a reference count of one. This means that your S-PLUS program has a lock on the object and the server application that created the object for your program cannot close until you release the references to it by reducing the reference count to zero.

If you assign the variable that represents one of these objects to another object, you should increment the object's reference count by one to indicate that an additional variable now has a lock on the object. If you remove a variable or a variable that represents an Automation object goes out of scope, you should decrement the reference count of the object the variable being destroyed represents.

To help you manage reference counting on Automation objects, S-PLUS provides the four functions listed in Table 13.7. For each of these functions, two optional character vector attributes called “error” and “warning” may be returned with each return value. These attributes can be used to stop the program and display warnings, errors, etc.

*Table 13.7: Functions for managing reference counting.*

Function	Description
<code>ole.reference.count(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function returns the value of the reference count for the instance. On error, -1 is returned.
<code>add.ole.reference(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function increments the reference count for the instance by 1 and returns the new value. On error, -1 is returned.
<code>release.ole.object(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function decrements the reference count for the instance by 1 and returns the new value. On error, -1 is returned.
<code>is.ole.object.valid(instance)</code>	The <i>instance</i> argument is an object previously created within the same session. This function returns T if the instance is valid or F otherwise.


Although reference counting must be handled manually, S-PLUS guards against the two major types of reference counting bugs: resource leaks and freezing due to using an invalid object handle.

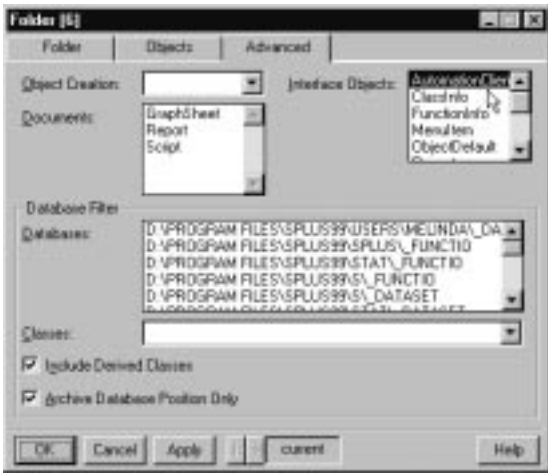
If you release all reference counts on an object and then attempt to get or set a property or call a method on this object, S-PLUS gives you an error that the object is no longer valid. You can check the validity of any Automation object by using the S-PLUS function `is.ole.object.valid`. You can also use the **Object Explorer** to check the validity of objects and adjust their reference counts. See the next section for details.

If you fail to release all references to objects you create during an S-PLUS session, the server application owning the objects will remain loaded and running. However, exiting S-PLUS automatically releases all objects, reduces all reference counts to zero, and closes all server applications.

**Managing  
Automation  
Objects Through  
the Object  
Explorer**

The **Object Explorer** provides a handy way to view all the valid and invalid Automation objects created during the current session. Simply insert a new folder that filters on interface objects of type AutomationClient:

1. With the **Object Explorer** in focus and no items selected, click the **New Folder** button  on the **Object Explorer** toolbar.
2. Enter a name for the new folder and press ENTER.
3. Right-click on the folder and select **Advanced** from the shortcut menu. The **Advanced** page of the **Folder** dialog opens, as shown in Figure 13.17.



*Figure 13.17: The **Advanced** page of the **Folder** dialog.*

4. Select **AutomationClient** from the **Interface Objects** list box and click **OK**. See Figure 13.18.

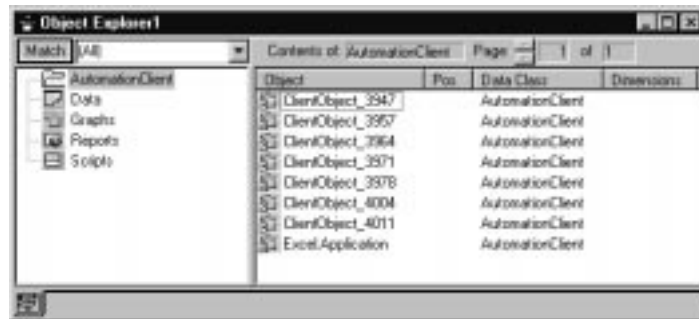


Figure 13.18: Displaying Automation objects in the **Object Explorer**.

When you release an Automation object by reducing its reference count to zero, its icon will be grayed-out in the right pane of the **Object Explorer**. This gives you a quick way to check which objects are invalid and which are still valid after running your S-PLUS script and thus is useful for debugging purposes.

You can also use the **Object Explorer** to create and destroy objects, adjust reference counts, and change the name to which an `AutomationClient` object refers.

- In the right pane of the **Object Explorer**, double-click an object's icon or right-click on it and select **Properties** from the shortcut menu. The **AutomationClient** dialog opens, as shown in Figure 13.19.

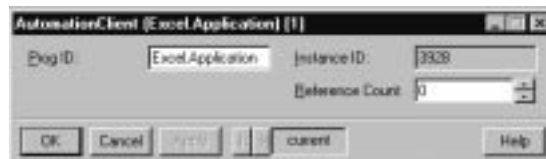


Figure 13.19: The **AutomationClient** dialog.

Changing the **Prog ID** field to a different Automation object name destroys the current object and creates a new object. Changing the **Reference Count** to zero destroys the current object and marks it as invalid.

## AUTOMATION EXAMPLES

S-PLUS ships with a number of example files that illustrate how to use S-PLUS as both an Automation server and an Automation client.

### Server Examples

The examples listed in Table 13.8 can be found in the **samples/oleauto** folder in the S-PLUS program folder.

*Table 13.8: Automation server examples.*

Name	Client Application	Description
<b>choosept</b>	Visual Basic 4.0	Project showing how to use the <code>ChooseGraphAndPlotType</code> Automation method to display the <b>Insert Graph</b> dialog and allow a user select the graph and plot type.
<b>creatept</b>	Visual Basic 4.0	Project showing how to use the following Automation methods: <ul style="list-style-type: none"> <li>• <code>CreatePlots()</code></li> <li>• <code>CreateConditionedPlots()</code></li> <li>• <code>CreateConditionedPlotsSeparateData</code></li> </ul>
<b>dialogs</b>	Visual Basic 4.0	Project showing how to use the following Automation methods: <ul style="list-style-type: none"> <li>• <code>ShowDialog()</code></li> <li>• <code>ShowDialogInParent()</code></li> <li>• <code>ShowDialogInParentModel ess()</code></li> </ul>
<b>gspages</b>	Visual Basic 4.0	Project showing how to embed a graph sheet with multiple pages, set up a tab control that has tabs for each page, and support changing active pages in the embedded graph sheet by clicking on tabs in the tab control.
<b>mixeddf</b>	Visual Basic 4.0	Project showing how to create a two-dimensional array of mixed data (that is, columns of different data types) and send it to a <code>data.frame</code> object in S-PLUS and also how to retrieve the data from the <code>data.frame</code> object into Visual Basic for display.

Table 13.8: Automation server examples. (Continued)

Name	Client Application	Description
<b>objects</b>	Visual Basic 4.0	Project showing how to use the following Automation methods: <ul style="list-style-type: none"> <li>• <code>ObjectContains()</code></li> <li>• <code>ObjectContainer()</code></li> <li>• <code>ClassName()</code></li> <li>• <code>PathName()</code></li> </ul>
<b>senddata</b>	Visual Basic 4.0	Project showing how to send data to S-PLUS data objects.
<b>vba</b>	Visual Basic for Applications with Excel 97	<ul style="list-style-type: none"> <li>• <b>Auto_vba.xls</b>: Example showing how to send and receive data and convert Excel ranges to arrays.</li> <li>• <b>Book1.xls</b>: Example showing how to pass data from an Excel worksheet to S-PLUS, which then performs a covariance estimation on the data and returns the resulting covariance matrix to Excel.</li> <li>• <b>Plotdata.xls</b>: Example showing how to embed a graph sheet and add and modify a plot in it.</li> <li>• <b>Xfertodf.xls</b>: Example showing how to transfer Excel ranges to S-PLUS data frames and back to Excel ranges.</li> </ul>
<b>vbclient</b>	Visual Basic 4.0	Project showing how to create a graph sheet, add an arrow to it, change the properties of the arrow, show a dialog for the arrow, execute S-PLUS commands, modify option values, get an object, and send and receive data.
<b>vbembed</b>	Visual Basic 4.0	Project showing how to embed an S-PLUS graph sheet, modify it, save it, and delete objects in it, and how to display an object dialog.
<b>vbrunfns</b>	Visual Basic 4.0	Project showing how to register an S-PLUS function, pass binary data to the function, and receive the result back into Visual Basic.
<b>vcembed</b>	Visual C++ 6.0	Project showing how to create and manipulate an embedded graph sheet in an MFC application. This example uses several MFC classes that make interacting with Automation objects from S-PLUS much easier in MFC code. See the <b>readme.txt</b> in the <b>vcembed</b> directory for more information.

*Table 13.8: Automation server examples. (Continued)*

Name	Client Application	Description
<b>autoclt</b>	Visual C++ 6.0	Non-MFC based C++ project showing how to use Automation to access the S-PLUS command line.
<b>vjclass1</b>	Visual J++ 6.0	Project showing how to create an S-PLUS application object and execute a command in S-PLUS syntax.

## Client Examples

The examples listed in Table 13.9 can be found in the **samples/oleauto/splus** folder in the S-PLUS program folder.

*Table 13.9: Automation client examples.*

Name	Server Application	Description
<b>Clitest.a.ssc</b>	Excel 97	Script showing how to use S-PLUS commands to start Excel and call method of Excel to convert inches to points and return the result in S-PLUS.
<b>Clitestb.ssc</b>	Excel 97	Script showing how to use S-PLUS commands to start Excel, get a property, and set a property.
<b>Clitestc.ssc</b>	Excel 97	Script showing how to use S-PLUS commands to set a range of data in an Excel worksheet with data from an S-PLUS vector and then how to get the data back from Excel into another vector.
<b>Clitestd.ssc</b>	Excel 97	Script showing how to get a property value from Excel.
<b>Cliteste.ssc</b>	Excel 97	Script showing how to send a vector from S-PLUS to Excel and transpose it to a row in Excel.
<b>Clitestf.ssc</b>	Excel 97	Script showing how to send a vector from S-PLUS to Excel and transpose it to a row in Excel using a different set of steps than in <code>cliteste.ssc</code> .
<b>Clitestg.ssc</b>	Excel 97	Script showing how to send the data from a data frame in S-PLUS into a new worksheet and range in Excel and then how to get the range data from Excel back into a new data frame in S-PLUS.



*Table 13.9: Automation client examples. (Continued)*

Name	Server Application	Description
<b>Clitesth.ssc</b>	Visual Basic 4.0	Example combining a Visual Basic Automation server project called <b>GetArray</b> with an S-PLUS Automation client script that calls it to retrieve data into a data frame (used in conjunction with the VB Automation server in the <b>Getarray</b> folder). The Automation server exposes an object type called <code>MyArrayObject</code> having a method called <code>GetArray</code> that gets a randomly generated, two-dimensional array of mixed data (that is, columns having different data types). The S-PLUS script uses this method to get the array and then set it into a newly created data frame.
<b>Clitesti.ssc</b>	Excel 97	Script showing how to retrieve data from an Excel worksheet, perform a covariance estimation on the data, and return the resulting covariance matrix to Excel.



---

<b>Introduction</b>	<b>532</b>
<b>About DDE</b>	<b>533</b>
Working With DDE in S-PLUS	535
Starting a DDE Conversation	537
Executing S-PLUS Commands: DDE Execute	538
Sending Data to S-PLUS: DDE Poke	539
Getting Data From S-PLUS: DDE Request	540
Using Application Names in Client Program Scripts	542

## INTRODUCTION

This chapter explains how to communicate with S-PLUS using DDE (Dynamic Data Exchange).

Communications between programs can take place in a number of ways. It is clearly possible to pass data between programs by having one program write an ASCII file to disk and having another program read that file. It is also possible for programs to exchange data using the Windows Clipboard. In each of these methods, the process is sequential and normally requires human intervention to coordinate the action.

DDE, on the other hand, permits one running program to communicate with another running program in real time without outside intervention.

There are occasions when S-PLUS users may need to use other programs to manipulate data. For example, the accounting department may have a spreadsheet that summarizes financial data for the company, and an analyst may be asked to summarize that data. DDE enables the calling of S-PLUS functions from within a spreadsheet.

---

## ABOUT DDE

Dynamic Data Exchange (DDE) is a mechanism supported by Microsoft Windows that permits two different programs running under Windows to communicate. This communication can take the form of passing data back and forth, or it can take the form of one program requesting the other program to take specific action. This communication can take place under program control (without human intervention) and as often as required.

In a DDE conversation, one program initiates the conversation and the other program responds. The program which initiates the conversation is called the *destination* program or the *client*. The program which responds is called the *source* program or the *server*.

S-PLUS can function as a DDE server, as described below. The basic DDE functions Connect, Execute, Poke, Request, Advise, Unadvise, and Terminate are supported. S-PLUS is not a DDE client.

DDE uses the metaphor of a conversation. Because an application can have multiple DDE conversations open at the same time, it is necessary to have an identifier for each conversation. We will call this the *channel number*. The channel number is established when a DDE conversation is initiated.

A DDE client application opens a conversation with a particular server. Each server has a name to which it will respond. We will call this the *server name*. The server name for S-PLUS is “S-PLUS”.

*Table 14.1: A DDE conversation must have a topic. S-PLUS supports the following topics for DDE conversations.*

Topic	Description	Example in Visual Basic for Applications
System [System]	The System topic is used to request special information from S-PLUS. This information includes the names of data objects (such as vectors and dataframes) that can be used in conversations, allowable conversation topic names, and other information.	<pre> Channel = Application.DDEInitiate( _     "S-PLUS", "System") Info = Application.DDERequest( _     Channel, "Topics" ) Application.DDETerminate Channel  Info = {     "System"     "[DataSheet]",     "[Execute]",     "SCommand" }</pre>
[DataSheet]	The [DataSheet] topic is an identifier used to specify that the conversation is about a block of data from a data object, and is optional. You can simply specify the name of the data object as the conversation topic.	<pre> Channel = Application.DDEInitiate( _     "S-PLUS", "[DataSheet]exsurf" ) Info = Application.DDERequest( _     Channel, "r1c1:r3c2" ) Application.DDETerminate Channel  Info = {     -2.00, -2.00;     -1.87, -2.00;     -1.74, -2.00 }</pre>

*Table 14.1: A DDE conversation must have a topic. S-PLUS supports the following topics for DDE conversations.*

Topic	Description	Example in Visual Basic for Applications
Any data object name (such as the name of a data frame or vector)	Same as the [DataSheet] topic above.	<pre>Channel = Application.DDEInitiate( _     "S-PLUS", "exsurf" ) Info = Application.DDERequest( _     Channel, "r1c1:r3c2" ) Application.DDETerminate Channel  Info = {     -2.00, -2.00;     -1.87, -2.00;     -1.74, -2.00}</pre>
[Execute] SCommand	[EXECUTE] or SCommand identifies that the conversation contains strings of valid S-PLUS commands or expressions to be executed by S-PLUS.	<pre>Channel = Application.DDEInitiate( _     "S-PLUS", "SCommand" ) Info = Application.DDERequest( _     Channel, "objects(0)" ) Application.DDETerminate Channel  Info = { ". Copyright . Options     . Program version" }</pre>

S-PLUS supports DDE conversations in the Microsoft CF\_TEXT format, which is simply an array of text characters terminated by a null character and with each line ending with a carriage return–linefeed (CR–LF) pair. Binary transfers are not supported.

Any application that supports DDE client functions can initiate a DDE conversation with S-PLUS. Examples of such applications include Visual Basic, Visual C++, Excel, Lotus 1-2-3, and PowerBuilder.

## Working With DDE in S-PLUS

S-PLUS supports two ways of working with DDE:

1. Using Copy/Paste Link from the Edit menu and
2. Using DDE functions built into other DDE client programs.

Copy/Paste Link is very easy to use but does not offer the flexibility and additional features of the DDE client program commands such as `DDERequest`, `DDEExecute`, and `DDEPoke` as supported by such languages as Visual Basic for Applications.

## Enabling/ Disabling S-PLUS Response to DDE Requests

At any time during a session with S-PLUS, you can suspend response to messages sent from DDE client applications by choosing **General Settings** from the **Options** menu and in the dialog that appears, uncheck the **Respond to DDE Requests** option. All links to any S-PLUS data objects will be temporarily suspended until the option is re-enabled.

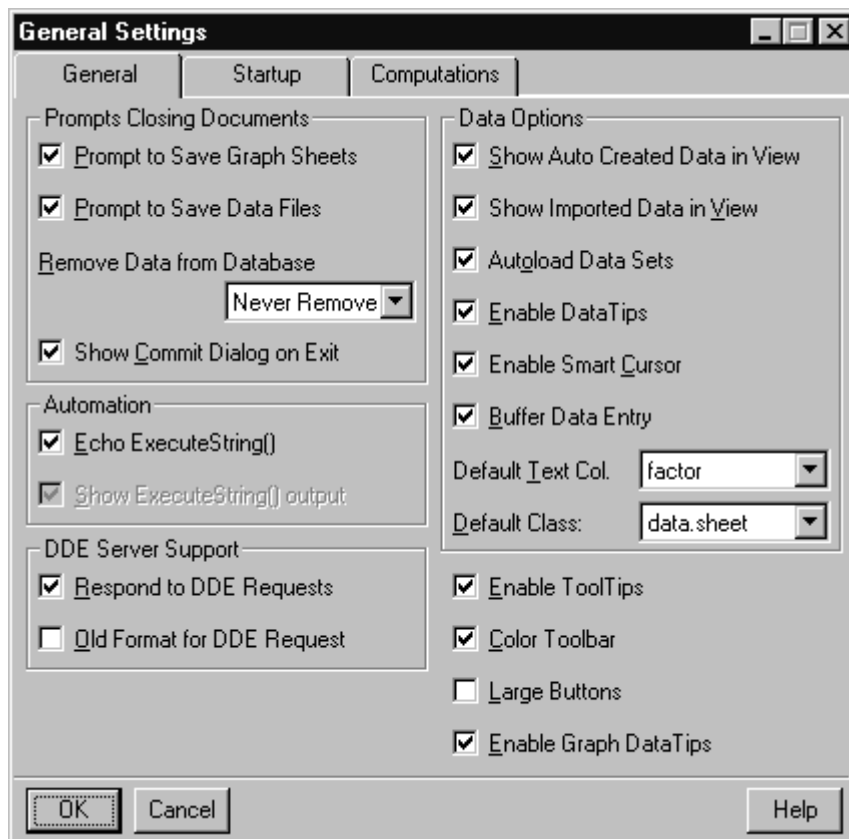


Figure 14.1: General Settings dialog box.

## Using Copy/Paste Link Using S-PLUS as a DDE server via the clipboard

You can copy data from S-PLUS data objects such as dataframes and vectors to the clipboard and paste that data into other OLE or DDE supporting programs as a DDE link to the data in S-PLUS. This connection is a hot link between a block of cells in an S-PLUS data object and a block in the document of another program. If DDE server support is enabled in S-PLUS, whenever you copy data from a data object to the clipboard, DDE link



information is transferred at the same time. Then when you paste into another program, the **Paste Special** or **Paste Link** option will be enabled if that client program supports DDE linking.

### Using S-PLUS as a DDE client via the clipboard

If you have copied data from a server program that supports DDE linking (such as Excel) into the clipboard, you can choose **Paste Link** from the **Edit** menu when a data object such as a data frame or a vector is in focus. This will paste the link into the data object that currently has the focus, starting the pasted block at the current cell location. This tells S-PLUS to request a DDE link to the data specified in the server program's document. Then whenever the data changes in the server document, the changes are automatically updated in the S-PLUS data object where you pasted the linked data.

## Starting a DDE Conversation

Visual Basic for Applications (VBA) has been chosen as a convenient client application for demonstrating access to S-PLUS as a DDE server. A sample Visual Basic DDE client program called **vbclient** is included on your distribution media. There are also example Excel spreadsheets with VBA and macro scripts demonstrating connection to S-PLUS via DDE.

The three steps of a DDE conversation are as follows:

1. Initiate the conversation
2. Issue one or more DDE commands
3. Terminate the conversation

Table 14.2 illustrates a simple `DDERequest` conversation with VBA (as used by Microsoft Word or Excel) as the client and S-PLUS as the server. You can use either `SCommand` or `[Execute]` for executing an S-PLUS command and requesting the result via a `DDERequest`.

The statement

```
exec_channel = Application.DDEInitiate("S-PLUS", _
    "SCommand")
```

initiates the conversation to S-PLUS with the topic `SCommand`. This statement returns the channel number for the conversation and assigns it to `exec_channel`. The channel number is then used in the subsequent `DDERequest` statement. This statement executes the command

`summary(corn.rain)` in S-PLUS and returns the result of the execution in an array called `ReturnResult`. Then the conversation is ended by calling `DDETerminate`.

*Table 14.2: Starting a DDERequest conversation from VBA.*

```
exec_channel = Application.DDEInitiate( "S-PLUS", "SCommand" )
ReturnResult = Application.DDERequest( exec_channel, "summary(corn.rain)" )
Application.DDETerminate exec_channel
```

## Executing S-PLUS Commands: DDE Execute

The `DDEExecute` command executes a command in S-PLUS and does not return output from the execution. Table 14.3 illustrates a simple `DDEExecute` routine in VBA. You can use either `SCommand` or `[Execute]` for the topic of a `DDEExecute` conversation.

First, the conversation is initiated with the `DDEInitiate` command. Then `DDEExecute` is used to execute the S-PLUS command `summary(corn.rain)`. The output from this command will go to either the S-PLUS Commands window if open, or to a Report window in S-PLUS. No output is returned to the calling program. Finally the conversation is ended with a call to `DDETerminate`.

A client program can issue multiple DDE commands including `DDEExecute`, `DDERequest`, and `DDEPoke` to S-PLUS as long as the DDE Conversation with the topic `SCommand` or `[Execute]` is open. `DDEPoke` using a conversation with the topic `SCommand` or `[Execute]` behaves in the same way as `DDEExecute` does, that is it executes the commands you poke and does not return any results. `DDERequest` using a conversation with the topic `SCommand` or `[Execute]` executes the commands specified and returns the result of the execution to the variable that is assigned to the `DDERequest` call. Both `DDERequest` and `DDEPoke` are discussed later in this chapter.

*Table 14.3: Starting a DDEExecute conversation from VBA.*

```
exec_channel = Application.DDEInitiate( "S-PLUS", "SCommand" )
Application.DDEExecute exec_channel, "summary(corn.rain)"
Application.DDETerminate exec_channel
```

The exact format of the `DDEExecute` command will vary from program to program. That is, Excel has its own format, as does Lotus 1-2-3 and Visual Basic. The essential features are the same: a channel number and a command string.

## Sending Data to S-PLUS: DDE Poke

When used in a conversation with the topic `SCommand` or `[Execute]`, the `DDEPoke` command behaves in the same way as the `DDEExecute` command. The following example of `DDEPoke` (written in a VBA script in Excel) executes commands in S-PLUS:

```
Channel = Application.DDEInitiate("S-PLUS", "SCommand")
Application.DDEPoke Channel, "", Sheets("Sheet1").Range("A1")
Application.DDETerminate Channel
```

First, `DDEInitiate` is used to open an `SCommand` conversation with S-PLUS. Next, `DDEPoke` is used to send the string located in the cell at A1 in Sheet1 of the current workbook from Excel to S-PLUS.

Like `DDEExecute`, `DDEPoke` must have a channel number specifying the conversation. The second parameter, called the *item text*, is ignored by S-PLUS when poking commands using an `SCommand` or `[Execute]` conversation. The *item text* parameter can be set to any value. VBA requires that this parameter be set to some value. The third parameter is a cell reference telling Excel where to find the string representing the commands to send to S-PLUS. Whatever is in that cell will be executed in S-PLUS.

S-PLUS then executes this string sending any output from the command to either the commands window if open or to a report window in S-PLUS. The conversation is then terminated using the `DDETerminate` command.

`DDEPoke` can also be used to send data from a DDE client program into existing data objects such as dataframes or vectors in S-PLUS when used in a conversation with the topic `[DataSheet]` or the name of the data object you wish to change data in.

Consider the following example of `DDEPoke` in a VBA script in Excel:

```
Channel = Application.DDEInitiate("S-PLUS", "exsurf")
Application.DDEPoke Channel, "r1c1:r3c2", Sheets("Sheet1").Range("A1:B3")
Application.DDETerminate Channel
```

First a conversation is initiated with the topic name set to the name of an existing data frame, in this case "exsurf". Next, the data in the spreadsheet "Sheet1" in the range "A1" through cell "B3" of the current workbook of Excel is sent to the data frame "exsurf" in the cells "r1c1:r3c2", that is to cells starting at row 1 and column 1 extending to row 3 and column 2.

The "r1c1:r3c2" string specifies the item string of the `DDEPoke` command and tells Excel where to put the data in the S-PLUS data frame named by the string "exsurf". The statement

`Sheets("Sheet1").Range("A1: B3")` is Excel syntax specifying the data for Excel to send to S-PLUS.

## Getting Data From S-PLUS: DDE Request

`DDEExecute` and `DDEPoke` let you send commands and data to S-PLUS. `DDERequest` lets you ask S-PLUS to send data back to the calling program.

When used in a conversation with the topic `SCommand` or `[Execute]`, the `DDERequest` command behaves in the same way as the `DDEExecute` command except that any output from execution is sent back to the calling application as an array of text strings.

The following example of `DDERequest` (written in a VBA in Excel) executes commands in S-PLUS and returns the results:

```
Channel = Application.DDEInitiate("S-PLUS", "SCommand")
ReturnData = Application.DDERequest(Channel, "summary(corn. rain)")
Application.DDETerminate Channel

stringRangeToFillSpec = "A1" + ":" + "A" + Trim(Str(UBound(ReturnData)))
Sheets("Sheet1").Range(stringRangeToFillSpec).Value = _
    Application.Transpose(ReturnData)
```

First, `DDEInitiate` is used to open an `SCommand` conversation with S-PLUS. Next, `DDERequest` is used to send the string `"summary(corn. rain)"` to S-PLUS for execution. The output from this command is passed back to Excel to the array `ReturnData`. The conversation is terminated using `DDETerminate`. Then, a string variable `stringRangeToFillSpec` is created using the upper bound of the `ReturnData` array (the call to `UBound(ReturnData)`) to specify the range of cells on Sheet1 in the current workbook to receive the return data. The `ReturnData` array is then transposed so each element of the array is in a row of the column A in Sheet1. Sheet1 column A now contains the summary data for the data frame `"corn. rain"`.

`DDERequest` must have a channel number specifying the conversation. The second parameter, called the *item text*, is used to specify the commands to execute in S-PLUS when using an `SCommand` or `[Execute]` conversation.

The format of the text in the array returned by `DDERequest` on an `SCommand` or `[Execute]` conversation can be controlled via an option in S-PLUS. By default, output is sent back using a method of most S-PLUS objects called `print`. This provides the best general purpose formatting for the returned results. However, if you depend on the format as returned in earlier versions

of S-PLUS, you can set this option to use the older formatting style. To change the formatting to that provided by older versions of S-PLUS, select **General Settings** from the **Options** menu. In the **General Settings** dialog, check the box for **Old Format for DDE Request**.

DDERequest can also be used to transfer data from existing data objects such as dataframes or vectors in S-PLUS into a DDE client program when used in a conversation with the topic [DataSheet] or the name of the data object you wish to transfer data from.

Consider the following example of DDERequest in a VBA script in Excel:

```
Channel = Application.DDEInitiate("S-PLUS", "exsurf")
ReturnData = Application.DDERequest(Channel, "r1c1:r3c2")
Application.DDETerminate Channel

sStartCell = "A1"
sSheetName = "Sheet1"

sRangeToFill = sStartCell + ":" + _
    Chr(Asc("A") + (Trim(Str(UBound(ReturnData, 2))) - 1)) + _
    Trim(Str(UBound(ReturnData, 1)))
Sheets(sSheetName).Range(sRangeToFill).Value = ReturnData
```

First a conversation is initiated with the topic name set to the name of an existing data frame, in this case "exsurf". Next, the data in the data frame "exsurf" in the cells "r1c1:r3c2", that is cells starting at row 1 and column 1 extending to row 3 and column 2, are sent to Excel into the array ReturnData. The "r1c1:r3c2" string specifies the item string of the DDERequest command and tells S-PLUS which cells from the data frame "exsurf" to send back to Excel. Then the conversation is terminated using DDETerminate. Two strings are assigned values specifying the starting cell ("A1") and the sheet ("Sheet1") where the subsequent commands are to copy the data in the ReturnData array. Finally, using the upper bounds of the ReturnData array (calls to UBound), the array of data is copied into the cells of the desired sheet.

Using  
Application  
Names in  
Client Program  
Scripts

Automation and DDE server support in S-PLUS and Axum share many of the same features. The main difference between writing automation and DDE client program scripts that use S-PLUS or Axum as a server is the name of the application or the DDE server name used in the script. By using values from the Windows system registry which are written every time S-PLUS or Axum is started, you can modify the automation and/or DDE client scripts you have so that they will work with either S-PLUS or Axum.

Every time you start up S-PLUS or Axum, several entries are made to the Windows system registry related to the application name, DDE server name, embedded graph sheet type name, and other information. If you uninstall either S-PLUS or Axum, these entries are removed. You can use commands in your automation and/or DDE client program to identify the name of the application for automation, the name of the DDE server, the type name to use for embedding a graph sheet, and so on.

The following discussion is directed toward individuals who are somewhat familiar with programming languages such as Visual Basic and have written programs to create and modify S-PLUS or Axum automation objects and/or manage DDE conversations.

Registry Keys

Every time S-PLUS or Axum is started, the following Windows registry key is written and/or updated:

**HKEY\_LOCAL\_MACHINE\Software\Mathsoft\LastAppStarted**

Under this key, several values are written depending on the application started:

<i>Value Name:</i>	<i>Description:</i>	<i>If S-PLUS, value is:</i>	<i>If Axum, value is:</i>
<b>ApplicationName</b>	The name of the application last started. This should be used when creating or modifying objects (such as in the Visual Basic command CreateObject) in automation client programs.	S-PLUS	Axum
<b>ApplicationPathName</b>	The full file pathname of the last application that was started.	Path plus executable file name and extension of S-PLUS.	Path plus executable file name and extension of Axum.

<b>DDEServerName</b>	The server name of the application which should be used when initiating a DDE conversation from a DDE client program.	S-PLUS	AxumDDE
<b>EmbeddedGSType-Name</b>	The type name for an embedded graph sheet that should be used when programmatically embedding new graph sheet objects in automation client programs.	SPLUSGraphSheetFileType	Axum6GraphSheetFileType
<b>Version</b>	A string representing the version number (both major and minor) of the application which was last started.	Version of S-PLUS.	Version of Axum.
<b>Edition</b>	A string representing the edition of the application which was last started.	Edition of S-PLUS.	Edition of Axum.

You can use these registry values in your own automation and/or DDE client programs to identify the application names to use.

An example function called `GetLastAppStartedValue()` written in Visual Basic 4.0, VBA, and Visual C++ is provided on disk in the “samples\lastapp” subdirectory of the application directory. This function is used throughout the examples for automation and DDE provided, and it is referred to in the discussion which follows. This function can be added to your automation or DDE client program by simply copying and pasting the text in the appropriate file in “samples\lastapp” for your client program language.

### Using LastAppStarted Values in Automation Scripts

A typical automation client program script might currently look like this one written with Visual Basic syntax that uses Axum as an automation server:

```
Sub Main()
    Dim pApp as Object
    Set pApp = CreateObject( "Axum. Application" )

    Dim sVersion as string
    sVersion = pApp.Version
End Sub
```

Notice that the `CreateObject()` call has the hard-wired application string "Axum.Application" in it. This prevents this script from working with S-PLUS. In order to allow this script to work with S-PLUS you would have to modify this string to read "S-PLUS.Application". However, once this modification is done, the script would have to be changed back again to work with Axum. Using the registry values described above allows your scripts to work with either Axum or S-PLUS without modifying them. This would be useful if, for instance, you developed some scripts for Axum, and then later needed to use them on systems where S-PLUS was installed. You could rewrite the above script as follows using a function that returns the registry value for the appropriate automation server name:

```
Sub Main()  
    Dim pApp as Object  
    Set pApp = CreateObject( _  
GetLastAppStartedValue( "ApplicationName" ) +  
    ".Application" )  
    Dim sVersion as string  
    sVersion = pApp.Version  
End Sub
```

The function `GetLastAppStartedValue()` needs to be defined only once in your automation script and then can be used wherever you are using the string "Axum". `GetLastAppStartedValue()` uses Windows API functions to retrieve the desired registry value that is passed in as an argument (that is, "ApplicationName") under the key **HKEY\_CURRENT\_USER\Software\Mathsoft\LastAppStarted**.

By writing the above script using `GetLastAppStartedValue()`, this script will now work with either Axum or S-PLUS depending on which was most recently started.

The function `GetLastAppStartedValue()` is defined and used in most of the automation examples provided in the **samples** directory installed with this program. Please refer to these sample scripts for more examples of using this function and for the definition of this function. This function can also be used to retrieve the correct DDE server name in DDE client scripts so that these scripts will also run without modification between Axum and S-PLUS.

### Using LastAppStarted Values in DDE Client Scripts

In the examples installed with the program in the **samples\dde** directory, there are several examples of using S-PLUS as a DDE server from client scripts in Excel and Visual Basic 4.0. These examples use the same function called `GetLastAppStartedValue()` as described above to get a string in the script representing the name of the DDE server you last started. This allows the examples to work with both Axum and S-PLUS without modification.



For example, this function could be used in a DDE client program to retrieve the server name:

```
Chan = Application.DDEInitiate( _  
    GetLastAppStartedValue("DDEServerName"), "System")  
Info = Application.DDERequest(Chan, "Topics")  
Application.DDETerminate Chan
```

In the above `DDEInitiate()` call, `GetLastAppStartedValue()` will return "AxumDDE" when Axum was last started and "S-PLUS" when S-PLUS was last started. This allows your DDE client script to work with either Axum or S-PLUS without further modifications.

You can use the function `GetLastAppStartedValue()` in your DDE client scripts that may have the DDE server name "Axum" in them. By using `GetLastAppStartedValue()`, your scripts will run without modification with either S-PLUS or Axum. You can also modify the example function `GetLastAppStartedValue()` provided to work with the programming language of DDE client programs you use.



---

# THE WINDOWS AND DOS INTERFACES

# 15

---

<b>Using the Windows Interface</b>	<b>548</b>
<b>Using the DOS Interface</b>	<b>552</b>
<b>Programming With the DOS and Windows Interfaces</b>	<b>554</b>
Functions for Archiving and Restoring Data	554
A Matrix Editor Using Microsoft Excel	556

Using the S-PLUS interfaces to Windows and DOS, you can run your favorite Windows and DOS applications from the S-PLUS prompt, or incorporate those applications in your S-PLUS functions. For example, the `fix` and `edit` functions use the Windows interface to start an editor on a file containing an S-PLUS object to be edited. Similarly, the `objdiff` function uses the DOS interface to run the `fc` command on ASCII dumps of two S-PLUS objects to be compared.

This chapter describes the DOS and Windows interfaces and provides several examples of S-PLUS functions you can write to incorporate spreadsheets, databases, and word processors into your S-PLUS programming environment.

## USING THE WINDOWS INTERFACE

To run a Windows application from S-PLUS, use the `system` function, which requires one argument: a character string containing a command suitable for the Windows “Run” command line. For example, to run the Windows Calculator, you could call `system` as follows:

```
> system("cal c")
```

The Windows Calculator accessory pops up on your screen, ready for your calculations. By default, S-PLUS waits for the Windows application to complete before returning an S-PLUS prompt. To return to S-PLUS, close the application window.

To run Windows applications concurrently with S-PLUS, so that you can type S-PLUS expressions while the Windows applications are running, use `system` with the `multi=T` argument:

```
> system("cal c", multi=T)
```

The Windows Calculator accessory pops up on your screen, and the S-PLUS prompt immediately appears in your S-PLUS Commands window. (You can, of course, always start Windows applications as usual.)

Commonly used calls should be written into function definitions:

```
calc <- function() { system("cal c", multi=T) }
notepad <- function() {system("notepad", multi=T) }
```

The command argument to `system` can be generated using the `paste` and `cat` functions. For example, the `ed` function, used by both `fix` and `edit` to actually call an editor, pastes together the name of an editor and the name of a file to create the command used by `system`. (The `ed` function distributed with S-PLUS actually uses `win3`, a wrapper for `system`, instead of calling `system` directly. The `win3` function is now deprecated, but it predates `system` on the Windows platform):

```
ed <- function(data, file=tempfile("ed."), editor="ed",
  error.expr)
{
  ...
  system(paste(editor, file), trans = T)
  ...
}
```

The argument `trans=T` is useful for converting strings containing UNIX-type directory names (e.g., `"/betty/users/rich"`) to strings containing DOS-type directory names (e.g., `"\\betty\\users\\rich"`). It can also save you typing, since it allows you to substitute forward slashes for the double backslashes required to represent a single backslash in S-PLUS. If `trans=T`, literal forward slashes must be enclosed in single quotes. For example, Notepad uses the flag `/p` to print a file. To print a file in `infile.txt` in the directory `C:\\RICH`, you could use system as follows:

```
> system("notepad ' /p' c:/rich/infile.txt", trans = T)
```

Note that the single quotes can surround the entire flag, not just the forward slash; in fact one set of quotes can surround all necessary flags.

If you try the above example on one of your own files, you will notice that the Notepad window appears on your screen with the text of the file while Notepad is printing. You can force Notepad to run in a minimized window by using the `mini mi ze=T` argument:

```
> system("notepad ' /p' c:/rich/infile.txt", trans=T,
        mini mi ze=T)
```

There are two arguments to `system` that control how it behaves when an error occurs in starting or exiting the application. The more commonly used is the `on.exec.status` argument, which controls how `system` behaves when an error occurs in starting the application. If the application specified in the `command` argument could not be started, S-PLUS queries the operating system for a character string that briefly describes the error that occurred. It then calls the function specified (as a character string) in the `on.exec.status` argument, passing it the error string. The default for the argument is `"stop"`, so that all current S-PLUS expressions are terminated. For example, if you wanted to run the Wordpad application, but the directory in which it resides is not in your `PATH` environment variable, you would get the following result:

```
> system("wordpad")
Error in system("wordpad"): Unable to execute 'wordpad',
exec.status=2 (the system cannot find the file specified.
Full path needed?)
Dumped
```

Specifying the full path to the Wordpad application successfully starts it:

```
> system("\\Program Files\\Windows NT\\Accessories\\wordpad")
```

You may substitute the name of another function for the `on.exec.status` argument, so long as the function's only required argument is a character string. For example, suppose you wanted to open a file for jotting down some

notes in an ASCII editor, but you weren't particular as to which editor you opened. You could write a `whi teboard` function to call Wordpad, but use `on. exec. status` to try Notepad if Wordpad wasn't available, as follows:

```
> whi teboard
functi on()
{
    system("wordpad", mul ti = T,
          on. exec. status = "tryi ng. notepad")
}
```

The `tryi ng. notepad` function is defined as follows:

```
> tryi ng. notepad
functi on(message = NULL)
{
    print(message)
    print("Tryi ng to start notepad.\n")
    system("notepad", mul ti = T,
          on. exec. status = "tryi ng. edi t")
}
```

As in the initial `whi teboard` function, `tryi ng. notepad` calls `system` with an alternative function as its `on. exec. status` argument. The `tryi ng. edi t` function uses a call to the `dos` function to start the MS-DOS editor, and uses the default `on. exec. status` behavior, that is, if you can't find any of Wordpad, Notepad or MS-DOS editor, `whi teboard` fails.

A less commonly used argument, because most Windows applications do not return a useful or documented exit status, is `on. exi t. status`, which controls how `system` behaves when the application returns a non-zero exit status. The default for this argument is "", so that no action is taken by `system`; you may substitute the name of any function for this argument, again so long as its only required argument is a character string. For example, if you had knowledge that a particular application returned a non-zero exit status when some condition was (or was not) met, you could have this condition reported as follows:

```
> system("myapp", on. exi t. status="my. report")
```

If `myapp` returned a non-zero exit status, the function `my. report` would be called with the argument "'myapp' returned with `exi t. status=n`", where `n` is the exit status value.

Rarely used in the `. 16. bi t` argument; it is used only to attempt running 16-bit applications modally (i.e., with `mul ti =F`) from S-PLUS. Normally, when a 32-bit process such as S-PLUS spawns a 16-bit subprocess, the 32-bit

process cannot be made to wait for the 16-bit subprocess to complete. S-PLUS attempts to get around this problem by using the Universal Thunk API of Win32s. If `multi=F` and S-PLUS is instructed to run an application as a 16-bit process, it will attempt to do so if it can load the DLL specified by the environment variable **S\_WIN32SDLL** (or **kernel32.dll**, if **S\_WIN32SDLL** is not set) and find the symbols `UTRegister` and `UTUnRegister` in that DLL. S-PLUS will automatically run an application as a 16-bit process if it detects that it is running on Win32s (i.e., in Windows 3.x). If S-PLUS detects that it is running Windows 95 or NT, you must specify `.16.bit=T` to get this behavior, and you must have the Win32s subsystem installed (which is why the `.16.bit` argument is rarely useful).

## USING THE DOS INTERFACE

While the Windows interface allows you to run Windows applications from S-PLUS, it cannot be used to run internal DOS commands (such as **dir** and **copy**), nor can it return a command's output as an S-PLUS vector. The DOS interface provides a way to perform these tasks. The DOS interface, however, cannot be used to run Windows applications.

To run internal DOS commands from S-PLUS, use the `dos` function. For example, to get a listing of files in your home directory, use `dos` with the **dir** command as follows:

```
> dos("di r")
[1] ""
[2] " Volume in drive C has no label "
[3] " Volume Serial Number is 6146-07CB"
[4] " Directory of C:\\RICH"
[5] ""
[6] ".                <DIR>      12-07-92  5:01p"
[7] ". .              <DIR>      12-07-92  5:01p"
[8] " __DATA"         <DIR>      12-07-92  5:02p"
[9] "DUMP           Q           74 01-14-93  2:51p"
[10] "WI NWORK    TEX      10053 12-13-92  4:08p"
...
```

By default, the output from the DOS command is returned to S-PLUS as a character vector, one element per line of output. In the case of the **dir** command, the first five to seven lines of output will seldom change. An S-PLUS function that strips off this repetitive information may be of more use than the simple call to `dos`:

```
di r <- function(directory="") {
  dos(paste("di r", directory))[-(1:5)]
}
```

Including the `directory` argument allows using the function to get a listing of an arbitrary directory.

If you don't want the output from the DOS command returned to S-PLUS, use the argument **output.to.S=F** in the call to `dos`:

```
> dos("copy file1 b:", output = F)
```



The output from the DOS command is displayed in a “DOS box”, which in this example will close automatically when the DOS command is completed. As with the `system` function, you can specify `minimize=T` to force the DOS box to run minimized. You can run DOS applications concurrently with S-PLUS by combining `output=F` with the `multi=T` argument. For example, to open the DOS text editor concurrently with S-PLUS, use `dos` as follows:

```
> dos("edit", output = F, multi = T)
```

A DOS box opens on your screen with the DOS text editor loaded.

**Warning**

When you use `dos` with `multi=T`, you must explicitly close the DOS box when you're done with it. It does not go away when the DOS command finishes executing.

Forward slashes can be translated to backslashes using the `trans=T` argument; this can save you typing (since one forward slash equals two backslashes), and is also useful if you are sharing files on a UNIX file system:

```
> dos("edit c:/rich/infile.txt", output = F, trans = T)
```

Two other arguments to `dos` are used less frequently---`input` and `redirection`. The `input` argument can be used to write data to a file that can then be passed as input to the DOS command. More often, however, such data is simply pasted into the command specified by the `command` argument. The `redirection` argument is a flag that can be used with `input`; if `redirection=T`, the input is passed to the command using the DOS redirection operator `<`, otherwise the input is passed as an argument to command. See the `dos help` file for more information.

## PROGRAMMING WITH THE DOS AND WINDOWS INTERFACES

The examples considered so far use the `dos` and `system` functions as simple analogues to the DOS box and Run command, respectively. For common operations, you will want to wrap up the call to `dos` or `system` in an S-PLUS function, as in the `dir` example of the previous section. As another example, consider the DOS **COPY** command. It can be used to build an S-PLUS copy function as follows:

```
copy <- function(source, target)
{
  dos(paste("copy", source, target),
      output = F, translate = T)
}
```

The `copy` function allows you to copy ordinary DOS files; it should not be used for copying S-PLUS objects between databases. For that, use `obj copy`.

Eventually you will want to use a DOS or Windows application to manipulate data created in S-PLUS. S-PLUS functions to perform such tasks are generally much more complicated than the “one-liners” considered so far, but they demonstrate the power of the DOS and Windows interfaces. In the following sections, we offer several comprehensive examples of S-PLUS functions that use the DOS and Windows interfaces to good advantage.

### Functions for Archiving and Restoring Data

Many DOS users have the shareware program **PKZIP**, available from PKWARE, Inc., Brown Deer, WI. The following function, `archive`, uses the S-PLUS `dump` function to create ASCII versions of the objects in an S-PLUS directory, then uses **PKZIP** to compress the resulting files into an archive file that can be stored on your hard drive or a floppy disk (you need to be sure before using this function that **PKZIP** (and its companion **PKUNZIP**) is in your DOS path):

```
archive <- function(file = "spl.us.arc", drive, where = 1)
{
  objectnames <- objects(where)
  filenames <- character(length(objectnames))
  dos("mkdir tmparch")
  on.exit({
    {
```

```

        unlink(filenamees)
        rmdir("tmparch")
    }
)
for (i in seq(along = objectnames)){
    objecti <- objectnames[i]
    dump(objecti, filenamees[i] <- paste("tmparch/",
        true.file.name(objecti), sep = ""))
}
cat(filenamees, file = "filelist", sep = "\n")
dos(paste("pkzip", file, "@filelist"))
if (!missing(drive)){
    dos(paste("copy", file, drive))
    unlink(c(file, "filelist"))
}
else {
    drive <- "C:\\\\"
    unlink("filelist")
}
cat("Created file", file, "on drive", drive, "\n")
}

```

We called `dump` inside a loop, rather than simply dumping all the objects to a single file, because we want to be able to use **PKUNZIP** to extract only selected objects and restore them to S-PLUS. The restoration function, `unarchive`, is shown below:

```

unarchive <- function(file = "spl.us.arc", Subjects, replace
= T)
{
    oddir <- objects()
    if(replace)
        flags <- "-o"
    else flags <- "-o-"
    if(!missing(Subjects)) {
        sfiles <- character(length(Subjects))
        for(i in seq(along = Subjects)) {
            sfiles[i] <- true.file.name(Subjects[i])
        }
        cat(sfiles, file = "objlist", sep = "\n")
        dos(paste("pkunzip", flags, file, "@objlist"))
    }
}

```

```
el se {  
  dos(paste("pkunzip", file, "-@objlist"))  
  dos(paste("pkunzip", flags, file, "@objlist"))  
  sfiles <- scan("objlist", what =character())  
}  
for(i in seq(along = sfiles)) {  
  source(sfiles[i])  
}  
unlink(c(sfiles, "objlist"))  
setdiff(objects(), olddir)  
}
```

In `unarchive`, the argument `Sobjects` can be a character vector of object names (such as the output of `objects`) to be restored from `file`. If provided, this vector is converted to a vector of file names, `sfiles`, that correspond to the names in the archive file. Otherwise, all the files in the archive are extracted; a list of files is obtained using the `-@objlist` construction (this construction works in **PKZIP** version 2 and higher). The `setdiff` function compares two objects and returns the elements of the first object that are not in the second; in `unarchive`, it is used to return a list of restored functions that were added to the working directory. (Restored functions that merely replaced existing functions are not listed.)

## A Matrix Editor Using Microsoft Excel

S-PLUS has a data editor that allows you to edit S-PLUS data objects such as data frames and matrices much as you would change values in a spreadsheet. In this section we show how to use the DOS and Windows interfaces to construct an alternative matrix editor that has, as its engine, the popular spreadsheet Microsoft Excel. (A much more effective Excel interface can be constructed using S-PLUS automation, using the commands described in the Chapter 13, Automation.)

The function presented uses the basic text format for Excel spreadsheets, and can be taken as a model for functions that use pure ASCII data transfer.

When you save a spreadsheet in text format in Microsoft Excel, the resulting text file has one spreadsheet row per line, each column separated by tabs. The following S-PLUS function writes such a text file:

```
write.excel.matrix <- function(data, file=tempfile("ex")) {  
  cat(t(data), file = file,  
      sep = c(rep("\t", ncol(data) - 1), "\n"))  
}
```

We use this function in our `matrix.ed` function. Rather than starting from scratch, we start with the `S-PLUS.ed` function, since it handles many of the details of writing temporary files and error reporting. We use the same arguments (although we supply a different default for the `editor`), and keep the opening statements unchanged. (We use the DOS 8.3 names for the Program Files and Microsoft Office directories; this is the standard location for Excel under Windows 95. You may need to modify this function slightly to specify an alternate path.):

```
matrix.ed <- function(data, file = tempfile("ed."),
  editor = "c:\\progra~1\\micro~1\\office\\excel",
  error.expr)
{
  drop <- missing(file)
  if(missing(data)) {
    if(!exists(".Last.file"))
      stop("Nothing available for re-editing" )
    file <- .Last.file
    data <- .Last.ed
  }
}
```

Our first real work is to save the `dimnames` of our original matrix, and use the auxiliary function `write.excel.matrix` to write the ASCII file:

```
datadn <- dimnames(data)
datadim <- dim(data)
write.excel.matrix(data, file)
```

Some routine error handling is followed by the heart of the function, the call to `system`:

```
if(!missing(error.expr))
  on.exit(error.expr)
else
  on.exit(cat("\nErrors occurred; Use a command
    like:\n", "\tmatrix.ed()\n",
    "to re-edit this object.\n", sep = ""))
on.exit(if(!missing(data)) {
  assign(".Last.ed", data, frame = 0)
  assign(".Last.file", file, frame = 0)
})
, add = T)
system(paste(editor, file), trans = T)
```

After editing the data with Excel, the following reads the saved text file back into S-PLUS:

```
data <- matrix(scan(file), ncol = datadim[2]), byrow = T)
```

Some routine cleanup follows, and then the saved dimnames are restored to the data, and the edited data are returned:

```
if(drop) {  
  unlink(file)  
  if(missing(data)) {  
    remove(".Last.ed", frame = 0)  
    remove(".Last.file", frame = 0)  
  }  
}  
on.exit(, add = F)  
dimnames(data) <- datadn  
data  
}
```

---

# INTERFACING WITH C AND FORTRAN CODE

# 16

---

<b>Overview</b>	<b>561</b>
<b>A Simple Example: Filtering Data</b>	<b>563</b>
<b>Using the C and Fortran Interfaces</b>	<b>566</b>
When Should You Consider the C or Fortran Interface?	566
Reasons for Avoiding C or Fortran	566
<b>Calling C or Fortran Routines From S-PLUS</b>	<b>568</b>
Calling C	568
Calling Fortran	569
<b>Writing C and Fortran Routines Suitable for Use in S-PLUS</b>	<b>572</b>
Handling IEEE Special Values	572
Allocating Memory	575
I/O in C Functions	576
I/O in Fortran Subroutines	577
Reporting Errors and Warnings	577
Generating Random Numbers	582
Returning Variable-Length Output Vectors	583
Calling Fortran From C	585
Calling C From Fortran	587
Calling S-PLUS Functions From C Code	587
<b>Using Dynamic Link Libraries (DLLs)</b>	<b>594</b>
Creating a DLL From C Source Code	594
Loading and Running the Code in the DLL	598
Unloading the DLL	600
Calling Functions in the S-PLUS Engine DLL	600
<b>Compiling and Loading Watcom Object Code</b>	<b>603</b>
Setting Up the Loading Environment	603
Using COMPILE to Create Watcom Object Files	605
Loading Watcom Object Files Using dyn.load	607
Statically Loading Watcom Object Code Using LOAD	608
Solving Problems With Static or Dynamic Loading	612

<b>Debugging Loaded Code</b>	<b>616</b>
Debugging C Code	616
Source-Level Debugging	618
<b>A Note on StatLib</b>	<b>622</b>



---

## OVERVIEW

A powerful feature of S-PLUS is that it allows you to extend its functionality, enabling you to *interface* with other languages, namely, C and Fortran. Interfaces to other languages allow you to combine the speed and efficiency of compiled code with the robust, flexible programming environment of S-PLUS. Your compiled routines can be loaded into S-PLUS using one of the following methods:

- *Dynamic loading.* Your object code or dynamic link library (DLL) is loaded dynamically, that is, while S-PLUS is running. There are two ways to load code dynamically:
  - *Use the `dll.load` function to load a DLL.* This is the preferred way to use compiled code within S-PLUS under Microsoft Windows. The advantage of DLL loading is that you are not restricted to using a Watcom 32-bit compiler. You may use any compiler that can produce a DLL with a 32-bit entry point, including Microsoft Visual C/C++ and Borland C/C++. Another advantage is that you can interface to other DLLs, including the S-PLUS engine DLL and the Windows API, and use their functions. Dynamic loading using `dll.load` is discussed in the section Using Dynamic Link Libraries (DLLs) (page 594).
  - *Use the `dyn.load` function to load object code.* The `dyn.load` function allows you to dynamically load object code compiled with a Watcom 32-bit compiler (see Release Notes for information on Watcom version compatibility). The advantage of object-code loading over DLL loading is that is simpler to create object code than a DLL, and `dyn.load` is somewhat simpler to use than `dll.load`. Object code compiled with Watcom also has greater compatibility with C library routines used in the S-PLUS engine DLL (up to 100% compatibility depending upon your version of the Watcom compiler). The disadvantage, for most users, is that the Watcom compilers used to create S-PLUS are often not available to the general public by the time S-PLUS ships. However, object code created with a newer version of the Watcom compiler is usually compatible with S-PLUS; unfortunately, MathSoft cannot guarantee this. Dynamic loading with `dyn.load` is discussed in the section Loading Watcom Object Files Using `dyn.load` (page 607).

- *Static loading.* S-PLUS is relinked to include your object code in a new S-PLUS engine DLL. You must have a Watcom 32-bit compiler to do this (see online Release Notes for Watcom version compatibility information). To statically load code, use the `LOAD` utility. In general, we discourage static loading, except when you are creating a system-wide copy of S-PLUS with site-specific modifications that you want all users to be able to access at all times. Static loading is discussed in the section *Statically Loading Watcom Object Code Using LOAD* (page 608).

After loading the compiled routines, use the `.C()` and `.Fortran()` functions to call compiled routines, directly from S-PLUS.

This chapter describes how to do the following tasks:

- Decide when and where to use compiled code.
- Call compiled routines from S-PLUS.
- Write C and Fortran routines suitable for use in S-PLUS.
- Create a dynamic link library (DLL).
- Use `dll.load()` to load a DLL.
- Use the librarian to build an import library to call functions in the S-PLUS engine DLL.
- Use **COMPILE** to create Watcom object files from source code.
- Use `dyn.load()` to dynamically load Watcom object code.
- Use **LOAD** to statically load the Watcom object code.
- Solve problems you may encounter with static loading or dynamic loading.
- Debug your compiled code.

Each of these tasks can become quite complicated, so we begin with a simple example that shows the basic flavor of writing, compiling, and using compiled code.

## A SIMPLE EXAMPLE: FILTERING DATA

In this section, we develop a function to apply a first order linear recursive filter to a vector of data. The S-PLUS function `filter` does what we want, but we'll ignore it for now in favor of the following pure S code:

```
Ar <- function(x, phi)
{
  n <- length(x)
  if (n>1)
    for (i in 2:n)
      x[i] <- phi * x[i - 1] + x[i]
  x
}
```

Looping is traditionally one area where S-PLUS tends to be significantly slower than compiled code, so we can rewrite the above code in C as follows, creating a file **Ar.c**:

```
void arsim(double *x, long *n, double *phi)
{
  long i;
  for (i=1; i<*n; i++)
    x[i] = *phi * x[i-1] + x[i];
}
```

This code is purely C language code; there are no dependencies on C libraries, or on S-PLUS, or on the Windows API. Such code should be portable to most operating systems. It is quite simple to create a DLL from this code using Visual C++:

1. Create the file **Ar.c** shown above.
2. Create a module definition file **ar.def** containing the following lines:

```
; *****
; ar.def module definition file
; *****

LIBRARY ar

EXPORTS

    arsim
```

3. Start Visual C++ (Microsoft Developer Studio), and from the File menu, select New.
4. From the New dialog, select Project Workspace.

5. In the Project Workspace dialog, specify a name for the project (such as “ar”), and for Project Type choose “Dynamic Link Library” (*not* MFCAppWizard (dll)).
6. From the Insert menu, choose Files into Project. Add `Ar.c` and `ar.def` to your project. To insert the `.def` file, use the Files of Type dropdown and select Definition files (.def).
7. From the Build menu, choose Rebuild All.

Visual C++ will build your DLL, in either the Debug or Release subdirectory beneath your project workspace (depending upon whether you select “Debug” or “Release” in the Default Project Configuration dialog accessed from Build/Set Default Configuration...). You can then load the DLL in S-PLUS as follows:

```
> dll.load("c:\\ar\\debug\\ar.dll", "arsi m", "cdecl ")
[1] 1
```

The call to `dll.load` returns 1 if the DLL is loaded successfully or 0 if it is already loaded. To run the filtering code, we can either call `.C` directly, or we can write a simple S function to do it for us. If we want to use our loaded call very often, it will save us time to define the function:

```
ar.compiled <-
function(x, phi)
{
  .C("arsi m",
     as.double(x),
     length(x),
     as.double(phi))[[1]]
}
```

Trying the code with a call to `ar.compiled` yields the following:

```
> ar.compiled(1:20, .75)
[1] 1.000000 2.750000 5.062500 7.796875 10.847656
[6] 14.135742 17.601807 21.201355 24.901016 28.675762
[11] 32.506822 36.380116 40.285087 44.213815 48.160362
[16] 52.120271 56.090203 60.067653 64.050739 68.038055
```

You lose some flexibility in the function by writing it in C. Our `ar.compiled` function converts all input data to double precision, so it won't work correctly for complex data sets nor objects with special arithmetic methods. The pure S-PLUS version works for all these cases. If complex data is important for your application, you could write C code for the complex case and have the S-PLUS code decide which C function to call. Similarly, to make `ar.compiled` work for data in classes with special arithmetic methods, you could have it call the C code only if the data were classless, so that it

could not invoke special arithmetic methods. This might be too conservative, however, as there could be many classes of data without arithmetic methods which could use the fast C code.

Another approach would be to make `ar.compiled` a generic function, for which the default method calls the C code for classless data. For classes with special arithmetic methods, pure S-PLUS code could be dispatched. Those classes of data without special arithmetic methods could include an `ar.compiled.class` method that would remove the class attribute and invoke the default method on the now classless data, thus using the fast compiled code, then postprocess the result if needed (perhaps just restoring the class). Using the object-oriented approach is more work to set up, but gives you the chance to combine the speed of compiled code with the flexibility of S-PLUS code.

## USING THE C AND FORTRAN INTERFACES

The key to effective use of compiled code is knowing when and when *not* to use such code. The following subsections provide some criteria for deciding whether compiled code is the right choice for your situation, and outline the basic procedure for using compiled code in S-PLUS.

### When Should You Consider the C or Fortran Interface?

Compiled C or Fortran code runs faster than interpreted S-PLUS code, but is neither as flexible nor as resilient as equivalent S-PLUS code. Mismatching data types and overrunning arrays are just two types of errors that can occur in compiled code but do not occur in S-PLUS code. The best time to use compiled code is when you have such code already written and tested. Another good time to use compiled code is when you cannot use S-PLUS's vectorized functions to solve your problem without explicit loops or recursion. Recursion in S tends to be very memory intensive; simulations that work for small cases may fail as the number of iterations rises. If the iterated computation is trivial, you can realize huge performance gains by moving that portion of the calculation to compiled code.

### Reasons for Avoiding C or Fortran

Compiled code deals only with data types fixed when the code is compiled (S-PLUS *modes* correspond to C or Fortran *types*) and the C and Fortran interfaces pass only the most basic modes, the numeric ones and character data. If your code does something numerical, it may be fine to convert all the inputs to double precision and return double precision results. If your code rearranges data, however, you probably don't want to change the modes of the data, so S-PLUS code would be better than compiled code. The C and Fortran interfaces ignore the class of datasets, so they are not object oriented.

It is usually harder to develop and debug compiled code than S-PLUS functions. With compiled code, you must make sure not only that the compiled code works, but also that the S-PLUS function that calls it works and is compatible with the compiled code.

Compiled code is usually not as portable as S-PLUS code. Other users who would like to use your code may not have the appropriate compilers or the compilers on other machines may not be compatible with one another. Your code may also depend upon certain libraries that others may not have.

A good strategy is to do as much as possible in S-PLUS code, including error checking, data rearrangements, selections and conversions, storage allocation, and input/output, and use compiled code to do only the numerical or

character string calculations required. When developing new functions in S-PLUS, you should probably write the entire function in S-PLUS code first. Then, if the pure S-PLUS version is too slow or memory intensive (and you expect it to be used a lot), look for bottlenecks and rewrite those parts in C.

## CALLING C OR FORTRAN ROUTINES FROM S-PLUS

### Calling C

To call a C function, use the S-PLUS function `.C()`, giving it the name of the function (as a character string) and one S-PLUS argument for each C argument. For example, a typical “vectorized” calculation, such as sine, requires you to pass an S-PLUS data object `x` and its length `n` to the C function performing the calculation:

```
.C("my_si_n_vec", x = as.double(x),
  n = as.integer(length(x)))
```

To return results to S-PLUS, modify the data pointed to by the arguments. The value of the `.C()` function is a list with each component matching one argument to the C function. If you name these arguments, as we did in the preceding example, the return list has named components. Your S-PLUS function can use the returned list for further computations or to construct its own return value, which generally omits those arguments which are not altered by the C code. Thus, if we wanted to just use the returned value of `x`, we could call `.C()` as follows:

```
.C("my_si_n_vec", x = as.double(x),
  n = as.integer(length(x)))$x.
```

All arguments to C routines called via `.C()` must be pointers. All such routines should be `void` functions; if the routine does return a value, it could cause S-PLUS to crash. S-PLUS has many data modes (types) that are not immediately representable in C. To simplify the interface between S-PLUS and C, the types of data that S-PLUS can pass to C code are restricted to the following S-PLUS *storage modes*: "single", "integer", "double", "complex", "logical", and "character". The following table shows the correspondence between S-PLUS modes and C types.

S-PLUS storage mode	Corresponding C type
"logical"	long *
"integer"	long *
"single"	float *
"double"	double *
"complex"	struct{double re, im;} *
"character"	char **



"name"	char *
"list"	void **

**Warning**

Passing data to compiled code is the one time you must be very specific about whether you want the mode "single", "double", or "integer"—specifying data to be "numeric" is not good enough. For example, if our routine `my_sin_vec()` expects an argument of type `double`, we need to ensure that the data passed from S-PLUS has the appropriate storage mode before calling `.C()`, by using `as.double()` as shown in the example. Equivalently, we could change the storage mode to `double` in our S-PLUS function before calling `.C()`.

**Warning**

Do *not* declare integer data as `C ints`, particularly if you want your code to be portable among machines that S-PLUS supports. While there is currently no difference on Windows, there is a distinction on other platforms.

S-PLUS does not supply a way to pass hierarchical structures between S-PLUS and your compiled code. The table above says that the S-PLUS mode "list" is passed as a C type "void \*\*", meaning that you cannot do anything with it except pass it to the C routine `call_S()`, supplied with S-PLUS, that knows how to interpret it. This means that you must use S-PLUS code to encode a hierarchical data structure as a set of vectors of numbers or character strings and decode that in your C code (and do the reverse for passing the data back from compiled code to S-PLUS). For instance, compiled code to work on a matrix must be given both the data in the matrix and its dimensions; you cannot give just the S-PLUS matrix and get the dimensions out of it using C. Functions dealing with trees or more general graphs must use fancier forms of encoding, for instance a vector of vertex values and a pair of vectors describing the endpoints of the edges (and perhaps another vector containing the weights for the edges). In addition, there are several C integral types that S-PLUS cannot directly represent, such as 1 and 2 byte integers and the unsigned types. Any C code called directly from S-PLUS must accept `C longs` and convert them to the type required.

**Calling Fortran**

To call a Fortran subroutine, use the S-PLUS function `.Fortran()`, giving it the name of the subroutine (as a character string) and one S-PLUS argument for each Fortran argument. For example, a typical "vectorized" calculation,

such as sine, requires you to pass an S-PLUS data object  $x$  and its length  $n$  to the Fortran subroutine performing the calculation:

```
.Fortran("my_sin_vec", x = as.double(x),
        n = as.integer(length(x)))
```

**Note**

You can call only Fortran *subroutines* from S-PLUS; you cannot call Fortran *functions*.

To return results to S-PLUS, modify the data pointed to by the arguments. The value of the `.Fortran()` function is a list with each component matching one argument to the Fortran subroutine. If you name these arguments, as we did in the preceding example, the return list has named components. Your S-PLUS function can use the returned list for further computations or to construct its own return value, which generally omits those arguments which are not altered by the Fortran code. Thus, if we wanted to return just the object  $x$ , we could call `.Fortran()` as follows:

```
.Fortran("my_sin_vec", x = as.double(x), n =
as.integer(length(x)))$x
```

S-PLUS has many data modes (types) that are not immediately representable in Fortran. To simplify the interface between S-PLUS and Fortran, the types of data that S-PLUS can pass to Fortran code are restricted to the following S-PLUS storage modes: "single", "integer", "double", "complex", "logical", and "character". The following table shows the correspondence between S-PLUS modes and Fortran types.

S-PLUS storage mode	Corresponding FORTRAN type
"logical"	LOGICAL
"integer"	INTEGER
"single"	REAL
"double"	DOUBLE PRECISION
"complex"	DOUBLE COMPLEX
"character"	CHARACTER(*)

**Warnings**

S-PLUS will not pass arrays of character strings to Fortran routines; only the first element.

The Fortran type `DOUBLE COMPLEX` (or `COMPLEX*16`) is a complex number made of double precision parts; it may not be available with all Fortran compilers, but it is available in the Watcom Fortran compilers.

Passing data to compiled code is the one time you must be very specific about whether you want the mode "single", "double", or "integer"—specifying data to be "numeric" is not good enough. For example, if our routine `my_sin_vec()` expects data of type double, we need to ensure that the data passed from S-PLUS has the appropriate storage mode before calling `.Fortran()`, by using `as.double()` as shown in the example. Equivalently, we could change the storage mode to double in our S-PLUS function before calling `.Fortran()`.

## WRITING C AND FORTRAN ROUTINES SUITABLE FOR USE IN S-PLUS

While the actual calls to `.C()` and `.Fortran()` are straightforward, you may encounter problems loading new compiled code into S-PLUS and we will discuss some common problems. We also describe some C procedures and macros which you may use to write more portable code, to generate random numbers from C code, to call S-PLUS functions from your C code, to report errors, to allocate memory, and to call Fortran procedures from C code.

In order to have access in C to most functions and macros described below, you will have to include the header file **S.h** in your source files:

```
#include <S.h>
```

and make sure that you specify the `%S_HOME%\include` directory in your compiler directives. That directory is specified automatically by the **COMPILE** utility (for use with Watcom compilers only). If you will be using any Windows API calls in your code, so that you need to include **windows.h**, include **windows.h** first, then **S.h** and any other include files you need.

### Handling IEEE Special Values

S-PLUS handles IEEE special values such as NaN, `Inf` or `-Inf`, for all supported storage modes (integer, single or double). NaN represents the number you obtain when you divide 0 by 0. `Inf` represents the number you obtain when you divide 1 by 0. `-Inf` represents the number you obtain when you divide -1 by 0. In addition, S-PLUS supports NA, which represents a missing value, i.e., a value to use when none is available. S-PLUS functions attempt to properly handle computations when missing values are present in the data. Both NaN and NA are displayed as NA, but the data values are properly kept as different values.

The `.C()` and `.Fortran()` functions have two arguments, the `NAOK` and the `specialok` argument, that you can use to specify whether your code can handle missing values or IEEE special values (`Inf` and `NaN`), respectively. Their default value is `FALSE`: if any argument to `.C()` or `.Fortran()` contains an NA (or `Inf` or `NaN`), you get an error message and your code is not

called. To specify these arguments, you must use their complete names, and you cannot use these names for the arguments passed to your C or Fortran code.

### Warning

The NAOK and special sok arguments refer to all of the arguments to your compiled code—you can allow NA's or IEEE special values in all of the arguments or none of them. Since typically you don't want NA's for certain arguments, such as the length of a data set, you must specially check those arguments if you use NAOK=T (or special sok=T).

Dealing with IEEE special values is easily done in C as long as you use the macros described below. It is possible, yet undocumented here, to do the same in Fortran, but refer to your Fortran compiler documentation for details.

It is often simplest to remove NA's from your data in the S-PLUS code, but is sometimes better done in C. If you allow NA's, you should deal with them using the C macros `is_na()` and `na_set()` described below. The arguments to `.C()` and `.Fortran()` cannot contain any NA's unless the special argument NAOK is T. The following macros test for and set NA's in your C code:

```
is_na(x, mode)
```

```
na_set(x, mode)
```

The argument `x` must be a pointer to a numeric type and the argument `mode` must be one of the symbolic constants LGL (S-PLUS mode "logical"), INT (S-PLUS mode "integer"), REAL (S-PLUS mode "single"), DOUBLE, or COMPLEX, corresponding to the type `x` points to: long, long, float, double, or complex, respectively. For example, the following C code sums a vector of double precision numbers, setting the sum to NA if any addends are NA:

```
#include <S.h>
void my_sum(double *x, long *n, double *sum) {
    long i;
    *sum = 0;
    for (i = 0; i < *n; i++)
        if (is_na(x[i], DOUBLE)) {
            na_set(sum, DOUBLE);
            break;
        }
    else
        *sum += x[i];
}
```

Use the following S-PLUS function to call this routine:

```
> my.sum <- function(x) .C("my_sum", as.double(x),
                           as.integer(length(x)),
                           double(1), NAOK = T)[[3]]
```

Call this from S-PLUS as follows:

```
> my.sum(c(1, NA, 2))
[1] NA
> my.sum(1:4)
[1] 10
```

If you omit the argument `NAOK=T` in the call to `.C()`, you get the following message:

```
> my.sum <- function(x)
  .C("my_sum", as.double(x),
      as.integer(length(x)), double(1))[[3]]
> my.sum(c(1, NA, 2))
Error in .C("my_sum", : subroutine my_sum: 1 missing
value(s) in argument 1
Dumped
```

### Warning

Both `is_na()` and `na_set()` have arguments that may be evaluated several times. Therefore don't use expressions with side effects in them, such as `na_set(x[i++], DOUBLE)`. Otherwise, the side effects may occur several times. The call `is_na(x, mode)` returns 0 if `*x` is not an NA and non-zero otherwise—the non-zero value is not necessarily 1. The return value tells what sort of value `*x` is: `IS_NA` meaning a true NA and `IS_NaN` meaning an IEEE not-a-number. To assign a NaN to a value, use the alternative macro `na_set3(x, mode, type)`, where `type` is either `IS_NA` or `IS_NaN`. The macro `na_set(x, mode)` is defined as `na_set3(x, mode, IS_NA)`.

You can use the macros `is_inf(x, mode)` and `inf_set(x, mode, sign)` to deal with IEEE infinities. If you allow IEEE special values, your code should be aware that `x != x` is TRUE if `x` is a NaN. In any case you should be aware that on machines supporting IEEE arithmetic (that includes most common workstations), `1/0` is `Inf` and `0/0` is NaN without any warnings given. You must set the `.C()` argument `special sok` to T if you want to let S-PLUS pass NaN's or Inf's to your C code. The call `is_inf(x, mode)` returns 0 if `*x` is not infinite and  $\pm 1$  if `*x` is  $\pm \infty$ , respectively. The call `set_inf(x, mode, sign)` sets `*x` to an infinity of the given mode and sign, where the sign is specified by the integer +1 for positive infinities and -1 for negative infinities.

## Allocating Memory

S-PLUS includes two families of C routines for storage allocation and reallocation. You can use either of these families, or use the standard library functions `malloc()`, `calloc()`, `realloc()`, and `free()`. However, be very careful to use only *one* family for any particular allocation; mixing calls using the same pointer variable can be disastrous. The first S-PLUS family consists of the two routines `S_malloc()` and `S_realloc()`, which may be used instead of the standard `malloc()` and `realloc()`. The storage they allocate lasts until the current evaluation frame goes away (at the end of the function calling `.C()`) or until memory compaction in an S-PLUS loop reclaims it. If space cannot be allocated, `S_malloc()` and `S_realloc()` perform their own error handling; they will not return a NULL pointer. You cannot explicitly free storage allocated by `S_malloc()` and `S_realloc()`, but you are guaranteed that the storage is freed by the end of the current evaluation frame. (There is no `S_free()` function, and using `free()` to release storage allocated by `S_malloc()` will cause S-PLUS to crash.) `S_malloc()` and `S_realloc()` are declared a bit differently from `malloc()` and `realloc()` (although `S_malloc` has many similarities to `calloc()`—for example, it zeroes storage and has two arguments). `S_malloc()` is declared as follows in **S.h**:

```
char * S_malloc(long n, int size);
```

Similarly, `S_realloc()` is declared as follows in **S.h**:

```
char * S_realloc(char *p, long new, long old, int size);
```

`S_malloc()` allocates (and fills with 0's) enough space for an array of `n` items, each taking up `size` bytes. For example, the following call allocates enough space for ten doubles:

```
S_malloc(10, sizeof(double))
```

`S_realloc()` takes a pointer, `p`, to space allocated by `S_malloc()` along with its original length, `old`, and `size, size`, and returns a pointer to space enough for `new` items of the same size. For example, the following expands the memory block size pointed to by `p` from 10 doubles to 11 doubles, zeroing the 11th double location:

```
S_realloc(p, 11, 10, sizeof(double))
```

The contents of the original vector are copied into the beginning of the new one and the trailing new entries are filled with zeros. You must ensure that `old` and `size` were the arguments given in the call to `S_malloc()` (or a previous call to `S_realloc()`) that returned the pointer `p`. The new length should be longer than the old. As a special case, if `p` is a NULL pointer (in which case `old` must be 0L), then `S_realloc()` acts just like `S_malloc()`.

The second S-PLUS family of allocation routines consists of the three macros `CalLoc()`, `RealLoc()`, and `Free()`; note the capitalization. `CalLoc()` and `RealLoc()` are simple wrappers for `calloc()` and `realloc()` that do their own error handling if space can not be allocated (they will not return if the corresponding wrapped function returns a NULL pointer). `Free()` is a simple wrapper for `free()` that sets its argument to NULL. As with `calloc()`, `realloc()`, and `free()`, memory remains allocated until freed—this may be before or after the end of the current frame.

### Warning

If you use `malLoc()` or `realLoc()` directly, you must free the allocated space with `Free()`. Similarly, when using `CalLoc()` or `RealLoc()`, you must free the allocated space with `Free()`. Otherwise, memory will build up, possibly causing S-PLUS to run out of memory unnecessarily. However, be aware that because S processing may be interrupted at any time (e.g., when the user hits the interrupt key or if further computations encounter an error and dump), it is sometimes difficult to guarantee that the memory allocated with `malLoc()` or `realLoc()` (or `CalLoc()` or `RealLoc()`) is freed.

### Note

If, in a call to `S_malloc()`, `S_realloc()`, `CalLoc()` or `RealLoc()`, the requested memory allocation cannot be obtained, those routines call `RECOVER()`. See the section Reporting Errors and Warnings (page 577) for more information on the `RECOVER()` macro.

## I/O in C Functions

File input and output is fully supported in C code called from S-PLUS, but input and output directed to the standard streams `STDIN`, `STDOUT`, and `STDERR` requires special handling. This special handling is provided by the header file **`newredef.h`**, which must be included after **`S.h`** in any code in which you plan to use stream I/O. For example, if you use the `printf()` function to add debugging statements to your code, you must include **`newredef.h`** to ensure that your messages appear in an S-PLUS GUI window rather than simply disappear.

The **`newredef.h`** file does not support `scanf()`; if you need to read user input from the GUI, use `fgets()` to read a line then use `sscanf()` to interpret the line.



## I/O in Fortran Subroutines

Fortran users cannot use any Fortran `WRITE` or `PRINT` statements since they conflict with the I/O in S-PLUS. Therefore, S-PLUS provides the following three subroutines as analogs of the S-PLUS `cat` function:

<code>DBLEPR</code>	Prints a double precision variable
<code>REALPR</code>	Prints an real variable
<code>INTPR</code>	Prints an integer variable

As an example of how to use them, here is a short Fortran subroutine for computing the net resistance of 3 resistors connected in parallel:

```

      SUBROUTINE RESIS1(R1, R2, R3, RC)
C      COMPUTE RESISTANCES
      RC = 1.0/(1.0/R1 + 1.0/R2 + 1.0/R3)
      CALL REALPR('First Resistance', -1, R1, 1)
      RETURN
      END

```

The second argument to `REALPR` specifies the number of characters in the first argument; the -1 can be used if your Fortran compiler inserts null bytes at the end of character strings. The fourth argument is the number of values to be printed.

Here is an S-PLUS function that calls `RESIS1`:

```

> parallel <-function(r1,r2,r3) {
  .Fortran("resis1", as.single(r1), as.single(r2),
    as.single(r3), as.single(0))[[4]]
}

```

Running `parallel` produces the following:

```

> parallel(25, 35, 75)
First Resistance
[1] 25
[1] 12.2093

```

## Reporting Errors and Warnings

S-PLUS provides two functions, `stop` and `warning`, for detecting and reporting error and warning conditions. In most cases, you should try to detect errors in your S-PLUS code, before calling your compiled code. However, S-PLUS does provide several tools to aid error reporting in your compiled code.

## C Functions

The include file **S.h** defines macros that make it easy for your C code to generate error and warning messages. The **PROBLEM** and **RECOVER** macros together work like the S-PLUS function **stop**:

```
PROBLEM "format string", arg1, ..., argn
RECOVER(NULL_ENTRY)
```

The **PROBLEM** and **WARNING** macros together work like the **warning** function:

```
PROBLEM "format string", arg1, ..., argn
WARNING(NULL_ENTRY)
```

The odd syntax in these macros arises because they are wrappers for the C library function **sprintf()**; the **PROBLEM** macro contains the opening parenthesis and the **RECOVER** and **WARNING** macros both start with the closing parenthesis. The format string and the other arguments must be arguments suitable for the **printf()** family of functions. For example, the following C code fragment:

```
#include <S.h>
double x ;
...
if (x <= 0)
    PROBLEM "x should be positive, it is %g", x
    RECOVER(NULL_ENTRY) ;
```

is equivalent to the S-PLUS code:

```
if (x<=0) stop(paste("x should be positive, it is", x))
```

Both print the message and exit all of the currently active S-PLUS functions calls. S-PLUS then prompts you to try again. Similarly, the C code:

```
#include <S.h>
double x ;
...
if (x <= 0)
    PROBLEM "x should be positive, it is %g", x
    WARNING(NULL_ENTRY) ;
```

is equivalent to the S-PLUS code:

```
if (x<=0) warning(paste("x should be positive, it is", x))
```

### Warning

The messages are stored in a fixed length buffer before printing, so you should not try to generate huge error messages. The buffer length is given by `ERROR_BUF_LENGTH` in **S.h** and is currently 4096 bytes. If your message exceeds this length, S-PLUS is likely to crash.

### Fortran Subroutines

Many of the I/O statements encountered in a typical Fortran routine arise in error handling—when the routine encounters a problem, it writes a message.

A previous section proposed using `DBLEPR`, `REALPR`, and `INTPR` for any necessary printing. An alternative approach in S-PLUS is to use the Fortran routines `XERROR` and `XERRWV` for error reporting, in place of explicit `WRITE` statements. For example, consider again the Fortran routine `RESIS1`, which computes the net resistance of 3 resistors connected in parallel. A check for division by 0 is appropriate, using `XERROR`:

```

      SUBROUTINE RESIS1(R1, R2, R3, RC)
C      COMPUTE RESISTANCES
      IF (R1 .EQ. 0 .OR. R2 .EQ. 0 .OR. R3 .EQ. 0) THEN
        CALL XERROR( ' Error : division by 0'
+LEN(' Error : division by 0' ), 99, 2)
        RETURN
      END IF
      RC = 1.0/(1.0/R1 + 1.0/R2 + 1.0/R3)
      CALL REALPR(' First Resistance', -1, R1, 1)
      RETURN
      END

```

`XERROR` takes four arguments: a character string message, an integer giving the length of the string in message, an error number (which must be unique within the routine), and an error level. If message is a quoted string, the length-of-message argument can be given as `LEN(message)`.

The `XERRWV` routine acts like `XERROR` but also allows you to print two integer values, two real values, or both.

The first four arguments to `XERRWV`, like the first four arguments to `XERROR`, are the message, the message length, the error ID, and the error level. The fifth and eighth arguments are integers in the range 0–2 that indicate, respectively, the number of integer values to be reported and the number of real (single precision) values to be reported. The sixth and seventh arguments hold the integer values to be reported, the ninth and tenth arguments hold the real values to be reported.

In the following call to `XERRWV`, the fifth argument is 1, to indicate that one integer value is to be reported. The sixth argument says that `n` is the integer to be reported:

```
XERRWV(MSG, LMSG, 1, 1, 1, n, 0, 0, 0.0, 0.0)
```

The following Fortran subroutine, `test.f`, shows a practical application of `XERRWV`:

```
subroutine test(x, n, ierr)
  real*8 x(1)
  integer n, ierr, LMSG
  character*100 MSG
  ierr = 0
  if (n.lt.3) then
    MSG = ' Integer (I1) should be greater than 2'
    LMSG = len(' Integer (I1) should be greater than 2' )
    CALL XERRWV(MSG, LMSG, 1, 1, 1, n, 0, 0, 0.0, 0.0)
    ierr = 1
    return
  endif
  do 10 i = 2, n
10  x(1) = x(1) + x(i)
  return
end
```

```
> .Fortran("test", as.double(1:2), length(1:2))
[[1]]:
[1] 1 2
[[2]]:
[1] 2
Warning messages:
1: Integer (I1) should be greater than 2 in:
   .Fortran("test", ....
2: in message above, i1=2 in:
   .Fortran("test", ....
```

The error message is duplicated because our S-PLUS code interprets the error status from the Fortran. The messages issued by `XERROR` and `XERRWV` are stored in an internal message table. S-PLUS provides several functions you can use to manipulate this message table within functions that call Fortran routines using `XERROR` and `XERRWV`:

`xerror.summary` prints out the current state of the internal message summary table, listing the initial segment of the message, the error number, the severity level, and the repetition count for each message.

<code>xerror.clear</code>	clears the message table. This function takes an optional argument, <code>doprint</code> . If <code>doprint=T</code> , the message table is printed before it is cleared.
<code>xerror.maxpr</code>	limits the number of times any one message is queued or printed. The default is 10.

For example, we can rewrite our S-PLUS test function to take advantage of these functions as follows:

```
test <- function(x)
{
  xerror.clear()
  val <- .Fortran("test",
                  as.double(x),
                  length(x),
                  err = integer(1))
  if(options()$warn == 0)
    xerror.summary()
  val[[1]][1]
}
```

Calling it as before (after setting the option `warn` to 0) yields the following result:

```
> test(1:2)
      error message summary
message start                nerr level count
Integer (I1) should be greater than 2      1      1      1
other errors not individually tabulated = 0

[1] 1
Warning messages:
1: Integer (I1) should be greater than 2 in:
  .Fortran("test", ....
2: in message above, i1 = 2 in:
  .Fortran("test", ....
```

See the `xerror` help file for more information on the S-PLUS functions used with `XERROR`, and the `XERROR` help file for more information on `XERROR` and `XERRWV`.

## Generating Random Numbers

S-PLUS includes user-callable C routines for generating standard uniform and normal pseudo-random numbers. It also includes procedures to get and set the permanent copy of the random number generator's seed value. The following routines (which have no arguments) each return one pseudo-random number:

```
double uni_f_rand(void);
```

```
double norm_rand(void);
```

Before calling either function, you must get the permanent copy of the random seed from disk into S-PLUS (which converts it to a convenient internal format) by calling `seed_in((long *)NULL)`. You can specify a particular seed using `setseed(long *seed)`, which is equivalent to the S-PLUS function `set.seed`. When you are finished generating random numbers, you must push the permanent copy of the random seed out to disk by calling `seed_out((long *)NULL)`. If you do not call `seed_in()` before the random number generators, they fail with an error message. If you do not call `seed_out()` after a series of calls to `uni_f_rand()` or `norm_rand()`, the next call to `seed_in()` retrieves the same seed as the last call and you get the same sequence of random numbers again. The seed manipulation routines take some time so we recommend calling `seed_in()` once, then calling `uni_f_rand()` or `norm_rand()` as many times as you wish, then calling `seed_out()` before returning from your C function. A simple C function to calculate a vector of standard normals is implemented as follows:

```
#include <S.h>
my_norm(double *x, long *n) {
    long i;
    seed_in((long *) NULL);
    for (i=0 ; i<*n ; i++)
        x[ i ] = S_DOUBLEVAL(norm_rand());
    seed_out((long *) NULL);
}
```

To call it from S-PLUS, define the function `my.norm` as follows:

```
my.norm <- function(n)
    .C("my_norm", double(n), as.integer(n))[[1]]
```

Of course it is simpler and safer to use the S-PLUS function `rnorm` to generate a fixed number of normal variates to pass into an analysis function. We recommend that you generate the random variates in C code only when you cannot tell how many random variates you will need, as when using a rejection method of generating non-uniform random numbers.

### Warning

Because of differences in the way Watcom C/C++ and Microsoft Visual C++ handle return values from floating point functions, the example above uses the `S_DOUBLEVAL` macro (defined when `S.h` is included). The `S_DOUBLEVAL` or `S_FLOATVAL` macros, defined in **compiler.h**, may be needed when calling floating point functions internal to S-PLUS from DLLs compiled with other non-Watcom compilers; see the section Calling Functions in the S-PLUS Engine DLL (page 600).

## Returning Variable-Length Output Vectors

Occasionally, we do not know how long the output vector of a procedure is until we have done quite a bit of processing of the data. For example, we might want to read all the data in a file and produce a summary of each line. Until we have counted the lines in the file, we don't know how much space to allocate for a summary vector. Generally, `.C` passes your C procedure a pointer to a data vector allocated by your S-PLUS function so you must know the length ahead of time. You could write two C procedures: one to examine the data to see how much output there is and one to create the output. Then you could call the first in one call to `.C`, allocate the correct amount of space, and call the second in another call to `.C`. The first could even allocate space for the output vector as it is processing the input and have the second simply copy that to the vector allocated by your S-PLUS function. Because this process can get clumsy, both the `.C` and `.Fortran` functions have an optional argument, `pointers`, to help you out.

The `pointers` argument is a logical vector with one entry for each of the ordinary arguments to `.C` (the *ordinary arguments* are those arguments which are not one of `NAME`, `NAOK`, `special`, and `pointers`). If the  $i$ th element of `pointers` is `TRUE`, then S-PLUS passes two arguments to your C procedure corresponding to the  $i$ th ordinary argument to `.C`. The first argument is a pointer to a pointer to the type corresponding to the mode of the  $i$ th ordinary argument to `.C`. The other argument is a pointer to a long. Your C procedure should use `S_alloc()` to allocate space for the output vector. It should pass back a pointer to the space via the first of the argument pair and its length back via the second.

Here is an example which takes a vector `x` of integers and returns a sequence of integers, of length `max(x)`:

```

#include <S.h>
void
makeseq(in, nin_p, out_p, nout_p)
long *in, *nin_p ;
long **out_p, *nout_p ;
{
    long nin = *nin_p ;
    long *out, nout = 0 ;
    long i ;
    if (nin == 0) {
        *nout_p = 0 ;
        *out_p = (long *)NULL ;
        PROBLEM "No input data" WARNING(NULL_ENTRY) ;
        return ;
    }
    /* compute maximum of in's */
    nout = *in ;
    for (i=1 ; i<nin ; i++)
        if (nout < in[i])
            nout = in[i] ;
    *nout_p = nout ;
    if (nout < 0) {
        PROBLEM
            "All input data is negative (max is %ld)",
            nout
        RECOVER(NULL_ENTRY) ;
    } else if (nout == 0) {
        *out_p = (long *)NULL ;
        return ;
    }

    /* allocate space for out vector and return *
     * pointer to it */
    *out_p = out = (long *) S_alloca(nout,
                                     sizeof(long));

    for (i=0 ; i<nout ; i++)
        out[i] = i + 1 ;
}

```

Use the following S-PLUS code to call makeseq():

```

makeseq.C <- function(x)
. C("makeseq",
    input = as.integer(x),
    input.length = length(x),
    output = integer(0),
    pointers = c(F, F, T))[[3]]

```



In `makeseq.C`, the third ordinary argument to `.C`, `output`, corresponds to the third and fourth arguments to C code for `makeseq()` because the third element of pointers is `TRUE`.

In more realistic examples, you use `S_realloc()` repeatedly to grow the output arrays as needed, as in the following code fragment. Note how we double the size of the array each time it overflows; other strategies are possible, but you usually waste time and space extending it only by as much as you need each time you need to add an entry:

```
nout_alloc = nout = 0 ;
out = (long *)NULL ;
for (i=0; i<n ; i++){
    if (predicate(i)) { /* add this index to output array */
        if (++nout > nout_alloc) {
            long new = 2 * nout_alloc + 1 ;
            out = (long *)S_realloc(out, new,
                                   nout_alloc, sizeof(long)) ;
            nout_alloc = new ;
        }
        /* now we know that nout <= nout_alloc */
        out[nout-1] = i ;
    }
}
*nout_p = nout ;
*out_p = out ;
```

Since `S_realloc()` acts like `S_alloc()` if the old pointer is `NULL` and the old length is 0, we don't have to prime the loop by calling `S_alloc()`.

## Calling Fortran From C

S-PLUS contains a few C preprocessor macros to help smooth over differences between machines in how to call C code from Fortran and vice versa. The following macros are needed to allow distinctions between the declaration, definition, and invocation of a Fortran common block or Fortran subroutine (coded in either C or Fortran):

<code>F77_NAME</code>	declaration of a Fortran subroutine.
<code>F77_SUB</code>	definition of a Fortran subroutine.
<code>F77_CALL</code>	invocation of a Fortran subroutine.
<code>F77_COMDECL</code>	declaration of a Fortran common block.
<code>F77_COM</code>	usage of a Fortran common block.

As an example of the proper use of the F77 macros, consider the following example C code fragment:

```
...
/* declaration of a common block defined in Fortran */
extern long F77_COMDECL(Forblock)[100];
...
/* declaration of a subroutine defined in Fortran */
void F77_NAME(Forfun)(double *, long *, double *);
...
/* declaration of a function defined in C, callable by
 * Fortran */
double F77_NAME(Cfun)(double *, long *);
...
/* usage of the above common block */
for (i = 0; i < 100; i++) F77_COM(Forblock)[i] = 0;
...
/* invocation of the above functions */
F77_CALL(Forfun)(s1, n1, result);
if (F77_CALL(Cfun)(s2, n2) < 0.0)
...
/* definition of the above 'callable by Fortran' function
 */
double F77_SUB(Cfun)(double *weights, long
 *number_of_weights);
...
```

If you are loading code originally written for a specific UNIX compiler (including some submissions to StatLib), you may find that that code does not compile correctly in Windows because not all of these macros are used. Usually, such code does not use the F77\_CALL macro to invoke the functions (using F77\_SUB instead), does not use the F77\_COMDECL macro to declare the Fortran common block (using F77\_COM instead), and leaves out the F77\_NAME macro altogether. If you attempt to load such code without substituting F77\_CALL for F77\_SUB at the appropriate places, you get compilation errors such as the following:

```
xxx.c(54): Error! E1063: Missing operand
xxx.c(54): Warning! W111: Meaningless use of an expression
xxx.c(54): Error! E1009: Expecting ';' but found 'fortran'
```

Similarly, if you attempt to statically load code without substituting F77\_COMDECL for F77\_COM where appropriate, you get a link error such as the following:

```
file xxx.obj(xxx.c): undefined symbol Forblock
```

Finally, if you attempt to statically load code without using `F77_NAME` to declare the subroutine, you get a link error of the following form:

```
file xxx.obj (xxx.c): undefined symbol Cfun
```

Fortran passes all arguments by reference, so a C routine calling Fortran must pass the address of all the arguments.

### Warning

Fortran character arguments are passed in many ways, depending on the Fortran compiler. It is impossible to cover up the differences with C preprocessor macros. Thus, to be portable, avoid using character and logical arguments to Fortran routines which you would like to call from C.

## Calling C From Fortran

You cannot portably call C from Fortran without running the Fortran through a macro processor. You need a powerful macro processor like `m4` (even it cannot do all that is needed) and then your code doesn't look like Fortran any more.

We can give some guidelines:

- Try not to do it.
- To be portable, do not use logical or character arguments (this applies to C-to-Fortran calls as well) because C and Fortran often represent them differently.
- Use pointer arguments in the C code.

## Calling S-PLUS Functions From C Code

To this point, we have shown how to call C and Fortran routines from S-PLUS functions. You can also call S-PLUS functions from C code, using the supplied C routine `call_S()`. The `call_S()` routine is useful as an interface to numerical routines which operate on C or Fortran functions, but it is not a general purpose way to call S-PLUS functions. The C routine calling `call_S()` must be loaded into S-PLUS, the arguments to the function must be simple, and the nature of the output must be known ahead of time. Because of these restrictions, `call_S()` cannot be used to call S-PLUS functions from an independent C application, as you might call functions from a subroutine library.

The C function `call_S()` calls an S-PLUS function from C, but `call_S()` must be called by C code called from S-PLUS via `.C()`. The `call_S()` function has the following calling sequence:

```
call_S(void *func, long nargs, void **arguments,  
       char **modes, long *lengths, char **names,  
       long nres, void **results);
```

where:

<code>func</code>	<p>is a pointer to a list containing one S-PLUS function. This should have been passed via an argument in a <code>.C</code> call, as follows:</p> <pre>.C("my_c_code", list(myfun))</pre> <p>This calls C code starting with the following lines:</p> <pre>my_c_code(void **Sfunc) {     ...     call_S(*Sfunc, ...);     ... }</pre> <p>The S-PLUS function must return an atomic vector or list of atomic vectors.</p>
<code>nargs</code>	<p>is the number of arguments to give to the S-PLUS function <code>func</code>.</p>
<code>arguments</code>	<p>is an array of <code>nargs</code> pointers to the data being passed to <code>func</code>. These can point to any atomic type of data, but must be cast to type <code>void*</code> when put into arguments.</p>
<code>modes</code>	<p>is an array of <code>nargs</code> character strings giving the S-PLUS names, e.g., "double" or "integer", of the modes of the arguments given to <code>func</code>.</p>
<code>lengths</code>	<p>is an array of <code>nargs</code> longs, giving the lengths of the arguments.</p>
<code>names</code>	<p>is an array of <code>nargs</code> strings, giving the names to be used for the arguments in the call to <code>func</code>. If you don't want to call any arguments by name, <code>names</code> may be <code>(char**)NULL</code>; if you don't want to call the <code>i</code>th argument by name, <code>names[i]</code> may be <code>(char*)NULL</code>.</p>
<code>nres</code>	<p>is the maximum number of components expected in the list returned by <code>func</code> (if <code>func</code> is expected to return an atomic vector, then <code>nres</code> should be 1).</p>

`results` is filled in by `call_S()`; it contains generic pointers to the components of the list returned by `func` (or a pointer to the value returned by `func` if the value were atomic).

Your C code calling `call_S()` should cast the generic pointers to pointers to some concrete type, like `float` or `int`, before using them. If `func` returns a list with fewer components than `nres`, the extra elements of results are filled with `NULL`'s. Notice that `call_S()` does not report the lengths or modes of the data pointed to by `results`; you must know this a priori.

To illustrate the use of `call_S()`, we construct (in Fortran) a general purpose differential equation solver, `heun()`, to solve systems of differential equations specified by an S-PLUS function. Other common applications involve function optimization, numerical integration, and root finding.

The `heun()` routine does all its computations in single precision and expects to be given a subroutine of the following form:

```
f(t, y, dydt)
```

where the scalar `t` and vector `y` are given and the vector `dydt`, the derivative, is returned. Because the `f()` subroutine calls the S-PLUS function, it must translate the function's argument list into one that `call_S()` expects. Since not all the data needed by `call_S` can be passed into `f()` via an argument list of the required form, we must have it refer to global data items for things like the pointer to the S-PLUS function and the modes and lengths of its arguments. The following file of C code, **dfeq.c**, contains a C function `f()` to feed to the solver `heun()`. It also contains a C function `dfeq()` which initializes data that `f()` uses and then calls `heun()` (which repeatedly calls `f()`):

```
#include <S.h>
extern void F77_NAME(heun)();
/* pointer to Splus function to be filled in */
static void *Sdydt ;

/* descriptions of the functions's two arguments */
static char *modes[] = {"single", "single"};
static long lengths[] = {1, 0};
/* neqn = lengths[1] to be filled in */
static char *names[] = { "t", "y" };

/*
   t [input]: 1 long ; y [input]: neqn long ;
   yp [output]: neqn long
*/
```

```

static void f(float *t, float *y, float *yp) {
    void *in[2] ; /* for two inputs to Splus function,
                  t and y */
    void *out[1] ; /* for one output vector of
                  Splus function */

    int i;
    in[0] = (void *)t;
    in[1] = (void *)y;
    call_S(Sdydt, 2L,
          in, modes, lengths, names, /* 2 arguments */
          1L, out/* 1 result */);

    /* the return value out must be 1 long – i.e., Splus
    function must return an atomic vector or a list of one
    atomic vector. We can check that it is at least 1 long. */

    if (!out[0])
        PROBLEM
        "Splus function returned a 0 long list"
        RECOVER(NULL_ENTRY);

    /* Assume out[0] points to lengths[1] single precision
    numbers. We cannot check this assumption here. */

    for(i=0; i<lengths[1]; i++)
        yp[i] = ((float *)out[0])[i] ;
    return ;
}

/* called via .C() by the Splus function dfreq(): */
void dfreq(void **Sdydtp, float *y, long *neqn,
          float *t_start, float *t_end, float *step,
          float *work) {
    /* Store pointer to Splus function and
    number of equations */
    Sdydt= *Sdydtp ;
    lengths[1] = *neqn ;

    /* call Fortran differential equation solver */
    F77_CALL(heun)(f, neqn, y, t_start, t_end, step, work);
}

```

**Warning**

In the C code, the value of the S-PLUS function was either atomic or was a list with at least one atomic component. To make sure there was no more than one component, you could look for 2 values in results and make sure that the second is a null pointer.

The following S-PLUS function, `dfeq`, does some of the consistency tests that our C code could not do (because `call_S` did not supply enough information about the output of the S-PLUS function). It also allocates the storage for the scratch vector. Then it repeatedly calls the C routine, `dfeq()`, to have it integrate to the next time point that we are interested in:

```
> dfeq <- function(func, y, t0 = 0, t1 = 1, nstep = 100,
  stepsize = (t1-t0)/nstep)
{
  if (length(func) != 3 ||
      any(names(func) != c("t", "y", "")))
    stop("arguments of func must be called t and y")
  y <- as.single(y)
  t0 <- as.single(t0)
  neqn <- length(y)
  test.val <- func(t = t0, y = y)
  if(neqn != length(test.val))
    stop("y and func(t0,y) must be same length")
  if(storage.mode(test.val) != "single")
    stop("func must return single precision vector")
  val <- matrix(as.single(NA), nrow = nstep + 1, ncol =
neqn)
  val[1, ] <- y
  time <- as.single(t0 + seq(0, nstep) * stepsize)
  for(i in 1:nstep) {
    val[i + 1, ] <- .C("dfeq", list(func), y=val[i, ],
      neqn=as.integer(neqn),
      t.start=as.single(time[i]),
      t.end=as.single(time[i + 1]),
      step=as.single(stepsize),
      work=single(3 * neqn))$y
  }
  list(time=time, y=val)
}
```

The following subroutine is the Fortran code, `heun.f`, for Heun's method of numerically solving a differential equation. It is a first order Runge-Kutta method. Production quality differential equation solvers let you specify a desired local accuracy rather than step size, but the code that follows does not:

```
C      Heun's method for solving dy/dt=f(t,y),
C      using step size h :
C      k1 = h f(t,y)
C      k2 = h f(t+h,y+k1)
C      ynext = y + (k1+k2)/2
```

```

      subroutine heun(f, neqn, y, tstart, tend, step, work)
      integer neqn
      real*4 f, y(neqn), tstart, tend, step, work(neqn,3)
C work(1,1) is k1, work(1,2) is k2, work(1,3) is y+k1
      integer i, nstep, istep
      real*4 t
      external f
      nstep = max((tend - tstart) / step, 1.0)
      step = (tend - tstart) / nstep
      do 30 istep = 1, nstep
         t = tstart + (istep-1)*step
         call f(t, y, work(1,1))
         do 10 i = 1, neqn
            work(i,1) = step * work(i,1)
            work(i,3) = y(i) + work(i,1)
10      continue
         call f(t+step, work(1,3), work(1,2))
         do 20 i = 1, neqn
            work(i,2) = step * work(i,2)
            y(i) = y(i) + 0.5 * (work(i,1) + work(i,2))
20      continue
30      continue
      return
      end

```

To try out this example of `call _S`, exercise it on a simple one-dimensional problem as follows:

```

> graphsheet()
> a <- dfreq(function(t,y)t^2, t0=0, t1=10, y=1)
> plot(a$time,a$y)
> lines(a$time, a$time^3/3+1) # compare to
                             #theoretical solution

```

You can increase `nstep` to see how decreasing the step size increases the accuracy of the solution. The local error should be proportional to the square of the step size and when you change the number of steps from 100 to 500 (over the same time span) the error does go down by a factor of about 25. An interesting three-dimensional example is the Lorenz equations, which have a strange attractor:

```

> chaos.func<-function(t, y) {
  as.single(c(10 * (y[2] - y[1]),
    - y[1] * y[3] + 28 * y[1] - y[2],
    y[1] * y[2] - 8/3 * y[3]))
}

```



```

> b <- dfreq(chaos.func, y=c(5, 7, 19), t0=1, t1=10,
             nstep=300)
> b.df <- data.frame(b$time, b$y)
> pairs(b.df)
    
```

The resulting plot is shown in Figure 16.1.

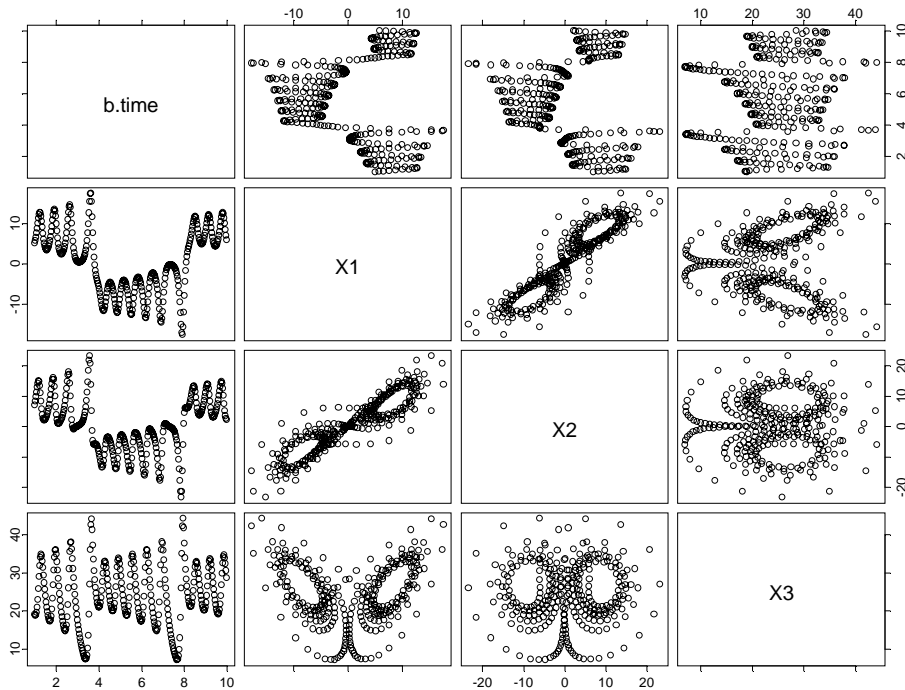


Figure 16.1: Viewing the Lorenz equations, as solved by `dfreq`.

### Warnings

Since `call_S` doesn't describe the output of the S-PLUS function it calls, you must “know” about it ahead of time. You can test the function for a variety of values before calling `call_S` to check for gross errors, but you cannot ensure that the function won't return an unacceptable value for certain values of its arguments.

The `call_S` function expects that the output of the function given to it has no attributes. If it does have attributes, such as dimensions or names, they are stripped.

## USING DYNAMIC LINK LIBRARIES (DLLS)

A Dynamic Link Library (DLL) is a library of functions that can be used by many different processes. When S-PLUS uses functions from a DLL, the object code in the DLL is added to the code space of the S-PLUS process. The S-PLUS engine creates references to the functions in the DLL at runtime. The S-PLUS function, `dll.load`, allows you to load a DLL from S-PLUS. Dynamic loading using `dll.load` offers the following advantages over dynamic loading using `dyn.load` (discussed in a later section):

- You do not have to use the Watcom compilers. You can use any compiler capable of producing DLLs with 32-bit entry points.
- You can interface to other DLLs and use their functions (in particular, the Windows API).

### Note

The restrictions on DLL loading in previous versions of S-PLUS do not apply to this version. Your DLL may reference C or Fortran functions that are internal to S-PLUS. For example, you can call the C functions or macros `S_alloc()`, `RECOVER()`, or `call_S()` in S-PLUS. You must include **S.h** to do this though. Also, when calling floating point functions internal to S-PLUS from a DLL compiled with a non-Watcom compiler, you may need to use the `S_DOUBLEVAL` or `S_FLOATVAL` macros defined in **compiler.h** (included by **S.h**); see the section Calling Functions in the S-PLUS Engine DLL (page 600).

### Note

When using `.C()` and `.Fortran()`, storage mode and type declarations are the same for DLL loading as they are for dynamic loading using `dyn.load` (see section Loading Watcom Object Files Using `dyn.load` (page 607)).

## Creating a DLL From C Source Code

The following routine is written in C code and calculates the Fibonacci numbers. This code can be compiled and dynamically loaded as a DLL using the Watcom or Visual C++ compilers. Other compilers can be used as well on the same source code, but you will have to adapt the specific instructions as to how to build the DLL itself.

The source code is written as follows (and also provided in the **library\progexam\fibdll.c** file in the S-PLUS distribution tree):

```
#i f defi ned(DLL_LOAD)
#i ncl ude <wi ndows. h>
/* Standard DLL entry/exi t procedure */
BOOL __stdcal l
DI lMai n(HI NSTANCE hDI lInstance, DWORD dwReason, LPVOID
lpReserved)
{
    swi tch (dwReason) {
    case DLL_PROCESS_ATTACH:
        /* i ni ti al i zati on code here */
        break;
    case DLL_PROCESS_DETACH:
        /* clean-up code here */
        break;
    }
    return(TRUE);
}
#endi f
/* Compute the nth Fibonacci number using recursion */
long
fi bi ter(long a, long b, long c) {
    i f (c <= 0L) return b;
    el se return fi bi ter(a+b, a, c - 1);
}

void fi b(long *n){
    *n = fi bi ter(1, 0, *n) ;
}
```

For many DLLs, your source code may involve a call to the `DI lMai n` function, as in the preceding example, to perform initialization of your library. Because this initialization routine is only called if the code is built as a DLL, you may want to isolate it with a C preprocessor `#i f defi ned` as we have done here.

The module definition file for this function, **fibdll.def**, is written as follows:

```
, *****
;
;   FIBDLL.DEF module definition file
; *****

LIBRARY FIBDLL
```

## EXPORTS

DI I Mai n

fi b

The following examples are intended to give you a sense of how different compilers use different tools to build DLLs. Regardless of which compiler you use, your DLL must obey the following rules:

- Must have a 32-bit entry point.
- Must follow the cdecl or stdcall calling conventions.
- Must be fully relocatable.

### Building a DLL Using Visual C++

To build the DLL, use Visual C++ as follows:

1. From Microsoft Developer Studio, select File, then New. The New dialog appears with a scrolled list of options.
2. Select Project Workspace from the New dialog. The New Project Workspace dialog appears.
3. Enter the project name “FIBDLL” in the Name text field, then select Dynamic Link Library in the Project Type list (*not* MFCAppWizard(dll)). If you want to specify a particular location for the project workspace, you can do that in this dialog as well.
4. Click Create. A new project workspace is created in the directory you specified.
5. From the Insert menu, choose Files into Project.
6. Use the Insert Files into Project dialog to select the source code file **fibdll.c** and the module definition file **fibdll.def**. To view **.def** files, use the Files of Type dropdown.
7. Select Build, then Settings, to bring up the Project Settings dialog. Choose the C/C++ tab.
8. Add the definition **DLL\_LOAD** to the Preprocessor Definitions text field in the Project Settings dialog.
9. Select Project and then Rebuild All from the menu bar. This builds the DLL and creates numerous files, including the DLL file, **fibdll.dll**.

## Building a DLL Using Watcom C/C++

To contrast how different compilers use different tools to build DLLs, we use the same **fibdll.c** file to build a DLL using a 32 bit Watcom compiler:

There are two steps involved in building a DLL: compiling the files and linking them into a DLL. Use the **COMPILE** utility to do the compiling, as follows (you can run this command from a DOS prompt, from a makefile, or even from with S-PLUS using the `dos` function):

**COMPILE -w4 -DDLL\_LOAD fibdll.c**

The **COMPILE** utility is described in more details in the section Using **COMPILE** to Create Watcom Object Files (page 605).

- The **-w4** option turns on the maximum warning level, which is good practice.
- The **-DDLL\_LOAD** option defines the symbol that causes the C pre-processor to allow compiling of the `DllMain` routine.

The linking stage involves building a Watcom linker directive file and passing that to **wlink**.

The general outline of the linker directive file is:

```
NAME <name-of-user-dll-to-build>
SYSTEM nt_dll
FILE <name-of-user-object-to-use>
EXPORT <name-of-user-symbols-that-can-be-called-from-S>
LIBRARY <name-of-user-libraries-to-use>
LIBRARY %S_HOME%\lib\sqpew.lib
LIBRARY kernel32.lib
LIBRARY user32.lib
LIBRARY gdi32.lib
```

The list of files or symbols in the `FILE`, `EXPORT`, and `LIBRARY` directives can either be comma-separated, or be on separate lines with the appropriate directive:

```
FILE y.obj
FILE z.obj
FILE t.obj

FILE y.obj, z.obj, t.obj
```

The first `LIBRARY` directive is optional, since users may not have any of their own object libraries to specify.

The last four `LIBRARY` directives may not be needed, but cover most of the cases of external symbols needing to be defined:

- The first one (**`sqpew.lib`**) is only needed if a call to a function internal to the S-PLUS engine is made (though it shouldn't be a problem including it anyway); see the section *Calling Functions in the S-PLUS Engine DLL* (page 600).
- The last three are the most likely ones to be needed if calls to Windows API functions are made (if such calls result in "undefined symbol" errors, the other libraries can be found in the `%WATCOM%\lib386\nt` directory).

Once the Watcom linker directive file is created, it should then be used to build the DLL as follows:

**`wlink @<directive-file>`**

If the Watcom linker directive file has an extension of **`.lnk`**, that extension does not have to be specified.

In our simple case, we can make do with the following linker file, **`fibdll.lnk`**:

```
NAME fibdll
SYSTEM nt_dll
FILE fibdll.obj
EXPORT fib
```

## Loading and Running the Code in the DLL

Once you have built the DLL, you can load it into S-PLUS using the `dll.load` function. The arguments to `dll.load` are as follows:

```
function(library, symbols=NULL, calling.convention="stdcall",
warn.level = 0, load.wo.sym = T, .16.bit = F)
```

where

<code>library</code>	is a character vector giving the name of the DLL file(s),
<code>symbols</code>	is a list specifying the name(s) of the called function(s),
<code>calling.convention</code>	is a vector of calling conventions for each called function name (cyclically repeated to the length of <code>symbols</code> ), with "stdcall" and "cdecl" as possible values.

The arguments `warn.level` and `load.wo.sym` are mostly useful for debugging and are described in detail in the help file for `dll.load`.

As an example, let us load the DLL we built from **fibdll.c** in the preceding section:

```
> dll.load("fibdll.dll", symbol.s="fib", "cdecl")
```

where `"fibdll.dll"` is the name of the DLL file, and `"fib"` is the name of the called function.

For convenience, an S-PLUS function to call this routine, once it is loaded, is as follows:

```
> fib.C <- function(n) .C("fib", as.integer(n))
```

Having loaded the DLL, we can run this function as follows:

```
> fib.C(4)
[[1]]
[1] 3
```

The calling convention `"cdecl"` in the call to `dll.load` is the default calling convention generated by Visual C++ or Watcom. If you use a different compiler, check with your documentation on how to specify the calling convention to use in the compiling phase, or how to specify the calling convention in your C code, when declaring and defining functions.

The syntax to use with Watcom C/C++ or Visual C++ in your source code is:

```
__declspec(dllexport) <return_type> __stdcall
<function_name>(<args>) {
...
}
```

or

```
__declspec(dllexport) <return_type> __cdecl
<function_name>(<args>) {
...
}
```

If you include **S.h**, and use either Watcom C/C++ or Visual C++, you can use the following macros to control your export directives and calling convention specifications:

<code>LibExport</code>	expands to the export directive notation appropriate for your compiler.
<code>CDECL</code>	expands to the <code>cdecl</code> calling convention notation appropriate for your compiler.

STDCALL expands to the standard calling convention notation appropriate for your compiler.

You can then use the following notation in your C code, for the `fib` function described above:

```
#include <S.h>
...

LibExport void CDECL fib(long * n) {
    ...
}
```

## Unloading the DLL

To unload the DLL (which you will need to do if you find that you have to rebuild your DLL), use the `dll.unload` function:

```
dll.unload("fib.dll.dll")
```

## Calling Functions in the S-PLUS Engine DLL

If your DLL calls internal S-PLUS functions, you will need an import library from the S-PLUS engine, **SQPE.DLL**, to resolve those calls. When you install the development libraries, you install one import library created with Microsoft Visual C++ Version 4 (**SQPE.LIB**) and also one created with Watcom C/C++ Version 10.5 (**SQPEW.LIB**). If you are using version 4 of Visual C++ or version 10.5 of Watcom C/C++, you are all set. If you are not using one of those compilers, the import library may not work with your DLL.

If you are using another version of Microsoft Visual C++, you can use the Library Manager, **LIB.EXE**, to create a compatible import library using the **SQPE.DEF** file provided with S-PLUS. Simply run the following command at a command prompt:

```
LIB /DEF:%S_HOME%\LIB\SQPE.DEF /OUT:SQPE.LIB
```

You must be sure to add **SQPE.LIB** to your list of included libraries, or include it in your Files in Project.

If you are using any non-Watcom compiler and your DLL calls an internal S-PLUS function that returns a floating point value, your source code may need to use the `S_DOUBLEVAL` or `S_FLOATVAL` macros defined in **compiler.h** (which is included by **S.h**). For example, instead of calling the pseudo-random number function directly:



```
...
double x = norm.rand();
...
```

wrap it with the appropriate macro:

```
...
double x = S_DOUBLEVAL(norm.rand());
...
```

If you are using another version of Watcom C/C++, you can create an import library directly from the file **SQPE.DLL** using the following command:

**WLIB SQPEW.LIB +%S\_HOME%\CMD\SQPE.DLL**

This has already been done for you if you are using version 10.5 of Watcom C/C++. You must be sure to add **SQPEW.LIB** to your list of included libraries.

If you are using a different compiler, you will probably have a different library manager. Check your compiler documentation for details on importing symbols or creating import libraries.

## Listing Symbols in Your DLL

If your DLL has just a few symbols, remembering them and listing them in the call to `dll_load` is not much of a problem. But if you are building a DLL with tens or hundreds of symbols, keeping track of them can be a chore. Luckily, most compilers offer utilities to help you list symbols in a DLL.

### Listing Symbols Using DUMPBIN

If you have Visual C++, you can use the **DUMPBIN** utility to view a list of exported symbols in your DLL. You run the **DUMPBIN** utility from a command prompt, as follows:

**DUMPBIN /exports [/out:filename] dllname**

The **/exports** switch tells **DUMPBIN** to report all exported symbols; the optional **/out** switch allows you to specify a file name for **DUMPBIN**'s output. (This is very useful if your DLL exports a lot of symbols.)

For example, to view the symbols exported from our **fibdll.dll**, we can use **DUMPBIN** as follows:

**C:\>dumpbin /exports fibdll.dll**

```
Microsoft (R) COFF Binary File Dumper Version 3.00.5270
Copyright (C) Microsoft Corp 1992-1995. All rights
reserved.
Dump of file fibdll.dll
```

File Type: DLL

Section contains the following Exports for  
FIBDLL.dll

```
0 characteristics
346B4464 time date stamp Thu Nov 13 10:18:12 1997
0.00 version
1 ordinal base
2 number of functions
2 number of names
ordinal hint name
2 0 DllMain (000012B8)
1 1 fib (00001110)
```

Summary

```
2000 .data
1000 .idata
1000 .rdata
1000 .reloc
1000 .rsrc
1000 .text
```

As we expected, there are only two exported symbols in this DLL. If we are obtaining a DLL compiled by someone else, **DUMPBIN /EXPORTS** may be an essential part of using the DLL.

### Listing Symbols Using the Watcom Librarian

If you have the Watcom compilers, you can list the exported symbols in a DLL using the Watcom librarian **wlib**. First, create an import library for the DLL. For example, the following command creates an import library for **fibdll.dll**:

```
wlib fibdll.lib +fibdll.dll
```

Then use **wlib** to list the symbols in the DLL's import library. For example, to print the exported symbols from **fibdll.dll** to the screen, use the following command:

```
wlib fibdll.lib
```

Adding the **-I** option with a filename will create a file of symbols:

```
wlib -I=fibdll.out fibdll.lib
```

## COMPILING AND LOADING WATCOM OBJECT CODE

If you have a Watcom compiler, you can often avoid the complexities of DLL creation by creating Watcom object code which you can either dynamically load with the `dyn. load` function or statically load with the **LOAD** utility. To statically load code, you must not only have a Watcom compiler, but it must be the same version of the compiler as was used to create the S-PLUS engine DLL in the first place. For dynamic loading with `dyn. load`, we have found that most versions of the Watcom compilers (10.0 or higher) will produce object code compatible with S-PLUS.

Besides the severe restrictions on the compilers that can be used to statically load code, static loading also has the disadvantages that it takes much longer than dynamic loading and that it creates a new S-PLUS engine DLL that occupies several megabytes of disk space. For these reasons, it is almost always better to use dynamic loading if possible. Static loading should be reserved for those rare cases where DLL loading or dynamic object code loading aren't possible, or when you actually do want to create a new S-PLUS engine DLL.

### Setting Up the Loading Environment

Before using **COMPILE** or **LOAD** for the first time, you need to have the S-PLUS development libraries installed. (If you did not install the development libraries when you installed S-PLUS, simply run S-PLUS Setup again and install them now.) You must also install the appropriate Watcom compilers. When installing the Watcom compilers, specify Windows NT/Windows 95/Win32s as the target operating system, stack-based calling in the 32-bit compiler options, and the host operating system you are using (either Windows NT/Windows 95 or Windows 3.1).

#### Note

The required target operating system and calling convention changed between S-PLUS version 3.3 and S-PLUS version 4.0. The other major change is that S-PLUS now requires DLLs with 32-bit entry points, where in version 3.3 it required DLLs with 16-bit entry points.

Also, before using **COMPILE** or **LOAD**, ensure that the following environment variables are set:  
S-PLUS environment variables:

- **SHOME** must be set to the directory under which the S-PLUS development libraries (subdirectory "**lib**") have been installed. For example, **SHOME** may be set as follows:

**SET SHOME=C:\SPWIN**

- If performing static loading, **S\_TMP** must be set (usually to an otherwise unused directory). For example, **S\_TMP** may be set as follows:

**SET S\_TMP=C:\TMP**

System environment variables:

- **PATH** must include the Watcom **BINNT** and **BINW** directories. For example, **PATH** may be set as follows:

**PATH=%PATH%;C:\WATCOM\BINNT;C:\WATCOM\BINW**

<b>Note</b>
The “% <i>varname</i> %” construct expands to the value of the environment variable <i>varname</i> .

Watcom environment variables:

- **WATCOM** must be set to the directory where the Watcom compiler(s) is installed. For example, **WATCOM** may be set as follows:

**SET WATCOM=C:\WATCOM**

- **INCLUDE** must be set to the Watcom **H** and **H\NT** directories. For example, **INCLUDE** may be set as follows:

**SET INCLUDE=C:\WATCOM\H;C:\WATCOM\H\NT**

or

**SET INCLUDE=%WATCOM%\H;%WATCOM%\H\NT**

- If using the Fortran compiler, **FINCLUDE** must be set. For example, **FINCLUDE** may be set as follows:

---

```
SET FINCLUDE=C:\WATCOM\SRC\FORTRAN\WIN;C:\WATCOM\SRC\FORTRAN
```

or

```
SET FI NCLUDE=%WATCOM%\SRC\FORTRAN\WI N; %WATCOM%\SRC\FORTRAN
```

If you want to perform static loading, the **LOAD** utility automatically determines which source files need to be compiled, based upon the arguments it is given. If you want to statically load C or Fortran code using additional compiler options other than the default ones (such as debugging or optimizing), you can set the environment variables **WCC\_FLAGS** or **WFC\_FLAGS**; otherwise, you must use the **COMPILE** utility, either from a DOS prompt or via a Makefile rule (see the section Options for Static Loading (page 609) for information on the **WCC\_FLAGS** and **WFC\_FLAGS** environment variables and on using Makefiles). If you want to perform dynamic loading using `dyn. load`, you must compile your C or Fortran code with the **COMPILE** utility, either from a DOS prompt, a Makefile, or from within S-PLUS using the `dos` function.

## Using COMPILE to Create Watcom Object Files

This section describes how to compile code for either static loading or dynamic loading using `dyn. load`, using the S-PLUS utility **COMPILE**. **COMPILE** requires you to use a Watcom 32-bit C or Fortran compiler. (For 100% compatibility, use Watcom Version 10.5 with S-PLUS Version 4.0.)

### Note

You do not need both C and Fortran Watcom compilers for doing static loading, unless you intend to use both. If you are using both C and Fortran source code, install the Watcom compilers for both in the same directory.

## Compiling C and Fortran Code With COMPILE

The **COMPILE** utility ensures that your code is compiled with compiler options compatible with those used by the code already in S-PLUS. Because you can run into problems if your code is not compiled in the same way that code already in S-PLUS is compiled, we strongly recommend you use the **COMPILE** utility instead of calling the compiler directly. The syntax for the **COMPILE** utility is as follows, whether you are running it from a DOS prompt, via a Makefile rule, or from within S-PLUS using the `dos` function:

```
COMPILE [options] filename.c (or .f or .for)
```

where *filename.c* (or *.f* or *.for*) is a source filename. The `COMPILE` utility allows only the file extension *.c* (for C code) or *.f* or *.for* (for Fortran code). You cannot compile more than one source file in the same call. The available options are the same as those for the Watcom compiler, which are listed in the *WATCOM C/C++32 User's Guide* and the *WATCOM Fortran 7732 User's Guide*. You can specify a *path* to the source filename in the call to **COMPILE**, but the object file that is created is put in the current directory.

For example, if you wanted to add the directory **\NEWDIR** to the list of directories that the Watcom C compiler will search for “include” files when it compiles the file **mycode.c**, do the following:

**COMPILE i#\NEWDIR mycode.c**

The *i* option uses a “#” instead of an “=” after it; this is to avoid problems with arguments to batch files. If there are no errors, **mycode.c** is compiled to create **mycode.obj**, and **mycode.obj** is put in the current directory.

#### Note

At most, nine arguments can be given to the **COMPILE** utility.

If you are compiling object code for dynamic loading with `dyn. load`, or for static loading, and your code might access data items in the S-PLUS engine (as, for example, in the `PROBLEM/RECOVER/WARNING` macros), you must use the “**-DS\_ENGINE\_BUILD**” option. For example:

`COMPILE -DS_ENGINE_BUILD mycode.c`

If your code might access data items in the S-PLUS engine and you do not use this option, you will get an undefined symbol error when you load the code.

#### Warning

Do *not* use the “**-DS\_ENGINE\_BUILD**” option if you are compiling code that will be linked into a DLL for `dll. load`. If you do, your code may compile, link, and load successfully, but cause an access violation when executed.

## Loading Watcom Object Files Using dyn.load

The `dyn.load` function reads your object code and adds it into the data space of the S-PLUS process, relocates your object code's references to data and functions so they point to the correct places, and stores the addresses of your functions so that you can execute them in subsequent calls to `.C` or `.Fortran` in the current session. You use the `dyn.load` function by specifying one or more object files created by **COMPILE** as the argument to `dyn.load`. In most cases, you dynamically load code with a simple call to `dyn.load` as follows:

```
dyn.load(object_files)
```

where *object\_files* is a character vector giving the file names of the object code files. With `dyn.load`, unlike `dll.load`, you do not have to specify the names of the symbols you wish to load; `dyn.load` determines this for you. The `dyn.load` function does not search C or Fortran libraries, so if your object code contains references to library functions that are absent from the S-PLUS engine (as indicated by “undefined symbol” errors), you must either use static or DLL loading, or extract the object modules containing those functions from the Watcom libraries and load those files as well. See the section Solving Problems With Static or Dynamic Loading (page 612) for details.

### Warning

If you dynamically load code using `dyn.load` and get a warning message of the form `dyn.load relocation problem`, your code may not be suitable for use with `dyn.load`. You may instead use static loading. *It is possible that dynamically loaded code with these types of warnings will work, but it should be tried with caution.*

### Note

If you dynamically load code with `dyn.load` and get a message of the form `undefined symbol __imp_`, your code may be attempting to access S-PLUS engine data items without having been compiled with the appropriate option. Try **COMPILE** again using the “**-DS\_ENGINE\_BUILD**” option, then reload your code. If the same message occurs, you will have to statically load your code or create a DLL. See the section Using the Watcom Libraries (page 613).

## Statically Loading Watcom Object Code Using LOAD

You statically load code with the **LOAD** utility. To do static loading, you must have the same Watcom 32-bit C or Fortran compiler as was used to create the S-PLUS engine DLL (Watcom Version 10.5 for S-PLUS 2000). **LOAD** creates a new engine DLL for S-PLUS (called **NSQPE.DLL** by default). The new copy of the S-PLUS engine takes up several megabytes of disk space. The syntax for the **LOAD** utility (typically run from a DOS prompt) is as follows:

**LOAD filename.dll loadinfo**

where *filename.dll* is the name of the new S-PLUS engine (if *filename.dll* is not specified, the new engine is named **NSQPE.DLL**). The *loadinfo* usually consists of the names of object files (having the same basename as the source files, but with a **.obj** extension) and libraries to be loaded into the new S-PLUS engine. For example, if you want to statically load a file of C code called **myfile.c**, do the following:

**LOAD myfile.dll myfile.obj**

If **myfile.obj** does not exist or is older than **myfile.c**, the **LOAD** utility compiles **myfile.c** to create **myfile.obj**. **LOAD** then links **myfile.obj** with the S-PLUS development libraries to create the **myfile.dll** engine, which contains your C function. You can specify a path for the newly created engine in the call to **LOAD**, but you can only specify a path to the object file if it already exists. If you have an additional source file, for example, **mynewfile.c**, which you want to load with **myfile.c**, perform the call as follows:

**LOAD myfile.dll myfile.obj mynewfile.obj**

The new **myfile.dll** contains both routines. In most cases, using **LOAD** as in the two preceding examples is all you need to know to load your code. However, to use additional compiler options to specify certain linker directives, or to automate the building of a large project when doing static loading, see the section Options for Static Loading (page 609).

### Warning

To prevent the overwriting of certain S-PLUS files, **LOAD** cannot be run from either the **cmd** or **lib** directories under **\$HOME**.



For example, to load the **fibdll.obj** file created in the previous section, call **LOAD** from the DOS prompt as follows:

**LOAD fibdll.obj**

### Warning

Be careful when mixing object code compiled for **dl l . load** with object code compiled for **dyn. load** or **LOAD**, as some options appropriate for one type of loading are not appropriate for the other type. In particular, if your code may access data items in the S-PLUS engine, you must use the “**-DS\_ENGINE\_BUILD**” option when using **COMPILE** for **dyn. load** or **LOAD**, but you must *not* use that option when using **COMPILE** for **dl l . load**.

It may take several minutes to create the new S-PLUS engine; in fact the Watcom linker is by default called twice by **LOAD**—the first time to determine all the exportable symbols in your code. You can eliminate the additional link step by creating your own **EXPORTS** linker directive file—see the section Options for Static Loading (page 609). After compiling and loading your source code for S-PLUS, you can run the newly created engine by enabling S-PLUS to find it. If the default name for the S-PLUS engine (**NSQPE.DLL**) was specified, it will be found by S-PLUS if it is in any of the following locations:

1. The directory containing the executable file for the S-PLUS session (usually, the **CMD** directory under **S\_HOME**).
2. The working directory for the S-PLUS session.
3. A directory listed in the **PATH** environment variable.

### Note

Regardless of whether the default name was specified, the full path to the new S-PLUS engine may be specified by the value of the environment variable/S-PLUS command line argument **S\_ENGINE\_LOCATION**.

## Options for Static Loading

When doing static loading, you may want to use the same set of additional compiler options. You can do this by setting the environment variables **WCC\_FLAGS** (referenced when C code is compiled) or **WFC\_FLAGS** (referenced when Fortran code is compiled). The options specified in this way apply to all compilations done by **LOAD** while those environment

variables are set. For example, if you want to compile many different C source files, and add the directory **C:\HFILES** to the compiler's "include" directory search for each compilation, do the following:

**SET WCC\_FLAGS=#C:\HFILES**

Notice that the **i** option uses a "#" instead of an "="; this is to avoid syntax errors when using the **SET** command. If this is done in a DOS box while running Windows, **WCC\_FLAGS** is only set for the duration of the DOS box (when you close the DOS box, **WCC\_FLAGS** is no longer set). You can unset **WCC\_FLAGS** as follows:

**SET WCC\_FLAGS=**

Subsequent compilations done by **LOAD** use the default options.

Sometimes you may want to specify additional compiler options for just a few of the files in a project that uses static loading, or automate the building of some parts of a particular project. You can do this with a Makefile. **LOAD** automatically compiles any source files by invoking Watcom's **wmake** utility using rules specified in a temporary makefile that **LOAD** creates. Normally, the temporary makefile contains only the rules specified in **makefile.tl** in the **S\_HOME** lib directory. However, if a Makefile exists in the current directory, **LOAD** adds the rules in it to the beginning of the temporary makefile, and those rules are used by **wmake**. For example, suppose you have source files **cfuns.c**, **ffuns.f**, and **ccode.c** (which includes **ccode.h**), and you create a **Makefile** containing the following lines:

```
myfuncs.lib : cfuns.obj ffuns.obj
    wlib -b -c -n myfuncs.lib cfuns.obj ffuns.obj
ccode.obj : ccode.h
    !compile -oi -ze ccode.c
```

If **ccode.obj** references functions in **myfuncs.lib** (for example, **ccode.obj** references **ffuns.obj**), then the following call causes **ccode.obj** to be loaded into a new version of the S-PLUS engine (by default, **NSQPE.DLL**):

**LOAD ccode.obj myfuncs.lib**

Sometimes you may want to specify certain linker directives (such as **DEBUG ALL**), or automate the building of a large project. You can do this with a loadfile, using the following syntax for the **LOAD** utility:

**LOAD filename.dll @ loadfile**

where *filename.dll* is the name of the new S-PLUS engine (if *filename.dll* is not specified, the new engine is named **NSQPE.DLL**). The loadfile is a file that contains the names of the object files and libraries to be loaded into the new S-PLUS engine, using directives as described in the Watcom Linker documentation. The loadfile may not use the following directives:

<b>FORMAT</b>	reserved by S-PLUS.
<b>NAME</b>	reserved by S-PLUS.
<b>SYSTEM</b>	reserved by S-PLUS.

A **Makefile** must exist in the current directory. The last line in this file must be of the following form:

**filename.dll: x.obj ...**

where *filename.dll* is the same as that supplied to the **LOAD** command (or **NSQPE.DLL** if that argument is not supplied), and the **x.obj**'s are the same ones specified in the loadfile using the **FILE** directive. Using this method allows you to specify that the same list of object files are to be loaded each time **LOAD** is run, and allows you to specify a list that is arbitrarily long. For example, suppose you have source files **hasbugs.c**, **cfile1.c**, **cfile2.c** and a library **fortlib.lib**, and you create a **Makefile** containing the following lines:

```
hasbugs.obj : notdone.h hasbugs.c
!compile -d2 hasbugs.c
testing.dll : hasbugs.obj cfile1.obj cfile2.obj fortlib.lib
```

and a loadfile called **testload** containing the following lines:

```
DEBUG all
FILE hasbugs.obj
FILE cfile1.obj
FILE cfile2.obj
LIBRARY fortlib.lib
```

You can then statically load the code with the following command:

**LOAD testing.dll @testload**

**LOAD** first compiles, if necessary, the source code files **hasbugs.c**, **cfile1.c**, and/or **cfile2.c**. Then the object files **hasbugs.obj**, **cfile1.obj**, and **cfile2.obj** are loaded into a new version of the S-PLUS engine. A source-level debugger can then be used to debug code in the file **testing.dll**. See the section Source-Level Debugging (page 618) for details.

By default, the Watcom linker is called twice by **LOAD**, the first time to determine for you all the exportable symbols in your code. You can eliminate this additional call to the linker by creating your own **EXPORTS** linker directive file. You will need to use a loadfile, and thus a Makefile, as described above. One of the directives in the loadfile must be of the form

**@exportsfile**

where **exportsfile** is the name of the **EXPORTS** linker directive file. The **EXPORTS** linker directive file should contain the names of all the symbols in your code that you want to export (so that they may be called from S-PLUS via .C). For example, suppose you have a source file **mycode.c** containing a routine `my_func` that you want to call from S-PLUS. Create a Makefile containing the following line:

```
nsqpe.dl I : mycode.obj
```

Create a loadfile called **myload** containing the following lines:

```
FILE mycode.obj
@myexps
```

Finally, create an **EXPORTS** linker directive file **myexps** containing the following line:

```
EXPORT my_func
```

The following call to **LOAD** then calls the Watcom linker only once:

```
LOAD @myload
```

## Solving Problems With Static or Dynamic Loading

The **COMPILE** and **LOAD** utilities simplify the process of compiling and loading your code, but you may still encounter difficulties. The most common difficulty is with undefined symbols. This section discusses the most common situations.

### Fortran Intrinsic Functions

The Fortran intrinsic functions, such as `sin`, may not be in S-PLUS (because most of S-PLUS is written in C). If you use `dyn.load` with Fortran code referring to the `sin` function you may get undefined symbol reports. If you do, the symbol referred to may be something like `Fsin` or `csin` (for single precision or complex sine). If you encounter this problem with your code, use static loading.

## Using the Watcom Libraries

Occasionally, the code you want to dynamically load may contain symbols that are not present in the S-PLUS engine. The routine you want to add may be part of the Windows API or in a Watcom standard library that doesn't happen to be part of the S-PLUS engine. In the former case you cannot use `dyn.load`; in the latter case you have to do some extra work to get the code from the library into S-PLUS, and part of the battle is finding out which Watcom library you need. This section points out some common problems and indicates promising lines of attack in the Windows environment.

As an example of the first case, suppose you have the following code in a file called **sysdir.c**:

```
#include <windows.h>
#include <stdlib.h>
#include "S.h"
void
S_systemdir(char **ret)
{
    *ret = S_alloc(_MAX_PATH, sizeof(char));
    GetSystemDirectory(*ret, _MAX_PATH);
}
```

If you try to dynamically load this code using `dyn.load`, you will get an error, since `GetSystemDirectory()` is not present in the S-PLUS engine:

```
> dos("COMPILE sysdir.c", out = F)
> dyn.load("sysdir.obj")
Error in dyn.load("sysdir.obj"):
  No definition for 1 symbols
In addition: Warning messages:
  undefined symbol "__imp__GetSystemDirectoryA@8" in:
  dyn.load("sysdir.obj")
Dumped
```

The undefined symbol in this case begins with “\_\_imp\_\_”. If this code attempted to access an S-PLUS data item (as is done via the `PROBLEM/WARNING/RECOVER` macros), you would need to recompile as follows:

```
> dos("COMPILE -DS_ENGINE_BUILT sysdir.c", out = F)
```

Since that is not the case with this code, recompiling will not help. Whenever a symbol beginning with “\_\_imp\_\_” that is not an S-PLUS data item is undefined, you cannot use `dyn.load`, and so must use one of your other options: statically loading the code or creating a DLL. If you statically load the code, it links and runs properly because the appropriate libraries are searched by the Watcom linker for `GetSystemDirectory`. Since `GetSystemDirectory` is part of the Windows API, the only other alternative is

to build a DLL and load it with `dll.load`. See the section Using Dynamic Link Libraries (DLLs) (page 594) for information on building DLLs with various compilers.

As an example of the second case, suppose you have the following code in a file called **mylength.c**:

```
#include "S.h"
#include "newredef.h"
void
my_file_length(char **file, long *ret)
{
    FILE *f;
    if ((f=fopen(*file, "r")) == NULL)
        PROBLEM "Unable to open file '%s'", *file
        RECOVER(NULL_ENTRY);
    *ret = file_length(file_no(f));
    fclose(f);
}
```

If you try to dynamically load this code using `dyn.load`, you will get an error, since `file_length()` is not present in the S-PLUS engine.

```
>dos("COMPILE mylength.c", OUT=F)
>dyn.load("mylength.obj")
Error in dyn.load("mylength.obj"): No definition for 1
symbol
In addition: Warning messages:
  Undefined symbol "file_length" in: dyn.load("mylength.obj")
Dumped
```

Again, one solution is to statically load this code. Since `file_length` is part of a Watcom standard library, you can also dynamically load the code if you search the Watcom libraries and extract the desired symbol. The following example shows how to search Watcom's subdirectories and extract `file_length`:

1. Determine the Watcom library that contains the undefined symbol. There are many Watcom libraries, all of which are located in Watcom's `lib386` directory (or its subdirectories). Most symbols are likely to be found in one of the following libraries (`%WATCOM%` being the directory in which your Watcom compiler has been installed):

```
%WATCOM%\lib386\math387s.lib
%WATCOM%\lib386\NT\NT.lib
%WATCOM%\lib386\NT\clib3s.lib
%WATCOM%\lib386\NT\flib7s.lib
```

Use the Watcom librarian **wlib** to list all the object modules and symbols contained in a particular library. To save the lists for the above libraries to a file, do the following from a DOS prompt:

```
wlib -q %WATCOM%\lib386\math387s.lib > math387s.lst
wlib -q %WATCOM%\lib386\NT\NT.lib > NT.lst
wlib -q %WATCOM%\lib386\NT\clib3s.lib > clib3s.lst
wlib -q %WATCOM%\lib386\NT\flib7s.lib > flib7s.lst
```

The **-q** parameter specifies “quiet mode.” You can use the DOS command **find** to find a particular string in a set of files. To find `filelength` in one of the **.lst** files, do the following from the DOS prompt:

```
for %i in (*.lst) do find “filelength” %i
```

Some lines of information appear on the screen similar to the following:

```
-----CLIB3S.LST
Fgets-----Fgets
filelength-----filelen
```

In this example, `filelength` is contained in **clib3s.lib**, in the object module `filelen`.

2. Extract the object module containing the undefined symbol from the library. Use the “extract” command (using the “\*” character) of the Watcom librarian. For our example, do the following from the DOS prompt:

```
wlib -q c:\watcom\lib386\NT\clib3s.lib *filelen
```

This creates the file **filelen.obj**. Now if you compile and dynamically load the code in **mylength.c**, it should successfully load, providing that you specify the correct order of dependencies in the object code. (If you specify **mylength.obj** *before* **filelen.obj**, you will still get an “undefined symbol error” when you try to dynamically load them.)

3. From the S-PLUS prompt, compile and dynamically load the code as follows:

```
> dos("COMPILE mylength.c", out = F)
> dyn.load(c("filelen.obj", "mylength.obj"))
```

You can use the function with a call like the following:

```
> .C("my_filelength", paste(getenv("S_HOME"), "cmd",
" sapiobj.dll", sep="\\"), ret = integer(1))$ret
[1] 167936
```

## DEBUGGING LOADED CODE

Frequently the code you are dynamically or statically loading is known, tested, and reliable. But what if you are writing new code, perhaps as a more efficient engine for a routine developed in S-PLUS? You may well need to debug both the C or Fortran code and the S-PLUS function that calls it. The first step in debugging C and Fortran routines for use in S-PLUS is to make sure that the C function or Fortran subroutine is of the proper form, so that all data transfer from S-PLUS to C or Fortran occurs through arguments. Both the input from S-PLUS and the expected output need to be arguments to the C or Fortran code. The next step is to ensure that the storage modes of all variables are consistent. This often requires that you add a call such as `as.single(variable)` in the call to `.C` or `.Fortran`. If the S-PLUS code and the compiled code disagree on the number, modes, or lengths of the argument vectors, S-PLUS's internal data may be corrupted and it will probably crash—by using `.C` or `.Fortran` you are trading the speed of compiled code for the safety of S-PLUS code. In this case, you usually get an application error message before your S-PLUS session crashes. Once you've verified that your use of the interface is correct, and you've determined there's a problem in the C or Fortran code, you can use an analog of the `cat` statement to trace the evaluation of your routine.

### Debugging C Code

If you are a C user, you can use C I/O routines, provided you include **`newredef.h`** inside your C code immediately after the directive to include **`S.h`**. Thus, you can casually sprinkle `printf` statements through your C code just as you would use `cat` or `print` statements within an S-PLUS function. (If your code is causing S-PLUS to crash, call `fflush()` after each call to `printf()` to force the output to be printed immediately.)

### Debugging C Code Using a Wrapper Function

If you cannot uncover the problem with generous use of `printf()`, the following function, `.Cdebug`, (a wrapper function for `.C`) can sometimes find cases where your compiled code writes off the end of an argument vector. It extends the length of every argument given to it and fills in the space with a flag value. Then it runs `.C` and checks that the flag values have not been changed. If any have been changed, it prints a description of the problem. Finally, it shortens the arguments down to their original size so its value is the same as the value of the corresponding `.C` call. The code includes references to the pointers argument to `.C`, described in the section *Returning Variable-Length Output Vectors* (page 583).



---

```

.Cdebug <- function(NAME, ..., NAOK = F, special sok = F,
pointers = NULL, ADD = 500, ADD.VALUE = -666)
{
  args <- list(...)
  tail <- rep(as.integer(ADD.VALUE), ADD)
  for(i in seq(along = args))
  {
    tmp <- tail
    storage.mode(tmp) <- storage.mode(args[[i]])
    args[[i]] <- c(args[[i]], tmp)
  }
  args <- c(NAME = NAME, args, NAOK = NAOK,
            special sok = special sok,
            pointers = list(pointers))
  val <- do.call(".", C", args)
  if(!is.null(pointers)) warning( "Cannot check for
                                overwriting when pointers used")
  else for(i in seq(along = val))
  {
    tmp <- tail
    storage.mode(tmp) <- storage.mode(args[[i]])
    taili <- val[[i]][seq(to = length(val[[i]]),
                          length = ADD)]
    if((s <- sum(taili != tmp)) > 0) {
      cat("Argument ", i, "(", names(val)[i],
          ") to ", NAME, " has ", s, " altered
          values after end of array\n",
          sep = "")
    }
    length(val[[i]]) <- length(val[[i]]) - ADD
  }
  val
}

```

For example, consider the following C procedure, oops():

```

oops(double *x, long* n)
long i;
{
  for (i=0 ; i <= *n ; i++) /* should be <, not <= */
    x[i] = x[i] + 10 ;
}

```

Because of the misused `<=`, this function runs off the end of the array `x`. If you call `oops()` using `.C` as follows, you get an Application Error General Protection Fault that crashes your S-PLUS session:

```
> .C("oops", x=as.double(1:66), n=as.integer(66))
```

If you use `.Cdebug` instead, you get some information about the problem:

```
> .Cdebug("oops", x=as.double(1:66), n=as.integer(66))
Argument 1(x) to oops has 1 altered values after end of
array
x:
[1] 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
[19] 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
[37] 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64
[55] 65 66 67 68 69 70 71 72 73 74 75 76
n:
[1] 66
```

The `.Cdebug` function cannot tell when you run off the beginning of an argument vector or when you write anywhere else in memory. If inspecting your source code and using S-PLUS functions like `.Cdebug` is not enough to pinpoint a problem, try the following:

1. Write a short main program that calls your procedure.
2. Compile and link the main program and your procedure for debugging.

## Source-Level Debugging

If your compiled routines are fairly complicated, you may want more help in debugging than can be provided by simple print statements. Both Microsoft Visual C++ and the Watcom compilers come with sophisticated source-level debuggers.

If you are using Microsoft Visual C++, you can easily do source-level debugging of your code. Simply follow the instructions for creating a DLL as outlined in the section A Simple Example: Filtering Data (page 563) or in the section Building a DLL Using Visual C++ (page 596). Before creating the DLL, you should ensure that the default project configuration (under Build, then Set Default Configuration...) is set to Win32 Debug. You will also need to specify the executable to be used for your debug session. To do this, select Build, then Settings..., to bring up the Project Settings dialog, and choose the Debug tab. Under Settings For:, select Win32 Debug, and in the Executable for debug session: field, enter the full path to the S-PLUS executable

(**SPLUS.EXE** in the **CMD** subdirectory of where S-PLUS is installed). When you have started your debug session, remember that the DLL will have been created in the **Debug** subdirectory of your project directory.

#### Note

The instructions referenced above are specific to the examples for which they are used. Obviously, you will want to add different files to your project; and in the second set of instructions, the step(s) to add the definition `DLL_LOAD` was for that example only and not generally necessary.

If you are using a Watcom 32-bit compiler, you can do source-level debugging (with either the character-mode or the "windowed" version of Watcom's debugger) of your code, provided it is either statically loaded, or dynamically loaded using the `dll_load` function; you cannot do source-level debugging of your code if it is dynamically loaded using the `dyn_load` function. You can use the **GCOMPILE** utility to compile your code with the appropriate debug options; you can also use the **GLOAD** utility to create, with the appropriate debug options, a new S-PLUS engine.

#### Note

The **COMPILE** and **GCOMPILE** utilities are the same, except that **GCOMPILE** adds the `"-d2"` option. All other options for **COMPILE** can also be used with **GCOMPILE**.

Also, the **LOAD** and **GLOAD** utilities are the same, except that **GLOAD** adds the `"DEBUG all"` linker directive and the new S-PLUS engine it creates is by default called **GSQPE.DLL** (instead of **NSQPE.DLL**). All other options for **LOAD** can also be used with **GLOAD**.

For example, suppose you wanted to do source-level debugging of the code in **library\progexam\fibdll.c** of the S-PLUS installation directory, as noted in the section Creating a DLL From C Source Code (page 594). Simply follow the instructions for building a DLL as outlined in the section Building a DLL Using Watcom C/C++ (page 597), except use **GCOMPILE** instead of **COMPILE**:

```
GCOMPILE -w4 -DDLL_LOAD fibdll.c
```

and add the following line to the linker directive file, just above the first **FILE** directive:

```
DEBUG all
```

The DLL built by the Watcom linker can then be "debugged" using either of the Watcom source-level debuggers. Upon starting the Watcom debugger you will need to enter, in the Program Name field of the New Program dialog, the full path to the S-PLUS executable (**SPLUS.EXE** in the **CMD** subdirectory of where S-PLUS is installed). Then select Break, and On Image Load..., to bring up the Break on Image Load dialog. Enter the same name as was specified in the **NAME** directive of your linker directive file (in this case, **fibdll**), and click OK. When you have started your debug session, a program break (pause) will occur when you use the `dll load` function to load your DLL. The module name (as specified in the **NAME** directive) of your DLL should appear in the Modules window of the debugger. You can then double-click (or right-click and select Source) to view your source code.

You can also use the **GLOAD** utility to create a source-level debuggable S-PLUS engine (by default, **GSQPE.DLL**), although you will only be able to view your own source files, not any S-PLUS ones. Simply follow the instructions as outlined in the section Statically Loading Watcom Object Code Using **LOAD** (page 608), except use **GLOAD** instead of **LOAD** (**GLOAD** automatically uses the "**DEBUG all**" linker directive; you do not have to use it yourself). If you choose to compile your code directly instead of through **GLOAD**'s default rules, you should use **GCOMPILE** to compile your code with the appropriate debug options. Remember that if your code attempts to reference S-PLUS engine data items, you will have to specify the "**-DS\_ENGINE\_BUILD**" option in using **GCOMPILE**; if you do not, you will get an undefined symbol error when you use **GLOAD**.

Starting a session of the Watcom debugger with a source-level debuggable S-PLUS engine is similar to the procedure outlined above: you will need to enter the full path to the S-PLUS executable (which is still **SPLUS.EXE**), plus you will have to enable the S-PLUS session to use your debuggable S-PLUS engine. You can do the latter by setting (either in your system environment or in the Arguments field of the Debugger's New Program dialog) the variable **S\_ENGINE\_LOCATION** to the full path of your debuggable S-PLUS engine; or, if you specified the default name (**GSQPE.DLL**) in **GLOAD**, you can copy or move your debuggable S-PLUS engine to the same directory as the S-PLUS executable (it is in the **CMD** subdirectory of the directory in which S-PLUS is installed). In the Break on Image Load dialog, you will still need to enter the same name as was specified in the **NAME** directive. However, that name will always be **SQPE**, regardless of the name of the S-PLUS engine DLL that **GLOAD** created; and the program break (pause) that occurs will happen very early in the S-PLUS session (you do *not* want to use the `dll load` function to load any S-PLUS engine DLL).

**Note**

In early releases of S-PLUS Version 4.0, a bug prevents S-PLUS from picking up the **S\_ENGINE\_LOCATION** from the command line. Use the `prog.name` function from your S-PLUS Commands window to determine the name of your engine DLL. If it is not the correct DLL, set **S\_ENGINE\_LOCATION** in your environment before starting S-PLUS.

## A NOTE ON STATLIB

StatLib is a system for distributing statistical software, data sets, and information by electronic mail, FTP, and the World Wide Web. It contains a wealth of user-contributed S-PLUS functions, many of which rely upon C and Fortran code that is also provided. Much of this code has been precompiled for use with S-PLUS for Windows.

- To access StatLib by FTP, open a connection to: **lib.stat.cmu.edu**. Login as **anonymous** and send your e-mail address as your password. The FAQ (frequently asked questions) is in **/S/FAQ**, or in HTML format at **<http://www.stat.math.ethz.ch/S-FAQ>**.
- To access StatLib with a web browser, visit **<http://lib.stat.cmu.edu/>**.
- To access StatLib by e-mail, send the message: **send index from S** to **statlib@lib.stat.cmu.edu**. You can then request any item in StatLib with the request **send item from S** where **item** is the name of the item.

If you find a module you want, check to see if it is pure S code or if it requires C or Fortran code. If it does require C or Fortran code, see if there's a precompiled Windows version—look in the **/DOS/S** directories. The precompiled versions generally require you to do nothing more than install the code.

# EXTENDING THE USER INTERFACE

# 17

---

<b>Overview</b>	<b>625</b>
Motivation	625
Approaches	625
Architecture	626
<b>Menus</b>	<b>627</b>
Creating Menu Items	627
Menu Item Properties	628
Modifying Menu Items	632
Displaying Menus	634
Saving and Opening Menus	634
<b>Toolbars and Palettes</b>	<b>636</b>
Creating Toolbars	636
Toolbar Object Properties	637
Modifying Toolbars	639
Creating Toolbar Buttons	640
ToolbarButton Object Properties	641
Modifying Toolbar Buttons	643
Displaying Toolbars	644
Saving and Opening Toolbars	646
<b>Dialogs</b>	<b>647</b>
Creating Dialogs	649
Creating Property Objects	650
Property Object Properties	650
Modifying Property Objects	652
Creating FunctionInfo Objects	653
FunctionInfo Object Properties	654
Modifying FunctionInfo Objects	655
Displaying Dialogs	656
Example: The Contingency Table Dialog	657
<b>Dialog Controls</b>	<b>660</b>
Control Types	660

Copying Properties	671
ActiveX Controls in S-PLUS dialogs	674
<b>Callback Functions</b>	<b>694</b>
Interdialog Communication	696
Example: Callback Functions	696
<b>Class Information</b>	<b>699</b>
Creating ClassInfo Objects	699
ClassInfo Object Properties	700
Modifying ClassInfo Objects	701
Example: Customizing the Context Menu	702
<b>Style Guidelines</b>	<b>706</b>
Basic Issues	706
Basic Dialogs	707
Modeling Dialogs	715
Modeling Dialog Functions	721
Class Information	727
Dialog Help	729



---

# OVERVIEW

In S-PLUS, it is easy to create customized dialogs and invoke them with toolbar buttons and menu items. Similarly, menus and toolbars can be created and modified by the user. This chapter describes in detail how to create and modify the dialogs, menus, and toolbars which make up the interface.

## Motivation

Users may be interested in customizing and extending the user interface to varying degrees. Possible needs include:

- Removing menu items and toolbars to simplify the interface.
- Changing menu item and toolbar layout.
- Creating new menu items or toolbars to launch scripts for repetitive analyses.
- Developing menus and dialogs for new statistical functions, either for personal use or for distribution to colleagues.

The tools for creating menus, toolbars, and dialogs in S-PLUS are quite flexible and powerful. In fact, all of the built-in statistical dialogs in S-PLUS are constructed using the tools described in this chapter. Thus users have the ability to create interfaces every bit as sophisticated as those used for built-in functionality.

## Approaches

This chapter discusses both point-and-click and command based approaches for modifying the user interface. The point-and-click approach may be preferable for individual interface customizations such as adding and removing toolbars. Programmers interested in sharing their interface extensions with others will probably prefer using commands, as this enables easier tracking and distribution of interface modifications.

## Architecture

The S-PLUS graphical user interface is constructed from *interface objects*. Menu items, toolbars, toolbar buttons, and dialog controls are all objects. The user modifies the interface by creating, modifying, and removing these objects.

The two components of extending the interface are:

- Creating user interface objects.
- Creating functions to perform actions in response to the interface.

The user interface objects discussed here are:

- `MenuItem` objects used to construct menus.
- `ToolBar` objects defining toolbars.
- `ToolBarButton` objects defining the buttons on toolbars.
- `Property` objects defining dialog controls, groups, and pages.
- `FunctionInfo` objects connecting dialog information to functions.
- `ClassInfo` objects describing right-click and context menu actions for objects in the Object Explorer.

The two types of functions used with the user interface are:

- *Callback functions* which modify dialog properties based on user actions within a dialog.
- Functions executed when OK or Apply is pressed in a dialog. While any standard S-PLUS function may be called from a dialog, we provide style guidelines describing the conventions used in the built-in statistical dialogs.

The following sections discuss these topics from both command and user interface perspectives.

# MENUS

Menus are represented as a hierarchy of MenuItem objects. Each object has a Type of Menu, MenuItem, or Separator:

- Menu creates a submenu.
- MenuItem causes an action to occur when selected.
- Separator displays a horizontal bar in the menu, visually separating two groups of menu items.

Different menus may be created by creating and modifying MenuItem objects. By default, the main menu in S-PLUS is `SPlusMenuBar`. MenuItems may be added to or deleted from this menu to modify the interface. Alternately, users may create whole new menus which may be saved, opened, and used as the default menu.

MenuItems are also used to construct context menus. These are the menus displayed when a user right-clicks on an object in the Object Explorer. Context menus are discussed in detail in the section Context Menu (page 728).

## Creating Menu Items

MenuItems may be created using commands or from within the Object Explorer.

## Using Commands

To create a menu item, use the `guiCreate` function with `classname="MenuItem"`. The Name of the object will specify the location of the menu item in the menu hierarchy. Specify `Type="Menu"` for a menu item which will be the "parent" for other menu items, `Type="MenuItem"` for a menu item which performs an action upon select, or `Type="Separator"` for a separator bar.

The following commands will create a new top-level menu item with two child menu items launching dialogs for the `sqrt` and `lme` functions:

```
guiCreate(classname="MenuItem",
  Name="$$SPlusMenuBar$MyStats",
  Type="Menu", MenuItemText="&My Stats", Index="11",
  StatusBarText="My Statistical Routines")
```

```

gui Create(classname="MenuItem",
    Name="$${SPI usMenuBar$MyStats$Sqrt}",
    Type="MenuItem", MenuItemText="&Square Root...",
    Action="Function", Command="sqrt")

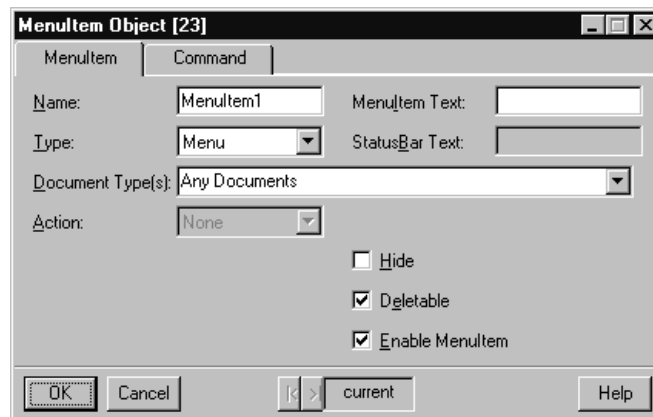
gui Create(classname="MenuItem",
    Name="$${SPI usMenuBar$MyStats$Lme}",
    Type="MenuItem", MenuItemText="Li near &Mi xed Effects...",
    Action="Function", Command="lme")

```

See the section Menu Item Properties (page 628) for details regarding property names and values.

## Using the Object Explorer

To create a menu item, first open the Object Explorer and filter by “MenuItem” to see the hierarchy of menu items. Navigate to the menu item above which the new menu item should appear. Right-click on this menu item, and select Insert MenuItem from the context menu. The property dialog shown in Figure 17.1 appears.



*Figure 17.1: The property dialog for a MenuItem object, MenuItem page.*

## Menu Item Properties

The properties of a MenuItem object determine characteristics such as the prompt for the menu item and the action performed when the menu item is selected. These properties may be specified and modified using the property dialog for the MenuItem object, or programmatically via the commands `gui Create` and `gui Modify`. See the `gui Create("MenuItem")` help file in the Language Reference help for syntax details.

---

The following properties are specified on the first tab of the MenuItem property dialog, shown in Figure 17.1:

**Name** The name of the MenuItem object. The name determines the menu to which the menu item belongs, and the position within the menu hierarchy.

**Type** The type of MenuItem object.

- Menu creates a submenu.
- MenuItem causes an action to occur when selected.
- Separator displays a horizontal bar in the menu, visually separating two group of menu items.

**Document Type** Depending on the type of document window type—Graph Sheet, Commands window, etc.—which has the focus, the item may or may not be visible. Document Type specifies the document types for which the item should be visible in the menu system. The choice “All Documents” causes the item to be always visible. The choice “No Documents” ensures that the item will be visible when no document window has the focus; for example, when no document window is open.

**Action** This applies to MenuItem objects of type MenuItem.

- NONE. No action is performed when the item is selected. This is useful when designing a menu system. It is not necessary to specify commands to execute when the type is set to NONE.
- BUILTIN. One of the actions associated with the default menus or toolbars is performed when the item is selected. These are listed on the Command page in the Built-In Operation dropdown box. This option allows you to use in a customized dialog any of the “intrinsic” menu actions, such as Window/Cascade.
- FUNCTION. Under this option, an S-PLUS function, either built-in or user-created, is executed when the item is selected.
- OPEN. The file specified on the Command page is opened when this item is selected. The file will be opened by the application associated to it by the operating system.
- PRINT. The file specified on the Command page is printed when

this item is selected. The file will be printed from the application currently associated to it by the operating system.

- **RUN.** The file specified on the Command page is opened and run as a script by S-PLUS when this item is selected.

**MenuItem Text** The text which will represent the item in the menu system. This does not apply to Separator items. Include an ampersand (&) to specify the keyboard accelerator value.

**StatusBar Text** The text which will appear in the status bar when the item has the focus in the menu.

**Hide** Logical value indicating whether to make the item invisible. When the item is hidden, its icon in the Object Explorer appears grayed out. This can also be specified through the context menu.

**Deletable** Logical values indicating whether to allow the item to be deleted.

The rest of the properties are found on the Command page, seen in Figure 17.2.

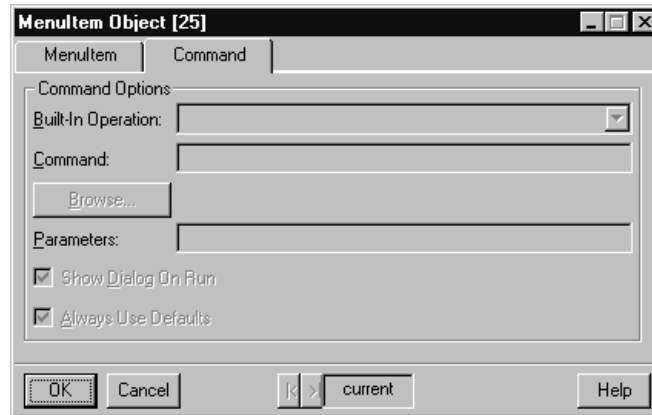


Figure 17.2: The property dialog for a MenuItem object, Command page.

**Built-In Operation** Operation desired when the Action is set to BUILTIN. Built-in actions are actions performed by the interface such as launching the Open file dialog.

**Command** The name of an S-PLUS function, or a path and filename. This field is enabled when Action is set to FUNCTION, OPEN, PRINT, or RUN on the MenuItem page. Use the Object Explorer to identify the folder.

**Parameters** This is relevant when the Action is set to FUNCTION. This property specifies the arguments for the function which will execute when the item is selected. The easiest way to specify these arguments is to work through the Customize dialog available through the context menu for the MenuItem in the Object Explorer. For details on doing this, see the section Using the Context Menu (page 632) below.

**Show Dialog On Run** This is relevant when Action is set to FUNCTION on the Command page. Logical value indicating whether to cause the dialog associated to the function to open when the item is selected. This can also be specified through the context menu.

**Always Use Defaults** This is relevant when Action is set to FUNCTION on the Command page. Logical value indicating whether to force the use of the default values when the function executes. This can also be specified through the context menu.

S-PLUS makes a distinction between the default argument values for a function as defined in the function's dialog (via the `FunctionInfo` object) and as defined by the function itself. Always Use Defaults refers to the "dialog" defaults. Table 17.1 summarizes how Show Dialog On Run and Always Use Defaults work together. In it, "function" refers to the S-PLUS

*Table 17.1: Summary of Show Dialog On Run and Always Use Defaults.*

Show Dialog On Run	Always Use Defaults	Action when the menu item is selected.
checked	checked	The dialog always opens in its default state when the menu item is selected. Changes are accepted, but do not persist as dialog defaults.
checked	unchecked	The dialog always opens when the menu item is selected. Changes are accepted and persist as dialog defaults.
unchecked	checked	The dialog does not appear and the function executes using the current dialog defaults.
unchecked	unchecked	The dialog will appear once; either when the menu item is selected or when Customize is selected from the menu item's context menu in the Object Explorer. After that, the dialog does not appear and the function executes using the current dialog defaults.

function associated to the menu item, and “dialog” refers to the dialog associated to that function.

## Modifying Menu Items

MenuItem objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

## Using Commands

The `gui Modify` command is used to modify an existing menu item. Specify the Name of the MenuItem to modify, and the properties to modify with their new values.

The following command will add status bar text for the “Square Root” dialog created in the section Creating Menu Items (page 627).

```
gui Modify(classname="MenuItem",  
          Name="$SPLUSMenuBar$MyStats$Sqrt",  
          StatusBarText="Calculate a square root.")
```

## Using the Property Dialog

MenuItem objects can be modified through the same property dialogs which are used to create them. To modify a MenuItem object, open the Object Explorer to a page with filtering set to “MenuItem.” Right-click on the MenuItem object’s icon in the right pane and choose MenuItem from the context menu. See the previous sections for details on using the property dialog.

## Using the Context Menu

MenuItem objects can be modified with their context menus which are accessible through the Object Explorer. The following choices appear after right-clicking on a MenuItem object in the Object Explorer.

**Insert MenuItem** Select this to create a new MenuItem object.

**Customize** This appears when Action is set to FUNCTION. Select this to open the dialog associated to the function. Any changes to the dialog persist as dialog defaults.

**Show Dialog On Run** This appears when Action is set to FUNCTION. Check this to cause the dialog associated to the function to open when the item is selected. See Table 17.1 for details.

**Always Use Defaults** This appears when Action is set to FUNCTION. Check this to force the use of the default values when the function executes. See Table 17.1 for details. S-PLUS makes a distinction between the default



argument values for a function as defined in the function's dialog (via the `FunctionInfo` object) and as defined by the function itself. Always **Use Defaults** refers to the “dialog” defaults.

**Hide** Select this to hide the menu item. It will not appear in the menu system and the `MenuItem` object icon will appear grayed out.

**Delete** Select this to delete the `MenuItem` object. The menu item will no longer be available.

**Save** Select this to save the `MenuItem` object, and any other `MenuItem` it contains in the menu hierarchy, to a file.

**Save As** Similar to **Save**, but this allows you to save a copy of the `MenuItem` object to a different filename.

**MenuItem** Select this to access the `MenuItem` page of the `MenuItem` object's property dialog.

**Command** Select this to access the **Command** page of the `MenuItem` object's property dialog.

**Show Menu In S-PLUS** Select this to cause the menu to be displayed in the main S-PLUS menu bar. This choice is available only for `MenuItem` objects having **Type Menu**.

**Restore Default Menus** Select this to restore the default S-PLUS menus in the main menu bar. For example, this will undo the effect of selecting **Show Menu In S-PLUS**. This choice is available only for `MenuItem` objects having **Type Menu**.

**Save MenuItem Object as default** Select this to make the `MenuItem` object the default. When a new `MenuItem` object is created, its property dialog will initially resemble that of the default object, except in the **Name** field.

**Help** Select this to open a help page describing `MenuItem` objects.

## Manipulating Menu Items in the Object Explorer

Menu items are easily copied, moved, and deleted through the Object Explorer.

### Moving Menu Items

To move a menu item into a different menu, locate the menu item icon in the Object Explorer. Select the icon, hold down the ALT key, and drag it onto the menu to which it is to be added.

To move the menu item within its current menu, hold down the SHIFT key and drag the menu item icon to the desired location.

### Copying Menu Items

To copy a menu item into a different menu, hold down the CTRL key and drag its icon onto the menu to which it is to be added.

To copy a menu item within its current menu, hold down the SHIFT and CTRL keys and drag the menu item icon to the desired location.

### Deleting Menu Items

To delete a menu item, right-click on the menu item in the Object Explorer and select Delete from the context menu.

## Displaying Menus

If the user modifies the default menu, which by default is named `SPI usMenuBar`, the modifications will be displayed upon changing the window in focus. If the user creates a new menu, the menu must be explicitly displayed in S-PLUS. This may be done programmatically, or in the Object Explorer.

### Using Commands

The function `gui Di spl ayMenu` will display the specified menu as the main menu in S-PLUS. As a somewhat nonsensical example, we can set the context-menu for `l m` to be the main menu bar, and then restore the menus to the default of `SPI usMenuBar`:

```
gui Di spl ayMenu("l m")  
gui Di spl ayMenu("SPI usMenuBar")
```

### Using the Object Explorer

After creating a menu system, in the Object Explorer right-click on the `MenuItem` object which you want used as the main menu. Select `Show Menu In S-PLUS` from the context menu to display the menu system.

To restore the default S-PLUS menus, select `Restore Default Menus` in the context menu for that same `MenuItem` object. Alternatively, select `Show Menu In S-PLUS` in the context menu for the `MenuItem` object which represents the default S-PLUS menus.

## Saving and Opening Menus

Menus may be saved as external files. These files may be opened at a later time to recreate the menu in S-PLUS.

---

**Using Commands**    The gui Save command is used to save a menu as an external file:

```
gui Save(classname="MenuItem", Name="SPI usMenuBar",  
         FileName="MyMenu. smn")
```

The gui Open command is used to open a menu file:

```
gui Open(classname="MenuItem", FileName="MyMenu. smn")
```

## Using the Object Explorer

To save a menu to an external file, right-click on the MenuItem object in the Object Explorer and select Save As in the context menu. Enter a filename in the Save As dialog and click OK. The extension .SMN is added to the filename.

To open a menu which has been saved in an external file, right-click on the default MenuItem object and select Open from the context menu. In the Open dialog, navigate to the desired file, select it, and click OK. The new menu is visible in the Object Explorer. Its name is the name of the external file, without the extension .SMN.

## TOOLBARS AND PALETTES

In S-PLUS, toolbars and palettes represent the same type of object. When a toolbar is dragged into the client area below the other toolbars, it is displayed there as a palette. When a palette is dragged to the non-client area, close to a toolbar or menu bar, it “docks” there as a toolbar.

Toolbars are represented in the Object Explorer as `Toolbar` objects. These contain `ToolbarButton` objects which represent their buttons.

While it is not hard to create or modify toolbars through the user interface, as shown in this section, it is sometimes easier to work with toolbars programmatically using the `guiCreate` and `guiModify` commands. For details on the syntax, see the `guiCreate("Toolbar")` and `guiCreate("ToolbarButton")` help files in the Language Reference help.

### Creating Toolbars

Toolbars may be created using commands or from within the Object Explorer.

### Using Commands

To create a menu item, use the `guiCreate` function with `classname="Toolbar"`.

The following command will create a new toolbar:

```
guiCreate(classname="Toolbar", Name="My Toolbar")
```

This will add a small empty toolbar which by default will be docked below the active document toolbar. Until we add buttons, the toolbar is not particularly interesting or useful.

### Using the Object Explorer

To create a `Toolbar` object, first open the Object Explorer and filter by “Toolbar” to see the toolbars and toolbar buttons. To create a new toolbar, right-click on the default object icon, labeled “Toolbar,” in the left pane of the Object Explorer. Select `New Toolbar` from the context menu.

(Alternatively, right-click in the S-PLUS application window, outside of any open document window, and choose New Toolbar from the context menu.) The New Toolbar dialog appears, as shown in Figure 17.3.



*Figure 17.3: The New Toolbar dialog.*

To modify the default settings that appear in this property dialog in the future, right-click on the default object icon, choose Properties from the context menu, fill out the dialog with the desired defaults, and click OK.

**Toolbar Name** Enter a name for the new toolbar.

**Make Toolbar for this Folder** Enter a path for a folder (directory). The new toolbar will contain a toolbar button for each file in the indicated folder. Use the Browse button, if desired, to identify the folder. If no folder is specified, the toolbar will contain a single button with the ToolbarButton defaults.

**Document Type** Select the document types which will, when in focus, allow the toolbar to be visible.

Click OK and a new Toolbar object appears in the Object Explorer.

## Toolbar Object Properties

The properties of a Toolbar object determine characteristics such as the name and location of the toolbar. These properties may be specified and modified using the property dialog for the Toolbar object, or programmatically via the commands `gui Create` and `gui Modify`. See the `gui Create("Toolbar")` help file in the Language Reference help for syntax details.

The following properties are specified in the Toolbar property dialog, shown in Figure 17.4.

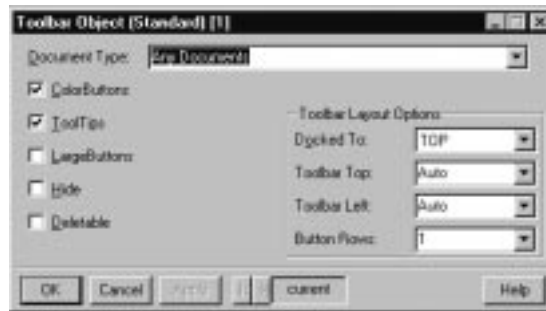


Figure 17.4: The Toolbar Property dialog.

**Document Type** Depending on the type of document window—Graph Sheet, Commands window, etc.—which has the focus, a toolbar may or may not be visible. Specify the document types for which the toolbar should be visible. The choice "All Documents" causes the toolbar to be always visible. The choice "No Documents" ensures that the toolbar will be visible when no document window has the focus; for example, when no window is open.

**ColorButtons** Logical value indicating whether to display button images in color.

**ToolTips** Logical value indicating whether to enable tool tips for the toolbar.

**LargeButtons** Logical value indicating whether to display large-sized buttons.

**Hide** Logical value indicating whether to hide the toolbar.

**Deletable** Logical value indicating whether to allow permanent deletion of the toolbar.

**Docked To** The side of the S-PLUS window to which the toolbar will be docked, or NONE to float the toolbar as a palette.

**Toolbar Top** The top coordinate of the toolbar in pixels.

**Toolbar Left** The left coordinate of the toolbar in pixels.

**Button Rows** The number of rows of buttons in the toolbar.

## Modifying Toolbars

Toolbar objects can be modified using either programming commands, their property dialogs or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

## Using Commands

The `gui Modify` command is used to modify an existing Toolbar. Specify the Name of the Toolbar to modify, and the properties to modify with their new values.

The position of the Toolbar on the screen is specified by the pixel location of the Top and Left corners of the Toolbar. The following command will automatically set these values so that the Toolbar is placed in the upper left corner of the screen:

```
gui Modify(classname="Tool bar", Name="My Tool bar",
           Top="Auto", Left="Auto")
```

## Using the Property Dialog

Toolbar objects can be modified through the same property dialogs which are used to create them. To modify a Toolbar object, open the Object Explorer to a page with filtering set to "Toolbar." Right-click on the Toolbar object's icon in the right pane and choose Properties from the context menu. See the previous sections for details on using the property dialog.

## Using the Context Menu

Toolbar objects can be modified with their context menus which are accessible through the Object Explorer. The following choices appear after right-clicking on a Toolbar object in the Object Explorer.

**New Toolbar** Select this to open a new toolbar.

**New Button** Select this to add a new button to the toolbar.

**Hide** Select this to hide the toolbar.

**Delete** Select this to delete the toolbar.

**Open** Select this to open a toolbar that has been saved in an external file.

**Save** Select this to save a toolbar to its external file, when one exists.

**Save As** Select this to save a toolbar to an external file.

**Unload** Select this to unload a toolbar from memory. The toolbar is no longer available for display. To reload a built-in toolbar, restart S-PLUS. To reload a toolbar that has been saved to an external file, open that file.

**Restore Default Toolbar** Select this to restore a built-in toolbar to its default state after it has been modified.

**Properties** Select this to display the property dialog for the Toolbar object.

**Buttons** Select this to display a dialog used for displaying or hiding different buttons on the toolbar.

**Refresh Icons** Select this to refresh the icon images on the toolbar buttons after they may have been modified.

**Save Toolbar Object as default** Save a modified version of a toolbar as the default for that toolbar.

**Help** Select this to display a help page on toolbars.

## Creating Toolbar Buttons

A Toolbar generally contains multiple toolbar buttons, each of which performs an action when pressed. Toolbar buttons may be created using commands or from within the Object Explorer.

### Using Commands

To create a `ToolBarButton`, use the `guiCreate` function with `className="ToolBarButton"`. The Name of the button determines the toolbar upon which it is placed.

The following command creates a toolbar button which launches the Linear Regression dialog:

```
guiCreate( "ToolBarButton", Name = "My Tool bar$Li nreg",  
          Action="Function", Command="menuLm")
```

Creating sophisticated dialogs such as the Linear Regression dialog is discussed later in the section Dialogs (page 647) and the section Style Guidelines (page 706).

Toolbar buttons can also be used to call built-in Windows interface commands. The following command will create a toolbar button which launches the standard file Open dialog:

```
guiCreate( "ToolBarButton", Name = "My Tool bar$Open",  
          Action="Bui l t l n",  
          Bui l t l nOperati on="$$SPI usMenuBar$No_Documents$Fi l e$Open")
```



## Using the Object Explorer

To add a button to an existing toolbar, right-click on the corresponding Toolbar object in the Object Explorer and select New Button from the context menu. The `ToolBarButton` property dialog appears, as in Figure 17.5.



Figure 17.5: The `ToolBarButton` property dialog.

## ToolBarButton Object Properties

The properties of a `ToolBarButton` object determine characteristics such as the button image for the menu item and the action performed when the button is selected. These properties may be specified and modified using the property dialog for the `ToolBarButton` object, or programmatically via the commands `gui Create` and `gui Modify`. See the `gui Create("ToolBarButton")` help file in the Language Reference help for syntax details.

The following properties are specified on the first tab of the `ToolBarButton` property dialog, shown in Figure 17.5:

**Name** The name of the button.

**Type** Select `BUTTON` to create a button, or select `SEPARATOR` to create a gap between buttons in the toolbar.

**Action** This applies to `ToolBarButton` objects of type `BUTTON`.

- `NONE`. No action is performed when the button is clicked.
- `BUILTIN`. One of the actions associated with the default menus or toolbars is performed when the item is selected. These are listed on the Command page in the Built-In Operation dropdown box. This option allows you to use in a customized toolbar any of the "intrinsic" menu or toolbar actions, such as Window/Cascade.

- **FUNCTION.** Under this option, an S-PLUS function is executed when the button is clicked. Optionally, the dialog for the function can be made to appear.
- **OPEN.** The file specified on the Command page is opened when the button is clicked. The file will be opened by the application associated to it by the operating system.
- **PRINT.** The file specified on the Command page is printed when the button is clicked. The file will be printed by the application currently associated to it by the operating system.
- **RUN.** The file specified on the Command page is opened and run as a script by S-PLUS when the button is clicked.

**Tip Text** The tool tip text for the button.

**Hide** Logical value indicating whether to make the button invisible. When the item is hidden, its icon in the Object Explorer appears grayed out. This can also be specified through the context menu.

**Deletable** Logical value indicating whether to allow the item to be deleted.

The next set of properties are found on the Command page of the ToolBarButton property dialog.

**Built-In Operation** Type of action to perform when the button is selected.

**Command** The name of an S-PLUS function, or path and filename. This field is enabled when Action is set to FUNCTION, OPEN, PRINT, or RUN on the button page. Use the Browse button to identify the folder.

**Parameters** This is relevant when the Action is set to FUNCTION. This property specifies the arguments for the function which will execute when the item is selected. The easiest way to specify these arguments is to work through the Customize dialog available through the context menu for the ToolBarButton in the Object Explorer. For details on doing this, see the section Using the Context Menu (page 643) below.

**Show Dialog On Run** This is relevant when Action is set to FUNCTION. Logical value indicating whether to display the dialog associated with the specified function when the button is selected.

**Always Use Defaults** This is relevant when Action is set to FUNCTION. Logical value indicating whether to force the use of the default values when the function executes.

S-PLUS makes a distinction between the default argument values for a function as defined in the function's dialog (via the `FunctionInfo` object) and as defined by the function itself. Always Use Defaults refers to the *dialog* defaults. Table 17.1 above summarizes how Show Dialog On Run and Always Use Defaults work together.

The next set of properties are found on the Image page of the `ToolBarButton` property dialog.

**Image FileName** The path and filename of a bitmap file whose image will be displayed on the toolbar button. Use the Browse button, if desired, to identify the file.

To modify a `ToolBarButton` object, use either the `ToolBarButton` property dialog described above or the context menu, described below.

## Modifying Toolbar Buttons

`ToolBarButton` objects can be modified using either programming commands, their property dialogs or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

## Using Commands

The `guiModify` command is used to modify an existing toolbar button. Specify the `Name` of the `ToolBarButton` to modify, and the properties to modify with their new values.

The following command will specify a new value for the tooltip text, which is the text displayed when the mouse is hovered over the button:

```
guiModify( "ToolBarButton", Name = "My Toolbar$Open",
  TipText="Open Document")
```

## Using the Property Dialog

`ToolBarButton` objects can be modified through the same property dialogs which are used to create them. To modify a `ToolBarButton` object, open the Object Explorer to a page with filtering set to "Toolbar". Right-click on the `ToolBarButton` object's icon in the right pane and choose "Button" from the context menu. See the previous sections for details on using the property dialog.

## Using the Context Menu

`ToolBarButton` objects can be modified with their context menus which are accessible through the Object Explorer. The following choices appear after right-clicking on a `ToolBarButton` object in the Object Explorer.

**Insert Button** Select this to insert a new toolbar button next to the current one.

**Customize** This appears when Action is set to FUNCTION. Select this to open the dialog associated to the function. Any changes to the dialog persist as dialog defaults.

**Hide** Select this to hide the toolbar button.

**Delete** Select this to delete the toolbar button.

**Edit Image** Select this to open the bitmap file, using the operating systems default bitmap editor, which contains the icon image of the toolbar button.

**Button.** Select this to open the Button page of the property dialog for the toolbar button.

**Command** Select this to open the Command page of the property dialog for the toolbar button.

**Image** Select this to open the Image page of the property dialog for the toolbar button.

**Save ToolbarButton Object as default** Select this to save a copy of the ToolbarButton object as the default ToolbarButton object.

**Help** Select this to open a help page on toolbar buttons.

## Manipulating Toolbars in the Object Explorer

Toolbar buttons are easily copied, moved, and deleted through the Object Explorer.

## Displaying Toolbars

### Using Commands

The `Hide` property of a toolbar determines whether or not it is displayed. To display a toolbar, set this property to false:

```
gui Modify (classname="Tool bar", Name="My Tool bar", Hide=F)
```

To hide the toolbar, set this property to true:

```
gui Modify (classname="Tool bar", Name="My Tool bar", Hide=T)
```

## Using the Toolbars Dialog

To hide (or unhide) a toolbar, right-click on the Toolbar object and select Hide (or Unhide) from the context menu. To selectively hide or display toolbars, right-click outside of any open windows or toolbars and select Toolbars from the context menu. A dialog like that shown in Figure 17.6

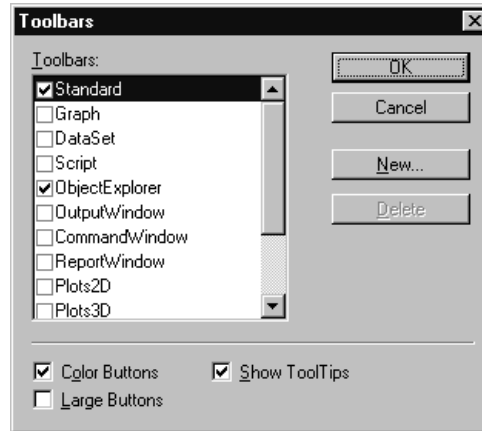


Figure 17.6: The Toolbars dialog.

appears. Use the checkboxes to specify which toolbars will be visible.

To hide (or unhide) a toolbar button, right-click on the ToolbarButton object and select Hide (or Unhide) from the context menu. To selectively hide or display the buttons in a toolbar, right-click the Toolbar object and select Buttons from the context menu. A dialog like that shown in Figure 17.7 appears. Use the checkboxes to specify which buttons will be visible in the toolbar.

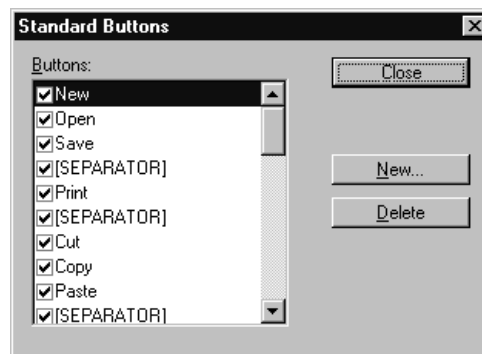


Figure 17.7: The Buttons dialog.

**Saving and Opening Toolbars**

A toolbar and the related toolbar buttons may be saved to an external file. This file may be opened at a later time to restore the toolbar and the toolbar buttons.

**Using Commands**    The gui Save command is used to save a toolbar as an external file:

```
gui Save(cI assname="Tool bar", Name="My Tool bar",  
        Fi l eName="MyTool bar. stb")
```

The gui Open command is used to open a toolbar file:

```
gui Open(cI assname="Tool bar", Fi l eName="MyTool bar. stb")
```

<b>Note</b>
Do not try to open a toolbar file while the toolbar it represents is loaded into S-PLUS; this results in an error message. You can see which toolbars are currently loaded by right-clicking in the S-PLUS window outside of any open document windows. To unload a toolbar, go to the Object Explorer and right-click on the toolbar item, and choose Unload.

**Using the Object Explorer**

To save a toolbar to an external file, right-click on the Toolbar object in the Object Explorer and select Save As in the context menu. Enter a filename in the Save As dialog and click OK. The extension **.STB** is added to the filename.

To open a toolbar which has been saved in an external file, right-click on the default Toolbar object and select Open from the context menu. In the Open dialog, navigate to the desired file, select it, and click OK. The new toolbar is visible in the Object Explorer. Its name is the name of the external file, without the extension **.STB**.

---

## DIALOGS

Almost all of the dialogs in S-PLUS have either a corresponding graphical user interface object or a corresponding function.

The dialog for a GUI object such as a BoxPlot displays the properties of the object, and allows the modification of these properties. When Apply or OK is pressed, the object is then modified to have the newly specified properties. While these dialogs are created using the same infrastructure as is discussed here, they are not generally modified by the user.

The dialog for a function allows the user to specify the arguments to the function. The function is then called with these arguments when Apply or OK is pressed. In S-PLUS, users may write their own functions and create customized dialogs corresponding to the functions. This section discusses the creation of such dialogs.

Think of a function dialog as the visual version of some S-PLUS function. For every function dialog there is one S-PLUS function, and for every S-PLUS function there is a dialog. The dialog controls in the dialog correspond to arguments in the function, and vice versa. In addition, all function dialogs are displayed with OK, Cancel, Apply (modeless) buttons that do not have any corresponding arguments in the functions. When the OK or Apply button is pressed, the function is executed with argument values taken from the current values of the dialog controls.

A dialog typically consists of one or more tabbed pages, each containing groups of controls. The characteristics of the controls in the dialog are defined by Property objects. Properties may be of type Page, Group, or Normal. A Page will contain Groups which in turn contain Normal properties. The primary information regarding Pages and Groups is their name, prompt, and what other properties they contain. Normal Properties have far more characteristics describing features such as the type of control to use, default values, option lists, and whether to quote the field's value when providing it in the function call. Together the Property objects determine the look of the dialog and its controls.

Filter by “Property” in the Object Explorer (Figure 17.8) to see objects of this type.

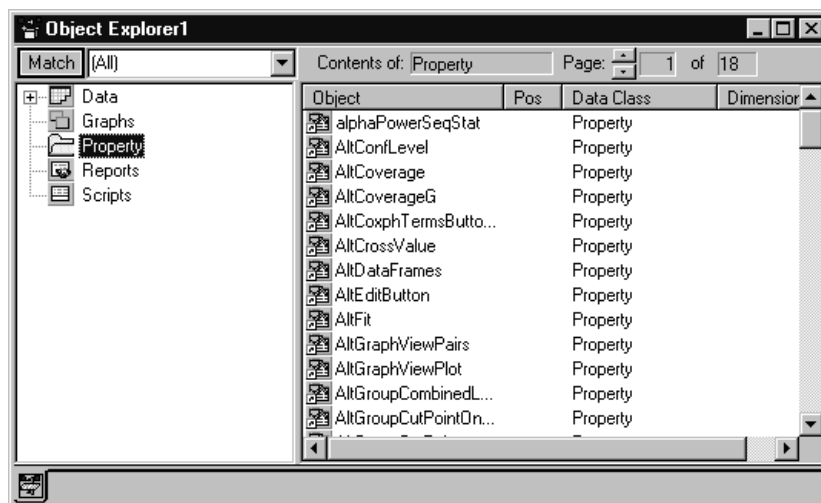


Figure 17.8: The Object Explorer showing all Property objects.

While the Property objects define the controls in a dialog, they do not contain information on which Property objects relate to each of the arguments in the function. This information is contained in a FunctionInfo object. Each function for which a dialog is constructed needs a FunctionInfo object describing what Property objects to use when constructing the dialog for the function, as well as other related information. If a function does not have a FunctionInfo object and its dialog is requested by a MenuItem or ToolbarButton, a simple default dialog will be launched in which an edit field is present for each argument to the function.



Filter by "FunctionInfo" in the Object Explorer (Figure 17.9) to see objects of this type.

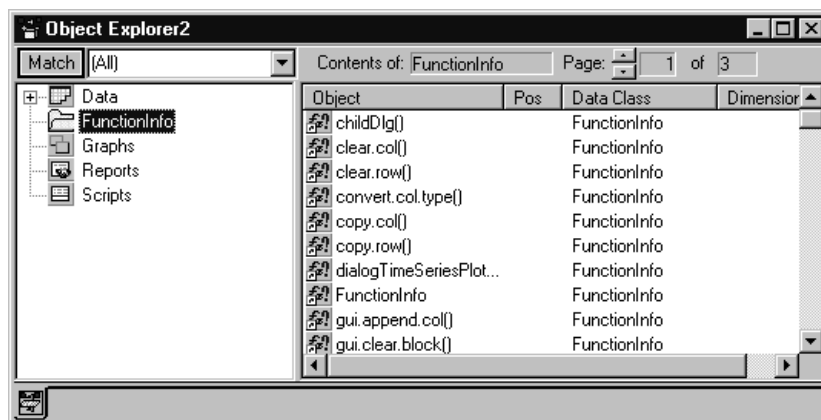


Figure 17.9: The Object Explorer showing all FunctionInfo objects.

While it is not hard to create or modify Property and FunctionInfo objects through the user interface, as shown in this section, it is usually preferable to work with them programmatically using the `gui Create` and `gui Modify` commands. For details on the syntax, see the `gui Create("Property")` and `gui Create("FunctionInfo")` help files in the Language Reference help.

## Creating Dialogs

To create a dialog in S-PLUS, follow these steps:

1. Identify the S-PLUS function which will be called by the dialog. This can be either a built-in or a user-created function.
2. Create the "Property" objects, such as pages, group boxes, list boxes, and check boxes, which will populate the dialog.
3. Create a "FunctionInfo" object having the same name as the function in step 1. The `FunctionInfo` object holds the layout information of the dialog, associates the values of the Property objects in the dialog with values for the arguments of the S-PLUS function, and causes the S-PLUS function to execute.

**Creating  
Property  
Objects**

Property objects may be created using commands or from within the Object Explorer.

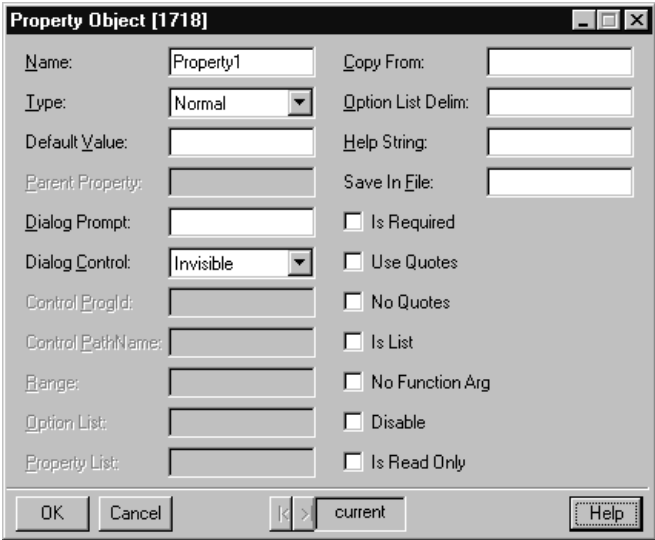
**Using Commands**

To create a Property object, use `gui Create` with `cl assname="Property"`. The following command will create a list box:

```
gui Create(cl assname="Property", Name="MyLi stProperty",  
Type="Normal ", Di al ogControl ="Li st Box",  
Di al ogPrompt="Method", Opti onLi st=c("MVE", "MLE",  
"Robust"), Defaul tVal ue="MLE")
```

**Using the Object  
Explorer**

To create a Property object, open the Object Explorer to a page with filtering set to "Property." Right-click on any property in the right pane and choose `Create Property` from the context menu. The property dialog shown in Figure 17.10 appears.



*Figure 17.10: The property dialog for a Property object.*

**Property  
Object  
Properties**

The properties of a Property object determine characteristics such as the prompt text, control type, and default value. These properties may be specified and modified using the property dialog for the Property object, or

programmatically via the commands `gui Create` and `gui Modify`. See the `gui Create("Property")` help file in the Language Reference help for syntax details.

The following properties are specified in the Property object property dialog, shown in Figure 17.10.

**Name** The name of the Property object. To create a Property object, a name must be specified.

**Type** The type of property. `Group` or `WideGroup` creates a group box. `Page` creates a tabbed page. `Normal` to creates any other type of Property object.

**Default Value** The default value for the Property object. This will be displayed when the dialog opens.

**Parent Property** The name of a parent property, if any. This is used by certain internal Property objects.

**Dialog Prompt** The text for the label which will appear next to the control in the dialog.

**Dialog Control** The type of control to use. Examples are `Button`, `Check Box`, `List Box`, and `Combo Box`. Control types are described in the section `Dialog Controls` (page 660).

**Range** The range of acceptable values for the function argument associated with this property. For instance, if the values must be between 1 and 10, enter `1: 10`.

**Option List** A comma-separated list. The elements of the list are used, for example, as the labels of `Radio Buttons` or as the choices in the dropdown box of a `String List Box`. A property may have either a range or an option list, but not both. Ranges are appropriate for continuous variables. Option lists are appropriate where there is a finite list of allowable values.

**Property List** A comma-separated list of the Property objects included in the `Group box` or on the `Page`. This applies to Property objects having `Type Page` or `Group`.

**Tip...**

A Property object may only be called once by a given `FunctionInfo` object.

**Copy From** The name of a Property object to be used as a template. The current Property object will differ from the template only where specified in the property dialog. See the section Dialog Controls (page 660) for lists of internal and standard Property objects that can be used in dialogs via Copy From.

**Option List Delim** A character used as the delimiter for Option List, such as comma, colon or semi-colon. Comma is the default.

**Help String** The text of the tool tip for this Property object.

**Save in File** The name of the file in which to save the Property definition.

**Is Required** Logical value indicating whether to require the Property object to have a value when OK or Apply is clicked in the dialog.

**Use Quotes** Logical value indicating whether to force quotes to be placed around the value of the Property object when the value is passed to the S-PLUS function.

**No Quotes** Logical value indicating whether to prohibit quotes from being placed around the value of the Property object when the value is passed to the S-PLUS function. This option is ignored when Is List (described below) is not checked.

**Is List** Logical value indicating whether to cause a multiple selection in a drop-down list to be passed as an S-PLUS list object to the S-PLUS function.

**No Function Arg** Logical value indicating whether to not pass the value of this Property object as an argument to the S-PLUS function. The Property object must still be referenced by the `FunctionInfo` object.

**Disable** Logical value indicating whether to cause the Property object to be disabled when the dialog starts up.

**Is Read Only** Logical value indicating whether the corresponding control is for read only.

## Modifying Property Objects

Property objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

**Using Commands** The `gui Modify` command is used to modify an existing Property object. Specify the Name of the Property object to modify, and the properties to modify with their new values.

```
gui Modify(classname="Property", Name="MyListProperty",
           DefaultValue="Robust")
```

**Using the Property Dialog** Property objects may be modified through the Property object property dialog.

To modify a Property object, open the Object Explorer to a page with filtering set to "Property." Right click on the Property object's icon in the right pane and choose Properties from the context menu. Refer to the previous sections for details on using the property dialog.

**Using the Context Menu** Property objects can be modified with their context menus. The context menu for an object is launched by right-clicking on the object in the Object Explorer. The context menu provides options such as creating, copying, and pasting the object, as well as a way to launch the property dialog.

**Creating FunctionInfo Objects** FunctionInfo objects may be created using commands or from within the Object Explorer.

**Using Commands** To create a FunctionInfo object, use the `gui Create` command with `classname="FunctionInfo"`.

As a simple example, we will create a function `my.sqrt` which calculates and prints the square root of a number. We will create a dialog for this function and add a menu item to the Data menu which launches the dialog. We will create a property `MySqrtInput` specifying the input value, and as we do not wish to store the result we will use the standard property `SProperty invisibleReturnObject` for the result.

```
my.sqrt <- function(x){
  y <- sqrt(x)
  cat("\nThe square root of ", x, " is ", y, ".\n", sep="")

  invisible(y)
}

gui Create(classname="Property", Name="MySqrtInput",
           DialogControl="String", UseQuotes=F,
```

```

DialogPrompt="Input Value")
gui Create(classname="FunctionInfo", Name="my.sqrt",
  DialogHeader="Calculate Square Root",
  PropertyList="SPropInvisibleReturnObject, MySqrtInput",
  ArgumentList="#0=SPropInvisibleReturnObject#1=MySqrtInput")
gui Create(classname="MenuItem",
  Name="$$SPUsMenuBar$Data$MySqrt",
  Type="MenuItem", MenuItemText="Square Root...",
  Action="Function", Command="my.sqrt")

```

### Using the Object Explorer

Open the Object Explorer to a page with filtering set to “FunctionInfo.” Right-click on any FunctionInfo object in the right pane and choose Create FunctionInfo from the context menu. The property dialog shown in Figure 17.11 appears.

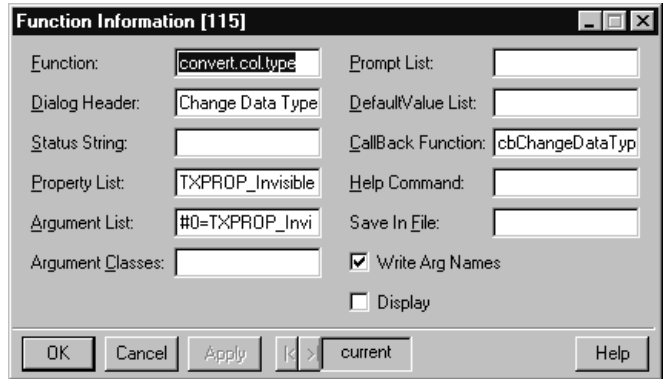


Figure 17.11: The property dialog for a FunctionInfo object.

### FunctionInfo Object Properties

The properties of a FunctionInfo object describe how the properties in a dialog correspond to the related function. These properties may be specified and modified using the property dialog for the FunctionInfo object, or programmatically via the commands `gui Create` and `gui Modify`. See the `gui Create("FunctionInfo")` help file in the Language Reference help for syntax details.

The following properties are specified in the FunctionInfo object property dialog, shown in Figure 17.11.

**Function Name** The name of the S-PLUS function which will execute when OK or Apply is clicked in the dialog. This is also the name of the FunctionInfo object.

**Dialog Header** The text that will appear at the top of the dialog.

**Property List** A comma-separated list of Property objects to be displayed in the dialog. A given Property object can only occur once in this list. If pages or group boxes are specified, it is not necessary to specify the Property objects that they comprise. Property objects in the list will be displayed in two columns, moving in order from top to bottom, first in the left-hand column and next in the right-hand column.

**Argument List** A comma-separated list in the form `#0 = PropName1, #1 = PropName2, ...`. Here *PropName1*, *PropName2*, ..., are names of Property objects, not including page and group objects, and #1, ..., refer in order to the arguments of the function indicated in Function Name. The argument names may be used in place of #1, #2, ... . The first item, #0, refers to the returned value of the function. Use Argument List if the order of the Property objects in the dialog differs from the order of the corresponding arguments of the S-PLUS function.

**Prompt List** A comma-separated list of labels for the Property objects in the dialog. These will override the default labels. The syntax for this list is the same as that for Argument List.

**Default Value List** A comma-separated list of default values for the Property objects. These will override the default values of the Property objects. The syntax for this list is the same as that for Argument List.

**Callback Function** The name of a function which will be executed on exit of any Property object in the dialog. Callback Functions are described in detail in the section Callback Functions (page 694).

**Help Command** The command to be executed when the Help button is pushed. This is an S-PLUS expression such as `help(my.function)`.

**Write Argument Names** Logical value indicating whether to have argument names written when the function call is made.

**Display** Logical value indicating whether to cause information about the `FunctionInfo` object to be written in a message window (or in the output pane of a script window when the dialog is launched by a script). This debugging tool is turned off after OK or Apply is clicked in the dialog.

## Modifying FunctionInfo Objects

FunctionInfo objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

**Using Commands** The `gui Modify` command is used to modify an existing `FunctionInfo` object. Specify the `Name` of the `FunctionInfo` object to modify, and the properties to modify with their new values.

```
gui Modify(classname="FunctionInfo", Name="my.sqrt",  
          DialogHeader="Compute Square Root")
```

**Using the Property Dialog** `FunctionInfo` objects may be modified through the `FunctionInfo` object property dialog.

To modify a `FunctionInfo` object, open the Object Explorer to a page with filtering set to “`FunctionInfo`.” Right click on the `FunctionInfo` object’s icon in the right pane and choose `Properties` from the context menu. Refer to the previous sections for details on using the dialog.

**Using the Context Menu** `FunctionInfo` objects can be modified with their context menus. The context menu for an object is launched by right-clicking on the object in the Object Explorer. The context menu provides options such as creating, copying, and pasting the object, as well as a way to launch the property dialog.

## Displaying Dialogs

There are several ways to display a dialog in S-PLUS.

- Locate the associated function in the Object Explorer and double-click on its icon. If a function is not associated with a `FunctionInfo` object, then double-clicking on its icon will cause a default dialog to be displayed.
- Click on a toolbar button which is linked to the associated function.
- Select a menu item which is linked to the associated function.
- Use the function `gui DisplayDialog` in a Script or Commands window.

```
gui DisplayDialog("Function", Name="menuLm")
```

- Write the name of the function in a script window, double-click on the name to select it, right-click to get a menu, and choose `Show Dialog`.



## Example: The Contingency Table Dialog

This example looks into the structure behind the Contingency Table dialog. The Contingency Table dialog in S-PLUS (Figure 17.12) is found under **Statistics ► Data Summaries ► Crosstabulations**.

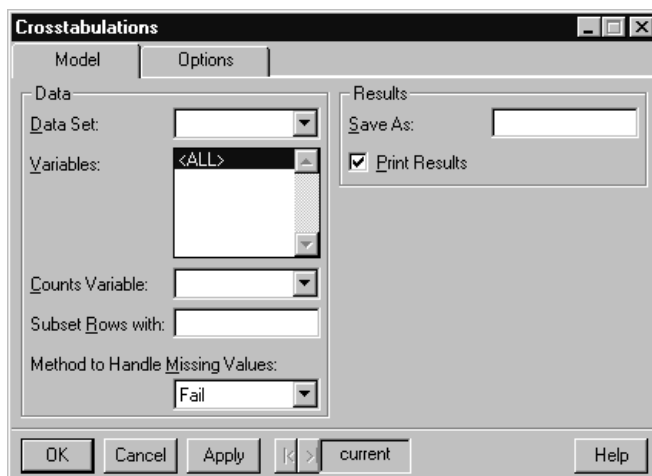


Figure 17.12: The Contingency Table dialog.

It has two tabbed pages, named Model and Options. On the Model page are two group boxes, named Data and Results.

The `FunctionInfo` object for this dialog is called `menuCrosstabs`; its property dialog is shown in Figure 17.13 and is described below.

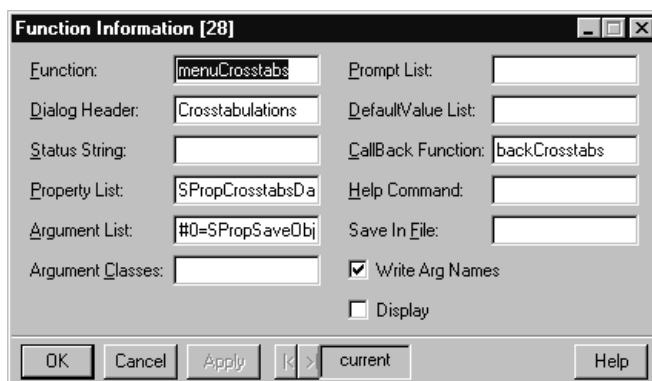


Figure 17.13: The property dialog for the `FunctionInfo` object `menuCrosstabs`.

**Function Name.** Notice that this value is also `menuCrosstabs`; the S-PLUS function associated with this dialog has the same name as the `FunctionInfo` object. To look at the code behind the function `menuCrosstabs`, type `menuCrosstabs`, or `page(menuCrosstabs)` at the prompt in the Commands window.

**Dialog Header** This is the header which appears at the top of the Contingency Table dialog. Try changing this and opening the dialog. The dialog will reflect the change. This change persists when S-PLUS is exited and restarted.

**Status String** This is currently empty. Try entering text here (do not forget to click Apply or OK) and opening the dialog.

**Property List** This shows only the Property objects for the two tabbed pages: `SPropCrosstabsDataPage` and `SPropCrosstabsOptionsPage`. To more easily see these values, right-click in the edit field and select Zoom. The zoom box shown in Figure 17.14 appears.



*Figure 17.14: The zoom box shows the Property List.*

Using the Object Explorer, open the property dialog for the first of these. This is shown in Figure 17.15.

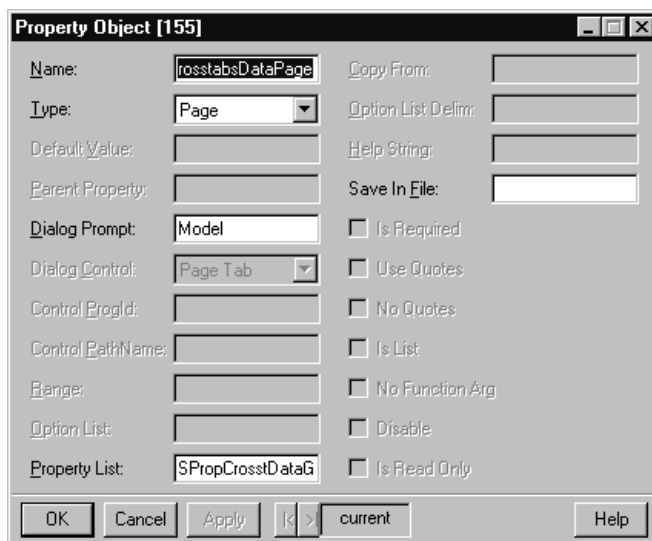


Figure 17.15: The Property dialog for the *SPropCrosstabsDataPage* Property object.

**Argument List** Use zoom, if desired, to view the assignments of Property object values to arguments of the function `menuCrosstabs`. Notice in Figure 17.13 that the return value is set to `SPropSaveObj`. This has been done consistently throughout the user interface.

**Prompt List** Since this is empty, fields in the dialog will have their default prompts (labels) as specified in their corresponding property objects.

**Default Value List** Since this is empty, fields in the dialog will have the default values as specified in their corresponding property objects.

**Call Back Function** The S-PLUS function `backCrosstabs` is executed each time a control in the dialog is exited. To look at the code behind the function, type

```
backCrosstabs
```

at the prompt in the Commands window. Callback functions are discussed in the section *Callback Functions* (page 694).

**Write Arg Names** This is currently empty.

**Display** This is not checked, so debugging messages will not be shown when the dialog is displayed.

# DIALOG CONTROLS

**Control Types** S-PLUS has a variety of dialog controls that can be used to represent the properties of an object (such as a user-defined function) in a dialog. Table 17.2 describes each control type. For more information on dialog controls, see the `gui Create("Property")` help file in the Language Reference help.

Table 17.2: Dialog control types.

Control Type	Description	Example
Invisible	A control which does not appear on the dialog.	<pre>gui Create("Property", Name = "ReturnVal ue", Type = "Normal ", Di al ogControl = "Invi si bl e" )</pre>
Button	A push-button control.  The "DialogPrompt" subcommand is used to set the text inside the button.	<pre>gui Create( "Property", Name = "myButton", Type = "Normal ", Di al ogPrompt = "&amp;MyButton", Di al ogControl = "Button" )</pre>
Check Box	A two-state check box control where one state is checked and the other is unchecked.  The "DefaultValue" subcommand is used to set the state of the check box. If set to "0" or "F", the box will be unchecked. If "1" or "T", the box will be checked.	<pre>gui Create( "Property", Name = "myCheckBox", Type = "Normal ", Defaul tVal ue = "T", Di al ogPrompt = "&amp;My CheckBox", Di al ogControl ="Check Box" )</pre>
Static Text	A text field that is not editable usually used before other controls to title them.  The "DialogPrompt" subcommand is used to specify the text of this static text field.	<pre>gui Create( "Property", Name = "myStati cText", Type = "Normal ", Di al ogPrompt = "MyStati cText", Di al ogControl = "Stati c Text" )</pre>

Table 17.2: Dialog control types.

Control Type	Description	Example
<b>String</b>	An editable field used to enter text.  If the subcommand “UseQuotes” is set to TRUE, the string returned to the user function from this dialog has quotes around it. If not specified, no quotes are added.	<pre>gui Create("Property", Name = "myString", Type = "Normal", DialogControl = "String", DialogPrompt = "&amp;My String", UseQuotes=T )</pre>
<b>Wide String</b>	Same as “String” except that this control takes up two dialog columns.	<pre>gui Create("Property", Name = "myString", Type = "Normal", DialogControl = "Wide String", DialogPrompt = "&amp;My String", UseQuotes=T )</pre>
<b>List Box</b>	A drop-list of strings. Only one string can be selected at a time. The selected string is not editable. The “DefaultValue” is used to specify the string from the list that is selected by default. The list of strings is specified as a comma-delimited list in “OptionList”. An optional subcommand “OptionListDelimiter” can be used to specify the delimiter.	<pre>gui Create( "Property", Name = "myListBox", Type = "Normal", DefaultValue = "Opt2", DialogPrompt = "A ListBox", DialogControl = "List Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", " )</pre>
<b>ComboBox</b>	Similar to a ListBox except that the selected string is editable. This allows the user to enter a string which is not part of the drop-list. Only one string can be selected at a time.	<pre>gui Create( "Property", Name = "myComboBox", Type = "Normal", DefaultValue = "Opt3", DialogPrompt = "A ComboBox", DialogControl = "Combo Box", OptionList = "Opt1, Opt2, Opt3", OptionListDelimiter = ", " )</pre>

*Table 17.2: Dialog control types.*

Control Type	Description	Example
<b>Float</b>	Similar to a String control. This control accepts floating point numbers.	<pre>gui Create( "Property", Name = "myFloat", Type = "Normal", DefaultVal ue = "2.54", Di al ogPrompt = "A Float", Di al ogControl = "Float" )</pre>
<b>Float Auto</b>	Similar to a ComboBox except that there is only one string "Auto" in the drop-list. You can enter a floating point number or select "Auto" from the drop list.	<pre>gui Create( "Property", Name = "myFloatAuto", Type = "Normal", DefaultVal ue = "2.54", Di al ogPrompt = "A FloatAuto", Di al ogControl = "Float Auto" )</pre>
<b>Float Range</b>	Similar to the Float control except that a range of values can be specified using the "Range" subcommand. If the value entered is outside of the range, then an error will be displayed and the dialog will remain open.	<pre>gui Create( "Property", Name = "myFloatRange", Type = "Normal", DefaultVal ue = "2.54", Di al ogPrompt = "A Float Range", Di al ogControl = "Float Range", Range = "1.00: 3.00" )</pre>
<b>Integer</b>	Similar to the Float control. This control accepts integer whole numbers.	<pre>gui Create( "Property", Name = "myInteger", Type = "Normal", DefaultVal ue = "2", Di al ogPrompt = "An Int", Di al ogControl = "Integer" )</pre>
<b>Integer Auto</b>	Similar to a "Float Auto" control except this control accepts integers.	<pre>gui Create( "Property", Name = "myIntAuto", Type = "Normal", DefaultVal ue = "2", Di al ogPrompt = "An IntAuto", Di al ogControl = "Integer Auto" )</pre>

*Table 17.2:Dialog control types.*

Control Type	Description	Example
<b>Integer Range</b>	Similar to a “Float Range” control except this control accepts integers.	<pre>gui Create( "Property", Name = "myIntRange", Type = "Normal", DefaultVal ue = "2", Di al ogPrompt = "An IntRange", Di al ogControl =     "Integer Range", Range = "1: 3" )</pre>
<b>Color List</b>	A drop-list that allows selection of one element from a list of strings representing colors (i.e. Red, Green, etc.) Each list element has drawn a color sample next to it if the string represents a valid color name. Use the “OptionList” subcommand to specify colors in the list.	<pre>gui Create( "Property", Name = "myCol orLi st", Type = "Normal", DefaultVal ue = "Red", Di al ogPrompt = "A Col orLi st", Di al ogControl = "Col or Li st", Opti onLi st = "Bl ue, Red,     Green" )</pre>

*Table 17.2:Dialog control types.*

Control Type	Description	Example
<b>New Line</b>	<p>Inserts an empty area the height of a single control between controls. Useful for inserting space in the second column of controls in a dialog so that a wide control can be used in the first column without overlapping controls in the second column. The “DialogPrompt” subcommand is not used.</p> <p>As an example, suppose you have six controls (not counting the invisible ReturnValue). The first is a String, the second is a Wide String, and all others are non-Wide controls. If you want to lay out the controls so that no overlap occurs in the second column from the second Wide String in the first column, you could insert a New Line control in the PropertyList subcommand for the FunctionInfo object.</p>	<pre>gui Create( "Property", Name = "xLINEFEED", Type = "Normal", DialogPrompt = "A New Line", DialogControl = "New Line" )  gui Create( "FunctionInfo", Function = "TestFn", DialogHeader = "Test", PropertyList = c( "ReturnValue", "aString1", "aWideString", "aString2", "aString3", "xLINEFEED", "aString4", "aString5" ), . . .</pre>
<b>Page Tab</b>	<p>Adds a page of controls and groups of controls to a dialog. You must have at least one group of controls on a page before the page will display.</p>	<pre>gui Create( "Property", Name = "myPageOne", Type = "Page", DialogPrompt = "Page 1", PropertyList = c("myGroup1", "myString"))</pre>



*Table 17.2:Dialog control types.*

Control Type	Description	Example
<b>Multi-select Combo Box</b>	Similar to the Combo Box control except that multiple selections can be made from the drop-list of strings.	<pre> gui Create( "Property", Name = "myMul ti Sel Combo", Type = "Normal", Defaul tVal ue = "Opt3", Di al ogPrompt =     "Mul ti Sel Combo", Di al ogControl =     "Mul ti -sel ect Combo Box", Opti onLi st = "Opt1, Opt2,     Opt3", Opti onLi stDel i mi ter = ", " ) </pre>
<b>Wide Multi-select Combo Box</b>	Similar to the Multi-select Combo Box except this control takes up two dialog columns.	<pre> gui Create( "Property", Name = "myWi deMul ti Sel Combo", Type = "Normal", Defaul tVal ue = "Opt3", Di al ogPrompt =     "Wi de Mul ti Sel Combo", Di al ogControl =     "Wi de Mul ti -sel ect Combo Box", Opti onLi st = "Opt1, Opt2,     Opt3", Opti onLi stDel i mi ter = ", " ) </pre>
<b>Multi-select List Box</b>	Similar to the List Box control except that multiple selections can be made from the string list.	<pre> gui Create( "Property", Name = "myMul ti Sel Li stBox", Type = "Normal", Defaul tVal ue = "Opt2", Di al ogPrompt =     "Mul ti Sel Li stBox", Di al ogControl =     "Mul ti -sel ect Li st Box", Opti onLi st = "Opt1, Opt2,     Opt3", Opti onLi stDel i mi ter = ", " ) </pre>

Table 17.2:Dialog control types.

Control Type	Description	Example
<b>Wide Multi-select List Box</b>	Similar to the Multi-select List Box control except this control takes up two dialog columns.	<pre>gui Create( "Property", Name = "myWideMultiSelListBox", Type = "Normal", DefaultVal ue = "Opt2", Di al ogPrompt =     "Wi de Mul ti Sel Li stBox", Di al ogControl =     "Wi de Mul ti -sel ect Li st Box", Opti onLi st = "Opt1, Opt2,     Opt3", Opti onLi stDel i mi ter = ", " )</pre>
<b>String List</b>	A list box of strings that allows only single selections. This control differs from the List Box and Combo Box controls in that the list of strings is always visible. The “OptionList” subcommand is used to fill the list.	<pre>gui Create( "Property", Name = "myStringList", Type = "Normal", DefaultVal ue = "Opt3", Di al ogPrompt = "Stri ng Li st", Di al ogControl =     "Stri ng Li st Box", Opti onLi st = "Opt1, Opt2,     Opt3")</pre>
<b>Radio Buttons</b>	A group of radio buttons which allow only one button to be selected. The buttons are exclusive which means that if one button is selected and another is clicked on, the original button is deselected and the button clicked on is selected. The “Option-List” subcommand is used to specify the names of the buttons in the group. This name is returned when a button is selected, as in the other list controls.	<pre>gui Create( "Property", Name = "myRadi oButtons", Type = "Normal", DefaultVal ue = "Opt3", Di al ogPrompt = "Radi o Buttons", Di al ogControl =     "Radi o Buttons", Opti onLi st = "Opt1, Opt2,     Opt3", Opti onLi stDel i mi ter = ", " )</pre>

*Table 17.2: Dialog control types.*

Control Type	Description	Example
<b>Integer Spinner</b>	An edit field with two buttons attached to increase or decrease the value in the edit field by some fixed increment. Use the “Range” subcommand to specify the start and end of the range of numbers allowed in the edit field and to specify the small and large increments. The small increment is used when you single-click once on the spinner arrows. The large increment is used when you click and hold on the spinner arrows.	<pre>gui Create( "Property", Name = "myIntSpinner", Type = "Normal", DefaultVal ue = "2", Di al ogPrompt = "Int Spi nner", Di al ogControl =     "Integer Spinner", Range = "-40: 40, 1, 5" )</pre>
<b>Float Spinner</b>	Similar to the Integer Spinner control except this control accepts floating point numbers.	<pre>gui Create( "Property", Name = "myFloatSpinner", Type = "Normal", DefaultVal ue = "2.5", Di al ogPrompt = "Fl oat Spi nner", Di al ogControl =     "Fl oat Spi nner", Range = "-40.5: 40.5, 0.1, 1.0" )</pre>
<b>Integer Slider</b>	A visual slider control that allows adjustment of a numeric value by dragging a lever from one side of the control to the other. The left and right arrow keys can be used to move the slider by the small increment and the page up and page down keys can be used to move by the large increment. Use the “Range” subcommand to specify the start and end of the range of numbers allowed and to specify the small and large increments.	<pre>gui Create( "Property", Name = "myIntSlider", Type = "Normal", DefaultVal ue = "2", Di al ogPrompt = "Int Sl i der", Di al ogControl =     "Integer Sl i der", Range = "-10: 10, 1, 2" )</pre>

*Table 17.2: Dialog control types.*

Control Type	Description	Example
<b>Float Slider</b>	Similar to the Integer Slider control except this control accepts floating point numbers and increments less than 1.	<pre>gui Create( "Property", Name = "myFloatSlider", Type = "Normal", DefaultVal ue = "2.1", Di al ogPrompt = "Fl oat Sl i der", Di al ogControl = "Fl oat Sl i der", Range = "-5:5,0.1,1" )</pre>
<b>OCX String</b>	<p>Adds any specially written ActiveX control to the dialog. Use the "ControlProgId" subcommand to specify the ProgID of the ActiveX control you want to add, and use the "ControlServerPathName" subcommand to specify the pathname of the ActiveX control server program.</p> <p>See the section "Dialog Controls In S-PLUS" in the S-PLUS Documentation Supplement for more information about ActiveX controls in S-PLUS dialogs.</p>	<pre>gui Create( "Property", Name = "myOCXControl", Type = "Normal", DefaultVal ue = "2", Di al ogPrompt = "My OCX", Di al ogControl = "OCX String", Control ProgId =     "MyOCXServer.MyCtrl.1", Control ServerPathName =     "c:\\myocx\\myocx.ocx" )</pre>
<b>Picture</b>	<p>A small rectangle taking up one dialog column which can contain a Windows metafile picture (either Aldus placable or enhanced).</p> <p>The picture to draw in this control is specified as a string in the "Default-Value" subcommand containing either the pathname to the WMF file on disk, or a pathname to a Windows 32-bit DLL followed by the resource name of the metafile picture in this DLL.</p>	<pre>gui Create( "Property", Name = "myPicture", Di al ogControl = "Pi ct ure", Di al ogPrompt = "My Pi ct ure", DefaultVal ue =     "c:\\pics\\mypi ct. wmf" );</pre>

*Table 17.2: Dialog control types.*

Control Type	Description	Example
<b>Wide Picture</b>	Same as the Picture control except this control takes up two dialog columns.	<pre>gui Create("Property", Name = "myWidePicture", DialogControl =     "WidePicture", DialogPrompt =     "My Wide Picture", DefaultValue =     "c:\\pics\\mypict.wmf" );</pre>
<b>Picture List Box</b>	<p>A scrolling list box control taking up one dialog column which can contain several Windows metafile pictures (either placeable or enhanced).</p> <p>The list of pictures to draw in this control is specified as a string list in the subcommand "OptionList.</p> <p>Each element in this option list is either the pathname to the WMF file on disk, or a pathname to a Windows 32-bit DLL followed by the resource name of the metafile picture in this DLL.</p>	<pre>gui Create("Property", Name = "PictureList", DialogControl =     "PictureListBox", OptionList = c(     "c:\\spl uswi n\\home\\met1.wmf",     "c:\\spl uswi n\\home\\met2.wmf",     "c:\\spl uswi n\\home\\met3.wmf" ), DialogPrompt =     "My Picture List" );</pre>
<b>Wide Picture List Box</b>	Similar to the Picture List Box control except this control takes up two dialog columns.	<pre>gui Create("Property", Name = "WidePictureList", DialogControl =     "WidePictureListBox", OptionList = c(     "c:\\spl uswi n\\home\\met1.wmf",     "c:\\spl uswi n\\home\\met2.wmf",     "c:\\spl uswi n\\home\\met3.wmf" ), DialogPrompt =     "My Wide Picture List" );</pre>

Table 17.2:Dialog control types.

Control Type	Description	Example
Multi-line String	Same as “String” except that this control is several rows long and accommodates multiple lines of text. Text is word-wrapped. Return is used to enter multiple lines.	<pre>gui Create("Property", Name = "myMultiLineString", Type = "Normal", DialogControl =     "Multi-Line String", DefaultVal ue =     "My default value.", UseQuotes = T )</pre>
Wide Multi-line String	Same as “Multi-line String” except that this control takes up two dialog columns.	<pre>gui Create("Property", Name =     "myWideMultiLineString", Type = "Normal", DialogControl =     "Wide Multi-Line String", DialogPrompt =     "&amp;My Multi-Line String", DefaultVal ue =     "My default value.", UseQuotes = T )</pre>

**Picture Controls** For both the Picture and the Picture List Box controls, you can specify either a pathname to a Windows metafile on disk or a pathname to a Windows 32-bit DLL and the resource name of the metafile in this DLL to use. The syntax for each of these is specified below:

Table 17.3:Picture control pathname syntax.

Pathname to Windows metafile	<pre>"[pathname]" Example: "c:\spl uswi n\home\Meta1.WMF"</pre>
DLL Pathname and resource name of metafile	<pre>"; [pathname to DLL], [metafile resource name] Example: "; c:\mydll\mydll.dll, MyMetaFile"</pre>

Please note that the leading semicolon is required in this case and the comma is required between the DLL pathname and the name of the metafile resource.

---

Several example S-PLUS scripts are available on disk which demonstrate how to use these new controls for your own dialogs. See the files **PictCtl1.ssc** and **PictCtl2.ssc** in the **Samples** directory within the directory where S-PLUS is installed.

## Copying Properties

When creating a new dialog, it is often desirable to have controls similar to those used in previously existing dialogs. To use a Property already present in another dialog, simply refer to this Property when creating the **FunctionInfo** object, and perhaps in the group or page containing the Property. Any of the properties used in the statistical dialogs are directly available for use by dialog developers.

Additionally, the dialog developer may wish to have a property which is a modified version of an existing property. One way to do so is to refer to the Property directly, and to overload specific aspects of the Property (such as the **DialogPrompt** or **DefaultValue**) in the **FunctionInfo** object.

Another way to create a new Property based on another Property is to specify the Property to **CopyFrom** when creating the new Property. The new Property will then be based on the **CopyFrom** Property, with any desired differences specified by the other properties of the object.

In this section we mention standard properties commonly used in S-PLUS dialogs, as well as internal properties useful for filling default values and option lists based on current selections.

## Standard Properties

Any Property used in a built-in statistics dialog is available for reuse. To find the name of a particular Property, start by looking at the Property List in the **FunctionInfo** object for the dialog of interest. This will typically list Page or Group properties used in the dialog in order of their appearance in the dialogs (from top left to lower right). For a single-page dialog, locate the name of the Group object containing the Property of interest, and then examine the Property List for that Group object to locate the name of the Property of interest. For multi-page dialogs, find the name of the Property by looking in at the **FunctionInfo** object for the Page name, then the Page object for the Group name, then the Group object for the desired Property name.

Once you know the name of the Property object, you may include it directly in a dialog by placing it in the Property List for the dialog or one of its groups or pages. Alternately, you may create a new Property using **CopyFrom** to base the new Property on the existing Property.

For easy reference, Table 17.4 lists some of the properties used in the **Linear Regression** dialog which are reused in many of the other statistical dialogs. For the names of additional properties, examine the FunctionInfo object for menuLm, and the related Property objects. Note that the naming convention used by MathSoft is generally to start property names with SProp. When creating new properties, users may wish to use some other prefix to avoid name conflicts.

*Table 17.4: Linear Regression dialog properties.*

Dialog Prompt	Property Name
<i>Data group</i>	
Data Frame	SPropDataFrameList
Weights	SPropWeights
Subset Rows with	SPropSubset
Omit Rows with Missing Values	SPropOmitMissing
<i>Formula group</i>	
Formula	SPropPFFormula
Create Formula	SPropPFButton
<i>Save Model Object group</i>	
Save As	SPropReturnObject
<i>Printed Results group</i>	
Short Output	SPropPrintShort
Long Output	SPropPrintLong
<i>Saved Results group</i>	



*Table 17.4: Linear Regression dialog properties.*

<b>Dialog Prompt</b>	<b>Property Name</b>
Save In	SPropSaveResultsObject
Fitted Values	SPropSaveFit
Residuals	SPropSaveResid
<i>Predict page</i>	
New Data	SPropPredictNewdata
Save In	SPropSavePredictObject
Predictions	SPropPredictSavePred
Confidence Intervals	SPropPredictSaveCI
Standard Errors	SPropPredictSaveStdErr
Confidence Level	SPropConfLevel

Some other widely used properties and their purpose are listed below.

### **SPropInvisibleReturnObject**

This Property object has an invisible control which does not appear in the dialog. It is used as the return value argument for dialogs whose results are never assigned.

### **SPropCurrentObject**

This Property object is an invisible control whose default value is the name of the currently selected object. It is used by method dialogs launched from context menus, as discussed in the section Method Dialogs (page 729).

### **SPropFSpace1, ..., SPropFSpace8**

These Property objects have a newline control. They are used to place spaces between groups to adjust the dialog layout.

**Internal Properties**

Internal properties are specifically designed to fill the default values and option lists based on the currently selected objects. For example, internal properties can be used to create a list box containing the names of the variables in the currently selected data frame.

If the dialog needs to fill these values in a more sophisticated way, this may be accomplished using callback functions. See the section Method Dialogs (page 729) for details.

Here are several internal property objects that can be used in dialogs either alone or by means of Copy From.

**TXPROP\_DataFrames**

This Property object displays a dropdown box listing all data frames filtered to be displayed in any browser.

**TXPROP\_DataFrameColumns**

This Property object displays a dropdown box listing all columns in the data frame selected in TXPROP\_DataFrames. If no selection in TXPROP\_DataFrames has been made, default values are supplied.

**TXPROP\_DataFrameColumnsND**

This Property object displays a dropdown box of all columns in the data frame selected in TXPROP\_DataFrames. If no selection in TXPROP\_DataFrames has been made, default values are not supplied.

**TXPROP\_SplusFormula**

This Property object causes an S-PLUS formula to be written into an edit field when columns in a data sheet view are selected. The response variable is the first column selected, and the predictor variables are the other columns.

**TXPROP\_WideSplusFormula**

This Property object differs from TXPROP\_SplusFormula only in that the formula is displayed in an edit field which spans two columns of the dialog, instead of one column.

**ActiveX Controls in S-PLUS dialogs**

S-PLUS supports the use of ActiveX controls in dialogs for user defined functions created in the S-PLUS programming language. This feature allows greater flexibility when designing a dialog to represent a function and its parameters. Any ActiveX control can be added to the property list for a

dialog, however, most ActiveX controls will not automatically communicate changed data back to the S-PLUS dialog nor will most tell S-PLUS how much space to give the control in the dialog. To fully support S-PLUS dialog layout and data communication to and from S-PLUS dialogs, a few special ActiveX methods, properties, and events need to be implemented in the control by the control designer.

Examples of ActiveX controls which implement support for S-PLUS dialog containment are provided on disk in the SAMPLES/OCX directory beneath the program directory. These examples are C++ projects in Microsoft Visual C++ 4.1 using MFC (Microsoft Foundation Classes). Any MFC ActiveX project can be modified to support S-PLUS dialogs easily, and this will be discussed later in this section. Also in SAMPLES/OCX are example scripts which use S-PLUS to test these ActiveX controls.

### Adding an ActiveX control to a dialog

To use an ActiveX control for a property in a dialog, when creating the property, specify a "DialogControl" of type "OCX String" and specify the program id (or PROGID) of the control using the "ControlProgId" subcommand. Below is an example S-PLUS script which creates a property that uses an ActiveX control:

```
gui Create("Property",
    name = "OCXStringField",
    DialogControl = "OCX String",
    ControlProgId = "TXTESTCONTROL1.TxTestControl1Ctrl.1",
    ControlServerPathName = "c:\\myocx\\myocx.ocx",
    DialogPrompt = "&OCX String");
```

If you are editing or creating a property using the Object Explorer, the Property object dialog for the property you are editing allows you to set the dialog control type to "OCX String" from the "Dialog Control" drop-down list. When this is done, the "Control ProgId" and "ControlServerPathName" fields become enabled allowing you to enter the PROGID of the ActiveX control and its location on disk, respectively. The "ControlServerPathName" value is used to autoregister the control, if necessary, before using the control.

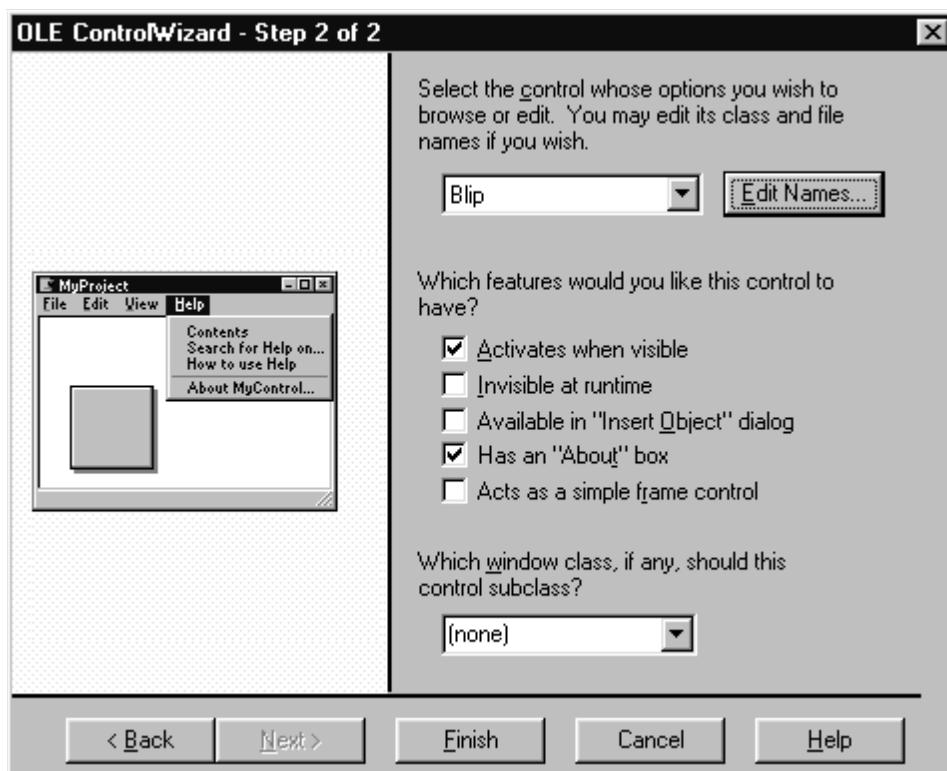
If you are editing or creating a property using the Object Explorer, the Property object dialog for the property you are editing allows you to set the dialog control type to "OCX String" from the "Dialog Control" drop-down list. When this is done, the "Control ProgId" field becomes enabled allowing you to enter the PROGID of the ActiveX control.

### Where can the PROGID for the control be found?

When you add an ActiveX control to an S-PLUS dialog, you need to specify its PROGID, as mentioned above. The PROGID is a string which uniquely identifies this control on your system. If you create controls using the ControlWizard in Developer Studio as part of Microsoft Visual C++ 4.0 or

higher, a default value for the PROGID is created by the ControlWizard during control creation that is based on the name of the project you use. For example, if your ControlWizard project name is “MyOCX”, then the PROGID that is generated is “MYOCX.MyOCXCtrl.1”. The pattern here is [Project name].[Control class name without the leading ‘C’].1. You can also find the PROGID used in an MFC ControlWizard project in the implementation CPP file of the control class. Search for the `IMPLEMENT_OLECREATE_EX()` macro in this file. The second parameter in this macro is the PROGID string you are looking for.

If you are using the OLE ControlWizard as part of Microsoft Visual C++ 4.0 or higher to develop your control, you can change the PROGID string for your control before it gets created by editing the names used for the control project. During the ControlWizard steps, you will see a dialog with the button “Edit Names” on it:



Click on this button and you will get another dialog allowing you to change the names used for classes in this project. Every control project in MFC has a class for the control and a class for the property sheet for the control. In the control class section of this dialog you will see the “Type ID” field. This is the PROGID for the control:

**Edit Names**

Short Name:

OK Cancel Help

**Control**

Class Name:	Header File:	Type Name:
<input type="text" value="CBlipCtrl"/>	<input type="text" value="BlipCtrl.h"/>	<input type="text" value="BlipCtrl"/>
	Implementation File:	Type ID:
	<input type="text" value="BlipCtrl.cpp"/>	<input type="text" value="BLIP.BlipCtrl.1"/>

**Property Page**

Class Name:	Header File:	Type Name:
<input type="text" value="CBlipPropPage"/>	<input type="text" value="BlipPpg.h"/>	<input type="text" value="Blip Property Page"/>
	Implementation File:	Type ID:
	<input type="text" value="BlipPpg.cpp"/>	<input type="text" value="BLIP.BlipPropPage.1"/>

## Registering an ActiveX control

It is important to register an ActiveX control with the operating system at least once before using it so that whenever the PROGID of the control is referred to (such as in the “ControlProgId” subcommand above), the operating system can properly locate the control on your system and run it. Registering an ActiveX control is usually done automatically during the creation of the control, such as in Microsoft Visual C++ 4.0 or higher. If the subcommand “ControlServerPathName” is specified in an S-PLUS script using the control, then this value will be used to register the control automatically. A control can also be registered manually by using a utility called “RegSvr32.exe”. This utility is included with development systems that support creating ActiveX controls, such as Microsoft Visual C++ 4.0 or higher. For your convenience, a copy of RegSvr32.exe is located in the

SAMPLES/OCX directory, along with two useful batch files, “RegOCX.BAT” and “UnRegOCX.BAT”, which will register and unregister a control. You can modify these batch files for use with controls you design.

You typically do not ever need to unregister an ActiveX control, unless you wish to remove the control permanently from your system and no longer need to use it with any other container programs such as S-PLUS. If this is the case, you can use RegSvr32.exe with the ‘/u’ command line switch (as in UnRegOCX.BAT) to unregister the control.

### **Why only “OCX String”?**

In S-PLUS, several different types of properties exist. There are string, single-select lists, multi-select lists, numeric, and others. This means that a property in a dialog communicates data depending on the type of property selected. A string property communicates string data to and from the dialog. A single-select list property communicates a number representing the selection from the list, a multi-select list communicates a string of selections made from the list with delimiters separating the selections. For ActiveX controls, only string communication has been provided in this version. This means that the control should pass a string representing the “value” or state of the control back to S-PLUS. In turn, if S-PLUS needs to change the state of the control, it will communicate a string back to the control. Using a string permits the most general type of communication between S-PLUS and the ActiveX control, because so many different types of data can be represented with a string, even for example lists. In future versions, other S-PLUS property types may be added for ActiveX controls.

### **Common error conditions when using ActiveX controls in S-PLUS**

The most common problem when using an ActiveX control in an S-PLUS dialog is that the control does not appear, instead a string edit field shows up when the dialog is created. This is usually caused by not registering the ActiveX control with the operating system. After a control is first created and before it is ever used, it must be registered with the operating system. This usually occurs automatically in the development system used to make the control, such as Microsoft Visual C++. However, you can also manually register the control by using a utility called “RegSvr32.exe”. This utility is included with development systems that support creating ActiveX controls, such as Microsoft Visual C++ 4.0 or higher. For your convenience, a copy of RegSvr32.exe is located in the SAMPLES/OCX directory, along with two useful batch files “RegOCX.BAT” and “UnRegOCX.BAT” which will register and unregister controls. You can modify these batch files for use with controls you design.

### **Designing ActiveX controls that support S-PLUS**

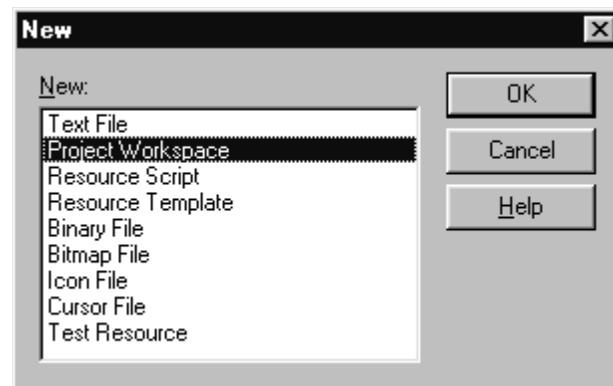
As mentioned earlier, examples of ActiveX controls which implement support for S-PLUS are provided on disk in the SAMPLES/OCX directory beneath the program directory. One of the examples in this directory is called MyOCX, and it is a C++ project in Microsoft Visual C++ 4.1 using MFC.

There is also an example S-PLUS script in MyOCX which shows how to use this ActiveX control in an S-PLUS dialog. This example will be used here to show how to implement ActiveX controls for S-PLUS. If you would rather skip this section and simply study the changes in the source files for MyOCX, all changes are marked in the source files with the step number (as listed below) that the change corresponds to. Just search for the string “S-PLUS Dialog change (STEP” in all the files of the MyOCX project to find these modifications.

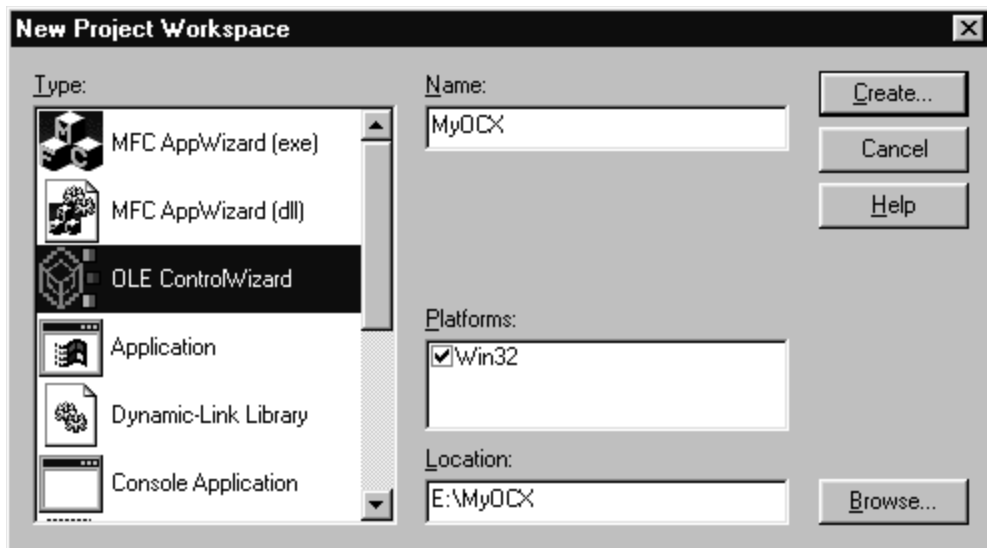
Version 4.0 or higher of Microsoft Visual C++ is used to demonstrate ActiveX control creation. Higher versions can also be used to create controls for S-PLUS but the dialogs and screens shown may be different.

### 1. Create the basic control

The first step to designing an ActiveX control in MFC should be to use the OLE ControlWizard that is part of the Developer Studio. Select New from the File menu in Developer Studio and then choose “Project Workspace” to start a new project.



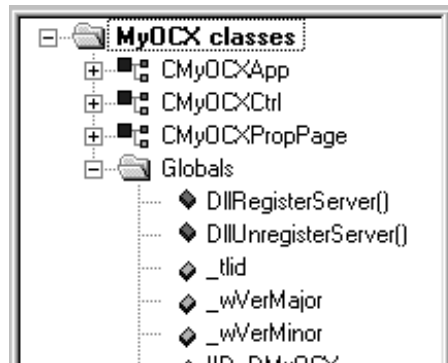
From the workspace dialog that appears, select “OLE ControlWizard” from the list of workspace types available. Enter a name for the project and specify the location, then click the “Create...” button.



After accepting this dialog, you will see a series of dialogs associated with the OLE ControlWizard, asking questions about how you want to implement your control. For now, you can simply accept the defaults by clicking “Next” on each dialog. When you reach the last dialog, click the “Finish” button. You will see a confirmation dialog showing you the choices you selected and names of classes that are about to be created. Click the “OK” button to accept and generate the project files.

In the “ClassView” page of the “Project Workspace” window in Visual C++, you will see the classes that the OLE ControlWizard created for your ActiveX control:





## 2. Add the S-PLUS support classes

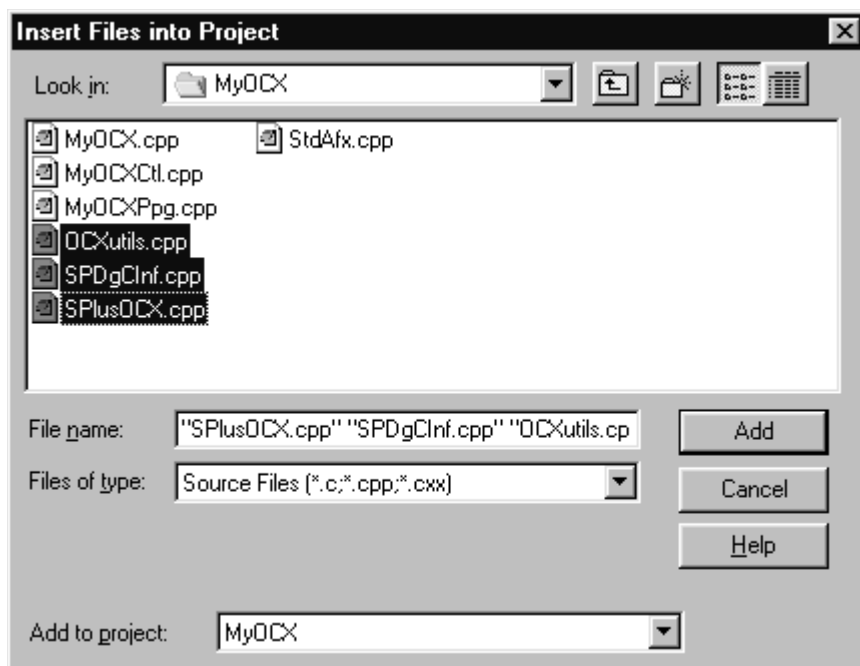
To start adding support for S-PLUS dialogs to your ActiveX control, copy the following files from the SAMPLES/OCX/SUPPORT control example directory into the new ActiveX control project directory you just created:

```
OCXUtil.s.cpp
OCXUtil.s.h
SPDgClnf.cpp
SPDgClnf.h
SPIusOCX.cpp
SPIusOCX.h
SPIusOCX.idl
```

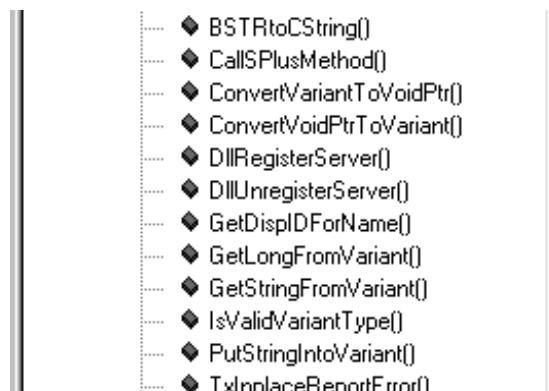
You also need to add these classes to your project before they will be compiled and linked to your control. To do this, select “Files into Project...” from the “Insert” menu in Visual C++. You will then see a standard file open dialog. Use this dialog to select the following files:

```
OCXUtil.s.cpp
SPDgClnf.cpp
SPIusOCX.cpp
```

To select all these files at once, hold down the CTRL key while using the mouse to click on the filenames in the list.



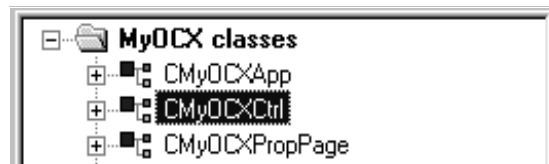
When these files are selected, click the “Add” button and the classes will appear as entries in your Project Workspace window.



### 3. Modify class inheritance

Next, we need to modify the inheritance of the class representing your ActiveX control so that it inherits from CSPlusOCX instead of from COleControl. CSPlusOCX is a parent class from which all ActiveX controls for which you desire support for S-PLUS dialogs can inherit. CSPlusOCX inherits directly from COleControl and its complete source code can be found in the SPlusOCX.cpp and SPlusOCX.h files.

To do this, first double-click on the class representing your ActiveX control in the “ClassView” page of the Project Workspace window to open the header for this class into your editor. In this example that is the CMyOCXCtrl class. Go to the top of this file in the editor.



Add the following line before the class declaration line for CMyOCXCtrl at the top of this header file:

```
#include "SPlusOCX.h"
```

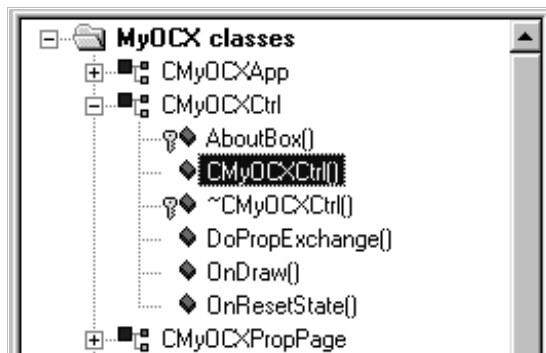
Modify the class declaration line

```
class CMyOCXCtrl : public COleControl
```

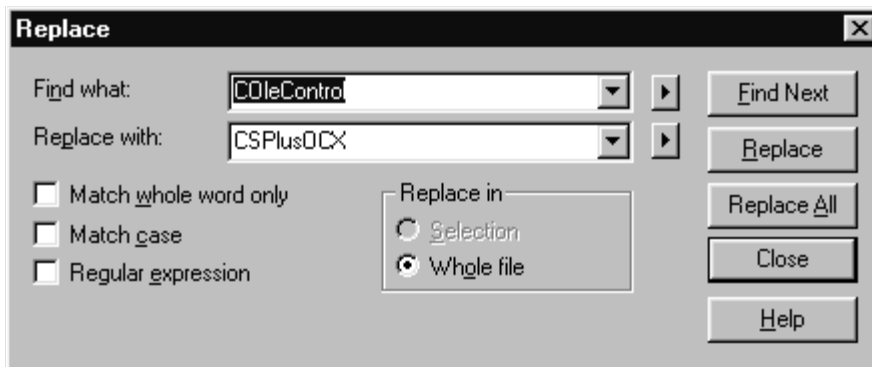
to read

```
class CMyOCXCtrl : public CSPlusOCX
```

Next, expand the class listing for CMyOCXCtrl so that all the methods are shown. To do this, click on the ‘+’ next to “CMyOCXCtrl” in the “ClassView” page of the Project Workspace window.

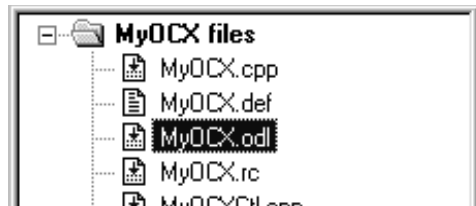


Then double-click on the constructor "CMyOCXCtrl()" to open the implementation CPP file for this class in your editor. Go to the top of this file. Using the find and replace function of the Developer Studio, replace all occurrences of "COleControl" base class with the new base class name "CSPlusOCX" in this file:



#### 4. Modify your control's type library definition file

Switch to the "FileView" page in the Project Workspace window and find the type library definition file (.ODL) for your ActiveX control. In this example it is "MyOCX.odl". Double-click on this entry in the list to open this file into your editor. Go to the top of this file.



Find the “properties” definition section for the dispatch interface “\_DMyOCX” in this file. It should look like:

```
di spi interface _DMyOCX
{
    properties:
        // NOTE - Class Wizard will maintain property information
        here.
        //      Use extreme caution when editing this section.
        //{{AFX_ODL_PROP(CMyOCXCtrl)
        //}}AFX_ODL_PROP
```

Add the following lines at the end of this section:

```
#define SPLUSOCX_PROPERTIES
#include "SPUSOCX.idl"
#undef SPLUSOCX_PROPERTIES
```

The section should now appear as follows:

```
di spi interface _DMyOCX
{
    properties:
        // NOTE - Class Wizard will maintain property information
        here.
        //      Use extreme caution when editing this section.
        //{{AFX_ODL_PROP(CMyOCXCtrl)
        //}}AFX_ODL_PROP
```

```
#define SPLUSOCX_PROPERTIES
#include "SPUSOCX.idl"
#undef SPLUSOCX_PROPERTIES

methods:
// NOTE - ClassWizard will maintain method information
here.
//      Use extreme caution when editing this section.
//{{AFX_ODL_METHOD(CMyOCXCtrl)
//}}AFX_ODL_METHOD

[id(DISPID_ABOUTBOX)] void AboutBox();
};
```

Now, add the following lines at the end of the “methods” section just below the “properties” section you just modified:

```
#define SPLUSOCX_METHODS
#include "SPUSOCX.idl"
#undef SPLUSOCX_METHODS
```

This whole section should now appear as follows:

```
dispinterface _DMyOCX
{
    properties:
// NOTE - ClassWizard will maintain property information
here.
//      Use extreme caution when editing this section.
//{{AFX_ODL_PROP(CMyOCXCtrl)
//}}AFX_ODL_PROP

#define SPLUSOCX_PROPERTIES
#include "SPUSOCX.idl"
#undef SPLUSOCX_PROPERTIES

methods:
// NOTE - ClassWizard will maintain method information
here.
//      Use extreme caution when editing this section.
//{{AFX_ODL_METHOD(CMyOCXCtrl)
//}}AFX_ODL_METHOD
```

---

```

        [id(DIALOG_ABOUTBOX)] void AboutBox();

#define SPLUSOCX_METHODS
#include "SPUSOCX.idl"
#undef SPLUSOCX_METHODS

};

```

Next, locate the event dispatch interface sections. In this example, it appears as:

```

dispatchinterface _DMYOCXEvents
{
    properties:
        // Event interface has no properties

    methods:
        // NOTE - ClassWizard will maintain event information
        here.
        // Use extreme caution when editing this section.
        //{AFX_ODL_EVENT(CMYOCXCtrl)
        //}AFX_ODL_EVENT
};

```

Add the following lines in the “events” section:

```

#define SPLUSOCX_EVENTS
#include "SPUSOCX.idl"
#undef SPLUSOCX_EVENTS

```

The section should now appear as:

```

dispatchinterface _DMYOCXEvents
{
    properties:
        // Event interface has no properties

    methods:
        // NOTE - ClassWizard will maintain event information
        here.
        // Use extreme caution when editing this section.
        //{AFX_ODL_EVENT(CMYOCXCtrl)
        //}AFX_ODL_EVENT

#define SPLUSOCX_EVENTS

```

```
#include "SPIusOCX.idl"
#undef SPLUSOCX_EVENTS

};
```

Do not modify any other parts of this file at this time.

## 5. Build the control

Now is a good time to build this project. To do this, click on the “Build” toolbar button or select “Build MyOCX.OCX” from the “Build” menu in the Developer Studio. If you receive any errors, go back through the above steps to make sure you have completed them correctly. You may receive warnings:

```
OCUtils.cpp(125) : warning C4237: nonstandard extension
used : 'bool' keyword is reserved for future use
OCUtils.cpp(216) : warning C4237: nonstandard extension
used : 'bool' keyword is reserved for future use
```

These warnings are normal and can be ignored.

Several overrides of CSPlusOCX virtual methods still remain to be added to your ActiveX control class, but compiling and linking now gives you a chance to review the changes made and ensure that everything builds properly at this stage.

## 6. Add overrides of virtual methods to your control class

To support S-PLUS dialog layout and setting the initial value of the control from an S-PLUS property value, you need to override and implement several methods in your control class. To do this, edit the header for your control class. In this example, edit the “MyOCXCtrl.h” file. In the declaration of the CMyOCXCtrl class, add the following method declarations in the “public” section:

```
virtual long GetSPIusDialogVerticalSize( void );
virtual long GetSPIusDialogHorizontalSize( void );
virtual BOOL SPIusOnInitializeControl( const VARIANT FAR&
    vInitialValue );
```



Next, open the implementation file for your control class. In this example, edit the file “MyOCXCtrl.cpp”. Add the following methods to the class:

```

long CMyOCXCtrl::GetSPUsDialogVerticalSize()
{
    return 3;    // takes up 3 lines in dialog
}

long CMyOCXCtrl::GetSPUsDialogHorizontalSize()
{
    return 1;    // takes up 1 column in dialog
}

BOOL CMyOCXCtrl::SPUsOnInitializeControl (const VARIANT
    FAR& vlnitialValue)
{
    CString slnitialValue; slnitialValue.Empty();
    if ( GetStringFromVariant(
        slnitialValue,
        vlnitialValue,
        "lnitialValue" ) )
    {
        // Set properties here
    }

    return TRUE;
}

```

These three methods should be implemented in the control class of any ActiveX control supporting S-PLUS dialogs fully. The first two methods support dialog layout, while the third supports setting values for the control from S-PLUS.

The value returned by `GetSPUsDialogVerticalSize()` should be a long number representing the number of lines the control takes up in an S-PLUS dialog. A line is the size of a String edit field property in an S-PLUS dialog. The value returned by `GetSPUsDialogHorizontalSize()` should be either 1 or 2. Returning 1 means that this control takes up only one column in an S-PLUS dialog. Returning 2 means the control takes up two columns. A column in an S-PLUS dialog is the width of a single String property field. There are at most two columns in an S-PLUS dialog. In the example above, the MyOCX control takes up three lines and only one column in an S-PLUS dialog.

`SPlusOnInitializeControl()` is called when the control is first enabled in the S-PLUS dialog and every time the property that this control corresponds to in S-PLUS is changed. It receives a variant representing the initial value or current value (if any) for the control. This method should return `TRUE` to indicate successful completion and `FALSE` to indicate failure. Included in the file “OCXUtils.h” (copied previously into your control project directory) are numerous helper functions such as the one used here `GetStringFromVariant()` which will convert the incoming variant into a string if possible. You can then use this string to set one or more properties in your control.

To use the `SPlusOnInitializeControl()` in this example ActiveX control, first add a member string to the control class. Edit the “MyOCXCtrl.h” file and add a `CString` member variable called “`m_sValue`” to the `CMyOCXCtrl` class:

```
private:
    CString m_sValue;
```

Next, initialize this value in the constructor for `CMyOCXCtrl` by modifying the constructor definition in “MyOCXCtrl.cpp”:

```
CMyOCXCtrl::CMyOCXCtrl()
{
    InitializeIDs(&IID_DMyOCX, &IID_DMyOCXEvents);
    // TODO: Initialize your control's instance data here.

    m_sValue.Empty();
}
```

Then, add lines to the definition of the override of `SPlusOnInitializeControl()` in your control class to set this member variable and refresh the control by modifying “MyOCXCtrl.cpp”:

```
BOOL CMyOCXCtrl::SPlusOnInitializeControl(
const VARIANT FAR& vInitialValue)
{
    CString sInitialValue; sInitialValue.Empty();
    if ( GetStringFromVariant(
        sInitialValue,
        vInitialValue,
        "Initial Value" ) )
    {
        // Set properties here
    }
}
```

```

        m_sValue = sInitialValue;
        Refresh();

    }

    return TRUE;
}

```

Finally, so we can see the effects of `SPlusOnInitializeControl()`, add a line to the “OnDraw” method of `CMyOCXCtrl` by editing the definition of this method in “MyOCXCtrl.h”:

```

void CMyOCXCtrl::OnDraw(
    CDC* pdc, const CRect& rcBounds, const CRect& rcInvalid)
{
    // TODO: Replace the following code with your
    // own drawing code.
    pdc->FillRect(rcBounds,
        CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)));
    pdc->Ellipse(rcBounds);

    // Display latest value
    pdc->DrawText(
        m_sValue, (LPRECT)&rcBounds, DT_CENTER | DT_VCENTER);
}

```

Rebuild the project now to test these changes.

## 7. Test your new control in S-Plus

To try out your new control in S-PLUS you’ll need to create an S-PLUS script which creates properties and displays a dialog. Open S-PLUS and open the script file from `SAMPLES/OCX/MyOCX` called “MyOCX.SSC”. Notice that the script begins by creating three properties, one for the return value from a function and the other two for the parameters of a function. The property for “MyOCX” uses the type “OCX String” and the PROGID for the control we just created:

```

guiCreate("Property",
    name = "MyOCX",
    DialogControl = "OCX String",
    ControlProgId = "MYOCX.MyOCXCtrl.1",
    DialogPrompt = "My &OCX");

```

Run the script “MyOCX.SSC” and you will see a dialog containing an edit field and the MyOCX control you just created. When the dialog appears, the ActiveX control contains the text “Hello” because this is set as the initial value in the S-PLUS script callback function:

```
callbackMyOCXExample <- function(df)
{
  if(!is.na(dialogMessage(df)) # Am I called to initialize
                                # the properties?
  {
    # Set the initial value of the MyOCX property
    df <- cbSetCurrValue(df, "MyOCX", "\"Hello\"")
  }
  ...
}
```

When you enter a string (use quotes around any string you enter in these dialog fields) in the edit field, the ActiveX control updates to show that string. When you click the OK or Apply buttons in the dialog, you will see the values of both properties printed in a report window.

### Summary of steps to support S-PLUS dialogs in ActiveX controls

To summarize the above steps, the list below shows you the tasks necessary to adapt your MFC ActiveX control project to support S-PLUS dialogs:

1. Add S-PLUS dialog support files to your project:

```
OCXUtils.cpp
OCXUtils.h
SPDgClnf.cpp
SPDgClnf.h
SPIusOCX.cpp
SPIusOCX.h
SPIusOCX.idl
```

2. Change the inheritance of your control class from base class COleControl to CSPlusOCX.
3. Modify your control's ODL (type library definition file) to include SPlusOCX.idl sections.
4. Add virtual overrides of key CSPlusOCX methods to your control class:

```
virtual Long GetSPIusDialogVerticalSize( void );
virtual Long GetSPIusDialogHorizontalSize( void );
```

```
virtual BOOL SPlusOnInitializeControl(const VARIANT
    FAR& vInitialValue);
```

## Examples of ACTIVE X controls included with S-PLUS

Examples of ActiveX controls which implement support for S-PLUS dialog containment are provided on disk in the **SAMPLES/OCX** directory beneath the program directory. These examples are C++ projects in Microsoft Visual C++ 4.1 using MFC (Microsoft Foundation Classes) and are intended for developers.

### samples/ocx

#### **myocx**

Microsoft Visual C++ 4.1 MFC project demonstrating how to write ActiveX controls that fully support S-PLUS dialogs.

#### **ocx1**

Microsoft Visual C++ 4.1 MFC project demonstrating how to write ActiveX controls that fully support S-PLUS dialogs.

#### **support**

Microsoft Visual C++ 4.1 MFC headers and source files necessary for making ActiveX controls that fully support S-PLUS dialogs.

# CALLBACK FUNCTIONS

In S-PLUS, virtually any GUI object has an associated dialog. For example, a line plot is an object whose properties can be modified via its associated dialog. Similarly, an S-PLUS function can have an associated dialog. The properties of a function object are mapped to the function arguments, which can then be modified through its associated dialog. The function dialog can have an attached *callback function*.

A callback function provides a mechanism for modifying and updating properties (controls) of a live dialog. It is a tool for developing complex dialogs whose properties are dynamically changing based on the logic written in the callback function. The dialog subsystem executes the callback function while its associated dialog is up and running, in the following instances:

- Once, just before the dialog is displayed.
- When a dialog property (control) value is updated or modified by another mechanism, such as by the user.
- A button is clicked.

The user associates a callback function with a dialog by specifying its name in the corresponding function info object. The callback function takes a single data frame as its argument. This data frame argument has the dialog property names as row names. The elements in the data frame define the present state of the dialog. The S-PLUS programmer can access and modify these elements directly, however, there is a set of utility functions that simplify this task. Table 17.5 lists the utility functions that can be used inside a callback function to modify a dialog state. To get more complete information on these functions see the Language Reference help.

Table 17.5: Utility functions for use inside a callback function.

<code>cbl sI ni tDi al ogMessage()</code>	Returns TRUE if the callback function is called before the dialog window is displayed on the screen.
<code>cbl sUpdateMessage()</code>	Returns TRUE if the callback function is called when the user updates a property.
<code>cbl sOkMessage()</code>	Returns TRUE if the callback function is called when the <b>OK</b> button is clicked.

*Table 17.5: Utility functions for use inside a callback function.*

<code>cbIsCancel Message()</code>	Returns TRUE if the callback function is called when the <b>Cancel</b> button is clicked.
<code>cbIsApply Message()</code>	Returns TRUE if the callback function is called when the <b>Apply</b> button is clicked.
<code>cbGetActiveProp()</code>	Gets the current active property in a dialog.
<code>cbGetCurrentValue()</code>	Gets the current value of a property.
<code>cbSetCurrentValue()</code>	Sets the current value of a property.
<code>cbGetEnableFlag()</code>	Gets the current state of the enable/disable flag of a property.
<code>cbSetEnableFlag()</code>	Sets the state of the enable/disable flag of a property.
<code>cbGetOptionList()</code>	Gets the list of items from list based properties, such as ListBox, ComboBox, Multi-selected ComboBox, and so on.
<code>cbSetOptionList()</code>	Sets the list of items from list based properties, such as ListBox, ComboBox, Multi-selected ComboBox, and so on.
<code>cbGetPrompt()</code>	Gets the Prompt string of a property.
<code>cbSetPrompt()</code>	Sets the Prompt string of a property.
<code>cbGetDialogId()</code>	Returns the unique ID of the dialog instance.

Since the usage of these functions facilitate readability and portability of the S-PLUS callback functions, we recommend that you use them instead of direct access to the data frame object.

Callback functions are the most flexible way to modify dialog properties such as default values and option lists. However, for specific cases it may be more straightforward to use a property based on an internal property, as described in the section Copying Properties (page 671). In particular, this is the easiest way to fill a field with the name of the currently selected data frame or a list of the selected data frame's variables.

## Interdialog Communication

In some circumstances it may be useful to launch a second dialog when a dialog button is pushed. For example, the Formula dialog is available as a child dialog launched by the Linear Regression dialog. Information may then be communicated between dialogs using interdialog communication.

The child dialog is launched using `gui DisplayDialog`. Communication between the dialogs is performed by the functions `cbGetDialogId` and `gui ModifyDialog`. The script file **samples/dialog/dlgcomm.ssc** in the S-PLUS directory contains an example of such communication.

## Example: Callback Functions

The example script below creates and displays a function dialog that uses a callback function to perform initialization, communication and updating properties within an active dialog. It is a complete script file (called **propcomm.ssc**) that can be opened into a script window and run.

```
#-----
# propcomm.ssc: creates and displays a function dialog.
#           It shows how to use a dialog callback function to perform
#           initialization, communication and updating properties within an
#           active dialog.
#-----

#-----
# Step 1: define the function to be executed when OK or Apply button is pushed
#-----

propcomm<- function(arg1, arg2) { print("Ok or Apply button in simple1 dialog
    is pushed!") }

#-----
# Step 2: create individual properties that we want to use for arguments in the
#         function
#-----

gui Create("Property", Name= "propcommInvisible", DialogControl = "Invisible");
gui Create("Property", Name= "propcommListBox", DialogControl = "List Box",
    DialogPrompt= "&Grade", DefaultValue= "3",
    OptionList= c("4", "3", "2", "1"))
gui Create("Property", Name= "propcommCheckBox", DialogControl = "Check Box",
    DialogPrompt= "&Numerical Grade");
```



```

#-----
# Step 3: create the function info object
#-----

guiCreate("FunctionInfo", Function = "propcomm", PropertyList =
  c("propcommInvisible", "propcommListBox", "propcommCheckBox"),
  CallbackFunction = "propcommCallback", Display = "T" )

#-----
# Step 4: define a callback function to be called by an instance of the dialog.
#         This callback mechanism is used to initialize, communicate and update
#         properties in an active dialog.
#-----

propcommCallback <- function(df)
{
  if(IsInitDialogMessage(df)) # Am I called to initialize the properties?
  {
    # override option list of a property
    df <- cbSetOptionList(df, "propcommListBox", "excellent, good, fair,
    poor, fail")

    # override default value of a property
    df <- cbSetCurrentValue(df, "propcommListBox", "fair")
    df <- cbSetOptionList(df, "propcommCheckBox", "F")

  }
  else if( cbIsOkMessage(df)) # Am I called when the Ok button is pushed?
  {
    display.messagebox("Ok!")
  }
  else if( cbIsCancelMessage(df)) # Am I called when the Cancel button is
    pushed?
  {
    display.messagebox("Cancel!")
  }
  else if( cbIsApplyMessage(df)) # Am I called when the Apply button is
    pushed?
  {
    display.messagebox("Apply!")
  }
  else # Am I called when a property value is updated?

```

```

{
  if (cbGetActiveProp(df) == "propcommCheckBox") # the check box was
  clicked?
  {
    # change the option list
    if (cbGetCurrValue(df, "propcommCheckBox") == "T")
    {
      df <- cbSetOptionsList(df, "propcommListBox", "4.0, 3.0, 2.0, 1.0,
0.0")
      df <- cbSetCurrValue(df, "propcommListBox", "4.0")

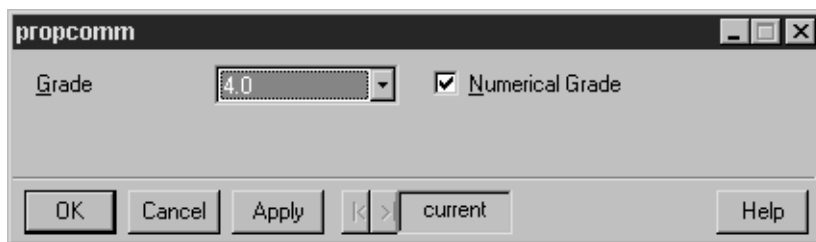
    }
    else
    {
      df <- cbSetOptionsList(df, "propcommListBox", "excellent, good, fair,
poor, fail")
      df <- cbSetCurrValue(df, "propcommListBox", "good")
    }
  }
}

df
}

#-----
# Step 5: display the dialog
#-----

guiDisplayDialog("Function", Name="propcomm");

```



*Figure 17.16: Clicking on Numerical Grade will illustrate the callback function working.*

# CLASS INFORMATION

## Overview

A `ClassInfo` object allows information to be specified about both user-defined and interface objects. It is similar to the `FunctionInfo` object, which allows information to be specified for functions (primarily for the purpose of defining function dialogs).

There are three main uses of the `ClassInfo` object:

1. Defining a context menu (right click menu) for objects.
2. Defining the double click action for objects. That is, you can use it to specify what will happen when the user double clicks or right clicks on an object in the Object Explorer.
3. It allows the dialog header and dialog prompts for interface objects to be overridden.

## Creating ClassInfo Objects

`ClassInfo` objects may be created using commands or from within the Object Explorer.

## Using Commands

To create a `ClassInfo` object, use `gui Create` with `classname="ClassInfo"`. The `lmsreg` robust regression function returns a model of class `"lms"`. The following commands will create a `ClassInfo` object indicating that the `print` function should be used as the double-click action, and define a context menu for `lms` objects:

```
gui Create(classname="ClassInfo", Name="lms",
ContextMenu="lms",
DialogHeader="Least Median Squares Regression",
DoubleClickAction="print")
```

```
gui Create(classname="MenuItem", Name="lms", Type="Menu",
DocumentType="lms")
```

```
gui Create(classname="MenuItem", Name="lms$summary",
Type="MenuItem", DocumentType="lms", Action="Function",
Command="summary", MenuItemText="Summary",
ShowDialogOnRun=F)
```

```
gui Create(classname="MenuItem", Name="I ms$pl ot",
    Type="MenuItem", DocumentType="I ms", Action="Function",
    Command="pl ot", MenuItemText="Pl ot",
    ShowDi al ogOnRun=F)
```

## Using the Object Explorer

Open the Object Explorer and create a folder with filtering set to “Cl assI nfo”. Right-click on a Classinfo object in the right pane, and choose Create ClassInfo from the context menu. The property dialog shown in Figure 17.17 appears.

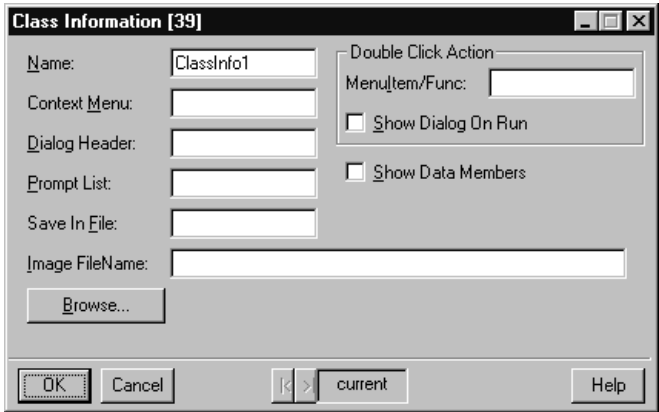


Figure 17.17: The property dialog for a ClassInfo object.

## ClassInfo Object Properties

The properties of a ClassInfo object determine characteristics such as the double-click action and context menu for the class of interest. These properties may be specified and modified using the property dialog for the MenuItem object, or programmatically via the commands `gui Create` and `gui Modi fy`. See the `gui Create("Cl assI nfo")` help file in the Language Reference help for syntax details.

The following properties are specified in the ClassInfo property dialog, shown in Figure 17.17:

The subcommand names of the properties are:

**Name** The name of the associated class. For instance, to specify information for the “I m” class, use this as the name. This also becomes the name of this instance of the Cl assI nfo object.

**ContextMenu** The name of the MenuItem object that defines the context menu (right click menu) for this object in the browser. This is the name of a

MenuItem of type “Menu”, which must have been defined in the standard way for menus.

**DoubleClickAction** The name of a MenuItem of type “MenuItem” (that is, it is a single item instead of an entire menu) or a function. This specifies the action that will happen when the user double clicks on the object in the browser. It allows a function to be called when the user double clicks.

**Show Dialog On Run** Logical value indicating whether the dialog for the MenuItem or function will be displayed before execution.

**DialogHeader** Text specifying the dialog header for the associated object. This is only useful for interface objects.

**PromptList** Allows dialog prompts to be specified (and overridden). The syntax is the same as it is for the corresponding property of FunctionInfo objects: `#0=" &My Prompt: ", #2="Another &Prompt: ", PropertySubcommandName="L&ast Prompt: ".` That is, it is a list of assignments, in which the left-hand side denotes the property whose prompt is going to be overridden, and the right-hand side denotes the new prompt. There are two ways of denoting the property: by position, starting with 0, with the number preceded by a #; and by property subcommand name. (In the example above, “#0” denotes the 0<sup>th</sup> property of the object; “PropertySubcommandName” is the subcommand name of the property to change.)

To find out the names of the properties of an object, you can use the following script:

```
gui GetPropertyNames("classname")
```

Note that all objects have two properties that may or may not be displayed on the dialog: `TXPROP_ObjectName` (subcommand name: `NewName`, always in position #0, but usually not displayed in a dialog) `TXPROP_ObjectPosIndex` (subcommand name: `NewIndex`, always in position #1, but usually not displayed in a dialog). To find out the argument names of the properties of an object, you can use the following script:

```
gui GetArgumentNames("classname")
```

The argument names are usually very similar to the corresponding prompts, so that figuring out which dialog field corresponds to which property should not be a problem.

## Modifying ClassInfo Objects

ClassInfo objects can be modified using either programming commands, their property dialogs, or their context menus.

If you are creating a GUI which you intend to distribute to others, it is usually preferable to revise the commands used to create the GUI. If you are

simply modifying the interface for your own use, using the property dialogs and context menus may be more convenient.

### Using Commands

The `gui Modify` command is used to modify an existing `ClassInfo` object. Specify the Name of the `ClassInfo` object to modify, and the properties to modify with their new values.

```
gui Modify(classname="ClassInfo", Name="IMS",  
          DoubleClickAction="plot")
```

### Using the Property Dialog

`ClassInfo` objects may be modified through the `ClassInfo` object property dialog.

To modify a `ClassInfo` object, open the Object Explorer to a page with filtering set to “`ClassInfo`.” Right-click on the `ClassInfo` object’s icon in the right pane and choose Properties from the context menu. Refer to the previous sections for details on using the property dialog.

### Using the Context Menu

`ClassInfo` objects can be modified with their context menus. The context menu for an object is launched by right-clicking on the object in the Object Explorer. The context menu provides options such as creating, copying, and pasting the object, as well as a way to launch the property dialog.

## Example: Customizing the Context Menu

This example shows how to add to the context menu for objects of class `data.frame` displayed in the Object Explorer. The new item automatically computes summary statistics for the selected data frame. To begin, open an Object Explorer page and filter by `ClassInfo` and `MenuItem`.

### 1. Creating a `ClassInfo` object for the `Class data.frame`

1. Right-click on a `ClassInfo` object and select Create `ClassInfo` in its context menu.
2. Enter “`data.frame`” in the Name field. This represents the name of the object class in which objects will have the context menu item specified below.
3. Enter `dfMenu` in the Context Menu field. This will be the name of the context menu.
4. Click OK.

### 2. Creating the Context Menu

1. Right-click on any `MenuItem` object and select Insert `MenuItem` from its context menu.

2. Enter dfMenu in the Name field. This corresponds to the Context Menu name given in to the ClassInfo object above.
3. Enter Menu in the Type field.
4. Click OK.
5. Right-click on dfMenu in the left pane and select Insert MenuItem from the context menu.
6. Enter desc in the Name field. This name is not important, as long as it does not conflict with that of an existing object.
7. Select MenuItem from the Type field.
8. Enter data.frame in the Document Type field; do not choose from the dropdown box selections. This corresponds to the object class which will have this context menu.
9. Select FUNCTION from the Action field.
10. Enter the text “Summary...” in the MenuItem Text field. This text will appear in the context menu.
11. Move to the Command page of the dialog.

**Tip...**

A FunctionInfo object must exist for the function which is called by the context menu item. Otherwise, the default dialog for that function will not appear.

12. Enter menuDescribe in the Command field. This is the function which is executed by the dialog which appears with Statistics/Data Summaries/Summary Statistics. There is a built-in FunctionInfo object by the same name.
13. **Show Dialog On Run.** This should be checked.
14. The MenuItem object desc is now found alongside dfMenu in the MenuItem tree. To move it underneath dfMenu, hold down the Alt key and drag the desc icon onto the dfMenu icon. To see the desc MenuItem object in its new position, click on the dfMenu icon in the left pane and look in the right pane.

### 3. Displaying and Testing the Context Menu

1. Click the Object Explorer button in the main toolbar to open a default Object Explorer window.

- When data frame objects are visible in the right pane, right-click on any data frame. Choose Summary, which should appear under Properties in the context menu, as shown in Figure 17.18.

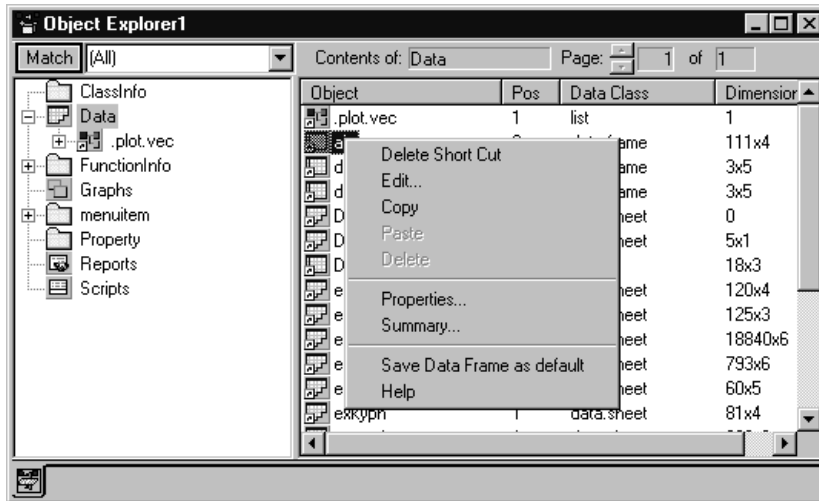


Figure 17.18: A context menu with the item Summary added.

The Summary Statistics dialog appears.

- By default, Data Frame is set to air in that dialog. Click OK and the Summary Statistics are sent to a Report window, unless the Command window is open to receive them.

Instead of the built-in `FunctionInfo` object `menuDescribe` and its associated built-in S-PLUS function, user-defined objects can also be used. The procedure for adding a context menu option is identical.

#### 4. Applying the Context Menu to a Class which Inherits from `data.frame`

- Use the Select Data dialog in the Data menu to select the `catalyst` data set. When it opens in the Data window, change the upper left cell to read "180", then change it back to 160. (This won't change the data, but it will write it to your working data, so it will appear in your Object Explorer.)
- Right-click on the object "catalyst". The context option Summary does not appear, because the object `catalyst` has class `design`, which inherits from `data.frame`. To confirm this, you can check Data Class and Inheritance in the Right Pane page of the Object Explorer



property dialog, if this is not already done, and view the information in the right pane of the Object Explorer, as in Figure 17.19. Make sure that Include Derived Classes is checked in the Object Explorer property dialog.

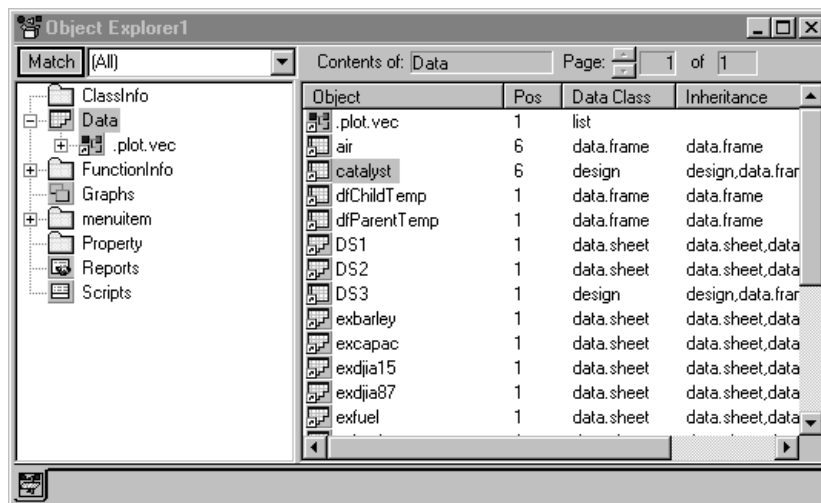


Figure 17.19: The Object Explorer showing the class and inheritance of the data object catalyst.

3. To enable the context menu for objects in the class design, open the property dialog for the MenuItem desc.
4. Enter data.frame,design in the Document Type field.
5. Click OK.
6. Return to the page showing data frames and right-click on the object catalyst. The context menu now contains Summary.

## STYLE GUIDELINES

Typically S-PLUS programmers will begin by writing functions for use in scripts and at the command line. These functions will generally fall into one of the following classes:

- Functions which compute some quantities and return a vector, matrix, data.frame, or list. If the result is assigned these values are stored, and if not they are printed using the standard mechanism. Functions such as `mean` and `cor` are of this type.
- Functions which take data and produce plots. The returned value is typically not of interest. Functions such as `xyplot` and `pairs` are of this type.
- A set of functions including a modeling function which produces a classed object, and method functions such as `print`, `summary`, `plot`, and `predict`. Functions such as `lm` and `tree` are of this type.

The custom menu and dialog tools allow the creation of a dialog for any function. Hence the programmer may create a dialog which directly accesses a function developed for use at the command line. While this may be acceptable in some cases, experience has shown that it is generally preferable to write a wrapper function which interfaces between the dialog and the command line function.

This section discusses the issues that arise when creating a function for use with a dialog, and describes how these issues are handled by the built-in statistical dialog functions. In addition, we discuss basic design guidelines for statistical dialogs.

### Basic Issues

Most functions will perform these steps:

- Accept input regarding the data to use.
- Accept input regarding computational parameters and options.
- Perform computations.
- Optionally print the results.

- Optionally store the results.
- Optionally produce plots.

Modeling functions have additional follow-on actions which are supported at the command line by separate methods:

- Providing additional summaries.
- Producing plots.
- Returning values such as fitted values and residuals.
- Calculating predicted values.

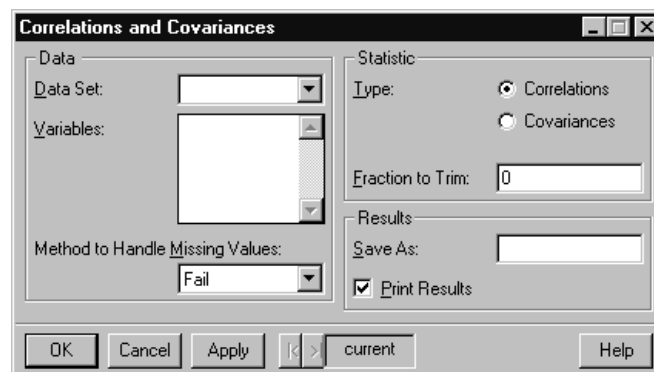
We will first discuss the basic steps performed by any function such as accepting input, performing calculations, printing results, saving results, and making plots. Then we will discuss the issues which arise for modeling functions with methods.

## Basic Dialogs

We will begin by discussing the **Correlations and Covariances** dialog. Exploring this dialog and the related analysis and callback functions will display the key issues encountered when constructing functions for dialogs.

### The Dialog

The **Correlations and Covariances** dialog is available from the **Statistics:Data Summaries:Correlations** menu item.



*Figure 17.20: The Correlations and Covariances dialog.*

This dialog provides access to the `cor` and `var` functions. It allows the user to specify the data to use, computation options, a name under which to save the results, and whether to print the results.

Note that the data to use is specified in the upper left corner of the dialog. The user first specifies which Data Frame to use, and then the variables of interest. (Some dialogs will accept matrices or vectors in the Data Frame field, but for simplicity users are encouraged to work with data frames.)

The Results group in the lower right corner of the dialog lets the user specify an object name under which to store the results, and provides a check box indicating whether the results should be printed.

Other options are placed between the Data group and the Results group.

## The Function

When **OK** or **Apply** is pressed in the dialog, the `menuCor` function is called. The naming convention for functions called by dialogs is to append `menu` to the command line function name, such as `menuLm`, `menuTree`, and `menuCensorReg`.

The `menuCor` function is:

```
> menuCor
function(data, variables = names(data), cor.p = F, trim = 0,
  cov.p = F, na.method = "fail", print.it = T,
  statistic = "Correlations")
{
  # Note cor.p and cov.p have been replaced with statistic.
  # They are left in solely for backwards compatibility.
  data <- as.data.frame(data)
  data.name <- deparse(substitute(data))
  if(!missing(variables))
    variables <- sapply(unpaste(variables, sep = ","),
      strip.blanks)
  if(!is.element(variables[[1]], c("<ALL>", "(All
    Variables)"))) {
    if(!length(variables))
      stop("You must select at least one variable\n")
    data <- data[, variables, drop = F]
  }
  dropped.cols <- !sapply(data, is.numeric) | sapply(data,
    is.dates)
  if(all(dropped.cols))
    stop("No numeric columns specified.")
  if(any(dropped.cols)) {
```

```

      warni ng(paste("Droppi ng non-numeri c col umn(s) ",
paste(names(data)[
      dropped.col s], collapse = ", "), ". ", sep = ""))
      data <- data[, !dropped.col s, drop = F]
    }
    na.method <- casefol d(na.method)
    i f(stati stic == "Correl ations" || (cor.p && !cov.p)) {
      coeff <- cor(data, trim = trim, na.method = na.method)
      header.txt <- paste("\n\t*** Correl ations for data
in: ", data.name,
      "***\n\n")
    }
    el se {
      coeff <- var(data, na.method = na.method)
      header.txt <- paste("\n\t*** Covari ances for data in:
      ", data.name,
      "***\n\n")
    }
    i f(print.it) {
      cat(header.txt)
      print(coeff)
    }
    i nvi si ble(coeff)
  }
}

```

## Input Values

The function arguments are:

```

functi on(data, vari ables = names(data), cor.p = F, trim =0,
cov.p = F, na.method = "fail", print.it = T,
stati stic = "Correl ations")

```

The function has one argument for each control in the dialog, with the exception of the **Save As** field specifying the name to which to assign the value returned by the function. Default values are present for all arguments except *data*. A default argument value will be used if the corresponding field in the dialog is left empty.

The first few lines in the function transform these inputs from a form preferable for a dialog field to the format expected by *cor* and *var*.

First the data is transformed to a data frame, to allow the handling of vectors and matrices. The name of the data is stored for use in printing the results:

```

data <- as.data.frame(data)
data.name <- deparse(substi tute(data))

```

Next the function constructs the names of the variables of interest. The `variables` argument passed by the dialog is a single string containing a comma delimited list of column names, and perhaps the string “(All Variables)”. This string is broken into a character vector of variable names. If it does not include “(All Variables)” and is not empty, the specified columns of the data are extracted.

```
if(!missing(variables))
  variables <- sapply(unpaste(variables, sep = ","),
    strip.blanks)
if(!is.element(variables[[1]], c("<ALL>", "(All
  Variables)"))) {
  if(!length(variables))
    stop("You must select at least one variable\n")
  data <- data[, variables, drop = F]
}
```

## Computations

After the desired set of data is constructed, the statistics are calculated:

```
if(Statistic == "Correlations" || (cor.p && !cov.p)) {
  coeff <- cor(data, trim = trim, na.method = na.method)
  header.txt <- paste("\n\t*** Correlations for data
in: ", data.name,
  "\n\t***\n\n")
}
else {
  coeff <- var(data, na.method = na.method)
  header.txt <- paste("\n\t*** Covariances for data in:
", data.name,
  "\n\t***\n\n")
}
```

The `Statistic` argument takes a string, either “Correlations” or “Covariances”; `cor.p` and `cov.p` arguments are logical values indicating whether to form the correlations or covariances which are supported for backward compatibility. The callback function (discussed later) enforces the constraint that only one of these is TRUE. Note that this could also have been implemented using Radio Buttons passing a character string rather than as separate Check Boxes.

The `trim` and `na.method` arguments are passed directly to the computational functions.

A character string is also constructed for use as a header when printing the results.

## Printing Results

The standard behavior in S-PLUS is to either print the results from a function or store them under a specified name using assignment. That is, a user may either see the results printed using

```
> cor(swi ss. x)
```

save the results using

```
> swi ss. cor <- cor(swi ss. x)
```

or do both by saving the results and then printing the object

```
> swi ss. cor <- cor(swi ss. x)
> swi ss. cor
```

Explicitly printing the results in a function is frowned upon unless the function is a `print` method for a classed object. The evaluation mechanism determines whether to print the result.

This convention is waived for the dialog functions, as it is necessary to provide a mechanism for both saving and printing the output within the function.

Another difference between using a function from the command line and from a dialog is that the command line alternately between an expression and the output related to that expression. Hence it is clear which expression and output go together. The output from a dialog is not preceded by an expression (the expression evaluated will be stored in the history log but is not printed to the output stream). Hence it is necessary to provide a header preceding the output which indicates the source of the output. The header lines also serve to separate subsequent sets of output.

If the user requests printed output, the header is printed with `cat`, and the result object with `print`:

```
header.txt <- paste("\n\t*** Covariance for data in:
", data.name, "***\n\n")
...
if(print.it) {
  cat(header.txt)
  print(coeff)
}
```

Generally `cat` is used to print character strings describing the output, and `print` is used for other objects.

Note that that convention for header lines is to use a character string of the form:

```
"\n\t***  Output  Descri pti on  ***\n\n"
```

## Saving Results

In this dialog, the results need not be explicitly saved within the function. The command is written such that the result is assigned to the name specified in **Save As** if a name is specified.

Note that the value is returned invisibly:

```
i nvi si bl e(coeff)
```

As we have already printed the result if printing is desired, it is necessary to suppress the autoprinting which would normally occur if the result were returned without assignment.

In some cases it is necessary to assign the result within the function. In particular, this is required if the function is creating the data and then displaying it in a Data window. For example, this is done in the `menuFacDesign` function, which creates a data frame `new.design` containing a factorial design, and displays this in a Data window.

```
i f(mi ssi ng(save.name))
    return(new.design)
el se {
    assi gn(save.name, new.design, where = 1,
           immediate = T)
    i f(i s.sgui .app() && show.p)
        gui OpenVi ew(cl assname = "data.frame",
                      Name = save.name)
    i nvi si bl e(new.design)
}
```

If `save.name` is not specified, the result is simply returned. Otherwise, the result is immediately assigned to the working directory. Then the data frame is displayed in a Data window if the Windows S-PLUS GUI is being run and the user specifies `show.p=T` by checking the **Show in Data Window** box in the **Factorial Design** dialog.

The explicit assignment is necessary because the data frame must exist as a persistent object on disk before it can be displayed in a Data window.

## Saving Additional Quantities

In some cases the user may want access to other quantities which are not part of the standard object returned by the function, such as residuals or predicted values. At the command line these functions can be accessed using extractor functions such as `resi d` and `predi ct`. In dialogs it may be preferable to save



these objects into specified data frames using the save mechanism as described above. The section `##Modeling dialog saved results##` discusses this situation.

## Plots

The Windows S-PLUS GUI supports multiple coexisting Graph sheets, each of which may have multiple tabbed pages. When a new graph is created it may do one of three things:

- Replace the current graph (typically the graph most recently created).
- Create a new tab on the current Graph sheet.
- Create a new Graph sheet.

The default behaviour is for a statistical dialog function to open a new Graph sheet before creating graphs. If the function produces multiple graphs, these appear on multiple tabs in the new Graph sheet.

This autocreation of new Graph sheets may annoy some users due to the proliferation of windows. The **Graphs Options** dialog has a **Statistics Dialogs Graphics: Create New Graph Sheet** check box which indicates whether or not to create a new Graph sheet for each new set of plots.

It is good form for any plots created from dialogs to follow the dictates of this option. This is done by calling `new.graphsheet` before plots are produced. This function will create a new Graph sheet if the abovementioned option specifies to do so. The `new.graphsheet` function should only be called if plots are to be produced, and should only be called once within the function as calling it multiple times would open multiple new Graph sheets.

The `menuAcf` function provides an example of the use of `new.graphsheet`:

```
if(as.logical(plot.it)) {
  new.graphsheet()
  acf.plot(acf.obj)
}
```

## The Callback Function

Most dialogs of any real complexity will have some interactions between the allowable argument values. In the **Correlations and Covariances** dialog the **Fraction to Trim** is only relevant for correlations. Hence this field should be disabled if **Variance/Covariance** is checked. The callback function `backCor` updates the values and enable status of controls based on actions in the dialog.

When the dialog is launched, **OK** or **Apply** is pressed, or a control is changed, the callback function is executed. The function is passed a data frame containing character strings reflecting dialog prompts, values, option lists, and enable status. These strings may be accessed and modified to make changes to the dialog.

This function starts by getting the name of the active property. This is the property which was last modified.

```
backCor <- function(data)
{
  activeprop <- cbGetActiveProp(data)
```

If the dialog has just been launched then **Fraction to Trim** should only be enabled if **Correlation** is checked. If **Correlation** is checked then **Variance/Covariance** should be unchecked, and vice versa. If which check box is checked changes, the enable status of **Fraction to Trim** must change. The next set of lines enforces these constraints.

```
  if(cbl$InitDialogMessage(data) || activeprop ==
     "SPropCorrP" || activeprop == "SPropCovP") {
    if(activeprop == "SPropCorrP") {
      if(cbGetCurrValue(data, "SPropCorrP") ==
         "F") {
        data <- cbSetEnableFlag(data,
                                "SPropTrim", F)
        data <- cbSetCurrValue(data,
                                "SPropCovP", "T")
        ...
      }
```

If the dialog has just been launched or the **Data Frame** has changed, the list of variables must be created. This is done by checking that an object of the specified name exists, and if so getting the object's column names and pasting them together with the (All Variables) string. Note that the list of variable names is passed as a single comma delimited string rather than as a vector of strings.

```
    if(activeprop == "SPropDataX2" || cbl$InitDialogMessage(
       data)) {
      if(exists(cbGetCurrValue(data, "SPropDataX2")))
      {
        x.names <- names(get(cbGetCurrValue(data,
                                              "SPropDataX2")))
        x.names <- paste(c("All Variables"),
                        x.names), collapse = ", ")
      }
```

---

```
data <- cbSetOptionList(data,
  "SPropVariableX2", x.names)
}
```

Lastly, the data frame containing the dialog status information is returned.

```
invisible(data)
```

The most common uses of callback functions are to fill variable lists and to enable/disable properties as is done by `backAcf`. For further examples, search for functions whose names start with `back`, or look at the `FunctionInfo` for a dialog with callback behaviour of interest to determine the name of the relevant callback function.

## Modeling Dialogs

A powerful feature of S-PLUS is the object-oriented nature of the statistical modeling functions. Statistical modeling is an iterative procedure in which the data analyst examines the data, fits a model, examines diagnostic plots and summaries for the model, and refines the model based on the diagnostics. Modeling is best performed interactively, alternating between fitting a model and examining the model.

This interactive modeling is supported in S-PLUS by its class and method architecture. Generally there will be a modeling function (such as `lm` for linear regression) which fits a model, and then a set of methods (such as `print`, `plot`, `summary`, and `anova`) which are used to examine the model. The modeling function creates a model object whose class indicates how it is handled by the various methods.

This differs from other statistical packages, in which all desired plots and summaries are typically specified at the time the model is fit. If additional diagnostic plots are desired the model must be completely refit with options indicating that the new plots are desired. In S-PLUS additional plots may be accessed by simply applying the plot method to the model object.

In moving from a set of command line functions to dialogs for statistical modeling, the desired level of granularity for action specification changes. At the command line the basic strategy would be to issue a command to fit the model, followed by separate commands to get the desired plots and summaries. The ability to use such follow-on methods is still desirable from a graphical user interface, but it should be a capability rather than a requirement. The user will generally want to specify options for fitting the model plus desired plots and summaries in a single dialog, with all results generated when the model is fit.

The design of the statistical modeling dialogs is such that the user may specify the desired summaries and plots at the time the model is fit, but it is also possible to right-click on a model object in the Object Explorer and access summary and plot methods as a follow-on action. Generally the Results, Plot, and Predict tabs on the modeling dialog are also available as separate dialogs from the model object context menu.

This section describes the general design of statistical modeling dialogs by examining the **Linear Regression** dialog. The following section describes the structure of the functions connected to the modeling and follow-on method dialogs. The statistical modeling dialogs follow the same design principles as are described here, but details will vary.

## Model Tab

The **Model** tab describes the data to use, the model to fit, the name under which to save the model object, and various fitting options. It is typical to have **Data**, **Formula**, and **Save Model Object** groups which are similar to those in the **Linear Regression** dialog.

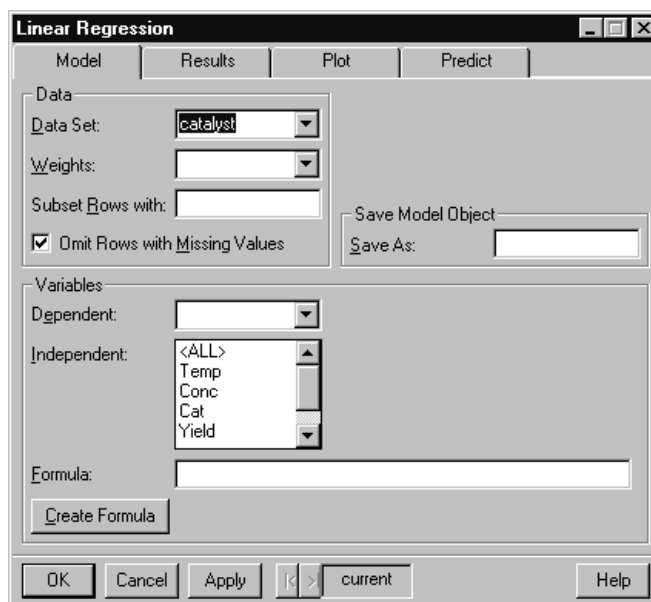


Figure 17.21: The Model tab of the Linear Regression dialog.

## Data Group

The **Data Set** property is a drop-down list of available data sets. This list is filled with the data sets which are in the working database, or have been displayed by filtering on other databases in the Object Explorer. This specifies the data argument to the modeling function.

The **Weights** property is a list of columns in the selected data set. The selected column will be used as weights in the model. This specifies the weights argument to the modeling function.

The **Subset Rows with** property takes an expression which is used as the subset expression in the model. This specifies the subset argument to the modeling function.

The **Omit Rows with Missing Values** check box specifies how missing values are handled. Checking this box is equivalent to specifying `na.action=na.omit`, while leaving it unchecked is equivalent to `na.action=na.fail`. Some dialogs (such as **Correlations and Covariances**) instead have a **Method to Handle Missing Values** list box, which provides additional missing value actions.

## Variables Group

The Variables group includes controls for specifying the **Dependent** and **Independent** variables. As you select or enter variables in these controls, they are echoed in the **Formula** control. The **Formula** specifies the form of the model, that is what variables to use as the predictors (independent variables) and the response (dependent variables). This specifies the formula argument to the modeling function.

Most modeling dialogs have a **Create Formula** button, which launches the Formula Builder dialog when pressed. This dialog allows point-and-click formula specification.

Dialogs in which the formula specifies a set of covariates rather than predictors and a response (such as **Factor Analysis**) have a **Variables** list rather than a **Create Formula** button.

## Save Model Object Group

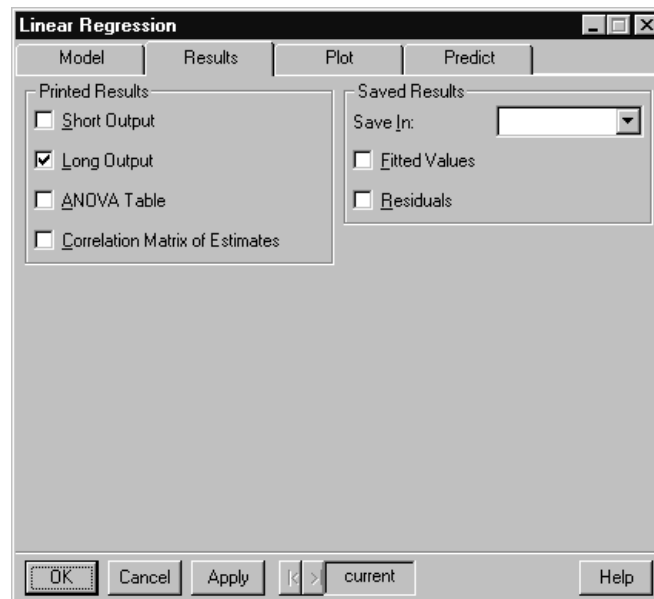
The **Save As** edit field specifies the name under which to save the model object. The returned value from the function called by the dialog is assigned to this name. Typically the default value is prefaced by `last` and is model specific, as in `last.im`.

## Options Tab

In the **Linear Regression** dialog, all of the necessary fitting options are available on the **Model** tab. Some other modeling dialogs, such as the **Logistic Regression** dialog, have more options which are placed on a separate tab. An **Options** tab may be useful either due to the availability of a large number of options, or to shelter the casual user from more advanced options.

## Results Tab

The **Results** tab generally has groups for specifying the printed and saved results.



*Figure 17.22: The Results tab of the Linear Regression dialog.*

### Printed Results Group

The **Printed Results** specify the types of summaries to print. These summaries will be displayed in a Report window, or in another location as specified in the **Text Output Routing** dialog.

Checking the **Short Output** check box generally indicates that the print method for the model object will be called.

Checking the **Long Output** check box generally indicates that the summary method for the model object will be called.

Other options will vary based on the statistical model.

## Saved Results Group

The **Saved Results** specify results to be saved separate from the model object, such as fitted values and residuals. These saved components are usually of the same length as the data used in the model. It is often of interest to plot these quantities against the data used to construct the model, and hence it is convenient to save these values in a data frame for later use in plotting or analysis.

The **Save In** edit field takes the name of a data frame in which to save the results. This may be a new data frame or the data frame used to construct the model. If the data frame named exists and has a different number of rows than are in the results, then a new name will be constructed and the results saved in the new data frame.

Check boxes specify what results to save. Common choices include **Fitted Values** and **Residuals**.

## Plot Tab

The Plot tab specifies which plots to produce and plotting options. Typically a **Plots** group provides check boxes to select plot types to produce. Other groups provide options for the various plots.



Figure 17.23: The Plot tab of the Linear Regression dialog.

## Predict Tab

The **Predict** tab specifies whether predicted values will be saved, using similar conventions as the **Saved Results** group on the **Results** tab.

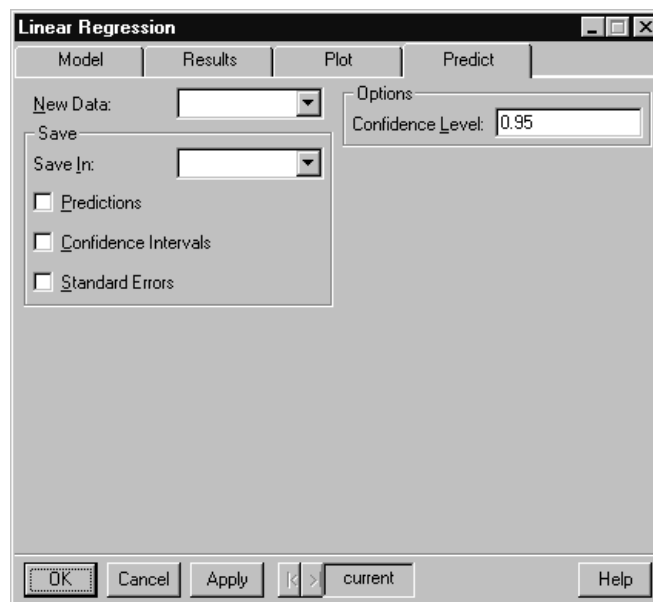


Figure 17.24: The Predict tab of the Linear Regression dialog.

The **New Data** edit field accepts the name of a data frame containing observations for which predictions are desired. This specifies the `newdata` argument to `predict`. If this is left empty the data used to fit the model will be used.

The **Save In** edit field takes the name of a data frame in which to save the results. This may be a new data frame or the data frame used to construct the model. If the data frame named exists and has a different number of rows than are in the results, then a new name will be constructed and the results saved in the new data frame.

Check boxes specify what results to save. Common choices include **Predictions**, **Confidence Intervals**, and **Standard Errors**.

Other options related to prediction may also be present.

## Other Tabs

A statistical model may have additional methods specific to that type of model. The dialog for this model may have additional tabs, such as the **Prune/Shrink** tab on the **Tree Regression** dialog. The only limitation on additional tabs is that each dialog is limited to at most five tabs.



## Modeling Dialog Functions

Command line functions for statistical models generally consist of a fitting function (such as `lm`) and method functions for that model (such as `print.lm`, `summary.lm`, `plot.lm`, and `predict.lm`). Similarly, the wrapper functions called by the dialogs consist of a main dialog function (such as `menuLm`) and method functions (`tabSummary.lm`, `tabPlot.lm`, and `tabPredict.lm`). The method functions are called by the main function, and also from separate dialogs available by right-clicking on the model object in the Object Explorer. This structure allows the methods to be called either at the time of fitting the model, or later for use with a fitted model object.

The convention is to name the main dialog function with the preface `menu`, and the method functions with the preface `tab`. The suffix of the function reflects the name of the related command line function.

## Main Function

The main dialog function has two primary purposes: fitting the model and calling the method functions. The main dialog function for linear regression is `menuLm`:

```
menuLm <-
function(formula, data, weights, subset, na.omit.p = T,
  print.short.p = F, print.long.p = T, print.anova.p = T,
  print.correlation.p = F, save.name = NULL, save.fit.p =
  F, save.resid.p = F, plotResidualVsFit.p = F,
  plotSqrtAbsResid.p = F, plotResponseVsFit.p = F,
  plotQQ.p = F, plotRFSpread.p = F, plotCook.s.p = F,
  smooths.p = F, rugplot.p = F, id.n = 3,
  plotPartialResid.p = F, plotPartialFit.p = F,
  rugplotPartialResid.p = F, scalePartialResid.p = T,
  newdata = NULL, predobj.name = NULL, predict.p = F, ci.p
  = F, se.p = F, conf.level = 0.95)
{
  fun.call <- match.call()
  fun.call[[1]] <- as.name("lm")
  if(na.omit.p)
    fun.call$na.action <- as.name("na.omit")
  else fun.call$na.action <- as.name("na.fail")
  fun.args <- is.element(arg.names(fun.call), arg.names(
    "lm"))
  fun.call <- fun.call[c(T, fun.args)]
  lmodj <- eval(fun.call)#
```

```
# Call summary function:
tabSummary.lm(lmobj, print.short.p, print.long.p,
              print.correlation.p, print.anova.p, save.name,
              save.fit.p, save.resid.p)#

# Call plot function:
if(any(c(plotResidualVsFit.p, plotSqrtAbsResid.p,
        plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
        plotCooks.p, plotPartialResid.p))) tabPlot.lm(
    lmobj, plotResidualVsFit.p,
    plotSqrtAbsResid.p, plotResponseVsFit.p,
    plotQQ.p, plotRFSpread.p, plotCooks.p,
    smooths.p, rugplot.p, id.n,
    plotPartialResid.p, plotPartialFit.p,
    rugplotPartialResid.p,
    scalePartialResid.p)#

# Call predict:
if(any(c(predict.p, ci.p, se.p)))
    tabPredict.lm(lmobj, newdata, predobj.name,
                 predict.p, ci.p, se.p, conf.level)
invisible(lmobj)
}
```

This function has one argument corresponding to each property in the dialog, with the exception of the **Save As** field and invisible fields used for formatting.

The first eight lines of the function are used to fit the model. The approach is to construct an expression which specifies the appropriate call to the model function, and then to evaluate this expression. The somewhat sophisticated syntax is used so that the `call` component in the model object has the appropriate names as specified in the dialog. This general recipe may be used for any function, with “`lm`” replaced by the name of the modeling function.

Note that `na.action` is specified as a check box in the dialog, but as a function name in `lm`. This necessitates modification of this argument before evaluating the call to `lm`.

After the model has been fit, the methods `tabSummary.lm`, `tabPlot.lm`, and `tabPredict.lm` are called. For efficiency, calls are only made to `tabPlot.lm` and `tabPredict.lm` if the options are specified such that calling these functions will produce some action.

Finally, the model object is returned. It is returned invisibly since any printing of the object has been handled by `tabSummary.lm`.

**Summary Method** The summary method produces printed summaries and saves specified results in a data frame separate from the model object. The summary method for the **Linear Regression** dialog is `tabSummary.lm`:

```
tabSummary.lm <-
function(lmobj, print.short.p = F, print.long.p = T,
  print.correlation.p = F, print.anova.p = F, save.name =
  NULL, save.fitt.p = F, save.resid.p = F)
{
  if(print.short.p || print.long.p || print.anova.p) {
    cat("\n\t*** Linear Model ***\n")
    if(print.short.p) {
      print(lmobj)
    }
    if(print.long.p) {
      print(summary(lmobj, correlation =
        print.correlation.p))
    }
    if(print.anova.p) {
      cat("\n")
      print(anova(lmobj))
    }
    cat("\n")
  }
  # Save results if requested:
  if(any(c(save.fitt.p, save.resid.p)) && !is.null(
    save.name)) {
    saveobj <- list()
    if(save.fitt.p)
      saveobj[["fitt"]] <- fitted(lmobj)
    if(save.resid.p)
      saveobj[["residuals"]] <- residuals(
        lmobj)
    saveobj <- data.frame(saveobj)
    n.save <- nrow(saveobj)
    if(exists(save.name, where = 1)) {
      if(inherits(get(save.name, where = 1),
        "data.frame") && nrow(get(
          save.name, where = 1)) == n.save
      )
        assign(save.name, cbind(get(
          save.name, where = 1), saveobj
```

```
      ), where = 1)
    else {
      newsave.name <- unique.name(
        save.name, where = 1)
      assign(newsave.name, saveobj,
        where = 1)
      warning(paste(
        "Fit and/or residuals saved in"
        ,
        newsave.name))
    }
  }
  else assign(save.name, saveobj, where = 1)
  invisible(NULL)
}
invisible(lmobj)
}
```

The first part of this function is responsible for printing the specified summaries. If any printed output is specified, a header will be printed demarcating the start of the output. Based on option values, the `print`, `summary`, and other methods for the model will be called.

The second part of the function concerns itself with saving the requested values. Extractor functions such as `fitted` and `residuals` are used to get the desired values. The remainder of the code specifies whether to add columns to an existing data frame, create a new data frame with the specified name, or create a new data frame with a new name to avoid overwriting an existing object.

The model object passed to this function is returned invisibly.

## Plot Method

The plot function opens a new Graph sheet if necessary, and produces the desired plots. The plot method for the **Linear Regression** dialog is `tabPlot.lm`:

```
tabPlot.lm <-
function(lmobj, plotResidualVsFit.p = F, plotSqrtAbsResidual.p =
  F, plotResponseVsFit.p = F, plotQQ.p = F,
  plotRFSpread.p = F, plotCooks.p = F, smooths.p = F,
  rugplot.p = F, id.n = 3, plotPartialResidual.p = F,
  plotPartialFit.p = F, rugplotPartialResidual.p = F,
  scalePartialResidual.p = T, ...)
{
```

```

if(any(c(plotResidualsFit.p, plotSqrtAbsResidual.p,
        plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
        plotCooks.p, plotPartialResidual.p)))
  new.graphsheet()
if(any(c(plotResidualsFit.p, plotSqrtAbsResidual.p,
        plotResponseVsFit.p, plotQQ.p, plotRFSpread.p,
        plotCooks.p))) {
  whichPlots <- seq(1, 6)[c(plotResidualsFit.p,
                           plotSqrtAbsResidual.p, plotResponseVsFit.p,
                           plotQQ.p, plotRFSpread.p, plotCooks.p)]
  plot.lm(lmobj, smooths = smooths.p, rugplot =
          rugplot.p, id.n = id.n, which.plots =
          whichPlots, ...)
}
if(plotPartialResidual.p) {
  partial.plot(lmobj, residual = T, fit =
              plotPartialFit.p, scale =
              scalePartialResidual.p, rugplot =
              rugplotPartialResidual.p, ...)
}
invisible(lmobj)
}

```

If any plots are desired, the function first calls `new.graphsheet` to open a new Graph sheet if necessary. The plot method for `lm` is then called. Some additional plots useful in linear regression are produced by the `partial.plots` function, which is called if partial residual plots are desired.

The model object passed to this function is returned invisibly.

## Predict Method

The `predict` function obtain predicted values for new data or the data used to fit the model. The `predict` method for the **Linear Regression** dialog is `tabPredict.lm`:

```

tabPredict.lm <-
function(object, newdata = NULL, save.name, predict.p = F,
        ci.p = F, se.p = F, conf.level = 0.95)
{
  if(is.null(newdata))
    predobj <- predict(object, se.fit = se.p || ci.p
                      )
  else predobj <- predict(object, newdata, se.fit = se.p ||

```

```
      ci.p)
if(ci.p) {
  if(conf.level > 1 && conf.level < 100)
    conf.level <- conf.level / 100
  t.value <- qt(conf.level, object$df.residual)
  lower.name <- paste(conf.level * 100, "% L. C. L.",
    sep = "")
  upper.name <- paste(conf.level * 100, "% U. C. L.",
    sep = "")
  predobj[[lower.name]] <- predobj$fit - t.value *
    predobj$se.fit
  predobj[[upper.name]] <- predobj$fit + t.value *
    predobj$se.fit
}
# remove prediction column and se column if not requested:
if(!predict.p)
  predobj$fit <- NULL
if(!se.p)
  predobj$se.fit <- NULL
predobj$residual.scale <- NULL
predobj$df <- NULL
predobj <- as.data.frame(predobj)
n.predict <- nrow(predobj)
if(exists(save.name, where = 1)) {
  if(inherits(get(save.name, where = 1),
    "data.frame") && nrow(get(save.name,
    where = 1)) == n.predict)
    assign(save.name, cbind(get(save.name,
    where = 1), predobj), where = 1)
  else {
    newsave.name <- unique.name(save.name,
    where = 1)
    assign(newsave.name, predobj, where = 1)
    warning(paste("Predictions saved in",
    newsave.name))
  }
}
else assign(save.name, predobj, where = 1)
invisible(NULL)
}
```

---

	<p>The function calls <code>predict.lm</code> to obtain predicted values and standard errors. It then calculates confidence intervals, and removes any undesired components from the output. The predictions are then stored in a data frame using the same type of algorithm as in <code>tabSummary.lm</code>.</p> <p>No value is returned.</p>
<b>Other Methods</b>	<p>If the dialog has additional tabs, other dialog methods will be available. For example, the <b>Tree Regression</b> dialog has a <b>Prune/Shrink</b> tab with a corresponding function <code>tabPrune.tree</code>.</p>
<b>Callback Functions</b>	<p>Modeling dialogs will have callbacks similar to those for a simple dialog. The callback function for <b>Linear Regression</b> is <code>backLm</code>.</p> <p>Method dialogs may also need callback functions for use when the related dialog is launched from the model object's context-menu. An example is the <code>tabPlot.princomp</code> callback function used by the <b>Principal Components</b> plot dialog. Method dialogs need callback functions less frequently than do the main modeling dialogs.</p>
<b>Class Information</b>	<p>Every model object has a class indicating what type of model the object is. For example, linear regression model objects are of class <code>lm</code>. At the command line, functions such as <code>print</code> look to see if there is a special function <code>print.lm</code> to use when they are given an <code>lm</code> object, and if not they use the default plot method for the object.</p> <p>Similarly, the Object Explorer has a limited set of actions it can perform on any object. In addition, it can allow class-specific actions. The <code>ClassInfo</code> object tells the Object Explorer what to do with objects of the specified class. In particular, the double-click action and the context menu may be specified.</p>
<b>Double-Click Action</b>	<p>The double-click action is the action to perform when the model object is double-clicked in the Object Explorer. The convention for statistical models is to produce printed summaries. In linear regression the <code>tabSummary.lm</code> function is called.</p>

# Context Menu

The context menu is the menu launched when the user right-clicks on a model object in the Object Explorer. Figure 17.25 displays the context menu for linear regression (lm) objects:

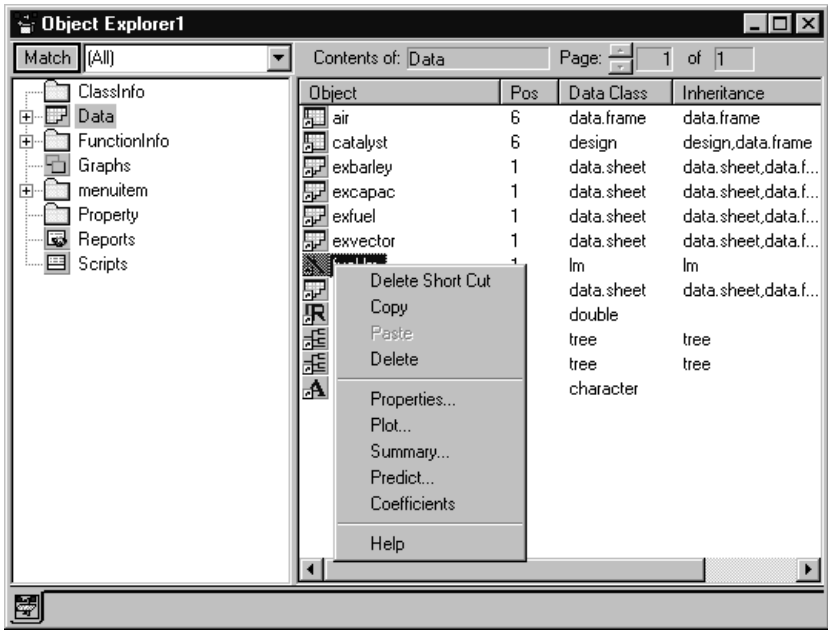
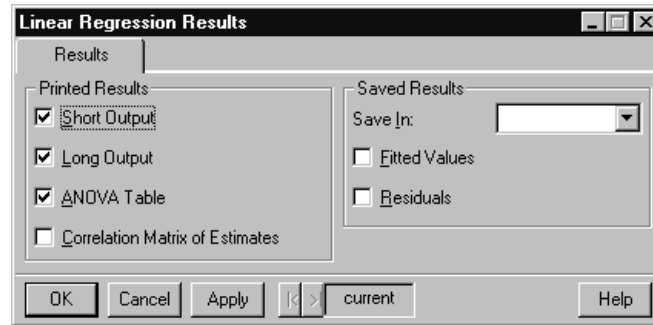


Figure 17.25: The context menu for linear regression objects.

This menu includes the methods made available through the dialog method functions such as **Summary**, **Plot**, and **Predict**. Each of these menu items launches a dialog which calls the respective function. For example, selecting **Summary** launches the **Linear Regression Results** dialog shown in Figure 17.26.



Some additional methods may be present, such as the **Coefficients** menu item which calls the `coef` function. This allows the user to quickly print the coefficients in the current Report window.



*Figure 17.26: The Linear Regression Results dialog.*

## Method Dialogs

The dialog functions are designed to make it easy to have dialogs for the various method functions.

A `FunctionInfo` object for each of the method dialogs defines the relevant dialog and its relationship to the method function. Most of the properties will be exactly the same as the properties in the modeling dialog. The `SPropCurrentObject` property is used as the first argument to indicate that the currently selected object is the model object of interest, and `SPropInvISIBLEReturnObject` is used for the return value.

Look at the properties of the `FunctionInfo` object for `tabSummary.lm` for details on the `FunctionInfo` object properties for the dialog given above.

## Dialog Help

The built-in statistical dialogs in S-PLUS have help entries in the main S-PLUS help file. As this is a compiled WinHelp file, it may not be extended by the user. However, the user may still make help information available for their dialogs.

The **HelpCommand** property of a `FunctionInfo` object specifies an S-PLUS expression to evaluate when the dialog's Help button is pushed.

If the user has created a help file for the command line function, the help for this function may be launched using an expression such as `help(menuLm)`. The help for the dialog might also be placed in a separate text file and displayed using an expression such as `system("notepad C:\\myhelp.txt")`. With creative use of `system`, the user might be able to display help files in another format such as HTML or Word.



---

<b>Introduction</b>	<b>732</b>
<b>Creating a Library</b>	<b>734</b>
Steps in Creating a Library	735
Creating Directories	735
Storing Functions	736
Storing Interface Objects	736
Copying Help Files	738
Storing Other Files	738
Start-up and Exit Actions	739
<b>Distributing the Library</b>	<b>740</b>

## INTRODUCTION

Preceding chapters have described how to create functions, dialogs, and other tools. S-PLUS libraries are a convenient way to package up user-created functionality in order to share it with other users.

A library will contain the following types of objects:

- Functions
- Example data sets
- Graphical user interface objects such as menus and dialogs
- Help files
- Compiled C and Fortran objects ready for dynamic or DLL loading

This chapter describes how to create and use libraries.

### Example Functions

In the preceding chapter, we created a function `my.sqrt` which calculates and prints the square root of a number. We also created a dialog for this function and a menu item on the Data menu which launches the dialog. In this chapter we will create a library for these objects.

The following commands create the function and the interface objects:

```
my.sqrt <- function(x){
  y <- sqrt(x)
  cat("\nThe square root of ", x, " is ", y, ".\n", sep="")
  invisible(y)
}

guiCreate(classname="Property", Name="MySqrtInput",
  DialogControl="String", UseQuotes=F,
  DialogPrompt="Input Value")
```

```
gui Create(classname="FunctionInfo", Name="my.sqrt",  
    DialogHeader="Calculate Square Root",  
    PropertyList="SPropList: si bl eReturnObject,  
        MySqrtInput",  
    ArgumentList="#0=SPropList: si bl eReturnObject,  
        #1=MySqrtInput")  
  
gui Create(classname="MenuItem",  
    Name="$$SPUsMenuBar$Data$MySqrt",  
    Type="MenuItem",  
    MenuItemText="Square Root...",  
    Action="Function", Command="my.sqrt")
```

---

## CREATING A LIBRARY

We can create libraries to distribute functions and related files in an organized manner. A library is a directory containing function and data objects, interface objects, help files, and related files such as compiled C or Fortran code. Attaching a library makes the objects in that library available to the user by adding the relevant `_Data` directory to the search list.

The library may contain a `First.lib` function indicating actions to perform upon attaching the library. Typically this might include dynamically loading C or Fortran code and loading interface objects such as menus, toolbars, and dialogs. A `Last.lib` function may specify actions to perform upon unloading the library or exiting.

Any directory containing an `_Data` directory may be used as a library. If the directory is a subdirectory of the library directory under the S-PLUS program directory then the library may be made available by using:

```
> library(Libraryname)
```

For example, `library(cluster)` will attach the built-in cluster library. It is not necessary for the library to reside in the library directory under the S-PLUS program directory. If it resides elsewhere the path to the directory containing your library directory must be specified with the `lib.loc` argument to `library`. For example, if your library is called *mylib*, and it exists as a directory under `c:\stat` then the library can be made available with

```
> library(mylib, lib.loc = "c:\\stat")
```

If the programmer shares a file system with the users to which the programmer intends to distribute routines, the library can reside anywhere on the file system and the end users need only be told to use the `library` command with the proper `lib.loc` argument.

To distribute the library to users who do not share a file system we can create a batch script to install the library properly. The most polished way to distribute a library is to create a self-extracting archive which has knowledge of where to install the library. Details of creating batch scripts and self-extracting executables depend upon the tools available to the programmer and are beyond the scope of this manual.

## Steps in Creating a Library

The steps in creating a library are:

- Create a main directory for the library in the library subdirectory of the S-PLUS program directory.
- Create `_Data` and `_Prefs` directories in this main directory.
- Store the function and data objects in the `_Data`.
- Store menu and dialog objects in `*.DFT` files placed in `_Prefs`.
- If help files are present, create a `_Help` directory under `_Data` and place the help files in this directory.
- Place other files related to the library in the library directory or in `_Prefs`.
- Write `First.lib` and `Last.lib` functions to specify actions to perform upon loading and unloading the library. For example, we might want to load `*.DFT` files and run commands to create menu items.

## Creating Directories

The easiest way to create directories is via Windows Explorer. If a programatic approach is preferred, the `mkdi r` command may be used.

First we will create directories using the function `mkdi r`. We will use `getenv("SHOME")` to find the location of the S-PLUS program directory.

```
> mkdi r(paste(getenv("SHOME"),
+             "\\l i brary\\myl i b", sep=" "))
> mkdi r(paste(getenv("SHOME"),
+             "\\l i brary\\myl i b\\_Data", sep=" "))
> mkdi r(paste(getenv("SHOME"),
+             "\\l i brary\\myl i b\\_Prefs", sep=" "))
> mkdi r(paste(getenv("SHOME"),
+             "\\l i brary\\myl i b\\_Data\\_Hel p", sep=" "))
```

## Storing Functions

Next we will use `library` to attach the new `mylib` library. We will use `first=T` to specify that it should be attached as the second directory in the search list.

```
> library(mylib, first=T)
```

We can use `assign` to copy our `mylib` related objects into this directory. We will make a vector of names of the functions we want to copy and then loop over this vector.

```
> mylib.funcs<-c("my.sqrt")
> for (i in 1:length(mylib.funcs)){
+   assign(mylib.funcs[i],
+   get(mylib.funcs[i]), where=2)
+ }
```

(In this example the loop is unnecessary, as we have only a single function. We provide the syntax for use with multiple objects.)

Note that this will produce warnings indicating that we are creating objects in the second directory on the search list with the same names as objects in other directories on the search list. These warnings are expected and in this case indicate that we will want to remove the objects from the first directory after making the copies to avoid duplicate names.

An alternate approach is to keep a text file **mylib.ssc** containing our functions. We can then attach the library directory as our working directory and use `source` to create the functions in this directory.

```
> attach(paste(getenv("SHOME"),
+   "\\library\\mylib\\_Data", sep=""), pos=1)
> source("mylib.ssc")
```

## Storing Interface Objects

User-created interface objects may include `MenuItem`, `Toolbar`, `ToolbarButton`, `Property`, `FunctionInfo`, and `ClassInfo` objects. In contrast to function and data objects, these objects are not stored in separate files contained under an `_Data` directory. Instead, they are serialized to class-specific files in the user's `_Prefs` directory.

`ClassInfo`, `FunctionInfo`, and `Property` objects are stored in `axclass.dft`, `axfunc.dft`, and `axprop.dft`. `Toolbar` and `ToolbarButton` objects are stored in `*.STB` files, with one file per toolbar. `MenuItem` objects are stored in `smenu.smn`.



When we create a library, we will want to package up the `ClassInfo`, `FunctionInfo`, and `Property` objects in files separate from the built-in objects shipped with S-PLUS. This is done by specifying where to save each interface object that we create.

Note that this procedure is only used for `ClassInfo`, `FunctionInfo`, and `Property` objects. `Toolbar` and `ToolbarButton` objects are already in separate files, and `MenuItems` are best created on the fly when loading the module.

## Save Path Name

Each interface object has a `SavePathName` property which may be used to specify the name of the file in which to save the object. We recommend defining variables for the desired file names at the top of the script used to generate the interface objects, and then referring to this variable when creating the objects.

For example, we would add a `SavePathName` argument when creating the `Property` object `MySqrtInput` and the `FunctionInfo` object `my.sqrt`:

```
prop.file <- "props.dft"
funcinfo.file <- "funcinfo.dft"
guiCreate(classname="Property", Name="MySqrtInput",
          DialogControl="String", UseQuotes=F,
          DialogPrompt="Input Value",
          SavePathName= prop.file)

guiCreate(classname="FunctionInfo", Name="my.sqrt",
          DialogHeader="Calculate Square Root",
          PropertyList="SPropInvisibleReturnObject,
                      MySqrtInput",
          ArgumentList="#0=SPropInvisibleReturnObject,
                      #1=MySqrtInput",
          SavePathName= funcinfo.file)
```

## Store Default Objects

The `guiStoreDefaultObjects` function will store all interface objects of a specific class into the files specified by their `SavePathName` property. An object without a `SavePathName` will be stored in the default file for objects of its class.

We will store the `Property` and `FunctionInfo` objects:

```
> guiStoreDefaultObjects("Property")
> guiStoreDefaultObjects("FunctionInfo")
```

The `gui LoadDefault tObjects` function loads all objects of a specified class from a specified file. It will update the `SavePathName` property for the objects read from the file to reflect the current path to the file. This may differ from the path to the file when the objects were stored.

## Copying Help Files

If we have help files which we want to include in the library we need to copy the text files into `library\mylib\_Data\_Help`.

It is important that we give each help file the same name as the object to which it is related. The name is the mechanism used by the `help` function to determine what help file to display. The `true.file.name` function will give the name under which each function is stored on disk. This is the same name to use for the help file.

Specify the name of the function and the location in the search list of the library to get the file name under which to save the help file.

```
> true.file.name("my.sqrt", where=2)
[1] "__3"
```

So we would save the help file for `my.sqrt` as `__3` in `library\mylib\_Data\_Help`.

## Storing Other Files

Other files may also belong in the library, and hence should be copied to `library\mylib`. Possibilities include:

- C and Fortran object files
- Documentation for the library

Files related to interface objects should be copied to `library\mylib\_Prefs`. This includes:

- Toolbar files (\*.STB)
- Bitmap files containing toolbar button images

## Start-up and Exit Actions

Attaching a library will make the functions and data sets in the library available. Additional steps are needed to create menus and dialogs, or to load object code.

The `.First.Lib` function is run when a library is attached. This may be used to perform actions such as loading interface objects and creating menu items when a library is attached:

```
.First.Lib<-function(Library, section){
  prefs.loc <- paste(getenv("SHOME"),
    "\\Library\\mylib\\_Prefs", sep="")
  guiLoadDefaultObjects("Property",
    paste(prefs.loc, "\\prop.dft", sep=""))
  guiLoadDefaultObjects("FunctionInfo",
    paste(prefs.loc, "\\funcinfo.dft", sep=""))
  guiCreate(classname="MenuItem",
    Name="$$SPI usMenuBar$Data$MySqrt",
    Type="MenuItem", MenuItemText="Square Root...",
    Action="Function", Command="my.sqrt")
  invisible()
}
```

The `.Last.Lib` function is run when a library is detached or the program exits normally. This may be used to perform cleanup actions upon exit.

## DISTRIBUTING THE LIBRARY

Once we have created the library we will want to package it up as a compressed archive or a self-extracting executable. One approach is to use a utility such as WinZip or pkzip to compress up the gaussfit library directory, and include a readme.txt file indicating how to unzip this as a subdirectory of the library directory. A more sophisticated approach is to use these tools to produce a self-extracting archive which unpacks to the proper location. Creating these archives is beyond the scope of this manual.

# COMMAND LINE OPTIONS

# 19

---

<b>Using the Command Line</b>	<b>742</b>
<b>Command Line Parsing</b>	<b>744</b>
Variables	745
Switches	754
<b>Working With Projects</b>	<b>757</b>
The Preferences Directory	757
The Data Directory	758
<b>Customizing Your Session at Start-up and Closing</b>	<b>759</b>
Setting S_FIRST	759
Customizing Your Session at Closing	760
<b>Enhancing S-PLUS</b>	<b>761</b>
Adding Functions and Data Sets to Your System	761
Adding Object Code	762

# USING THE COMMAND LINE

S-PLUS accepts a number of optional commands on the **splus.exe** executable's command line that give users significant control over the operation of S-PLUS.

These facilitate running S-PLUS in an automated or batch environment. They also make it possible to readily alter the behavior of S-PLUS on a session by session basis. Some users may find it handy to have several shortcuts (or program manager icons if running older versions of Windows), each of which starts S-PLUS with specific projects and options selected by default.

**Note**

This chapter refers to the **splus.exe** command line that is used to start execution of S-PLUS, not the "Commands window" that is used to enter commands once S-PLUS has started. The S-PLUS command line refers to anything that follows the name of the executable (**splus.exe** by default) in the shortcut, program manager icon, or batch file from which S-PLUS may be started. On the S-PLUS command line only certain switches are permitted and have their own syntax as discussed in this section.

Command line processing is only implemented for **splus.exe**, not for **sqpe.exe**.

Filenames that follow a @ symbol on the command line are expanded in place. The command line is then tokenized, with the following classes of tokens identified:

- Variables: Any environment variable is identified by the trailing equal sign and uses a `Variable=Value` syntax. S-PLUS recognizes certain variables (Table 19.1), and user written routines may also query and react to these.
- Switches: A number of predefined switches (listed below) may be specified on the command line. They use a `/Switch [Value1 [Value2 [Value3 [...]]]` syntax and are parsed based on the leading / or - symbol.
- Script Files: Remaining tokens are considered script files to be run by the S-PLUS interpreter.

An example command line might be:

```
SPLUS. EXE ScriptFile1 /REGKEY SPLUS1 S_PROJ=c:\Project1
```

Most options that may be set on the command line are for advanced users. Some more generally useful options are the BATCH switch, and Script file processing, and for intermediate users, S\_TMP, S\_FIRST, and the ability to set up S-PLUS to run with different project directories, using the S\_PROJ variable. See the following section for more information about using multiple projects.

## COMMAND LINE PARSING

The operating system passes the command line unaltered to S-PLUS.

### File Expansion

S-PLUS expands files specified in the command line. Anything between a '@' character and the first subsequent delimiter (@ sign, double quote, single quote, or the standard delimiters: space, tab, newline, linefeed) is considered a filename and the entire file will be expanded in place.

The @ token can be escaped by placing a backslash before it, for example, in "EnvVar1=EnvValueWith@Filename" the @ sign will be active, and in "EnvVar2=EnvValueWith\@NoFilename" it will be ignored. The escape character is removed during this stage.

Multiple file names in the command line are fine, as are further filenames embedded within a file. Files specified with a relative path are sought first relative to the current working directory, and second in the same directory as **splus.exe**.

There is no way to specify a filename with spaces in it, nor to avoid a trailing delimiter after the filename, nor to avoid a trailing delimiter after the expanded file contents.

We recommend using file expansion to work around length limitations to the command line:

- Windows 3.x and Windows 95 have 127 character command line length restrictions, unless modifications similar to the following are made to **config.sys**:

```
shell =c: \wi ndows\command.com /p /u: 250
```

- Windows NT currently limits command lines to 1028 characters.

### Tokenizing

The command line is then broken into tokens. Standard command line delimiters are space, tab, newline, and linefeed and any combination of these are ignored at the start of the command line, and between tokens.

If the first character of a token is a single or double quote then it will be matched with another of the same type of quote, and anything between will be considered part of the token, but nothing afterwards.



Otherwise, a token begins with any non-delimiter and goes to the first delimiter or equal sign (the only way to "escape" a delimiter or equal sign is to place the entire token in quotes).

**Variables**

If the token is followed by an equal sign, it is considered to be part of a variable-value pair. (This is true even if the token begins with a "-" or "/"). If a delimiter is found trailing the equal sign, the variable is assigned an empty string for the value. (This can be used to cancel or override environment variables in the process environment.) Variables are then assigned the specified value.

**Switches**

Any token (not followed by an equal sign) that has either "-" or "/" as its first character is considered a switch. Each switch takes a variable number of successive tokens. Switches are evaluated in a case-insensitive manner. Switches are separated from successive tokens by the normal delimiters. Unknown switches are ignored.

**Script Files**

Remaining tokens are then considered script files and their contents sent to the S-PLUS interpreter. (Also see the /BATCH switch for an alternative mechanism for automating S-PLUS sessions.)

**Variables**

The following is a list of the variables recognized by S-PLUS. You are *not* required to set them.

*Table 19.1: Variables.*

Variable	Description
HOME	Deprecated, replaced by the synonymous <b>S_PROJ</b> .
S_CMDFILE	Name of the file used to initialize, and optionally append to, the Commands History window.
S_CMDLINEBUF	Sets the maximum number of characters that can be entered to specify a command in the Commands window. By default, this is 32000.
S_CMDSAVE	Number of commands to save for Commands History recall.

*Table 19.1: Variables.*

<b>S_CWD</b>	Directs S-PLUS to set the current working directory to this directory at startup. Subsequent file I/O will be done relative to this directory.
<b>S_DATA</b>	A series of directory paths, which is searched for a suitable database 1 (which stores user S-PLUS functions and data sets).
<b>S_ENGINE_LOCATION</b>	File or directory where S-PLUS should look for an alternate engine. Typically set only after using the <b>LOAD</b> or <b>GLOAD</b> utilities.
<b>S_FIRST</b>	S-PLUS function evaluated at start-up. See section Setting S_FIRST (page 759).
<b>S_HOME</b>	Specifies the directory where S-PLUS is installed. By default, this is set to the parent directory of the program executable.
<b>S_NOAUDIT</b>	Tells S-PLUS not to write the audit file. Set by default.
<b>S_NO_RECLAIM</b>	Tells S-PLUS not to ask the operating system to reclaim memory at the top of S-PLUS's memory "heap" after every top-level expression.
<b>S_NOSYMLOOK</b>	Tells S-PLUS not to look for symbols in the executable when loading custom object code.
<b>S_PATH</b>	A list of directories used for the initial S-PLUS search path, set by default to the S-PLUS system databases.
<b>S_PREFS</b>	Directory to use for storing user preferences.
<b>S_PRINT_COMMAND</b>	Windows command to use for printing the Commands Window. By default, the command is " <b>Notepad /p</b> ".
<b>S_PROJ</b>	Sets default values for <b>S_CWD</b> , <b>S_DATA</b> , and <b>S_PREFS</b> . See the section Working With Projects (page 757).
<b>S_PS2_FORMAT</b>	Tells S-PLUS to put a CTRL-D at the end of any PostScript file it generates. By default, S-PLUS does not do this.
<b>S_SCRSAVE</b>	KB of Commands Window output to save for scrollbar.
<b>S_SILENT_STARTUP</b>	Disable printing of copyright/version messages.
<b>S_TMP</b>	Specifies the directory where S-PLUS creates temporary scratch files.
<b>S_WIN32SDLL</b>	Used to work around the problem of S-PLUS not waiting for a spawned 16-bit process to complete.

Table 19.1: Variables.

<b>S_WORK</b>	Deprecated; replaced by <b>S_DATA</b> , except for use by the <b>TRUNC_AUDIT</b> utility under Win32s (Windows 3.1x and Windows for Workgroups).
<b>SHOME</b>	Deprecated, replaced by the synonymous <b>S_HOME</b> , except for use by the <b>LOAD</b> and <b>COMPILE</b> utilities.

Many of the variables in this section take effect if you set them to any value, and do not take effect if you do not set them, so you may leave them unset without harm. For example, to set **S\_NOAUDIT** you can enter:

**SPLUS.EXE S\_NOAUDIT=X**

on the command line and S-PLUS will not write an audit file, because the variable **S\_NOAUDIT** has a value (any value); this is the default for that variable. If you want S-PLUS to begin writing the audit file again during your next S-PLUS session, unset **S\_NOAUDIT** on the command line. To unset a variable that has previously been set in some way, enter no value (or a space) after typing the equal sign:

**SPLUS.EXE S\_NOAUDIT=**

Now, **S\_NOAUDIT** is not set, and S-PLUS writes the audit file.

Variables are currently implemented using environment variables. As a consequence, advanced users can specify these variables by altering their system or process environment variables using standard operating system specific techniques (for example, via the Control Panel's System applet, or for Win32s users, modifications to the **autoexec.bat** file). Win32s users may alternatively modify the [Environment] section of the **splus.ini** file. Variables specified on the command line are placed in the process environment at start-up and hence take precedence over any previously defined for the process.

User code can check the current values for these variables by using `getenv` from C or S code.

**Note**

In S-PLUS 3.3, we recommended placing variables in the SPLUS.INI file. Now we recommend placing them on the command line. If you wish to have multiple shortcuts use some of the same variables or switches, we recommend you place those common settings in a file and place the file name on the command line preceded with the `@` sign. For specifics see the File Expansion section above.

**S\_CMDFILE=filePath**

By using the **S\_CMDFILE** variable you can initialize the Commands History window to contain the commands found in a named text file. Optionally, the commands from your current session are appended to this file. Below are several examples illustrating the use of the **S\_CMDFILE** variable. These lines would be placed on the S-PLUS start-up command line.

```
S_CMDFILE=d:\splus\cmdhist.q  
S_CMDFILE=d:\splus\cmdhist.q+  
S_CMDFILE=+d:\splus\cmdhist.q  
S_CMDFILE=+history.q
```

In all cases, a path and filename are specified, and any commands found in the named file are placed in the Commands History window. In the first example, new commands from the current session will not be appended to the file. Placing a "+" immediately after the path and filename, as in the second example, causes the commands from the current session to be appended to the named file. Placing a "+" immediately before the path and filename, as in the third example, causes the commands from the current session to be appended to the named file and causes S-PLUS to create a new file if the named file does not exist. (The directory must already exist; only the file is created automatically.) In the final case, S-PLUS uses the file **history.q** in the start-up directory; it creates the file if it does not already exist. If you later change your start-up directory, another **history.q** will be created in that directory. You can also use the auditing facility in S-PLUS to automatically save your commands history. For this to work, you must turn on auditing and set the **S\_CMDFILE** variable to **\_AUDIT** by placing the following on the S-PLUS start-up command line:

```
SPLUS.EXE S_NOAUDIT= S_CMDFILE=+_AUDIT
```

You need the "+" to avoid an error message when you start up in a new working directory. When you use auditing, S-PLUS saves more than commands in the **\_audit** file. However, the Commands History window will show you only the S-PLUS commands when you use the auditing facility for your commands history. The **\_audit** file used by the auditing facility is found in the data directory. The Commands History window will look in that directory for **\_audit** when the variable **S\_CMDFILE** is set to **\_AUDIT**.

**S\_CMDLINEBUF=integer**

Set the **S\_CMDLINEBUF** variable to increase or decrease the maximum number of characters that can be entered for any one command in the Commands window. The default is 32000.

**S\_CMDSAVE=integer**

Set the **S\_CMDSAVE** variable if you want to limit the number of commands saved for command line/Commands History recall. For example, to limit the number of commands stored to the most recent 100, set this variable as follows:

**S\_CMDSAVE=100**

**S\_CWD=directoryPath**

Every operating system process, including S-PLUS, has a “Current Working Directory” that determines where file input/output occurs (or is relative to). The S-PLUS process is assigned a directory when it is started from a shortcut (or a program manager icon in earlier versions of Windows) or from a batch file or DOS prompt. Specifying the **S\_CWD** variable causes S-PLUS to ignore the default directory assigned by the parent process and use a specific one instead. Note that **S\_CWD** defaults to **S\_PROJ**.

**Note**

Previous versions of S-PLUS used the **S\_WORK** variable to refer to the “working directory”. To avoid confusion with the term “current working directory”, the terminology has changed and now we use the **S\_DATA** variable to refer to the “data directory”.

**S\_DATA=directoryPath[;directoryPath[...]]**

**S\_DATA** specifies a list of directories that is searched for a suitable database 1. Thus the first valid directory in the list is used by S-PLUS to store user data and functions. It traditionally is named **\_Data**.

**S\_DATA** defaults to **\_Data;%S\_PROJ%\\_Data**, so S-PLUS will seek a **\_Data** directory under the current working directory (see **S\_CWD**), and otherwise seek a **\_Data** directory under the project directory. If that then fails, a dialog will ask the user for the directory path. **S\_DATA** replaces **S\_WORK** that was used in previous versions of S-PLUS.

**S\_ENGINE\_LOCATION=directoryOrfilePath[:directoryOrfilePath[...]]**

**S\_ENGINE\_LOCATION** specifies a list of directories or files where S-PLUS should look for an alternate engine. This is typically set only after using the **LOAD** or **GLOAD** utility. Normally, S-PLUS uses as its engine the first file it finds with the name **GSQPE.DLL**, **NSQPE.DLL**, or **SQPE.DLL**, in that order, in the S-PLUS executable directory, the current working directory, or the directories listed in the **PATH** environment variable. By default, only **SQPE.DLL** exists, in the S-PLUS executable directory. Use **S\_ENGINE\_LOCATION** to change the default searching behavior; use the prefix “dir:” to indicate a directory in which to search.

For example, to force S-PLUS to use a particular file as its engine, give the full path to the file as the complete value for **S\_ENGINE\_LOCATION**:

**S\_ENGINE\_LOCATION=c:\myproj\myloaded.dll**

To have S-PLUS search only for engine files having the name **SQPE.DLL** (suppressing the search for **GSQPE.DLL** and **NSQPE.DLL**), use the following setting:

**S\_ENGINE\_LOCATION=sqpe.dll**

In this case, S-PLUS still searches for **sqpe.dll** in the standard places (the S-PLUS executables directory, the current working directory, and the directories listed in the **PATH** environment variable, in that order), but doesn't look for either **GSQPE.DLL** or **NSQPE.DLL**.

To have S-PLUS ignore the **PATH** environment variable in the search, and reverse the order it searches the current working directory and the S-PLUS executables directory, use the following setting:

**S\_ENGINE\_LOCATION=dir.:dir:c:\spwin45\cmd**

**S\_FIRST=function**

**S\_FIRST** specifies a function that will be executed upon start-up, immediately after S-PLUS finishes its initialization. It can be used to execute routine tasks related to setting up your work environment, for department wide functions or other initialization. If set, it overrides `.First`. See the section Customizing Your Session at Start-up and Closing (page 759) for specifics.

## **S\_HOME=directoryPath**

S\_HOME refers to the directory where S-PLUS is installed, which contains the S-PLUS application files. S-PLUS libraries, data sets, and other related files are stored in subdirectories under the top-level directory specified by S\_HOME. S-PLUS determines the location of the S-PLUS installation by referring to the parent directory where the executable is stored. You should never need to change S\_HOME in S-PLUS but expert users can explicitly define it if they move their S-PLUS files to another directory.

### **Note**

S\_HOME has replaced the SHOME variable to avoid conflicts with earlier versions of S-PLUS. Internally SHOME remains a synonym for S\_HOME for compatibility with previous versions.

## **S\_NOAUDIT=[value]**

If you set this variable (to any value), S-PLUS does not write an audit file. This is useful if you do not need a record of the commands you've typed into S-PLUS. (The default is to not write an audit file.) If this variable is not set, S-PLUS maintains a record of your S-PLUS commands (and other information) in a file called **\_audit** in your data directory. The audit file accumulates the commands from each S-PLUS session, so it may naturally grow large. The following setting causes S-PLUS not to maintain this record:

### **S\_NOAUDIT=YES**

If **S\_NOAUDIT** is set to any value, the **\_audit** file will not be opened or written into.

If you keep an audit file, it can grow very large. To reduce the size of the audit file, use the **/TRUNC\_AUDIT** command line switch. See Page 756 for details.

## **S\_NO\_RECLAIM=[value]**

By default, at the end of each top-level expression, S-PLUS asks the operating system (Windows 95 or Windows NT) to reclaim memory at the top of S-PLUS's memory heap. This request normally results in much lower S-PLUS memory usage at the cost of a very slight increase in processing time. In very rare circumstances, this memory reclamation may not be desired. To turn off memory reclamation, set **S\_NO\_RECLAIM** (to any value).

**S\_NOSYMLOOK=[value]**

This variable is only useful for developers writing custom libraries to link into S-PLUS. If you set this variable (to any value), S-PLUS does not do an extra search through the executable file when a symbol is not found by a search through an internal list of symbols and addresses. Normally, S-PLUS does the extra search through the executable file so the S-PLUS language can access (through the `.C`, `.Fortran`, and `.S` functions) any symbol known internally, whether the symbol was originally intended to be accessible to the S-PLUS language or not. Setting this variable restricts the S-PLUS language to symbols that were originally intended to be accessible.

**S\_PATH=directoryPath[;directoryPath[...]]**

Set the **S\_PATH** variable only if you want to override the standard S-PLUS search list every time you start S-PLUS. By default, **S\_PATH** is set by S-PLUS. It includes the built-in S-PLUS functions and data sets. You can display it using the `S` command `search`. As with other variables and their values, enclose the value with matched quotes if the directory paths include spaces.

**S\_PREFS=directoryPath**

Directory to use for storing user preferences. It defaults to **%S\_PROJ%\\_Prefs**. See the section *Working With Projects* (page 757).

**S\_PRINT\_COMMAND=WindowsCommand**

Windows command to use for printing the Commands window. By default, the command is **"Notepad /p"**.

**S\_PROJ=directoryPath**

**S\_PROJ** sets the defaults for **S\_PREFS**, **S\_DATA**, and **S\_CWD**. You may decide to change **S\_PROJ** if you wish to move your S-PLUS data files to another directory, or if you begin a new project. For example, if your name is Jay, you might create a home directory for your personal use. First, create the directory **C:\JAY**. Then set **S\_PROJ** to **C:\JAY** by setting it as a command line variable:



---

**SPLUS.EXE S\_PROJ=C:\JAY****Note**

**S\_PROJ** has replaced the **HOME** variable used in previous versions of S-PLUS. Internally **HOME** remains a synonym for **S\_PROJ** for compatibility with previous versions.

**S\_PS2\_FORMAT=[value]**

If you set the **S\_PS2\_FORMAT** variable, to any value, S-PLUS puts a CTRL-D character at the end of any PostScript file it generates. This is for compatibility with older PostScript formats. By default, S-PLUS does not put the CTRL-D character in the file.

**S\_SCRSAVE=integer**

Set the **S\_SCRSAVE** variable if you want to limit the amount of output saved for scrollbar in the Commands window. For example, to limit the number of characters saved to a maximum of 100KB, set this variable as follows:

**S\_SCRSAVE=100**

**Note**

This variable also imposes a limit on the number of commands available for recall.

**S\_SILENT\_STARTUP=value**

If you set the **S\_SILENT\_STARTUP** variable (to any value), the location of **S\_DATA** is not displayed when the Commands window is opened. The S-PLUS Commands window appears with a prompt, and nothing else, when created.

**S\_TMP=directoryPath**

Set the **S\_TMP** variable to the name of the directory where you want S-PLUS to create temporary scratch files. By default **S\_TMP** is unset, so temporary files are created in **S\_CWD**, the process current working directory.

If the directory specified by **S\_TMP** does not exist or cannot be accessed, it will be ignored. If you want S-PLUS to create temporary scratch files in the **C:\TEMP** directory, first create the directory **C:\TEMP**. Then, set **S\_TMP** to **C:\TEMP**:

**SPLUS.EXE S\_TMP=C:\TEMP**

**S\_WIN32SDLL=UTThunkDllPath**

Set the **S\_WIN32SDLL** to the path of the Universal Thunk DLL when S-PLUS will not wait for a spawned 16-bit process to complete. See Chapter 15, The Windows and DOS Interfaces, for more details.

## Switches

### Note

Unlike variables, switches do not use equal signs between the switch and any necessary values. If you need to include an equals sign, use quotes around the entire token.

**/BATCH stdin [stdout [stderr]]**

**BATCH** may be followed with one, two or three file names indicating where stdin, stdout, and stderr, respectively, should be redirected. Specify 'stdin', 'stdout', or 'stderr' to maintain the default input and output processing for any of these values. Stdin is typically the name of a text file containing valid S-PLUS commands.

When S-PLUS is run in batch mode, a small dialog appears to notify the user that it is running rather than the normal S-PLUS window. Once completed running a **BATCH** session, S-PLUS automatically terminates. Use a script file for running S-PLUS commands automatically without automatically terminating S-PLUS when done, although that does not allow one to redirect stdin, stdout, or stderr.

**/BATCH\_PROMPTS**

**BATCH\_PROMPTS** specifies whether any progress, non-fatal warning or error, and/or exit status dialog should be displayed. By default, whenever S-PLUS runs in batch mode, it displays a "Pre-processing" and then a "thermometer" dialog to indicate the batch session's progress. When batch mode is completed, the progress dialogs disappear, but no completion dialog

is displayed to indicate the batch session's success or failure. Also, if a non-fatal warning or error occurs in interactive mode, that dialog is suppressed and its message (and default response) is written to the batch output file. Use the **/BATCH\_PROMPTS** switch to change this default behavior. The value "Yes" turns on all dialogs, "No" turns off all dialogs. To apply these values to a particular dialog, use the prefixes "progress:"; "non-fatal:"; and "exitstatus:". For example, to display the exit status dialog but suppress the progress dialogs, use the following switch:

**/BATCH\_PROMPTS progress:no,exitstatus:yes**

When specifying the prefix/word pairs, use either "," or ";", but no spaces, to separate them.

If an error occurs in one of the batch files while S-PLUS is in batch mode, a dialog is displayed indicating the error, unless both progress and exitstatus dialogs are suppressed.

## **/COMMANDS\_WINDOW\_ONLY**

Starts S-PLUS with only the Commands window displayed, overriding any "Open at Startup" preferences.

### **Note**

The next four switches below provide a convenient mechanism for developers to run the batch file of the same name (**/COMPILE** runs **COMPILE.BAT** and so on). All values that follow the command line switch are passed directly to the batch file, once the **\$HOME** environment variable is set in the spawned process.

**/COMPILE values**     Compiles C or Fortran code with release (no debug) options.

**/GCOMPILE values**     Compiles C or Fortran code with debug options.

**/GLOAD values**         Creates a new S-PLUS engine with debug link options.

**/LOAD values**           Creates a new S-PLUS engine with release (no debug) link options.

## **/MULTIPLEINSTANCES**

Specifies that multiple instances are allowed.

**/Q**                         Causes S-PLUS to automatically quit after any script file processing. This is automatically set when the **/BATCH** switch is specified.

**/REGISTEROLEOBJECTS**

Registers S-PLUS's Automation components in the registry. This permits other programs to access S-PLUS functions programmatically via ActiveX Automation (previously known as OLE Automation). See Chapter 13, Automation, for more information.

**/REGKEY KeyName**

Specify alternative registry key under:  
**HKEY\_CURRENT\_USER\Software\Mathsoft.**

This is useful for expert users who wish to maintain multiple versions of S-PLUS on one system. KeyName defaults to S-PLUS.

**/TRUNC\_AUDIT integer**

Spawns the **TRUNC\_AUDIT** program that truncates the **\_audit** file (found in the data directory), removing the oldest part of the file. The integer argument specifies the maximum size (in characters) to keep in the audit file and defaults to 100,000; **TRUNC\_AUDIT** only removes entire commands, not partial commands.

Under Win32s, the use of **TRUNC\_AUDIT** requires that the environment variable **S\_WORK** be set in your global environment to the directory in which your **\_audit** file is located.

**/UNREGISTEROLEOBJECTS**

Removes the Automation components (installed by **/REGISTEROLEOBJECTS**) from the registry.

**/UTILITY ProgramName values**

Runs the specified ProgramName, passing it all values that follow, once the **\$HOME** environment variable is set in the spawned process.

**Script Files**

See the *User's Guide*, Chapter 8, for information about using script files.

---

## WORKING WITH PROJECTS

This section illustrates ways intermediate users can use the various switches discussed above to customize S-PLUS on a project by project basis.

S-PLUS now makes it quite easy to maintain distinct data and preferences directories for each of your projects. For each project:

1. Make a copy of the S-PLUS desktop shortcut (or program item in earlier versions of Windows) by holding the CTRL key down as you drag a copy of the shortcut to its new location,
2. Edit the command line (right-click, or, for older versions of Windows, select it and press Alt-Enter) to set S\_PROJ:

`SPLUS. EXE S_PROJ=c: \Banana\Survey`

When you use the new shortcut, S-PLUS will use c:\Banana\Survey\\_Data for the project data and c:\Banana\Survey\\_Prefs for the project preferences.

There are many ways to customize this to fit your particular needs. For instance you may want to use various \_Data directories and a common \_Prefs directory.

S\_PROJ is used to set the default values for  
S\_CWD (to %S\_PROJ%), and  
S\_PREFS (to %S\_PROJ%\\_Prefs), and  
S\_DATA (to %S\_CWD%\\_Data;%S\_PROJ%\\_Data).

### The Preferences Directory

S-PLUS is more customizable than any previous version of S-PLUS. These user preferences are all stored in a variety of files in a directory with the name \_Prefs.

Note
The only exception to this is that the Options object is stored in the S_DATA directory for backward compatibility reasons.

Upon creation of a new \_Prefs directory, it is populated with “template” preference files from the %S\_HOME%\CMD directory, so expert users who wish to permanently set a particular preference may edit the templates. You should make a copy of the original templates for future reference.

# The Data Directory

Whenever you assign the results of an S-PLUS expression to an object, using the `<-` operator within an S-PLUS session, S-PLUS creates the named object in your data directory. The data directory occupies position 1 in your S-PLUS search list, so it is also the first place S-PLUS looks for an S-PLUS object. You specify the data directory with the variable `S_DATA`, which can specify one directory or a colon-separated list of directories. The first valid directory in the list is used as the data directory, and the others are placed behind it in the search list.

Like other variables, `S_DATA` cannot be changed during an S-PLUS session. To change the data directory during a session, use the `attach` function with the optional argument `pos=1`, as in the following example that specifies `MYSPLUS\_FUNCS` as the data directory:

```
attach("C:\MYSPLUS\FUNCS", pos=1)
```

If `S_DATA` is not set, S-PLUS sets the data directory, to one of two directories according to the following rules:

1. If a subdirectory named `_Data` exists in `S_CWD`, the current working directory, S-PLUS sets the data directory to this `_Data` subdirectory.
2. Otherwise S-PLUS checks to see if the `%S_PROJ%\_Data` directory exists. If so, it will be used as the default location for objects created in S-PLUS. If not, it will be created.

## Note

Although `S_DATA` may be used to provide alternative directory names (other than `_Data`), in practice some code depends on this being set to `_Data`. Therefore it is recommended that `S_DATA` be used primarily to set the path to a particular directory named `_Data`, and not to change the name of the directory itself.

## CUSTOMIZING YOUR SESSION AT START-UP AND CLOSING

If you routinely set one or more options each time you start S-PLUS, you can store these options and have S-PLUS set them automatically whenever it starts. You can store the options by doing one of the following:

- Set the S-PLUS command line variable `S_FIRST` as described below, or
- Create an S-PLUS function named `.First` containing the desired options.

When S-PLUS starts up, it checks whether the `S_FIRST` variable exists. If not, S-PLUS runs the `.First` function, if the function exists, from your data directory. If `S_FIRST` is set, S-PLUS ignores the `.First` function. If S-PLUS encounters any errors in your `.First` function, it starts without executing it.

### Setting `S_FIRST`

To store a sequence of commands in the `S_FIRST` variable, use the following syntax:

```
SPLUS. EXE S_FIRST=S-PLUS expression
```

For example, the following command tells S-PLUS to start the default graphics device:

```
SPLUS. EXE S_FIRST=graphicsheet()
```

To avoid misinterpretation by the command line parser, it is safest to surround complex S-PLUS expressions with a single or double quote (whichever you do *not* use in your S-PLUS expression). For example, the following command starts S-PLUS and modifies several options:

```
SPLUS. EXE S_FIRST='options(digits=4); options(expressions=128)'
```

Due to operating system specific line length limitations, or for ease of use, you can also place the commands in a file and put the filename on the command line, preceded by the `@` symbol:

```
Echo options(digits=4); options(expressions=128) >  
c:\mylnitialization.txt
```

```
SPLUS. EXE S_FIRST=@c:\mylnitialization.txt
```

You can also combine several commands into a single S-PLUS function, then set `S_FIRST` to this function. For example:

```
> startup <- function() { options(digits=4)
+ options(expressions=128)}
```

You can call this function each time you start S-PLUS by setting `S_FIRST` as follows:

```
SPLUS.EXE S_FIRST=startup()
```

Variables cannot be set while S-PLUS is running, just at initialization. Any changes to `S_FIRST` will only take effect upon restarting S-PLUS.

### Creating the .First Function

Here is a sample `.First` file that starts the default graphics device:

```
> .First <- function() graphicsheet()
```

After creating a `.First` function, you should always test it immediately to make sure it works. Otherwise S-PLUS will not execute it in subsequent sessions.

### Customizing Your Session at Closing

When S-PLUS quits, it looks in your data directory for a function called `.Last`. If `.Last` exists, S-PLUS runs it. A `.Last` function can be useful for cleaning up your directory by removing temporary objects or files.

Example:

```
.Last <- function () dos(paste("del ", getenv("S_Tmp"),
+ "/*.*.Tmp, dep+"), Trans=T)
```



# ENHANCING S-PLUS

With the instructions in this section, you can:

- Add functions, or modify system functions to change default values or use different algorithms.
- Load object code with the `.C` and `.Fortran` functions.

## Note

Keep a careful log of how you modify S-PLUS, so that you can restore your changes when you receive the next update.

## Adding Functions and Data Sets to Your System

You may need to add or modify S-PLUS functions. This section describes how to add or modify functions.

1. Start S-PLUS.
2. Create a version of the function or data set you want to add or modify with a command such as the one below, where *my.function* is the name of the function or data set you want to modify:

```
> fix(my.function)
```

3. Run *my.function* to make sure that it works properly (you don't want to install a bad version).
4. Create a directory for your new and modified functions:

```
new.database ("modfuncs")
```

where *modfuncs* is the name of your directory.

5. Use the `search` command to see a list of available S-PLUS system directories:

```
> search()
```

The position of a directory in this list is called its *index*. Each row in the list displays the index of the first directory in the row.

6. Create a function `.First.local` in the directory in search position 2, as follows:

```
> assign(" .First.local ", function()
+ attach(modfuncs, pos = 2),
+ where=2)
```

Each time you start S-PLUS, S-PLUS executes `.First.local` if it exists. This `.First.local` attaches your new and modified functions directory ahead of all built-in S-PLUS functions, but behind your data.

7. Attach your *modfuncs* directory to your current session:

```
> attach("modfuncs", pos=2)
```

8. Assign your new or modified function to its permanent home:

```
> assign("my.function", my.function, where=2)
```

### Warning

Be careful when you modify system functions, because you may have to repeat the installation procedure if you make a mistake. You should keep a careful change log, both to guide your own troubleshooting and to assist support staff in solving any problems you report.

## Adding Object Code

If you have C or Fortran code that you want to access from within S-PLUS, you can create a Dynamic Link Library (DLL) with most 32 bit compilers and dynamically link to your code from S-PLUS.

Alternatively, in some cases, it may be necessary to link your C and Fortran object code into a core S-PLUS DLL. This is done with the S-PLUS COMPILE and LOAD utilities, although this requires the same compiler used to produce S-PLUS:

- WATCOM C/C++/32 bit compiler.
- If loading Fortran code, WATCOM F77 32 bit compiler.

Please see the Release Notes for Watcom version compatibility and ordering information.

# COMPUTING ON THE LANGUAGE

# 20

---

<b>Introduction</b>	<b>764</b>
<b>Symbolic Computations</b>	<b>766</b>
<b>Making Labels From Your Expressions</b>	<b>768</b>
<b>Creating File Names and Object Names</b>	<b>770</b>
<b>Building Expressions and Function Calls</b>	<b>771</b>
Building Unevaluated Expressions	771
Manipulating Function Definitions	772
Building Function Calls	776
<b>Argument Matching and Recovering Actual Arguments</b>	<b>780</b>
Interpreting Complicated Argument Lists	782

## INTRODUCTION

One of the most powerful aspects of the S-PLUS language is the ability to reuse intermediate expressions at will. The simplest example is the ability to use arbitrary S-PLUS expressions as arguments to functions. While evaluating an expression, S-PLUS stores the entire expression, including any function calls, for further use. The stored expressions and function calls can be retrieved and manipulated using a wide array of functions. The key to such manipulation, which is called *computing on the language*, is that each step of any S-PLUS calculation results in a new S-PLUS object, and objects can always be manipulated in S-PLUS. Chapter 21, Data Management, discusses several uses of these techniques.

Computing on the language is useful for a number of tasks, including the following:

- Symbolic computations.
- Creating labels and titles for a graphics plot, using part or all of the expression used to create the plot.
- Capturing function calls for labeling return values.
- Building expressions and function calls within S-PLUS functions.
- Debugging S-PLUS functions.
- Intercepting user input, a technique that can be useful when building custom user interfaces.

This chapter discusses the first four of these tasks. Computing on the language for debugging purposes is also described in the Chapter 6, Debugging Your Functions.

Most of these tasks involve some variant of the following basic technique

1. Create an unevaluated expression (with `substitute`, `expression`, or `parse`).
2. Perform some intermediate computations.

3. Generate finished output, using either or both of the following methods:
  - a. Deparsing the unevaluated expression (with `deparse`).
  - b. Evaluating the created expression (with `eval` ).

## SYMBOLIC COMPUTATIONS

Symbolic computations involve manipulating *formulas* of *symbols* representing numeric quantities, without explicitly evaluating the results numerically. Such computations arise frequently in mathematics:

$$\frac{d}{dx} \sin(x) = \cos(x) \quad (20.1)$$

$$\int (x^2 + 3x + 4) dx = \frac{x^3}{3} + \frac{3x^2}{2} + 4x + C \quad (20.2)$$

To perform symbolic computations in S-PLUS, you must interrupt S-PLUS's usual pattern of evaluation to capture *unevaluated* expressions to represent formulas, then perform the desired manipulations and return the result as an S-PLUS expression. The returned expression can then, in general, be evaluated just as any S-PLUS expression would be.

The key to capturing unevaluated expressions, and thus to symbolic computations in general, is the `substitute` function. In its most common use, you call `substitute` from inside a function, giving the formal name of one of the function's arguments as the argument to `substitute`. S-PLUS returns the actual argument corresponding to that formal name in the current function call. For example, S-PLUS has a function, `D`, that takes an S-PLUS expression and returns a symbolic form for the expression's derivative. The form required, by default, is rather arcane:

```
> D(expression(3*x^2), "x")
3 * (2 * x)
```

The following “wrapper” function allows you to find derivatives in a much more natural way:

```
my.deriv <-  
function(mathfunc, var) {  
  temp <- substitute(mathfunc)  
  name <- deparse(substitute(var))  
  D(temp, name)  
}
```

**For example:**

```
> my.deriv(3*x^2, x)
```

```
3 * (2 * x)
```

```
> my.deriv(4*z^3 + 5*z^(1/2), z)
```

```
4 * (3 * z^2) + 5 * (z^((1/2) - 1) * (1/2))
```

## MAKING LABELS FROM YOUR EXPRESSIONS

When you plot a data set with `plot`, the labels for the  $y$ -axis and possibly the  $x$ -axis are drawn directly from what you type: if you type `plot(corn.rain, corn.yield)`, the resulting plot has `corn.rain` as the  $x$ -axis label and `corn.yield` as the  $y$ -axis label. And it is not simply names that appear this way; if you type `plot((1:10)^2, log(1:10))`, you get  $x$ - and  $y$ -axis labels of  $(1:10)^2$  and  $\log(1:10)$ , respectively. S-PLUS is using the *unevaluated* form of these expressions, in the form of character strings.

In general, these character strings result from `substitute` (which returns the unevaluated actual argument) coupled with `deparse`, which *de-parses* the expression and returns a character vector containing your originally typed string. For example, here is a function that plots mathematical functions:

```
mathplot2 <-
function(f, bottom = -5, top = 5)
{
  fexpr <- substitute(f)
  ylabel <- deparse(fexpr)
  x <- seq(bottom, top, length = 100)
  y <- eval(fexpr)
  plot(x, y, axes = F, type = "l",
       ylab = paste( "f(x) =", ylabel ))
  axis(1, pos = 0, las = 0, tck = 0.02)
  axis(2, pos = 0, las = 2)
}
```

Whenever you call `mathplot2`, the actual argument you type as `f` is stored as `fexpr`. Thus, in the following call:

```
> mathplot2(sin(x)^2
```

the expression `sin(x)^2` is stored in the temporary variable `fexpr`.

For creating labels, simply using the unevaluated expression is not enough, because in the course of creating the label S-PLUS will evaluate the parsed expression. To protect the expression from evaluation, it must be *deparsed*, that is, converted to a character string corresponding to the unevaluated expression. Thus, the  $y$ -axis labels in `mathplot` are created by deparsing the expression previously stored in `fexpr`.



In `mathplot2`, `fexpr` was needed in at least two places, but in most simple applications, you need not store the substituted expression before deparsing. For example, in the `ulorigin` function described in the online help, we pointed out that the labels could be improved. Here is how to do so:

```
ulorigin2 <-
function(x, y, ...)
{
  labx <- deparse(substitute(x))
  laby <- deparse(substitute(y))
  plot(x, -y, axes = F, xlab = labx, ylab = laby, ...)
  axis(3)
  yaxp <- par("yaxp")
  ticks <- seq(yaxp[1], yaxp[2], length = yaxp[3])
  axis(2, at = ticks, labels = -ticks, srt = 90)
  box()
}
```

Many functions, including `plot`, use the `deparse(substitute(x))` construction to create default labels.

Sometimes you want to title a plot using the complete expression that generated the plot. This is also very easy, using the `sys.call` function. Here is a modified version of `mathplot` that prints the function call as the plot's main title:

```
mathplot3 <-
function(f, bottom = -5, top = 5)
{
  fexpr <- substitute(f)
  ylabel <- deparse(fexpr)
  x <- seq(bottom, top, length=1000)
  y <- eval(fexpr)
  plot(x, y, axes = F, type = "l",
       ylab = paste("f(x) =",
                    ylabel), main=deparse(sys.call()))
  axis(1, pos = 0, las = 0, tck = 0.02)
  axis(2, pos = 0, las = 2)
}
```

## CREATING FILE NAMES AND OBJECT NAMES

Another use of the `deparse(substitute(x))` syntax is in the following simplified version `fix` function, a simple but useful wrapper for the `vi` function. The `fix` function eliminates the need to explicitly assign the output of `vi` back to a function you are trying to alter (if you've ever typed `vi(my.func)` and then watched several hours' worth of fixes scroll by on your screen, you'll see the usefulness of `fix`):

```
my.fix <-
function(fcn, where = 1)
{
    deparse(substitute(fcn)), vi(fcn, where = where)
}
```

Often, you will create a useful function, like `my.fix`, that you want to make available to all of your S-PLUS sessions. If you have many different `.Data` directories, it makes sense to place all of your utility functions in a single directory (better yet, create a *library* for these functions) and attach this directory whenever you start S-PLUS. The following function makes it easy to move functions (or other objects) between directories. Here the default destination for the moved object is a directory of objects labeled `".Utilities"`:

```
move <-
function(object, from = 1, to = ".Utilities")
{
    obj name <- deparse(substitute(object))
    assign(obj name, get(obj name, where = from),
           where = to)
    remove(obj name, where = from)
}
```

# BUILDING EXPRESSIONS AND FUNCTION CALLS

Expressions, function definitions, and function calls are all recursive S-PLUS objects, like lists, and thus can be manipulated in exactly the same way you manipulate lists. A common programming technique is to build unevaluated expressions or function calls as you would build a list, then use `eval` to evaluate the expression or call. The following subsections give an introduction to this technique.

## Building Unevaluated Expressions

The `substitute` function, as we have seen, is one way of building an unevaluated expression. The `expression` function is another. You can use `expression` as a way to protect input from evaluation:

```
> fexpr <- expression(3 * corn.ra.in)
> fexpr

expression(3 * corn.ra.in)

> eval(fexpr)

1890: 28.8 38.7 29.7 26.1 20.4 37.5 39.0 30.3 30.3 30.3
1900: 32.4 23.4 48.6 42.3 31.8 30.0 34.5 40.8 36.3 36.0
1910: 27.9 23.1 33.0 20.7 28.5 49.5 27.9 28.2 26.1 28.5
1920: 34.8 36.3 24.0 32.1 41.7 33.9 34.8 31.2
```

The `expression` function can be useful in building user interfaces, such as menus. For example, here is a function that generates a random number from a user-selected distribution:

```
RandomNumber <-
function()
{
  rand.choi ce <- expression(Gaussian = rnorm(1),
                             Uniform = runif(1), Exponential = rexp(1),
                             Cauchy = rcauchy(1))
  pi ck <- menu(names(rand.choi ce))
  i f(pi ck)
    eval(rand.choi ce[pi ck])
}
```

The `expression` function returns an object of mode `expression`. Such objects are recursive, like lists, and can be manipulated just like lists. The `RandomNumber` function creates the expression object `rand.choice`, with the named components `Gaussian`, `Uniform`, `Exponential`, and `Cauchy`. The `menu` function uses these names to provide a choice of options to the user, and the user's selection is stored as `pick`. If `pick` is nonzero, the selected component of `rand.choice` is evaluated.

A similar approach can be used to give a user a choice of graphical views of a data set:

```
Visualize <-
function(x)
{
  view <- expression(Scatterplot = plot(x),
                     Histogram = hist(x), Density = plot(density(x),
width = 2 * (summary(x)[5] - summary(x)[2])), xlab = "x",
ylab = "", type = "l"), QQplot = qqnorm(x); qqline(x) ) )
  repeat
  {
    pick <- menu(names(view))
    if(pick) eval (view[pick])
    else break
  }
}
```

## Manipulating Function Definitions

Function definitions are also recursive objects and like expression objects they can be manipulated just like lists. A function definition is essentially a list with one component corresponding to each formal argument and one component representing the body of the function. Thus, for example, you can see the formal names of arguments to any function using the `names` function:

```
> names(hist)

[1] "x"           "nclass" "breaks" "plot"
[5] "probability" "... "    "xlab"    ""
```

The empty string at the end of the return value corresponds to the function body; if you are writing a function to return only the argument names, you can use a subscript to omit this element:

```
argnames <- function(funcname)
{
  names(funcname)[ - length(funcname)]
}
```

Thus, for example,

```
> argnames(hist)

[1] "x"           "nclass"  "breaks"  "plot"
[5] "probability" "... "     "xlab"
```

You can use the list-like structure of the function definition to *replace* the body of a function with another function body that uses the same argument list. For example, when debugging your functions, you may want to trace their evaluation with the `browser` function, or some other tracing function. The `trace` function creates a copy of the traced function with the body modified to include a call to the tracing function. (See Chapter 6, Debugging Your Functions, for more information on the `trace` function.) For example, if we trace the `argnames` function (and specify `browser` as the tracing function), then look at the definition of `argnames`, we see the call to `browser` embedded:

```
> trace(argnames, browser)
> argnames
function(funcname)
{
  if(.Traceon)
  {
    .Internal(assign(".Traceon", F, where = 0),
               "S_put")
    cat("On entry: ")
    browser()
    .Internal(assign(".Traceon", T, where = 0),
               "S_put")
  }
  {
    names(funcname)[ - length(funcname)]
  }
}
```

Here is a simplified version of `trace`, called `smp.trace` that shows how the temporary version of the traced function is created. The material to be added to the body of the traced function is created as an expression. In our simplified version, we have hard-coded the call to `browser` as well as the message `cat("On entry: ")`. The subscript on the expression object indicates that only the first component is desired:

```

. . .
texpr <- expression( if(.Traceon)
{
  .Internal(assign(".Traceon", F, where = 0), "S_put")
  cat("On entry: ")
  browser()
  .Internal(assign(".Traceon", T, where = 0), "S_put") )
}
[[1]]
. . .

```

The actual substitution is performed as follows:

```

for(i in seq(along = what))
{
  name <- what[i]
  . . .
  fun <- get(name, mode = "function")
  n <- length(fun)
  body <- fun[[n]]
  e.expr <- expression( { NULL NULL } )
  [[1]]
  e.expr[[1]] <- texpr
  e.expr[[2]] <- body
  fun[[n]] <- e.expr
  assign(name, fun, where = 0)
}

```

The complete `smp.trace` function is shown below:

```

smp.trace <-
function(what = character())
{
  temp <- .Options
  temp$warn <- -1
  assign(".Options", temp, frame = 1)
  assign(".Traceon", F, where = 0)

```

```

    if(!is.character(what))
    {
        fun <- substitute(what)
        if(!is.name(fun))
            stop("what must be character or name" )
        what <- as.character(fun)
    }
    texpr <- expression(if(.Traceon)
        .Internal(assign(".Traceon", F, where = 0),
            "S_put")
        cat("On entry: ")
        browser()
        .Internal(assign(".Traceon", T, where = 0),
            "S_put")
    )
    )[[1]]
    tracefuns <- if(exists(".Tracelist"))
        get( ".Tracelist", where = 0)
    else
        character()
    for(i in seq(along = what))
    {
        name <- what[i]
        if(exists(name, where = 0))
        {
            remove(name, where = 0)
            if(!exists(name, mode = "function") )
                stop(paste( "no permanent definition
of", name))
        }
        fun <- get(name, mode = "function")
        n <- length(fun)
        body <- fun[[n]]
        e.expr <- expression({ NULL NULL })
        [[1]]
        e.expr[[1]] <- texpr
        e.expr[[2]] <- body
        fun[[n]] <- e.expr
        assign(name, fun, where = 0)
    }
    tracefuns <- unique(c(what, tracefuns))
    assign(".Tracelist", tracefuns, where = 0)
    assign(".Traceon", T, where = 0)
    invisible(what)
}

```

## Building Function Calls

A function call object is a recursive object for which the first component is a function name and the remaining components are the arguments to the function. You can create an unevaluated function call in many ways. We have seen one simple way: wrap an ordinary function call inside the `expression` function, and extract the first component:

```
> expression(hist(corn.rai n))[[1]]

hist(corn.rai n)
```

Analogous to the `expression` function, but specific to function calls, is the `call` function, which takes a character string giving the function name as its first argument, then accepts arbitrary arguments as arguments to the function:

```
> call("hist", corn.rai n)

hist(structure(.Data = c(9.6, 12.9, 9.9, 8.7, 6.8, 12.5,
 13, 10.1, 10.1, 10.1, 10.8, 7.8, 16.2, 14.1, 10.6, 10,
 11.5, 13.6, 12.1, 12, 9.3, 7.7, 11, 6.9, 9.5, 16.5, 9.3,
 9.4, 8.7, 9.5, 11.6, 12.1, 8, 10.7, 13.9, 11.3, 11.6,
 10.4), .Tsp = c(1890, 1927, 1)))
```

To prevent S-PLUS from reading the arguments into memory until evaluation, the idiom `as.name("argument")` can be useful:

```
> call("hist", as.name("corn.rai n"))

hist(corn.rai n)
```

A typical use of `call` is inside a function that offers the user a range of functionality and calls different functions depending upon the options specified by the user. For example, here is a version of the `ar` function that uses `call`:

```
my.ar <-
function(x, aic = T, order.max, method = "yul e-wal ker")
{
  if(!missing(order.max))
    arglist$order.max <- order.max
  imeth <- charmatch(method, c("yul e-wal ker", "burg"),
    nomatch = 0)
  method.name <- switch(imeth + 1,
    stop("method should be either yul e-wal ker or
burg" ),
    "ar.yw",
```



```

      "ar.burg")
    z <- call(method.name, x = x, aic = aic)
    ar <- eval(z, local = sys.parent(1))
    ar$series <- deparse(substitute(x))
    return(ar)
}

```

A more common idiom in S-PLUS programming, however, is to create an ordinary *list* of the appropriate form, then change the *mode* of the list to `call`. This idiom, in fact, is used by the actual `ar` function distributed with S-PLUS:

```

> ar
function(x, aic = T, order.max, method = "yule-walker")
{
  arglist <- list(x = x, aic = as.logical(aic))
  if(!missing(order.max))
    arglist$order.max <- order.max
  imeth <- charmatch(method, c("yule-walker", "burg"),
    nomatch = 0)
  method.name <- switch(imeth + 1,
    stop("method should be either yule-walker or
burg" ),
    as.name("ar.yw"),
    as.name("ar.burg"))
  z <- c(method.name, arglist)
  mode(z) <- "call"
  ar <- eval(z, local = sys.parent(1))
  ar$series <- deparse(substitute(x))
  return(ar)
}

```

The call list is created by combining the argument list `arglist` (created with the `list` function in the first line of the function body) and the name of the appropriate method. Since `arglist` is of mode `list`, the value of the `c` function is also of mode `list`. Changing the mode of an object can also be used to construct function calls as follows. Suppose you have a character string representing the name of a function. You can coerce the string to a name using `as.name`, then evaluate the function with a call of the following form:

```

> eval(function)(args)

```

Note the parenthesization; the function *name* is an argument to `eval`, but the argument list is an argument to the *function* to which the name evaluates. Thus, for example, we have the following:

```
> my.list <- as.name("list")
> eval(my.list)
function(...)
.Internal(list(...), "S_list", T, 1)
> eval(my.list)(stuff="stuff")

$stuff:
[1] "stuff"
```

The method can be used to revise `ar` again as follows:

```
my.ar2 <-
function(x, aic = T, order.max, method = "yul e-wal ker")
{
  arglist <- list(x = x, aic = as.logical(aic))
  if(!missing(order.max))
    arglist$order.max <- order.max
  imeth <- charmatch(method, c("yul e-wal ker", "burg"),
    nomatch = 0)
  method.name <- switch(imeth + 1,
    stop("method should be either yul e-wal ker or
burg" ),
    as.name("ar.yw"),
    as.name("ar.burg"))
  ar <- eval(method.name,
    local = sys.parent(1))(unlist(arglist))
  ar$series <- deparse(substitute(x))
  return(ar)
}
```

In the `ar` example, the unevaluated function call was needed so that `eval` could choose the appropriate *frame* for evaluation (see Chapter 21, Data Management, for more details). If you don't need to exercise such control over the evaluation, you can build the argument list of evaluated arguments, then construct and evaluate the function call using a single call to the `do.call` function. The `do.call` function is convenient when a function's arguments can be generated computationally. As a trivial example, consider the following function, which generates several graphical parameters randomly, then calls `do.call` to construct and evaluate a call to `plot`:

```

Sample.plot <-
function(x)
{
  cex <- sample(seq(0.1, 3, by = 0.1), 1)
  pch <- sample(1:20, 1)
  type <- sample(c("p", "l", "b", "o", "n", "s", "h"),
    1)
  main <- "A random plot"
  do.call("plot", list(x = x, cex = cex, pch = pch,
    type = type, main = main))
}

```

One other method of constructing function calls, used frequently in statistical modeling functions such as `lm`, uses the `match.call` function, which returns a call in which all the arguments are specified by name. Typically, the returned call is the call to the function that calls `match.call`, although you can specify any call. The `lm` function uses `match.call` to return the call to `lm`, then it changes the first component of the returned call (i.e., the function name) to create a new function call to `model.frame` with the same arguments as the original call to `lm`:

```

> lm
function(formula, data, weights, subset, na.action, method
= "qr", model = F, x = F, y = F, contrasts = NULL, ...)
{
  call <- match.call()
  m <- match.call(expand = F)
  m$method <- m$model <- m$x <- m$y <- m$contrasts <-
m$... <- NULL
  m[[1]] <- as.name("model.frame")
  m <- eval(m, sys.parent())
  ...
}

```

We discuss `match.call` further in the following section.

## ARGUMENT MATCHING AND RECOVERING ACTUAL ARGUMENTS

We have met two functions, `substitute` and `missing`, that take a formal argument of the calling function and return, respectively, the unevaluated actual argument and a logical value stating whether the argument is missing. If you want to manipulate *all* the arguments of the calling function, you can use two other functions, `amatch` and `match.call`.

The `amatch` function mimics the argument matching performed inside the evaluator and returns a list with one component for each formal argument. Each component is an S-PLUS object of the recursive mode argument. Such objects have length 2—the first element is the actual argument, the second is the formal. The `amatch` function requires two arguments—a function name and an unevaluated function call:

```
> amatch(hi.st, expressi.on(hi.st(corn.rai.n, hori.z=T)))
```

```
$x:
```

```
.Argument(corn.rai.n, x = )
```

```
$ncl.ass:
```

```
.Argument(, ncl.ass = )
```

```
$breaks:
```

```
.Argument(, breaks = )
```

```
$plot:
```

```
.Argument(, plot = TRUE)
```

```
$probability:
```

```
.Argument(, probability = FALSE)
```

```
$...:
```

```
.Argument((hori.z = T), ... = )
```

```
$xlab:
```

```
.Argument(, xlab = deparse(substitute(x)))
```

```
attr(,"missing"):
```

```
[1] F T T T T F T
```

In the above call, the argument `corn.rain` is returned as the actual argument to the formal name `x`, and the argument `hori z = T` is returned as the actual argument to the formal argument `<ms-program-code>...`. The missing status of each argument is stored in the `missing` attribute.

The `match.call` function, as we saw in the previous section, returns the call to the function calling `match.call` with all arguments named. For example, suppose we define the following simple function:

```
fcn.F <-
function(x, y, z)
{
  match.call()
}
```

Calling the function with arbitrary values for `x`, `y`, and `z` yields the following:

```
> fcn.F(7, 11, 13)
fcn.F(x = 7, y = 11, z = 13)
```

If a function has the `<ms-program-code>...` formal argument, an optional argument to `match.call` can be used to specify whether the `<ms-program-code>...` arguments are shown like the named arguments, separated by commas (the default), or are grouped together in a list:

```
> fcn.G
function(x, y, z, ...)
{
  match.call()
}
> fcn.G(7, 11, 13, "paris", "new york")
fcn.G(x = 7, y = 11, z = 13, "paris", "new york")
> fcn.H
function(x, y, z, ...)
{
  match.call(expand.dots = F)
}
> fcn.H(7, 11, 13, "paris", "new york")
fcn.H(x = 7, y = 11, z = 13, ... = list("paris", "new york"))
```

Both `amatch` and `match.call` return *unevaluated* arguments. In general, `match.call` is easier to use for routine manipulations, because it does not involve the `.`Argument structure. Subscripting is identical to that for any named list, as we saw in the `lm` example:

```
> lm
function(formula, data, weights, subset, na.action, method
= "qr", model = F, x = F, y = F, contrasts = NULL, ...)
{
  call <- match.call()
  m <- match.call(expand = F)
  m$method <- m$model <- m$x <- m$y <- m$contrasts <-
m$... <- NULL
  m[[1]] <- as.name("model.frame")
  m <- eval(m, sys.parent())
  . . .
}
```

## Interpreting Complicated Argument Lists

We may want our functions to accept data in several different formats. For example, most of the plotting functions require *x* and *y* arguments, but convert a two column matrix, a vector of complex numbers, or a list containing *x* and *y* components into the *x* and *y* arguments. Here is a simple example of how to do this:

```
plotstrings <-
function(x, y, strings, ...)
{
  if (is.list(x) !is.null(x$x) !is.null(x$y))
  {
    strings <- y
    y <- x$y
    x <- x$x
  } else if (is.complex(x))
  {
    strings <- y
    y <- Im(x)
    x <- Re(x)
  } else if (is.matrix(x) && ncol(x)==2)
  {
    strings <- y
    y <- x[, 2, drop=F]
    x <- x[, 1, drop=F]
  }

  # now x, y, and strings have their values
  # the rest of the function will call up a graphics device
  # and plot them
  . . .
}
```

In this example, we check the `x` argument to determine if it will yield both the `x` and `y` arguments. If so, we set `strings <- y` to make `strings` the second argument. This technique is fine if there are only one or two positional (instead of named) arguments. If we had seven or eight arguments before the `<ms-program-code>...`, however, we would need to shift all of them as we shifted `strings`. This can get tedious.

The low level `x-y` plotting functions (e.g., `lines`, `points`, and `text`) use a different strategy. They call an auxiliary function which uses `amatch` to examine the argument list of its caller, extracts the `x` and `y` vectors from this list, and then calls another function which can only handle argument lists containing `x` and `y` vectors. The following auxiliary function, `return.xy`, illustrates the strategy. For concreteness, let's assume that `return.xy` is called by `plotstrings`. The `return.xy` function begins by storing (as `caller`) the frame number of the function that calls `plotstrings` and (as `plot.call`) the unevaluated parse tree of the call to `plotstrings`:

```
return.xy <-
function()
{
  caller <- sys.parent(2)
  plot.call <- sys.calls()[[sys.parent(1)]]
  . . .
```

Next, `return.xy` stores the argument list generated by calling `amatch` on the call `plot.call`:

```
. . .
args <- amatch(function(x, y, ...)
{
}
, plot.call)
default <- attr(args, "missing")
if(default[1])
  stop("no x or y data")
. . .
```

The function `amatch` takes the argument list of its first argument (which should be a function) and the parse tree of a function call and returns a list of arguments matched by name or order just as S-PLUS usually does when it evaluates a function call. The result of `amatch()` has an attribute named `missing`, which is a logical vector with one entry per element of the argument list. If `amatch` cannot find an argument in the call to match the `i`th argument in the function definition, `missing[i]` is `T`. The elements in `args` are named `x`, `y`, and `<ms-program-code>...`. They generally have values of the form `Argument(expression, x=)`, which we want to convert to `x =`

*evaluated expression* (if the expression is trivial, like a number, `.Argument(number, x=)` is collapsed to just the number). The test `is.language(xexpr)` yields T if `xexpr` is of the form `.Argument()`. In that case we want to evaluate the expression in the argument (i.e., its first component):

```
. . .
xexpr <- args$x
if(is.language(xexpr))
  xexpr <- xexpr[[1]]
x <- eval(xexpr, caller)
other <- args$...[[1]]
y <- NULL
. . .
```

Now we try to find an implicit `y` variable embedded in the `x` variable. It may be the `y` component of a list, the imaginary part of a complex vector, or the second column of a matrix. If the first argument is a time series, then `x=time(series)` and `y=series`.

```
. . .
if(is.list(x))
{
  if(any(is.na(match(c("x", "y"), names(x)))))
    stop("cannot find x and y in list")
  y <- x$y
  x <- x$x
} else if(is.complex(x))
{
  y <- Im(x)
  x <- Re(x)
} else if(is.matrix(x) && ncol(x) == 2)
{
  y <- x[, 2]
  x <- x[, 1]
} else if(default[2])
{
  y <- x
  x <- time(x)
}
if(length(x) == 0)
  stop("zero length x data")
. . .
```

If `y` is still `NULL`, then the `x` variable doesn't contain an implicit `y`—it must be the next argument in the argument list returned by `amatch`:



```

    if(is.null(y))
    {
        yexpr <- args$y
        if(is.language(yexpr))
            yexpr <- yexpr[[1]]
        y <- eval(yexpr, caller)
    }
    . . .

```

Otherwise, *y* was found within the first argument so we shift all the other arguments over by one:

```

    . . .
else
{
    other <- amatch(function(x, ...)
    {
    }
    , plot.call)[[2]][[1]]
}
if(length(y) == 0)
    stop("zero length y data")
    . . .

```

At this point, we can construct an explicit list of *x* and *y* vectors and return it:

```

    . . .
    list(x = x, y = y)
}

```

The complete `return.xy` function is shown below:

```

return.xy <-
function()
{
    caller <- sys.parent(2)
    plot.call <- sys.calls()[[sys.parent(1)]]
    args <- amatch(function(x, y, ...) {}, plot.call)
    default <- attr(args, "missing")
    if(default[1])
        stop("no x or y data")
    xexpr <- args$x
    if(is.language(xexpr))
        xexpr <- xexpr[[1]]
    x <- eval(as.name("x"), sys.parent(1))
    other <- args$...[[1]]
}

```

```
    y <- NULL
    if(is.list(x))
    {      if(any(is.na(match(c("x", "y"), names(x)))) )
          stop("cannot find x and y in list")
          y <- x$y
          x <- x$x
    }
    else if(is.complex(x))
    {      y <- Im(x)
          x <- Re(x)
    }
    else if(is.matrix(x) && ncol(x) == 2)
    {      y <- x[, 2]
          x <- x[, 1]
    } else if(default[2])
    {      y <- x
          x <- time(x)
    }
    if(length(x) == 0)
        return(NULL)
    if(is.null(y))
    {      yexpr <- args$y
          if(is.language(yexpr))
              yexpr <- yexpr[[1]]
          y <- eval(yexpr, caller)
    } else
    {      other <- amatch(function(x, ...)
        {
        }
        , plot.call)[[2]][[1]]
    }
    if(length(y) == 0)
        stop("zero length y data")
    list(x = as.vector(x), y = as.vector(y))
}
```

---

<b>Frames, Names and Values</b>	<b>788</b>
Frames and Argument Evaluation	793
Quick Call Functions	793
Creating and Moving Frames	793
<b>Databases in S-PLUS</b>	<b>795</b>
Database Dictionaries	799
Directory Databases and Object Storage	800
Recursive Objects as Databases	802
User-Defined Database Classes	803
<b>Matching Names and Values</b>	<b>805</b>
<b>Commitment of Assignments</b>	<b>807</b>

What happens when you associate an S-PLUS object with a name? How does S-PLUS find the object again once assigned? For the most part, when you are using S-PLUS, the answers to such questions are unimportant. However, when writing functions, you may be surprised by an apparently correct function returning with an error saying,

Object "a" not found

In such cases, some knowledge of how S-PLUS maintains data objects can be helpful in debugging. Knowledge of S-PLUS data management can also help you write more efficient functions, a topic we will turn to in Chapter 22, Using Less Time and Memory.

This chapter discusses the main features of S-PLUS data management.

## FRAMES, NAMES AND VALUES

A *frame* is essentially a list associating names with values. Within a single frame, each name can have at most one value. When we say a function is evaluated in a certain frame, we are saying that the evaluation uses that frame's associated names and values.

Frames are important in S-PLUS because they serve to limit the scope of assignments. Assignments associate names with values, and the S-PLUS evaluator uses this association to determine the values associated with each name during the evaluation of a function. New frames are created by most S-PLUS functions when they are called; each function call uses its own frame to determine the values associated with each name during the evaluation of the function.

Every time you type an expression on the S-PLUS command line, S-PLUS creates a frame called variously the *expression frame*, *top-level frame*, or simply *frame 1*. The expression frame contains the unevaluated expression and a flag specifying whether automatic print is enabled or disabled. If the expression is a function call, S-PLUS creates a new frame, *frame 2*, for the called function, containing the arguments to the function, (unevaluated until needed), plus any names assigned values in the body of the function. For example, suppose we have the following function definition:

```
fcn.B <-
function(x, y)
{
  a <- sqrt(x)
  print(a)
  b <- log(y)
  C <- a + b
  sin(C)
}
```

Now, suppose we have the following vectors A and B:

```
> A
[1] 1 2 3 4 5 6 7 8 9 10

> B
[1] 10 12 14 16 18 20 22 24 26 28
```

If we call `fcn.B` with the expression `fcn.B(A, B)`, `fcn.B`'s frame at first contains only the unevaluated arguments:

```
$x:
. Argument(A, x = )
```

```
$y:
. Argument(B, y = )
```

In the course of evaluating the expression `a <- sqrt(x)`, `x` is evaluated and its name-value pair is inserted into the frame, as is the name `a` and the value assigned to it:

```
$a:
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
[6] 2.449490 2.645751 2.828427 3.000000 3.162278
```

```
$x:
[1] 1 2 3 4 5 6 7 8 9 10
```

```
$x:
. Argument(A, x = )
```

```
$y:
. Argument(B, y = )
```

The original, unevaluated arguments are retained in the frame, so they can be accessed by functions such as `substitute`, if necessary.

Just before `fcn.B` completes, its frame looks like the following:

```
$C:
[1] 3.302585 3.899120 4.371108 4.772589 5.126440
[6] 5.445222 5.736794 6.006481 6.258097 6.494482
```

```
$b:
[1] 2.302585 2.484907 2.639057 2.772589 2.890372
[6] 2.995732 3.091042 3.178054 3.258097 3.332205
```

```
$y:
[1] 10 12 14 16 18 20 22 24 26 28
```

```
$a:
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
[6] 2.449490 2.645751 2.828427 3.000000 3.162278
```

```
$x:
[1] 1 2 3 4 5 6 7 8 9 10
```

```

$x:
.Argument(A, x = )

$y:
.Argument(B, y = )

```

Expressions that do not involve assignment (such as `print(a)` and `sin(C)` in our example) are not reflected in the frame list; this is only natural, since the list is just an association of names and values. If there is no assignment, a value has no name associated with it. Frames are organized hierarchically, with the expression frame at the top, and subsequent function evaluation frames below. A frame is said to be the *parent* of the frame immediately below it in the hierarchy. For example, Figure 21.1 shows a hierarchy of three frames, generated by a call to `my.nest`, which is defined as follows:

```

my.nest <-
function(x)
{   my.sqrt(x)
}.

```

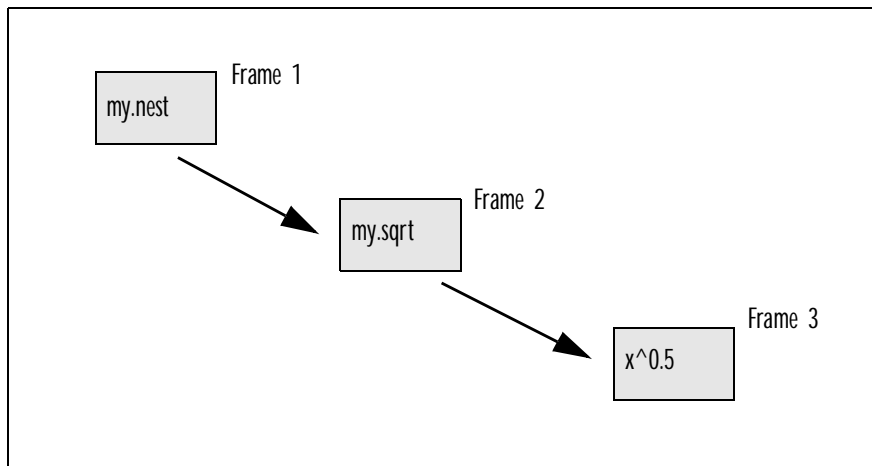


Figure 21.1: The frame hierarchy.

The function `my.sqrt`, in turn, is defined as follows:

```

my.sqrt <-
function(x) {x^0.5}

```

The expression frame is the parent of the function frame of `my.nest`. The function frame of `my.nest`, in turn, is the parent of the function frame of `my.sqrt`. Each frame has a number associated with it. Frames can be referred to by number—as we have already mentioned, the expression frame is frame 1 and the frame of the top-level function call is frame 2.

The complete frame hierarchy is maintained in a list called the *list of frames*. Each frame corresponds to a component of the list of frames. You can view the list of frames with the `sys.frames` function, which is most informative when called from within a nested function call. For example, suppose we redefine `my.sqrt` as follows:

```
my.sqrt <-
function(x)
{
  x^0.5
  sys.frames()
}
```

A call to `my.nest` now yields the following:

```
> my.nest(4)

[[1]]:
[[1]]$.Auto.print:
[1] T

[[1]]$.Last.expr:
expression(my.nest(4))

[[2]]:
[[2]]$x:
[1] 4

[[3]]:
[[3]]$x:
[1] 4

[[3]]$x:
.Argument(x, x = )
```

Here we see the expression frame in component `[[1]]`: it contains the autoprint flag and the current expression. The evaluation frame for `my.nest` consists of just the single argument 4, which needs no evaluation. Finally, the evaluation frame for `my.sqrt`, shown as component `[[3]]`, contains both the unevaluated argument `x` and the evaluated argument 4. S-PLUS includes several functions for using the data stored in frames. To see the actual contents of the frames use `sys.frame` (to view the current frame), or `sys.frames` (to view all frames). To obtain the current frame's position in

the frame hierarchy, use `sys.nframe`. Use `sys.parent(n)`, where *n* is the number of generations to go back to find the position of a frame's parent, grandparent, or *n*th ancestor frame. In the example above, if we replace `sys.frames` with `sys.parent` in `my.sqrt`, we get the following result:

```
> my.nest(4)
```

```
[1] 2
```

indicating that the parent frame of `my.sqrt` is frame number 2 in the frame hierarchy. If you know in which frame a particular name-value pair is located, you can use the frame number, along with the `get` function, to retrieve the correct value. For example, consider the following function definitions:

```
top.level.func <-  
function(x)  
{  
  a <- sqrt(x)  
  next.level.func()  
}  
  
next.level.func <-  
function()  
{  
  get("a", frame = sys.parent()) * 2  
}
```

When we call `top.level.func`, we obtain the following result:

```
> top.level.func(25)
```

```
[1] 10
```

This is one method for passing assignments within functions to nested function calls. A more useful method is to pass the assigned names as arguments. This is discussed more fully in the section *Frames and Argument Evaluation* (page 793). Each function is evaluated in its own frame (with the exception of the so-called “quick call” functions, discussed in the section *Quick Call Functions* (page 793)).



## Frames and Argument Evaluation

When a function is called, arguments are placed, unevaluated, into the function's evaluation frame. As soon as the calling function references the argument, S-PLUS evaluates the argument in the *parent* frame of the function's evaluation frame. This ensures that constructions such as the following will work:

```
my.plot <-
function(x, y)
{
  a <- sqrt(x)
  b <- log(y)
  plot(a, b)
}
```

Here the actual arguments to `plot` are evaluated in `my.plot`'s evaluation frame. If argument evaluation took place in the `plot`'s evaluation frame, S-PLUS would be unable to find the appropriate values for `a` and `b`, which belong to the frame evaluating `my.plot`. Because the parent frame continues to exist at least as long as any of its child frames exist, arguments do not have to be evaluated until needed. This is the key to S-PLUS's *lazy evaluation*, described in Chapter 5, Writing Functions in S-PLUS. Default values however, are evaluated in `plot`'s frame.

## Quick Call Functions

Not all functions generate their own evaluation frame. Those that do not are a subset of the functions that are implemented as `.internal` functions, that is, of the functions that are defined as part of S-PLUS's underlying C code. These functions are evaluated in the frame of the calling function, much as arguments are evaluated. By not creating a new frame for their evaluation, S-PLUS makes their already efficient internal implementation that much more efficient.

## Creating and Moving Frames

Most frames in S-PLUS are created automatically when the evaluator encounters a function call. Sometimes, however, it is helpful to create frames explicitly to exercise more control over the evaluation. For example, the `eval` function allows you to evaluate any S-PLUS expression. It takes an optional second argument, `local`, that can be either a number (interpreted as one of the existing frames) or an explicit list object, with the named elements defining the name-value pairs. Thus, for example, suppose we have a simple list object `myframe`:

```
> myframe <- list(a=100, b=30)
```

In evaluating the following, S-PLUS uses `myframe` as the frame for evaluation:

```
> eval (expressi on(a + b), myframe)
```

This construction lets you share a set of name-value bindings over many different expressions, without relying on the parent frame to maintain the list. An important application of this is in conjunction with the `new.frame` function, which lets you explicitly create a new frame. For example, we can use `myframe` as a frame for a number of calculations by defining a function as follows:

```
manycal c <-  
functi on()  
{  
  n <- new.frame(myframe)  
  a <- eval (expressi on(max(a, b)), n)  
  b <- eval (expressi on(mean(c(a, b))), n)  
  clear.frame(n)  
  a - b  
}
```

The output from a call to `manycal c` is shown below:

```
> manycal c()
```

```
[1] 35
```

In this example, we take advantage of `myframe` to reuse the variable names `a` and `b` in `manycal c`'s evaluation frame. Because the evaluation of `max` and `mean` are performed in `myframe`, the assignment to `a` does not affect the calculation of `mean(c(a, b))` at all. The max of `a` and `b` in `myframe` is 100, and the mean of `a` and `b` is 65. Taking the difference of these two values, in `manycal c`'s frame, yields the answer, 35. We used the `clear.frame` function to get rid of frame `n` when we were done with it.

## DATABASES IN S-PLUS

An S-PLUS *database* is simply a collection of named objects. In this respect it is closely related to a frame. The distinction is primarily one of duration. Objects stored permanently by name are found in databases, while objects stored temporarily are found in frames. For this reason, the session frame, which we met previously as frame 0, can also be thought of as a database, since it endures for an entire S-PLUS session. Two types of databases are in common use: file directories and recursive (list-like) objects, particularly data frames. Other database types exist, and still others can be defined, but these two types are adequate for most purposes. For a description of the other types currently available, see Chambers [Chambers:91]. If an object is referred to in an S-PLUS expression, and its name-value binding is not found in the current frame, S-PLUS searches the expression frame. If the binding is not found there, S-PLUS searches through databases, starting with database 0 and continuing along a user-specified search path. When you start S-PLUS, this path consists of your *working directory* and eleven directories of S-PLUS functions and data sets. You can see the search path at any time using the search function:

```
> search()

[1] "C:\\Program Files\\spl us2000\\home\\_Data"
[2] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Function"
[3] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Function"
[4] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Function"
[5] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Dataset"
[6] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Dataset"
[7] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Dataset"
[8] "C:\\PROGRAM FILES\\SPLUS2000\\library\\trellis\\_Data"
[9] "C:\\PROGRAM FILES\\SPLUS2000\\library\\sapi\\_Data"
[10] "C:\\PROGRAM FILES\\SPLUS2000\\library\\sgui\\_Data"
[11] "C:\\PROGRAM FILES\\SPLUS2000\\library\\editdata\\_Data"
[12] "C:\\PROGRAM FILES\\SPLUS2000\\library\\menu\\_Data"
```

Databases can be added to the search list with the attach function:

```
> attach("c:\\dox\\_Data")
> search()

[1] "C:\\Program Files\\spl us2000\\home\\_Data"
[2] "c:\\dox\\_Data"
[3] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Function"
[4] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Function"
[5] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Function"
```

```
[6] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Dataset"
[7] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Dataset"
[8] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Dataset"
[9] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\trell is\\_Data"
[10] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sapi \\_Data"
[11] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sgui \\_Data"
[12] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\edi tdata\\_Data"
[13] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\menu\\_Data"
```

Similarly, databases can be removed from the search list with the detach function:

```
> detach("c:\\dox\\_Data")
> search()
```

```
[1] "C:\\Program Files\\spl us2000\\home\\_Data"
[2] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Functi o"
[3] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Functi o"
[4] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Functi o"
[5] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Dataset"
[6] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Dataset"
[7] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Dataset"
[8] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\trell is\\_Data"
[9] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sapi \\_Data"
[10] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sgui \\_Data"
[11] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\edi tdata\\_Data"
[12] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\menu\\_Data"
```

By default, new databases are added in position two of the search list, immediately after the working data. You can override this positioning with the pos argument:

```
> attach("c:\\dox\\_Data", pos=8)
> search()
```

```
[1] "C:\\Program Files\\spl us2000\\home\\_Data"
[2] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Functi o"
[3] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Functi o"
[4] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Functi o"
[5] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Dataset"
[6] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Dataset"
[7] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Dataset"
[8] "c:\\dox\\_Data"
[9] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\trell is\\_Data"
[10] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sapi \\_Data"
[11] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sgui \\_Data"
[12] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\edi tdata\\_Data"
[13] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\menu\\_Data"
```

You can also provide an "alias" for a directory with the `name` argument:

```
> attach("c:\\dox\\_Data", pos=3, name="myfuncs")
> search()

[1] "C:\\Program Files\\spl us2000\\home\\_Data"
[2] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Functi o"
[3] "myfuncs"
[4] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Functi o"
[5] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Functi o"
[6] "C:\\PROGRAM FILES\\SPLUS2000\\s\\_Dataset"
[7] "C:\\PROGRAM FILES\\SPLUS2000\\stat\\_Dataset"
[8] "C:\\PROGRAM FILES\\SPLUS2000\\spl us\\_Dataset"
[9] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\trell is\\_Data"
[10] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sapi \\_Data"
[11] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\sgui \\_Data"
[12] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\edi tdata\\_Data"
[13] "C:\\PROGRAM FILES\\SPLUS2000\\li brary\\menu\\_Data"
```

Naming databases in the search list is particularly useful if you are attaching and manipulating databases within a function, because it means you don't have to keep track of the database's position within the search list. (As the examples above show, a given directory can occupy many different places in the list depending on where and when different databases are attached.) For example, if you have a function that makes extensive use of a particular group of functions that are stored together in a directory, you might want to attach the directory at the beginning of the function, to ensure that the group of necessary functions is available, then detach it at the end of the function:

```
function()
{
  attach("/usr/lib/groupfuns.S", pos=2,
        name="groupfuns")
  on.exit(detach("groupfuns")) . . .
}
```

The contents of databases in the search list can be manipulated using the following generic functions:

Table 21.1: Functions to manipulate the contents of databases.

Function	Purpose
<code>exists</code>	Tests whether a given object exists in the search path.
<code>get</code>	Returns a copy of the object, if it exists. Otherwise, returns an error.
<code>assign</code>	Creates a new name-value pair in a specified database.
<code>remove</code>	Deletes specified objects, if they exist, from the specified database.
<code>objects</code>	Returns a character vector of the names of the objects in the specified database.

Except for `get` and `exists`, these functions operate on the working directory (or the current frame, if called within a function) unless another database is specified. The `exists` and `get` functions search the entire search path before returning an answer.

**Warning**

The `assign` and `remove` functions modify the contents of databases. In particular, if you assign a value to a name that already has a value in the specified database, the old value is lost. For example,

```
> get("x", where=2)

[1] "White" "Black" "Gray" "Gray" "White" "White"

> assign("x", 1:10, where=2)
> get("x", where=2)

[1] 1 2 3 4 5 6 7 8 9 10
```

These functions are the *only* way to manipulate S-PLUS objects having names that do not follow S-PLUS's syntactic conventions, that is, functions with names *not* made up solely of alphanumeric characters and periods, not beginning with a number. For example, in Chapter 5, Writing Functions in S-PLUS, we mentioned that the name `2do` was *not* syntactically correct. However, virtually any quoted string can be used as an object name, with the limitation that S-PLUS does not automatically recognize such strings as names. Thus, we can create an object `"2do"` in the usual way:

```
> "2do" <- 1:10
```

However, if we type the object's name, we do not see the expected behavior:

```
> 2do
```

```
Syntax error: name ("do") used illegally at this point:
2do
```

Because `2do` is not a syntactic name, the parser tries to interpret it as best it can; it reads "2" as a numeric literal, then complains when it is immediately followed by a name, "do." To get around this problem, we can use the database manipulation functions:

```
> get("2do")
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

### Assignments and quoted strings

If you assign a value to a quoted string that *is* a syntactically correct name, S-PLUS strips the quotes from the string during the assignment. Thus, the assignments

```
zoo <- 1:10
"zoo" <- 1:100
```

result in only one object, `zoo`, with value the vector of integers from 1 to 100.

## Database Dictionaries

Most databases are searched by means of *dictionaries*, which are simply tables consisting of either the names of the objects in the database, or the list of name-value pairs. The dictionary is constructed when the database is attached, and the database manipulation functions update it as necessary. However, events outside the current S-PLUS session are not reflected in the dictionary. You can look at the dictionary for any database with the `dbobjects` function:

```
> dbobjects("/betty/users/ri ch/. Data")
```

```
[1] "lnw"      "inflow"      "x.cancer"    "snews.par"
[5] "lillist"   "aaa"          "fac2"        "foo"
[9] "diff.hs.ts" "allocated2"  "bi grm"      "fi x"
[13] "tut.choices" . . .
```

```
> dbobjects(kyphosis)
```

```
[1] "Kyphosis" "Start" "Age" "Number"
```

If `dbobjects` returns `NULL`, the database has no associated dictionary. Databases without dictionaries must be searched using queries to `exists`; this procedure is significantly slower than searching using a dictionary.

## Directory Databases and Object Storage

File system directories are used to store S-PLUS objects, and they are thus the most common form of database for general use. Objects are stored in a special binary format that includes both the object itself and information about the object's S-PLUS structure. Because S-PLUS object files are neither readable by humans nor necessarily under their own names, you should manipulate them only from within S-PLUS. For example, use the `objects` function within S-PLUS to view a list of data objects. Similarly, you should use the S-PLUS `rm` and `remove` functions. If you want to edit a data object, use the S-PLUS `Edit`, or possibly `fix`, commands, rather than editing the corresponding file outside of S-PLUS. Most, but not all, objects are stored as files under their own names. Objects with names incompatible with the file system's file naming conventions are *mapped* to files with names chosen by S-PLUS. S-PLUS maintains a list of all mapped objects and the file names under which the objects are stored.

The mapping works as follows: when an object is assigned to a given name in a given directory, S-PLUS checks whether the name can be accommodated by the file system, and if so, creates a file with that name. If the file system cannot accommodate the name, S-PLUS generates a "true file name" internally that is consistent with the current file system, creates a file with this name, and maps the S-PLUS object name to this file.

On DOS systems and other systems with restrictive naming conventions, you can expect many more objects to be mapped than on systems with less restrictive conventions. The true file names of mapped objects have the form `__n`, where *n* indicates that the object is the *n*th mapped object created. If you attempt to create an object with a name of this form, S-PLUS maps it:

```
> "either/or" <- 1:10
```

```
> dos("DIR _data")
```

```
[1] ""
```

```
[2] " Volume in drive C has no label "
```

```
[3] " Volume Serial Number is 0146-07CB "
```

```
[4] " Directory of C:\\RICH\\_DATA "
```

```
[5] ""
```

```
[6] ".          <DIR>  01-21-93  3:24p "
```



```
[7] ". . . <DIR> 01-21-93 3: 24p"
[8] "_AUDIT 31913 01-29-93 10: 51a"
[9] "__DB3 1 0 01-21-93 3: 29p"
[10] "___NONFI 42 01-21-93 5: 24p"
[11] "__1 156 01-28-93 2: 02p"
[12] " 7 file(s) 32345 bytes"
[13] " 131129344 bytes free"
```

```
> "__1" <- 10: 20
> dos("DIR _data")
```

```
. . .
[10] "___NONFI 42 01-21-93 5: 24p"
[12] "__2 234 01-28-93 2: 05p"
[12] "__1 156 01-28-93 2: 02p"
. . .
```

Here "either/or" needs to be mapped because DOS does not permit file names with slashes, so S-PLUS maps the object to the file \_\_1 and records the true file name and the mapped object name in the file \_\_\_nonfi le. The new object "\_\_1", while having a perfectly valid file name, conflicts with the S-PLUS mapping scheme, so it is itself mapped, to \_\_2. To see that this is the case, remove "either/or":

```
> remove("either/or")
> dos("DIR _data")
```

```
. . .
[10] "___NONFI 42 01-21-93 5: 24p"
[12] "__2 234 01-28-93 2: 05p"
. . .
```

To use the mapping scheme from S-PLUS functions, you can use the S-PLUS function `true.file.name`. The only required argument is the S-PLUS object name. You can also specify the position of the database in the search list, and set a flag specifying whether the true file name should be added to the \_\_\_nonfi le file, if it isn't there already. For example, we can verify that our object "1" has the true file name \_\_2:

```
> true.file.name("__1")

[1] "__2"
```

## Recursive Objects as Databases

Any recursive object with named components can be used as a database, just as such objects were used as frames in the section *Frames, Names and Values* (page 788). When you attach such an object, a dictionary for the database is created containing copies of all the named components of the object. Unlike directory databases, the dictionary for an object database actually contains the data, not just the names. Thus, finding a given component is faster for an attached object than for an object in a directory. Attaching the object also simplifies calling those components. For example, to view the `Age` component of the `kyphosi s` data frame, you could use the following:

```
> kyphosi s$Age

[1] 71 158 128 2 1 1 61 37 113 59 82 148
[13] 18 1 168 1 78 175 80 27 22 105 96 131
[25] . . .
```

After attaching `kyphosi s` as a database, however, you can access its components as whole objects:

```
> attach(kyphosi s)
> Age

1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 17 18 19
71 158 128 2 1 1 61 37 113 59 82 148 18 1 168 1 78 175
20 21 22 23 24 25 26 27 29 30 31 32 33 34 35 36
80 27 22 105 96 131 15 9 8 100 4 151 31 125 130 112
. . .
```

The simplified syntax is useful when you are constructing complicated expressions involving different components. For example, compare the following expressions:

```
i f (any(kyphosi s$Age > 130))
    sqrt(kyphosi s$Start)
e l s e exp(kyphosi s$Number)

i f (any(Age > 130))
    sqrt(Start)
e l s e exp(Number)
```

If you attach a recursive data object in position 1 in the search path, you can modify the components of the objects directly, and save the changes to the usual working data (typically, the `.Data` or `_Data`) when you detach the object. This is a useful technique for adding variables to a data frame—

simply assign a vector the same length as the existing variables to the data frame. For example, to add a weight per displacement variable to the `cu. specs` data frame, you could proceed as follows:

```
> attach(cu. specs, pos=1)
> wpd <- Weight/Di sp.
> detach(1, save="cu. specs. new")
> names(cu. specs. new)

[1] "Wei ght"  "Ti res"   "Steeri ng"
[4] "Turni ng" "Di sp. "  "HP"
[7] "Trans1"   "Gear. Rati o" "Eng. Rev"
[10] "Tank"     "Model 2"   "Di st. n"
[13] "Ti res2"   "Pwr. Steer" ". empty. "
[16] "Di sp2"    "HP. revs"  "Trans2"
[19] "Gear2"     "Eng. Rev2" "wpd"
```

Attaching a data frame in position 1 ensures that the variables in the data frame are not masked by objects of the same name in the usual working data

#### Warning: Assignments to data frames

All assignments to a data frame must be to objects the same length as the variables in the original data frame, or the assignments will be lost when the object is saved. Thus, do not try to carry out an entire S-PLUS session with a data frame attached in position 1.

## User-Defined Database Classes

Databases are ubiquitous in modern computing environments, because they offer such a convenient way of storing and retrieving information. A familiar example is the *relational database*, such as Oracle or Informix SQL. Equally useful, however, are databases consisting of, for example, statistical data sets, such as those from the SAS or SPSS software packages. Often, it is desirable to use S-PLUS's data analysis techniques on the contents of such a database. For example, it might be useful to apply S-PLUS's time series modules to data stored in a sales database. S-PLUS can be extended to allow such an analysis, by creating a database *class* corresponding to the sales database, and then defining *methods* to read from or write to the database, view a list of objects

in the database, and remove objects that no longer exist. The specific methods are used with the following generic functions, and must have the indicated effect:

Table 21.2:Generic database functions.

Function	Purpose
<code>dbread(database, name)</code>	Reads <i>name</i> from <i>database</i> . Returns the S-PLUS object corresponding to <i>name</i> .
<code>dbwrite(database, name, object)</code>	Writes a permanent version of <i>object</i> into <i>database</i> , with S-PLUS object name <i>name</i> .
<code>dbremove(database, name)</code>	Removes the object stored as <i>name</i> on <i>database</i> , or return a warning if the object does not exist.
<code>dbobjects(database)</code>	Returns one of the following: 1. A character vector giving the names of the objects currently in <i>database</i> . 2. A list giving both the names of the objects currently in <i>database</i> and their contents. 3. The value NULL.
In the first two cases ( <code>dbread</code> and <code>dbwrite</code> ), a dictionary is formed when the database is attached. The dictionary consists of either the names or the names and contents of the database.	

## MATCHING NAMES AND VALUES

When S-PLUS evaluates an expression, it must match any names in the expression with the appropriate S-PLUS objects and their values. The search begins in the current frame, typically the evaluation frame of a function. Name-value pairs that are not found in the current frame are then searched for in the expression frame, and then in the session frame (database). If a name is not matched in any of these three frames, S-PLUS searches in turn each of the databases currently in the search path, starting with the working directory. Other existing *frames* are not searched; this is of particular importance when considering the evaluation of nested function calls. For example, suppose we have two functions defined as follows:

```
fcfn.C <-
function()
{
    x <- 3
    fcn.D()
}
fcfn.D <-
function()
{
    x^2
}
```

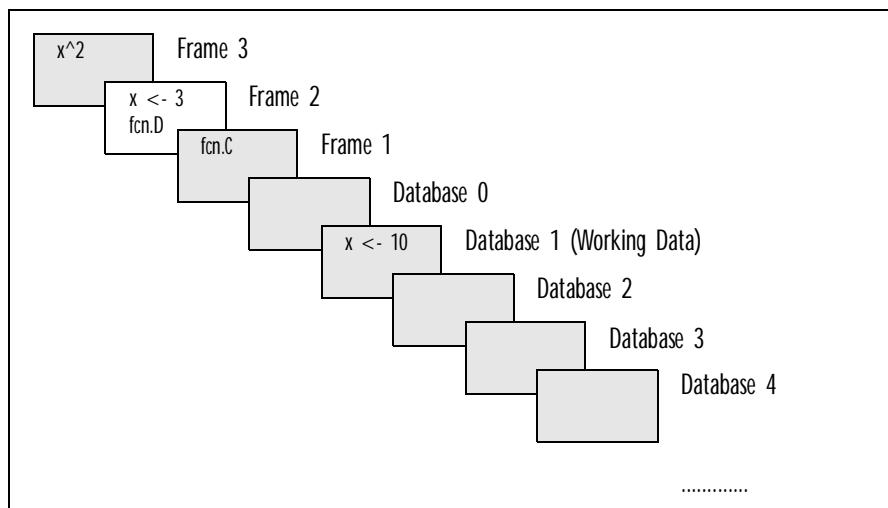


Figure 21.2: The search path through the frames.

If we call `fcn. C`, the call to `fcn. D` within `fcn. C` creates frame 3. Objects referred to in frame 3 are first looked for there, then, if not found there, are looked for in frame 1. Notice that *frame 2* is not searched. Figure 21.2 shows the search path for this example; only the shaded boxes are searched, top to bottom. The list of searched databases can be arbitrarily long.

You can override the normal search path by specifying either the `where` or `frame` argument (it is an error to specify both) to `get`.

---

## COMMITMENT OF ASSIGNMENTS

Because permanent assignments alter the contents of S-PLUS's permanent databases, there is a safety mechanism to prevent such assignments when the top-level expression encounters an error or interrupt. Thus, if you have the line `x <- letters[1:10]` inside a function `fcn.E`, and `fcn.E` stops with an error (any error), nothing is assigned to name `x`. Thus, if `x` already exists, it is unchanged, and if it does not exist, it is not created. For example, suppose we have an existing object `A` and a function `fcn.E` defined as follows:

```
> A

[1] 1 2 3 4 5 6 7 8 9 10

> fcn.E
function(y)
{
  A <- letters[1:10]
  stop("Any error")
}
```

When we call `fcn.E` and then look at `A`, we find it is unchanged, even though the assignment occurs *before* the error:

```
> fcn.E()

Error in fcn.E(): Any error
Dumped

> A

[1] 1 2 3 4 5 6 7 8 9 10
```

This safety mechanism is called *backout* protection. For backout to be possible, assignments remain pending until the top-level expression completes, at which time the assignments are actually *committed*. For purposes of evaluation, however, the assignments do take place immediately. Thus, consider the following function:

```
assn.use <-  
function()  
{  
  assign("x", 10:1, where = 2)  
  print(2 * get("x", where = 2))  
  stop("Nothing is committed")  
}
```

The value printed by the second line of `assn.use` reflects the assignment to `x` on database 2, even though `x` has not yet been permanently committed:

```
> assn.use()  
  
[1] 20 18 16 14 12 10 8 6 4 2  
Error in assn.use(): Nothing is committed  
Dumped
```

Because assignments remain pending until the completion of the top-level expression, S-PLUS must retain the previous value, if any, throughout evaluation of the top-level expression. Thus, repeated assignment of large data objects can cause excessive memory buildup inside S-PLUS. The `immediate` argument to the `assign` function can reduce this memory buildup. When you use `assign` with `immediate=T`, S-PLUS overrides the normal backout mechanism and commits the assignment *immediately*. (Similar memory growth can occur simply from *reading* too many data objects, because by default S-PLUS stores any newly-read object in memory in case it is needed elsewhere in the top-level expression. To prevent this caching, read objects using `get` with the `immediate=T` argument.)

Another way to override the backout mechanism is to use the `synchronize` function, which commits all pending assignments and thus *synchronizes* the databases and their dictionaries. If called with a numeric vector *i* as its argument, `synchronize` reattaches the databases in the positions given by *i* in the search list. Reattaching not only updates the dictionaries and databases with respect to commitment, but also with respect to changes made by other processes. For example, if an object has been edited in one window and you want to use it in another, a call to `synchronize` with the object's database position as its argument will ensure that you are getting the latest object.



# USING LESS TIME AND MEMORY

# 22

---

<b>Time and Memory</b>	<b>810</b>
How S-PLUS Allocates Memory	811
Why and When S-PLUS Copies Data	812
<b>Writing Good Code</b>	<b>816</b>
Use Vectorized Arithmetic	816
Avoid for Loops	817
Avoid Growing Data Sets	819
Avoid Looping Over Named Objects	820
Keep It Simple!	820
Reuse Computations	821
Reuse Code	822
Avoid Recursion	822

In Chapter 5, Writing Functions in S-PLUS, we described several rules of thumb for writing functions that run faster, consume less memory, or both. We offered those rules with little explanation, just the assurance that if you followed them, your functions would be more efficient. In this chapter, we explore some of the justifications for those rules. We begin with a brief description of what we mean by time and memory, followed by a brief discussion of how S-PLUS allocates and manages memory. This information is important for understanding why, for example, for loops are less efficient in S-PLUS than vectorized expressions. This information should help you write functions which minimize the required number of copies of large data sets.

## TIME AND MEMORY

Time and memory, when used as measures of the efficiency of a function, can have many meanings. The *time* required by a function might be any of the following:

- *CPU time*: the time the computer spends actively processing the function.
- *elapsed time*: the time elapsed on a clock, stopwatch, or other time-keeping device. Generally, you are most interested in elapsed time, but unfortunately, elapsed time is not a particularly reliable measure of efficiency because it can vary widely from one execution to the next. Disk activity and other programs on the system can have a profound effect on the elapsed time. Disk activity in S-PLUS functions arises from the following:
  - reading and writing S-PLUS data sets and functions
  - reading and writing temporary files
  - paging to swap space when memory usage exceeds main memory
- *programmer time*: the time it takes the programmer to write and maintain a function (and, less directly, the time it takes the programmer to learn to write functions). Often, “computationally efficient” functions take much longer to write than simpler alternatives. If the function will not be called frequently, it may not be worth the extra effort.

Like the word “time,” the word “memory” can have different meanings depending on the context. Originally, “memory” meant simply *main memory*, or *RAM*, and referred to physical memory built into the computer and accessible from the CPU during processing. The advent of *virtual memory* complicated the issue. With virtual memory, when a program uses more memory than is physically available, data that is not actively being processed is *swapped* to a hard disk, freeing main memory for further processing. A portion of the hard disk must generally be committed to supporting swap operations. This portion is called *swap space*. Using virtual memory allows your programs to be larger than they would otherwise be, but incurs a

significant time penalty when memory use exceeds main memory. Operations involving hard disks are significantly slower than operations in RAM, and each swap involves disk IO.

S-PLUS makes extensive use of virtual memory. To your S-PLUS session, your total memory is the sum of your main memory and swap space. S-PLUS is free to use both, and in fact it doesn't know which it is using at any particular time. Functions that seem slow may actually be fast algorithms that use a lot of memory, causing S-PLUS to swap. Particularly if you need to analyze large data sets, it is wise to equip your computer or workstation with as much RAM as you can afford. This will help you avoid the time penalty for swapping and make your functions more efficient.

## How S-PLUS Allocates Memory

To write the most efficient S-PLUS functions, you should have some understanding of how S-PLUS allocates and frees memory. Such understanding takes you a long way toward understanding why loops and recursion are discouraged in S-PLUS programming.

In Chapter 21, Data Management, we described frames as lists associating names and values. Those lists must be maintained in memory, and obviously different frames can have different associations of names and values. Thus, it is not too surprising that, at a high level, memory allocation in S-PLUS corresponds to memory allocation for the various frames. At a lower level, memory is allocated by *arenas* and *buckets*.

A standard arena is a block of memory (4088 bytes in S-PLUS 2000) which, once allocated, can be filled by any number of S-PLUS objects. Atomic data objects consist of the data plus a 36 byte vector header. Recursive data objects such as lists also consist of a vector header plus the data, but recursive data is a combination of vector headers and atomic data. Atomic objects larger than a user-specified size (`options(scrap.bytes)`, by default 500) are put into a custom arena, exactly large enough to fit the object. The vector headers for all objects in a given frame are stored in buckets. Each bucket can hold up to 75 in headers, although most frames use only a handful of the available vector headers in their associated buckets. A frame is created for every new top-level S-PLUS expression, and for most (though not all) function calls generated by the top-level expression. Within each frame, arenas and buckets are allocated as necessary to maintain the frame. When S-PLUS is finished with the frame, values are returned to the parent frame in one of two ways. If the value occupies a custom arena, the arena is simply reassigned to the parent frame. If the value is contained in a standard arena, the value is copied to one of the parent frame's standard arenas. Once any necessary values are transferred to the parent frame, the frame is broken down and all its associated memory is freed.

## Why and When S-PLUS Copies Data

S-PLUS has many attributes of a functional language: functions generally look at the data in their arguments and return values, rather than alter the data given to them as arguments. Contrast this with typical Fortran code, in which a subroutine is given a pointer to an array and then modifies parts of that array. If an S-PLUS function alters parts of a vector given to it as an argument, it does so on a private version of that vector so the calling function does not see the change. If we want the calling function to get the changed vector, we have the called function pass it back as a return value. (Replacement functions such as "`[<-`" and "`names<-`" do alter their arguments, but these are the exception in S-PLUS.) In order to ensure that a function cannot alter data in its caller's frame, S-PLUS must copy arguments to functions. It does not always copy arguments, but if it is unsure whether it must it will err on the side of caution and make the copy. In particular, if a data set is named, its value cannot be changed by arbitrary functions, so it will be copied. S-PLUS may also copy data when returning the value of a function call, although it tries to avoid copying, if possible. Since returning a function value involves moving data from the called frame to the caller's frame and the caller's frame is about to be destroyed, S-PLUS usually just rearranges some internal pointers so the memory in the called frame is transferred to the caller's frame and no copies are required. However, since vectors smaller than 500 bytes (or the value specified in `options("scrap.size")`) are stored in blocks shared with other vectors, S-PLUS does copy such short vectors so that it doesn't have to move the unused part of those shared blocks into the caller's frame. Thus, if your function returns a large object consisting of short vectors it will be copied instead of being moved to the caller's frame. If you are writing a function to process a very large data set, it may be worth your time to see how many copies of that data set will be in memory at once. You may be able to avoid some copies by rearranging calculations or by not naming temporary results. You may also give temporary results the same name as a previous temporary result or use the `remove` function to remove a temporary result from the frame.

To get a feeling for how many copies are required in various situations, evaluate your function with a large argument, say 125,000 double precision numbers (1 million bytes), and see how it pushes up the amount of memory used. It is best to do this as the first function called in an S-PLUS session, so the results are not confounded by memory fragmentation caused by previous function calls. The function `mem.tally.report` can help you view the memory used in evaluating an S-PLUS function. It reports the maximum amount of memory used in S-PLUS evaluation frames since the last call to another function, `mem.tally.reset`. Thus, to get baseline memory usage, call `mem.tally.reset` followed by an expression combining the function

you want to measure and a call to `mem.tally.report`. Here is a simple example. We want to calculate the sum of a vector of many random numbers plus 1. One function to calculate this is as follows:

```
f <- function(n=125000) {x <- runif(n); sum(x + 1)}
```

To estimate the amount of space required, do the following:

- Call `mem.tally.reset()`:

```
> mem.tally.reset()
```

- When the S-PLUS prompt returns, create a braced expression containing your function call and a call to `mem.tally.report`:

```
> {f(); mem.tally.report()[2]}
evaluation
1997276
```

So you can see that evaluating `f` required approximately 2 million bytes, or space for two copies of `x`. The first copy was required to generate the random numbers, the second was needed to store the value of `x+1`. If you didn't name `x`, but called `sum(runif(n)+1)` then the value `runif(n)+1` could be stored in the space originally used by `runif(n)`, because we did not ask S-PLUS to save that data under a name. To see this, define a new function, `f1`, as follows:

```
f1 <- function(n = 125000) {sum(runif(n) + 1)}
```

Calling `f1` after another call to `mem.tally.reset` shows the savings:

```
> mem.tally.reset()
> {f1(); mem.tally.report()[2]}
evaluation
997276
```

When temporary variables are needed, using the same name for ones of the same size that are not required simultaneously can avoid an unneeded copy as well. For example, consider the following function:

```
g <- function(n = 125000)
{
  tmp <- runif(n)
  tmp1 <- 2 * tmp + tmp^2
  tmp2 <- tmp1 - trunc(tmp1)
  mean(tmp2 > 0.5)
}
```

This requires 6 million bytes to complete, while the following slightly modified version needs only 4.5 million:

```
g1 <- function(n = 125000)
{
  tmp <- runif(n)
  tmp <- 2 * tmp + tmp^2
  tmp <- tmp - trunc(tmp)
  mean(tmp > 0.5)
}
```

(The 0.5 million byte chunks come from the logical vectors such as `tmp>0.5` and `is.na(x)` in the call to `mean`.)

Some memory that cannot be accounted for is due to memory fragmentation. If a block of  $n$  bytes between two other memory blocks is freed and then we need a block of  $n+1$  bytes, we cannot get that space from the space just freed. This fragmentation depends on the exact sequence of function calls and you may find that adding an innocuous looking function call may actually decrease the memory requirements. One kind of memory fragmentation that you can often avoid is due to growing a vector in a loop. Each time a bit is added to the end S-PLUS must find a new location for the data vector because it no longer fits in its original space. If you preallocate the vector to the length you expect it to get, or even a bit bigger, you will often save some space.

Analyzing these functions to determine exactly where the copies are being made can quickly get confusing, so measuring simple examples of proposed constructs can be very instructive.

Another way to measure where memory is going it to call the following function, `storage.summary`:

```
storage.summary <- function(nframe = sys.parent()) {
  m <- mem.tally.report()[2]
  st <- storage()
  bytes <- sum(st$allocated[st$frame == nframe])
  + 16024 * sum(st$"frame (h)" == nframe)
  if(nframe > 0)
    frame.size <- object.size(sys.frame(nframe))
  else
    frame.size <- NA
  cat("nframe=", nframe, " bytes=", bytes,
      " frame size=",
      frame.size, " memory.size=", m, "\n ", sep = "")
  NULL
}
```

The `storage.summary` function is designed to be called with no arguments. It provides statistics on the frame from which it is called without affecting that frame. (It may affect the total memory used, by causing some fragmentation, but should not cause the frame being analyzed to change.)

The column labeled `bytes=` is the number of bytes of memory allocated in the frame, and `frame size=` is the number of bytes used for named objects in the frame. The difference represents both “scratch space,” used for temporary data, and data headers not yet used.

For instance, we can put a call to `storage.summary` before and after every line of `g` and `g1` and see how reusing the `tmp` in `g1` slows the growth of the frame. Also note how adding the calls to `storage.summary` decreases the ultimate memory size reported by `g1`:

```
> g()
nframe=5 bytes=4044 frame size=226 memory.size=0
nframe=5 bytes=1004044 frame size=1000320 memory.size=1000000
nframe=5 bytes=2004044 frame size=2000365 memory.size=2986464
nframe=5 bytes=3004044 frame size=3000410 memory.size=4007696
nframe=5 bytes=4004044 frame size=3000410 memory.size=6007696
NULL
> g1()
nframe=5 bytes=4044 frame size=226 memory.size=0
nframe=5 bytes=1004044 frame size=1000320 memory.size=1000000
nframe=5 bytes=1004044 frame size=1000320 memory.size=2986464
nframe=5 bytes=1004044 frame size=1000320 memory.size=2986464
nframe=5 bytes=2004044 frame size=1000320 memory.size=3493232
NULL
```

## WRITING GOOD CODE

### Use Vectorized Arithmetic

S-PLUS is set up to operate on whole vectors quickly and efficiently. If possible, you should always set up your calculations to act on whole vectors or subsets of whole vectors, rather than looping over individual elements. Your principal tools should be subscripts and built-in vectorized functions. For example, suppose you have a set *x* of thirty observations collected over time, and you want to calculate a weighted average, with the weights given simply by the observation index. This is a straightforward calculation in S-PLUS :

```
> wt.ave <- sum(x*1:30)/sum(1:30)
```

Because you may want to repeat this calculation often on data sets of varying lengths, you can easily write it as a function:

```
wt.ave <-  
function(x) { wt <- seq(along=x); sum(x * wt)/sum(wt) }
```

Here we created weights for each element of *x* simply by creating a weights vector having the same length as *x*. S-PLUS performs its mathematics vectorially, so the proper factor is automatically matched to the appropriate element of *x*.

Even if you only want to calculate with a portion of the data, you should still think in terms of the data object, rather than the elements that make it up. For example, in diving competitions, there are usually six judges, each of whom assigns a score to each dive. To compute the diver's score, the highest and lowest scores are thrown out, and the remaining scores are summed and multiplied by the degree of difficulty:

```
diving.score <-  
function(scores, deg.of.diff = 1)  
{  
  scores <- sort(scores)[ - c(1, length(scores))]  
  sum(scores) * deg.of.diff  
}
```

We use `sort` to order the scores, then use a negative subscript to return all the scores except the highest and lowest.

By now, these examples should be obvious. Yet seeing that these are indeed obvious solutions is a crucial step in becoming proficient at vectorized arithmetic. Less obvious, but of major importance, is to use logical subscripts



instead of `for` loops and `if` statements. For example, here is a straightforward function for replacing elements of a vector that fall below a certain user-specified threshold with 0:

```
over.thresh <-
function(x, threshold)
{
  for (i in 1:length(x))
    if (x[i] < threshold)
      x[i] <- 0
  x
}
```

The “vectorized” way to write this uses the `ifelse` function:

```
over.thresh2 <-
function(x, threshold)
{
  ifelse(x < threshold, 0, x)
}
```

But the fastest, most efficient way is to simply use a logical subscript:

```
over.thresh3 <-
function(x, threshold)
{
  x[x < threshold] <- 0
  x
}
```

(This is essentially what `ifelse` does, except that `ifelse` includes protection against NAs in the data. If your data have no missing values, you can safely use logical subscripts.)

## Avoid for Loops

In Chapter 5, Writing Functions in S-PLUS, we offered the following rule:

- Avoid `for`, `while`, and `repeat` loops.

We provided several examples, with timings, to demonstrate that looping was generally much slower and used much more memory than equivalent constructions performed in a “vectorized” way.

Loops are primarily expensive because of the memory they use; time efficiency is lost mainly when the number of iterations becomes extremely high (on the order of 10,000 or so). Unlike function calls, which generally create a new frame, loops are analyzed within the current frame. After each function call completes, its associated frame disappears and its memory is

freed. After each iteration of a loop, however, only a portion of the memory used by the loop is freed. The process of freeing memory from within loops is called *compaction*. S-PLUS looks at the memory allocated during the loop, checks whether any of the data stored there is being used and if so copies it to another part of memory. When copying is complete, the memory is freed. As the number of iterations increases, this compaction takes somewhat longer each time, primarily because it becomes more difficult to determine what data is actually being used.

Because of compaction, though, loops are more efficient than simply rewriting the loop as the equivalent series of “unrolled” expressions. That is, writing `for (i in 1:100) x=` is more efficient computationally than simply typing `x=` a hundred times. In a function, so that `x` is in a frame, not a database, a body consisting of the line `x` repeated  $n$  times executes faster than a body containing the expression `for (i in 1:n) x` for small  $n$ . As soon as the memory used by the former grows above memory size, however so that paging begins, the former becomes very slow.

It is not always possible to avoid loops in S-PLUS. Two common situations in which loops are required are the following:

- Operations on individual elements of a list. The `apply` family is recommended for this purpose, but all of these functions are implemented (in S-PLUS code) as `for` loops. These functions, are however, implemented as efficiently as possible.
- Operations on vectors that contain dependencies, so that `result[i]` depends on `result[i-1]`. For example, the `cummax` function calculates the cumulative maximum vector, so that

```
> cummax(c(1, 3, 2, 4, 7, 5, 6, 9))
```

```
[1] 1 3 3 4 7 7 7 9
```

The  $i$ th term cannot be calculated until the  $i-1$ st term is known. In these situations, loops are unavoidable. When you must use loops, following a few rules will greatly improve the efficiency of your functions:

- Avoid growing a data set within a loop. Always create a data set of the desired size before entering the loop; this greatly improves the memory allocation. If you don't know the exact size, overestimate it and then shorten the vector at the end of the loop.

- Avoid looping over a named data set. If necessary, save any names and then remove them by assigning NULL to them, perform the loop, then reassign the names.

These rules, and the rationale behind them, are discussed in the following sections.

## Avoid Growing Data Sets

Avoid “growing” data sets, either in loops or in recursive function calls. S-PLUS maintains each data object in a contiguous portion of memory. If the data object grows, it may outgrow the available contiguous memory allotted to it, requiring S-PLUS to allocate a new, different contiguous portion of memory to accommodate it. This is both computationally inefficient (because of the copying of data involved) and memory wasteful (because while the copying is taking place approximately twice as much memory is being used as is needed by the data set). If you know a value can be no larger than a certain size (and that size is not so enormous as to be a memory drag by its very allocation), you will do better to simply create the appropriate sized data object, then fill it using replacement.

For example, consider the following simple function:

```
grow <-
function()
{
  x <- NULL
  for(i in 1:100)
  {
    x <- rbind(x, i:(i + 9))
  }
  x
}
```

The “no grow” version allocates memory for the full 1000 element matrix at the beginning:

```
no.grow <-
function()
{
  x <- matrix(0, nrow = 100, ncol = 10)
  for(i in 1:100)
    x[i, ] <- i:(i + 9)
  x
}
```

The detrimental effect of growing data sets will become very pronounced as the size of the data object increases.

## Avoid Looping Over Named Objects

If you are creating a list in a loop, add component names after the loop, rather than before:

```
. . .
for (i in seq(along=z))
  z[[i]] <- list(letters[1:i])
names(z) <- letters[seq(along=z)]
. . .
```

instead of

```
. . .
names(z) <- letters[seq(along=z)]
for (i in seq(along=z))
  z[[i]] <- list(letters[1:i])
. . .
```

S-PLUS stores the data separately from the names, so extracting data from named data sets takes longer than extracting data from unnamed data sets. Since replacement uses much of the same code as extraction, it too takes significantly longer for named data sets than unnamed. The effect is noticeable even on small examples; on large examples it can be dramatic.

## Keep It Simple!

If you are an experienced programmer, you probably already know that the simpler you can make your program, the better. If you're just beginning, it is tempting to get carried away with bells and whistles, endless bullet-proofing, complicated new features, and on and on. Most S-PLUS functions don't need such elaboration. If you can get a function that does what you want, or most of what you want, reliably and easily, consider your work on the function done. Often, new features are more easily implemented as new functions that call old functions.

Also, because S-PLUS evaluates functions in frames, it is more efficient (in memory usage, not necessarily run time) to write a set of small functions, all of which are called from a top-level function, than to write a single large function. For example, suppose you wanted to write a function to perform a statistical analysis and provide a production-quality graphic of the result. Write one function to do the analysis, another to do the graphics, then call these functions from a third. Such an approach is more efficient, and yields functions which are easier to debug.

Use the simplest data representation possible. Operating on vectors and matrices is much simpler and more efficient than operations on lists. As we have seen, you must use loops if you want to replace list elements. Thus, even simple operations become complicated for lists. The `lapply` and `sapply` functions hide the loops, but do not reduce the computational complexity.

Use matrix multiplication instead of `apply` for simple summaries (sums and means). Matrix multiplication can be 4 to 10 times as fast (see the section Constructing Return Values (page 163)).

## Reuse Computations

If you need the result of a calculation more than once, store the value the first time you calculate it, rather than recalculating it as needed. For most explicit numeric calculations, such as  $x + 2$ , assigning the result is probably second nature. But the same principle applies to *all* calculations, including logical operations, subscripting, and so on.

Conversely, if you know a calculation will *not* be reused, you save memory by *not* assigning the intermediate value. Once named, an object must be copied before being modified. If you name all temporary results, you can essentially replicate your data many times over. Avoiding such replication is often the point of using an S-PLUS *expression* as an argument to a function. For example, consider the following fragment:

```
y <- log(x)
z <- y + 1
```

Here `y` is used only once, but creates an object as large as the original `x`. It is better to replace the two line fragment above with the following single line:

```
z <- log(x) + 1
```

Some times, you may need a result several times during one portion of the calculation, but not subsequently. In such cases, you can name the object as usual, with the result being written to the appropriate frame. At the point where the result is no longer needed, you can use `remove` to delete the object from the frame:

```
y <- log(x)
# numerous calculations involving y
remove(y, frame=2)
```

## Reuse Code

The efficiency of a piece of software needs to be measured not only by the memory it uses and the speed with which it executes, but also by the time and effort required to develop and maintain the code. S-PLUS is an excellent prototyping language precisely because changes to code are so easily implemented. One important way you can simplify development and maintenance is to reuse code, by packaging frequently used combinations of expressions into new functions. For example, many of the functions in Chapter 10, *Object-Oriented Programming in S-PLUS*, allow the user a broad choice of formats for input data (vectors, lists, or matrices). Each function checks the form of the input data and converts it to the format used by the function.

If you take care to write these “building block” functions as efficiently as possible, larger functions constructed from them will tend to be more efficient, as well.

## Avoid Recursion

One common programming technique is even more inefficient in S-PLUS than looping—recursion. Recursion is memory inefficient because each recursive call generates a new frame, with new data, and all these frames must be maintained by S-PLUS until a return value is obtained. For example, our original Fibonacci sequence function used recursion:

```
fib <-
function(n)
{
  old.opts <- options(expressions = 512 + 512 * sqrt(n))
  on.exit(options(old.opts))
  fibiter <- function(a, b, count)
  {
    if(count == 0) b else Recall(a + b, a,
                                count - 1)
  }
  fibiter(1, 0, n)
}
```

---

It can be more efficiently coded as a `while` loop:

```
fib.loop <-  
function(n)  
{  
  a <- 1  
  b <- 0  
  while(n > 0)  
  {  
    tmp <- a  
    a <- a + b  
    b <- tmp  
    n <- n - 1  
  }  
  b  
}
```





---

<b>Working With Many Datasets</b>	<b>826</b>
Many Iterations and the For Function	827
The Advantages of lapply	827
Using the For Function	827
<b>A Simple Bootstrap Function</b>	<b>829</b>
<b>Monitoring Progress</b>	<b>831</b>
Recovery After Errors	832
<b>Summary of Programming Tips</b>	<b>833</b>

In the Chapter 22, Using Less Time and Memory, we described how you could use some knowledge of how S-PLUS organizes its computations in order to write functions that use less time and memory than those you might otherwise write. The main point of that chapter was to use vectorized S-PLUS functions to do as much as possible with each function call. In this chapter, we consider some special problems arising in doing large simulations with S-PLUS. We are interested here in cases where the calculation cannot be vectorized, either because of its complexity or because the vectors would be too large to fit into virtual memory. We will deal with working on many datasets in loop, doing large numbers of iterations (>50,000) in a loop, predicting the amount of time required for a simulation and monitoring its progress, and recovering after errors.

## WORKING WITH MANY DATASETS

Whenever S-PLUS needs to read a new permanent data set it reads the data from disk and stores it until the end of the top-level expression. The next time S-PLUS sees a reference to the data set, it uses the stored version so it doesn't waste time reading the disk. (Writing to disk is typically about 1,000 times slower than writing to main memory.) When asked to save a permanent data set, S-PLUS stores the data set until the end of the expression before writing it, so that although it may alter the same data many times in the top-level expression, it takes the time to write it to disk only once.

While this caching is a good trade-off of memory for speed for most S-PLUS purposes, it can use excessive memory when you must read or write many large permanent datasets in a loop. Each of those datasets is stored in main memory until the end of the current top-level expression. You can prevent this expression-level caching by using the argument `immediate=TRUE` in the `get` and `assign` functions.

For example, you can create a number of large datasets with the following loop:

```
make.data <- function(n=10)
{
  for (i in 1:n)
    assign(where=1, immediate=T,
           paste("x", i, sep="."), rnorm(100000))
}
```

For `n=10`, this requires 1.6 megabytes of virtual memory to execute with `immediate=T` (enough space for 2 copies of one data set), while it requires 10.4 megabytes (13 copies) with `immediate=F`. As you increase the number of datasets, the space required does not rise when you use `immediate=T`. To read the datasets, use `get(immediate=T, ...)` in the same manner:

```
analyze.data <- function(n=10) {
  result <- matrix(NA, nrow=n, ncol=2,
                  dimnames=list(NULL, c("Mean", "Spread")))
  for (i in 1:n)
  {
    x.i <- get(where=1, immediate=T,
               paste("x", i, sep="."))
    result[i, "Mean"] <- mean(x.i)
    result[i, "Spread"] <- diff(range(x.i))
  }
  result
}
```

## Many Iterations and the For Function

An S-PLUS loop tends to slow after many iterations. This is usually not noticeable until about 10,000 iterations, and not terribly important until about 50,000 iterations. The slowdown happens because the functions invoked by “quick calls” (see the section Quick Call Functions (page 793)), such as arithmetic, comparison, and subscripting functions, leave behind a 32-byte chunk of memory with each call, and that memory is not freed until the current function completes. The memory compaction mechanism spends time testing to see if it can free those chunks, and as they build up, it spends most of its time dealing with these chunks that will never be freed. Thus, asymptotically, the time it takes to evaluate a `for` loop is quadratic in the number of iterations, although the quadratic factor has such a small coefficient that you cannot detect it until about the 10,000th iteration of the loop. If your loop does nothing more than call another function, you can delay this slowdown as much as possible.

## The Advantages of `lapply`

Different methods of doing a loop use significantly different amounts of memory. Typically `for` loops use less memory if most of the work is done inside a function. Using `lapply` is better than using a `for` loop. So, replace this:

```
for(i in 1:n){
  some lines of code
  results[i] <- final.result
}
```

with this:

```
f <- function() { some lines of code; final.result }
for(i in 1:n) { results[i] <- f() }
```

or with this:

```
f <- function...
result <- lapply(1:n, f)
```

## Using the For Function

If you will be running a simulation for this many iterations, you may consider using the `For` function. This function creates a file consisting of the contents of the `for` loop repeated `n` times, as separate top-level expressions, so as to avoid both the overhead of long-running `for` loops and the memory overhead of caching data for the duration of a top-level expression. In general, the top-level `for` loop

```
for(i in 1:n)
  r[i] <- func(i)
```

gives the same results as

```
For(i=1:n, r[i]<-func(i))
```

but the latter will not slow down as *n* gets very large.

Because `For` evaluates its expressions in another S-PLUS session, all of the datasets referred to in the `For` expression must be permanent datasets. If you are running `For` from within a function, be sure to assign the data it needs to the working directory before running it. `For` also creates a permanent data set containing the current value of the index variable, in the above example *i*—this overwrites any other permanent variable by that name.

Running each iteration of a loop as a top-level expression may save a bit of memory, but is much slower than doing a group of iterations as a top-level expression. Each top-level expression spends time setting up things and writing results to disk, and by doing more in each expression, we avoid that overhead. Thus, the `For` function has a `grain.size` argument to control how many iterations to do in each top-level iteration. (These iterations are not done in a `for` loop, but are just repeated in the command file, so we don't reclaim memory with the compaction that occurs in loops.) If `grain.size` is too large, memory requirements grow too much, and if it is too small, you waste too much time reading and writing disk files. Since results are saved to disk every `grain.size` iterations, if the computer crashes or S-PLUS must be killed, you only lose the last `grain.size` results. A good setting for `grain.size` is such that each top-level expression takes a few minutes to evaluate, because the overhead of a top-level expression ranges from a fraction of a second to a few seconds, depending on how much data you are accessing from disk. You can predict how long the simulation will take by running `grain.size` iterations and linearly scaling from there.

`For` has a few special arguments that are useful in certain situations. The first argument allows you to specify an expression to evaluate before any of the iterations—remember that `For` evaluates its expressions in a new session of S-PLUS, so you may need to attach some databases or start a graphics device. If the expression given to `For` is large, or the number of iterations is very large, `For` itself may run out of memory while making the command file, or the command file may be too large to fit on your disk. If this is a problem, make a function out of your expression, or save the expression itself and have `For` call the function or evaluate the saved expression.

## A SIMPLE BOOTSTRAP FUNCTION

The following simple bootstrap function demonstrates some ideas for efficient simulations.

The basic idea of the looping procedures below is to do blocks of bootstrap samples, within a for loop. The block size involves a tradeoff. Short block sizes require that `. Random. seed` be saved many times. Because `. Random. seed` is slated for storage on a permanent database, every call to any sampling function uses 81 bytes of memory that are not reclaimed until all functions and loops are done. Large block sizes require storage of a large matrix of indices. The default block size below may need to be reduced if `n` is large, for example:

```
if (n * block * 6 + 32 > options("object.size"))
```

The speed is not very sensitive to the block size, except that large `n` and large `block` require too much of memory, and very small block sizes cause sample to be called too often.

In addition to the basic idea, there are a number of subtle points that affect memory use; these are noted below.

The function is written so that a user interrupt will save as many blocks of bootstrap results as have been completed. It also saves results after exits due to some memory problems.

```
simple.bootstrap <- function(X, FUN, ..., B = 1000, seed = 0, block = 50){
  # demonstration program for nonparametric bootstrapping
  # X is a matrix or data frame, rows are observations
  # FUN(X, ...)
  # This version of bootstrap assumes that FUN() returns a scalar and
  # that B is a multiple of block
  # B bootstrap replications
  # seed is an integer between 0 and 1000
  # block is the block size, number of bootstrap values computed simultaneously
  set.seed(seed) # So results are reproducible
  if(is.null(dim(X))) X <- as.matrix(X)
  n <- nrow(X)
  call.stat <- function(i, X, FUN, indices, ...){
    FUN(X[indices[,i], ], ...)
  }
  # The call.stat() function will be called by lapply() to
  # do the actual bootstrapping
  nblocks <- ceiling(B/block) # number of blocks
  result <- numeric(B) # Create space for results
  indices <- matrix( integer(n*block), nrow=n)
  temp <- 1:block
  on.exit({
    cat("Saving replications 1:", (i-1)*block, " to .bootstrap.results\n")
    assign(".bootstrap.results", replicates, where=1, immediate=T)
  }) # In case function is interrupted
  for(i in 1:nblocks){
    indices[] <- sample(1:n, n*block, T) # Sample the indices
    result[temp+block*(i-1)] <-
      unlist(lapply(temp, call.stat, X, FUN, indices, ...))
  }
  on.exit()
  result
}
```

*Figure 23.1: An interruptible bootstrap function.*

The bootstrap function in Figure 23.1 saves results automatically if you interrupt the function, or if it fails due to trying to allocate too large a dataset for indices.

## MONITORING PROGRESS

After your simulation has been running for a while, you may want to know how far it has gotten. You cannot safely interrupt its progress, examine its status, and resume execution. You should include some code to periodically record its status in a file. It should be in a file rather than in an S-PLUS data set so that you can look at it without using S-PLUS and so that it will be written to disk immediately. In addition, appending some text to a large file is quicker than reading a large S-PLUS data set with that information, adding to it, then writing it out again. This status information can include the iteration number, a summary of results for each iteration, and enough information to restart the simulation at the last checkpoint if S-PLUS or the computer should crash during the simulation.

You can use `options("error")` or `on.exit()` to put a message into the status file when something goes wrong:

```
analyze.data <- function(n=10, logfile)
{
  result <- matrix(NA, nrow=n, ncol=2,
                  dimnames=list(NULL, c("Mean", "Spread")))
  dimnames(result) <- list(NULL, c("Mean", "Spread"))
  if (!missing(logfile))
    on.exit(cat(file=logfile, append=T,
              "Error in iteration", i, "\n "))
  for (i in 1:n)
  {
    x.i <- get(where=1, immediate=T,
               paste("x", i, sep=". "))
    result[i, "Mean"] <- mean(x.i)
    result[i, "Spread"] <- diff(range(x.i))
    if (!missing(logfile))
      cat("result[", i, ", ] =", result[i, ], "\n ",
          file=logfile, append=T)
  }
  if (!missing(logfile))
    on.exit() # cancel the error report
  result
}
```

If we run this in a directory that only contained `x.1` and `x.2`, we get:

```
> analyze.data(n=10, logfile="/tmp/log")

Error in get.defaultt(where = 1, immediate = T, pas...:
Object "x.3" not found
Dumped
```

The log file contains:

```
result[ 1 , ] = 0.672007595235482 0.916557230986655
result[ 2 , ] = 0.509014049381949 0.945636332035065
Error in iteration 3
```

## Recovery After Errors

For a variety of reasons, a simulation function may crash after doing many iterations. Sometimes a rare sequence of random variables may trigger a bug in the function, or the function may run out of memory, or the computer may have to be rebooted for unrelated reasons.

So you don't waste the time tied up by the iterations already done, you should write the function so it can be restarted at the point where it died. This involves making sure the current state of the simulation is saved on disk periodically, and that you can use this state to restart the function at a point close to where it stopped.

Often, the state information required is just the iteration number. If you are using random number generators, the seed of the generator, `.Random.seed`, must be saved as well. Its value is updated every time some random numbers are generated, but, like any other data set, it is not committed to disk until the successful completion of a top-level expression.



# SUMMARY OF PROGRAMMING TIPS

Some of the key points from this chapter are:

- It is best to reduce the number of calls to random functions like `runif`, because repeatedly changing `.Random.seed` is very inefficient.
- `lapply` is currently the single most efficient way to do looping. There are some subtle ways to make `lapply` slightly more efficient; to use a first argument "temp" (can have any name) which is the vector `1:n` with blank names. Thus we might recommend replacing

```
for(i in 1:n) result[i] <- f(i)
```

with

```
temp <- 1:n
result <- unlist( lapply( temp, f ), recursive=F)
```

The use of `recursive=F` is in case `func` returns a list.

- a hybrid of `for` and `lapply` can be more efficient than using just either one in simulations like bootstrapping, where trying to do everything within a single `lapply` requires saving huge matrices of random indices.
- listen to your hard disk; if it is running a lot you are probably paging, and things will get *veryslow*. In this case it is better to do the simulation in smaller parts, e.g. using `For`, to prevent memory requirements from growing to the point that paging occurs.



---

<b>S-PLUS Syntax and Grammar</b>	<b>836</b>
Literals	837
Calls	839
S-PLUS Evaluation	839
Internal Function Calls	840
Generic Dispatch	842
Assignments	843
Conditionals	844
Loops and Flow of Control	844
Grouping	846

To this point, we have for the most part simply assumed that S-PLUS knows how to translate what you type into something it can understand, and then knows what to do with that something to produce values and side effects. In this chapter, we describe precisely how S-PLUS works, from parsing input to printing values. Together with the information in Chapter 21, Data Management, this chapter provides you with a full account of the machinery that runs the S-PLUS environment.

# S-PLUS SYNTAX AND GRAMMAR

When you are using the S-PLUS Commands window, your keyboard's standard input is directed immediately to the S-PLUS parser, which converts the characters you type into expressions evaluable by the S-PLUS evaluator. (If you start in batch mode, the parser's input is provide by the *in* file.) When you press ENTER (or the parser encounters the next line of a file), the parser checks to see if the parsed text constitutes a complete expression, in which case the expression is passed to the evaluator. If not, the parser prompts you for further input with a continuation prompt (usually `+` ). A semicolon (`;`) can also be used to terminate an expression, but if the expression is incomplete S-PLUS issues an error message. A *complete expression* is any typed expression that falls into one of the seven broad classes, shown in Table 24.1.

Table 24.1: *Classes of expression.*

Class	Expression
Literals	Literals are the simplest objects known to S-PLUS. Any individual number, character string, or name is a literal.
Calls	Perhaps the most common S-PLUS expression, a call is any actual use of a function or operator.
Assignments	Assignments associate names and values.
Conditionals	Conditionals allow branching depending upon the logical value of a user-defined condition.
Loops	Loops allow iterative calculations.
Flow-of-control statements	Flow-of-control statements direct evaluation out of a loop or the current iteration of a loop
Grouping statements	Grouping allows you to control evaluation by overriding the default precedence of operations or by modifying the expected end-of-expression signal.

A complete expression may contain many expressions as sub-expressions. For example, assignments often involve function calls, and function calls usually involve literals. In the following sections, we describe the complete syntactic, lexical, and semantic rules for each of the seven classes.

## Literals

All literals fall into one of the following six classes:

### 1. Numbers.

Numbers are further subdivided as follows:

1. *Numeric*. Numeric values represent real numbers. Numeric values can be expressed in any of the following forms:
  - As ordinary *decimal* numbers, such as 11, -2.3, or 14.943.
  - As S-PLUS expressions that generate real values, such as `pi`, `exp(1)`, or `14/3`.
  - In scientific notation (exponential form), which represents numbers as powers of 10. For example, 100 is represented as `1e2` in scientific notation and 0.002 is `2e-3`.
  - As the IEEE special value `Inf`. This value may be either assigned to objects or returned from computations. `Inf` represents infinity and results from, for example, division by zero.

```
> c(5/0, -2.1/0)
```

```
[1] Inf -Inf
```

#### Internal storage format

Numeric data is stored internally in one of three *storage modes*: "integer", "single", and "double". These storage modes are important when declaring variables in C and Fortran code. Use the `storage.mode` function to view the storage mode of a numeric data object.

2. *Complex*. Complex values are similar to numeric values, except they represent *complex* numbers. Complex values are specified in the form `a+bi` (or simply `bi`), where `a` is the real part and `b` is the imaginary part. The imaginary part `b` must be expressed in decimal form, and there must be no spaces or other symbols between `b` and `i`. In particular, there is no `*` between `b` and `i`.

## 2. Strings.

Strings consist of zero or more characters typed between two apostrophes (') or double quotes ("). The table lists some special characters for use in string literals. These special characters are for carriage control, obtaining characters that are not represented on the keyboard, or delimiting character strings.

*Table 24.2: Special Characters.*

Character	Description
<code>\t</code>	tab
<code>\b</code>	backspace
<code>\n</code>	new line
<code>\"</code>	" (double quotes)
<code>\'</code>	' (apostrophe)
<code>\###</code>	ASCII character as an octal number, # is in the range 0 to 7

## 3. Names.

Syntactic names are unquoted strings, consisting of alphanumeric characters and periods (.), that do not start with a number. (As described in Chapter 21, Data Management, objects can be named using virtually any quoted string. Only syntactic names, however, are directly recognized by the parser. Thus the need for the more general functions `get`, `assi gn`, etc.

## 4. Comments.

Anything typed between a # character and the end of a line is a comment. Comments are attached to the S-PLUS object created by the parser, but not all objects can have comments attached, so that comments in functions may not be printed in the position in which they were originally inserted.

## 5. Functions.

A function consists of the word `function`, a parenthesized set of formal arguments (this may be empty), and a complete expression. Function literals, in general, appear only during function assignment, on the right side of the assignment arrow.

## 6. Symbolic constants.

S-PLUS reserves the following symbolic constants:

`TRUE T FALSE F NULL NA Inf NaN`

These constants are syntactically names, but S-PLUS prevents assignment to them.

### Reserved names

Attempts to associate objects with these names results in an error:

`if else for while repeat next break in function return`

The names of built-in symbolic constants are also reserved:

`TRUE T FALSE F NULL NA Inf NaN`

## Calls

Most complete expressions involve at least one call. Calls are the expressions that do most of the work in S-PLUS. They fall into three basic categories:

1. *Simple calls.* Simple calls are calls of the form: *function-name(arglist)*
2. *Operations.* Operations are calls using infix or unary operators. Examples include `2 * 3` and `-5`. The parser interprets operations as simple calls of the form *"op"(args)*. Thus, for example, the expression `5 + 4` is interpreted by the parser as the simple call *"+"(5, 4)*.
3. *Subscripting.* Subscripting extracts elements from atomic and recursive data and also extracts named components from recursive data. The parser also interprets subscript expressions as simple calls, converting expressions such as the following:  
`object[i] object$i object[[i]]`  
into equivalent function calls  
`"["(object, i) "$"(object, i) "[["(object, i)`  
Thus all calls have essentially the same evaluation process.

## S-PLUS Evaluation

Evaluation of simple calls is the principal activity of the S-PLUS evaluator. Most simple calls generate at least two frames—the top-level expression frame and the function's evaluation frame. Functions which consist solely of calls to `.Internal`, however, do not generate an evaluation frame. Calls to these functions are called *quick calls*, because they avoid the overhead of creating an evaluation frame (which isn't needed because there are no S-PLUS assignments in the function body, and the arguments are matched in the top-level expression frame).

To evaluate the call, S-PLUS first finds the definition of the function. The formal arguments in the function definition are matched against the actual arguments in the function call, and S-PLUS creates a new frame containing the argument expressions (unevaluated) and information on which formal arguments were not specified in the actual call.

When a new frame is created the following internal lists are updated:

- *The frames list.* The new frame is appended at the end of the existing frames list, which is accessible via the `sys.frames` function. The new frame is frame number `Nframe`.
- *The system calls list.* The current call is appended at the end of the system calls list, which is accessible via the `sys.calls` function.
- *The list of parent frames.* The number of the parent frame of the new frame is appended at the end of the list of parent frames, which is accessible via the `sys.parents` function.
- *The list of argument counts.* This list, which is not directly accessible, holds a count of actual arguments for each frame in the list of frames.
- *The list of return flags.* This list, also not directly accessible, holds a logical value for each frame, specifying whether a return expression has been evaluated in the frame. Once the frame has been built, the evaluator extracts the body of the function definition and evaluates that. The value of the function call, as previously mentioned, is simply the value of the last expression in the function body, unless the body includes a return expression. If the body includes a return expression, that expression stores a return value in the internal return list, and this value is returned as the value of the function call.

The value is returned to the parent frame, and the evaluation frame is broken down.

## Internal Function Calls

Functions defined as calls to `.Internal` are evaluated somewhat differently from calls to functions written wholly in S-PLUS or to functions using the interfaces to C, Fortran, or the operating system. Such functions do not generate their own evaluation frame; instead, in most cases the evaluator



evaluates the arguments to the call and passes these to the internal C code, which performs the computations and returns a pointer to an S-PLUS object that is the value of the call.

A few internal functions do not evaluate their arguments at the top level. These functions pass the entire unevaluated expression to the internal C code, which then performs the computations. Most of these functions, which are listed below, use the form of the expression itself to determine how the evaluation is to proceed:

- `&&`, `||`

The Control And (`&&`) and Control Or (`||`) operators evaluate their first argument as a logical condition. If the first argument is `TRUE`, Control And proceeds to evaluate its second argument, while Control Or immediately returns `TRUE`. If the first argument is `FALSE`, Control And immediately returns `FALSE`, while Control Or proceeds to evaluate its second argument.

- `switch`

When the evaluator encounters a call to `switch`, it evaluates the first argument. If the value is of mode "character", the evaluator matches it against the names of the remaining arguments, and if a match is found, evaluates the first non-missing argument that follows the match. If no match is found, the first unnamed argument is evaluated. If there is no match and there are no unnamed arguments, `switch` returns `NULL`. If the value of the first argument is of mode "numeric", the value is matched against the sequence `1:nargs()-1` corresponding to the remaining arguments. If a match is found, the evaluator evaluates the first non-missing argument that follows the match. Otherwise, `switch` returns `NULL`.

- `missing`

The `missing` function takes a formal argument to the current function and returns `FALSE` or `TRUE` depending on whether there was an actual argument corresponding to that formal argument.

- `expression`, `substitute`

Both `expression` and `substitute` return unevaluated expressions, suitable for passing to the evaluator. The `substitute` function takes its unevaluated first argument and tries to replace any name within it by the value of an object with the same name in a specified list or frame, by default the local frame. Unlike most functions which use a frame to match names and objects, `substitute` searches the frame list back-to-front, so that arguments are matched in their unevaluated form. Names that are not matched are left

alone. The mode of the returned expression is the mode of the unsubstituted expression. By contrast, `expression` always returns an object of mode "expression", which is a list of one or more expressions.

- `UseMethod`, `NextMethod`

The `UseMethod` and `NextMethod` functions are the principal components of S-PLUS's generic dispatch mechanism. See the section *Generic Dispatch* (page 842) for complete details.

- `Recall`

`Recall` makes recursion independent of the function's name, and permits recursive function definitions inside other functions. The evaluator finds the definition of the function calling `Recall`, creates a new frame by matching the arguments to `Recall`, then evaluates the recalled definition in the new frame.

# Generic Dispatch

*Generic dispatch* is the mechanism by which object-oriented methods are called from user calls to generic functions. Thus, a user call, such as `print(x)`, with `x` a factor, results in a generic dispatch to the method `print.factor`. The object `x` is said to *drive* the dispatch. (Some methods are also dispatched directly via the `.Internal` interface.) Calls to generic functions are evaluated as if they were calls to specific methods. That is, an evaluation frame is created for the generic call and this frame becomes the evaluation frame for the method. The arguments to the method are matched as if the call had been directly to the method.

*Table 24.3: Special objects in the evaluation frame.*

Object	Description
<code>.Class</code>	The <code>class</code> attribute corresponding to the current method. If <code>NextMethod</code> is called by the method, <code>NextMethod</code> saves <code>.Class</code> as the previous attribute to the <code>.Class</code> object, then sets <code>.Class</code> to the inheriting class.
<code>.Generic</code>	The name of the generic function.
<code>.Method</code>	The name of the method being used.
<code>.Group</code>	The name of the function group (Summary, Ops, or Math), if applicable.

Calls to `NextMethod` do not generate a new frame; instead, the existing frame is used as the frame for evaluating the call to the next method. If the object driving the dispatch has been modified before the call to `NextMethod`, `NextMethod` acts upon the modified object, but the modification *does not* alter the choice of method dispatched by `NextMethod`. If you want the modified object to drive the dispatch, you should call the generic again, rather than using `NextMethod`.

The default call to `NextMethod` is evaluated in much the same way as if the call were directly to the dispatched method. Arguments are passed from the current method to the next method with their values at the time `NextMethod` is called. (Unevaluated arguments remain unevaluated and missing arguments remain missing.) Arguments passed through the ... mechanism are passed with the correct name to the next method.

## Assignments

Syntactical assignments are expressions of mode `"<-"` or `"<<-"`. Simple assignments are assignments with a name or string on the left hand side of the assignment arrow. *Replacements* are assignments with a function call on the left hand side, where the function call may be subscripting operation. *Nested replacements*, in which several function calls are nested on the left hand side, are common. For example, consider the following expression:

```
di mnames(state.x77)[[2]][8] <- "Land Area"
```

This expression extracts the eighth element of the second list element in the `di mnames` attribute of the `state.x77` data set and replaces it with the value on the right hand side.

Simple assignments are evaluated simply: if the mode of the assignment is `"<<-"` or the assignment is in frame 1, the name or string on the left hand side is associated with the value on the right hand side in the working data, otherwise the association is added to the current frame. If the assignment is to the working data, the dictionary is updated to reflect the new association.

Simple replacements of the form `f(x) <- value` are somewhat more complicated: the extraction function `f` is replaced by the corresponding *replacement function* `"f<-"` to create a function call of the following form:

```
"f<-"(x, value)
```

This function call is evaluated, then the name `x` is associated with the function call's return value. The frame in which the replacement is performed is determined as for simple assignments: in the working directory if the replacement is evaluated in frame 1, in the local frame otherwise.

Nested replacements are expanded internally into an equivalent series of simple replacements. For example, the expression

```
di mnames(state.x77)[[2]][8] <- "Land Area"
```

might be expanded as follows:

```
assi gn("..a", di mnames(state.x77)[[2]], frame=Nframe)
..a[8] <- "Land Area"
assi gn("..b", di mnames(state.x77), frame=Nframe)
..b[[2]] <- ..a
di mnames(state.x77) <- ..b
```

## Conditionals

Conditionals are expressions of the form `if (cond) expr1 else expr2`. The `else expr2` may be omitted. The condition *cond* may be any expression that evaluates to a single logical value TRUE or FALSE. Common test conditions include tests on the mode of an object and simple comparison tests using the operators `>`, `>=`, `==`, `<`, and `<=`.

The evaluation of conditionals begins with evaluation of *cond*. If the evaluation does not yield a logical value, or yields multiple values, an error message is returned. If *cond* is true, *expr1* is evaluated. If *cond* is false, and the `else` has not been omitted, *expr2* is evaluated. Otherwise, S-PLUS returns NULL.

## Loops and Flow of Control

S-PLUS supports three types of loops: `repeat`, `while`, and `for`. The three loops are evaluated similarly, primarily differing in how they are exited. The `while` and `for` loops have specific completion tests, but may also be interrupted by flow-of-control instructions. Loops involving `repeat`, however, have no specific completion test, so in general must be exited using the flow-of-control instructions.

Because S-PLUS has no explicit jumps or GOTOs, flow of control is handled by flags which are checked by the evaluator each time it is recursively called. There are three flags—`Break. flag`, `Next. flag`, and `Return. flag`—which correspond to the three flow-of-control instructions `break`, `next`, and `return`. As we saw in the section `Calls` (page 839), when a return expression is evaluated, the `Return. flag` is set to TRUE and the return value is stored in the `Return` list. Similarly, within a loop, when a `next` or `break` flag is evaluated, the corresponding flag is set to TRUE. On the next recursive call to

the evaluator, it checks the three flags and breaks out of the loop if either the `Break.flag` or `Return.flag` is `TRUE`. If the `Next.flag` is `TRUE`, the evaluator skips to the next iteration of the loop.

This flag-checking essentially defines the evaluation of `repeat` loops. The `repeat` loop simply evaluates its body, checks the three flags, and continues until one of `Break.flag` or `Return.flag` is `TRUE`.

The value of a loop expression is the value of the last completed iteration, that is, an iteration not interrupted by `break` or `next`. Iterations interrupted by `return`, of course, have the value specified by the `return` expression.

Evaluation of a `while` loop corresponds to evaluation of a `repeat` loop with a conditional corresponding to the `while` condition at the top of the loop. For example, the following `while` loop:

```
while(n < 3)
{
  cat("hello\n ")
  n <- n + 1
}
```

is interpreted by the evaluator as the following `repeat` loop:

```
repeat
{
  if (n >= 3) break
  cat("hello\n ")
  n <- n + 1
}
```

Unlike `repeat` and `while` loops, `for` loops introduce the notion of a *looping variable* that takes on a new value during each iteration of the loop. To maintain the looping variable correctly, and also to permit arbitrary names for the looping variable, evaluation of `for` loops is somewhat more complicated than that of the other loops.

First, the name of the looping variable is extracted, and the current value associated with that name is stored. A vector of values that the looping variable will take on is created, and then the looping begins. When the loop is completed, the name of the looping variable is reassigned to the stored value, if any.

Loops are evaluated in the local frame—no special loop frame is created. To control memory growth during loop evaluation, the evaluator has a mechanism for *compacting* loops. A trigger value for compaction is specified by the S-PLUS option `compact`; by default, this value is `1e5`. When the loop evaluation has allocated more memory than this value, the compaction mechanism is triggered. During compaction, the evaluator checks all arenas allocated since the beginning of the loop or since the last compaction and

determines which stored values are still being used. It copies all such values to another part of memory, then frees all the checked arenas. This compaction means that loops within functions make more efficient use of memory than the “unrolled” loops they are equivalent to. That is:

```
for (i in 1:300000) cat("hello\n ")
```

will use less memory than

```
cat("hello\n ") cat("hello\n ") . . .
```

repeated 300000 times.

## Grouping

Two types of grouping affect S-PLUS evaluation:

1. *Braces* group sets of expressions separated by semicolons or newlines into a single expression of mode "{". Braced expressions make up the body of most function definitions and iterative loops. The braced expression is evaluated by evaluating each subexpression in turn, with the usual checks for `return`, `break`, and `next`. The value of the braced expression is the value of the last evaluated subexpression, or the value stored in the `Return` vector if the braced expression includes an evaluated `return`.
2. *Parentheses* are used to group subexpressions to control the order of evaluation of a given expression. Syntactically, they differ from braces in that they can surround only one complete expression (that is, an expression terminated by a semicolon or newline). For grouping purposes, braces may be used anywhere parentheses are.

---

<b>Outline of the Validation Routines</b>	<b>848</b>
<b>Running the Tests</b>	<b>851</b>
<b>Creating Your Own Tests</b>	<b>853</b>

Using the S-PLUS function `val i date`, you can check the accuracy of S-PLUS algorithms and routines as they run on your system. The `val i date` function draws upon a suite of validation tests which refer to published examples of both typical and extreme data sets in a variety of statistical routines and distribution lookups. You can also create your own validation tests and call them with `val i date`, using the supplied test files as templates.

This chapter details the coverage of the supplied routines, describes the syntax of `val i date`, and gives examples of its output. The last section of the chapter shows how to create your own tests.

# OUTLINE OF THE VALIDATION ROUTINES

Table 25.1 describes the coverage of the built-in tests. Tests are grouped in the left-hand column into functional areas. The middle column lists the high-level S-PLUS functions tested. The right-hand column contains a brief description, when relevant, of the different cases covered by the tests. You can expand the scope of the tests provided, or write routines for other S-PLUS functions—including your own functions.

*Table 25.1: Table of Built-in Validation Tests.*

Functional Area	High-Level Functions	Test Cases
Descriptive Statistics	mean medi an var standard dev. (var, sqrt)	Descriptive statistics calculated for various numeric vectors, including very large and very small numeric values.
	cor	Simple correlation coefficient between two vectors.
Regression	l m	Simple linear regression including hypothesis testing; multiple regression including hypothesis testing; polynomial regression; multiple regression, no intercept, including hypothesis testing.
	gl m	Logistic regression including chi-square goodness of fit; log-linear regression; gaussian linear model.
	nl s	Michaelis-Menten model with 4 parameters.
	l me	Random intercept; random intercept with AR(1) errors; random factors; random factors with AR(1) errors.
Hypothesis Tests	bi nom. test	2-sided alternative, both 1-sided alternatives.



*Table 25.1: Table of Built-in Validation Tests.*

	prop. test	2-sample test, 2-sided alternative.
	chi sq. test	2x2 contingency table, with and without continuity correction; 4x5 contingency table, with and without continuity correction.
	t. test	1-sample, 2-tailed and both 1-sided alternatives; 2-sample, 2-tailed and 1-tailed (less) alternative; paired test.
	wilcox. test	1-sample signed rank test, 1-sided (greater) alternative; 2-sample rank sum test using large sample approximation, no continuity correction; paired-sample signed rank test, 2-sided alternative.
	cor. test	Pearson's product moment correlation coefficient, t-test for significance, 2-sided alternative; Spearman's rank correlation; Kendall's tau statistic.
	fisher. test	Exact test for 2x2 contingency table, 2-sided alternative.
	mantel haen. test	2x2x3 contingency table with continuity correction.
	mcnemar. test	2x2 table.
	kruskal . test	1-way layout with 3 groups, 2-sided alternative.
	friedman. test	2-way unreplicated layout.
	var. test	2-tailed variance ratio test, 2-tailed and 1-tailed (less) alternative.
	ks. gof	1-sample test, 2-sided and both 1-sided alternatives.
	chi sq. gof	Testing continuous data from normal distribution with given $\mu$ and $\sigma$ ; expected values the same in each of the specified number of classes; testing discrete data from normal distribution; expected values calculated from specified class endpoints.

*Table 25.1: Table of Built-in Validation Tests.*

ANOVA	<code>aov</code>	1-way balanced layout with replicates; 2-way layout without replication; 2-way layout with replicates; complete balanced block design, with and without 2-way interaction term; complete unbalanced block design, no interaction term; $2^3$ design; split-plot design; repeated measures.
	<code>manova</code>	Simple; repeated measures.
Multivariate	<code>princomp</code>	Component calculated based on both correlations and covariances.
	<code>factanal</code>	Principal component extraction with varimax rotation; maximum likelihood factor solutions with varimax rotation.
Survival Analysis	<code>survfit</code>	Kaplan-Meier estimator with 2 groups.
	<code>survdiff</code>	Log-rank test.
	<code>coxph</code>	Simple survival data set, modeled by group; multivariate failure time data with repeated failures modeled as strata; Andersen-Gill fit on multivariate failure time data;
	<code>survreg</code>	gaussian, Weibull, and exponential models.
Statistical Distributions	<code>pnorm</code> , <code>qnorm</code>	z-values between -3.0902 and 3.0902.
	<code>pchisq</code>	Probabilities from 0.001 to 0.995 and degrees of freedom from 1 to 25 by 5 and 30 to 100 by 10.

# RUNNING THE TESTS

To run validation tests using the `validate` function, no more is required than the simple call

```
> validate()
```

which runs all tests available in the directory `$SHOME/splus/lib/validate`. This is where the built-in files are found after installing S-PLUS. The `validate` function invisibly returns `TRUE` when all tests run successfully and `FALSE` otherwise. Assuming all tests run successfully, output ends with the following summary.

VALIDATION TEST SUMMARY:

All tests PASSED

To run specific built-in tests, indicate a selection with the `file` argument. The choices correspond to the Functional Areas in Table 25.1: `sdistrib`, `descstat`, `regress`, `hypotest`, `anova`, `multivar`, and `survival`. For example, the following runs the analysis of variance and regression tests. In this example, the argument `verbose` is set to `TRUE`, and the details of each test are returned by `validate`.

```
> validate(file=c("anova", "hypotest"), verbose=T)

----- Analysis of Variance -----
test:
{
#Functions: aov, summary.aov
#Data: Sokal and Rohlf, Box 9.4 p. 220
#Reference: Sokal, R. and F. J. Rohlf. 1981.
#Biometry, 2nd edition.
#W. H. Freeman and Company
#Description: 1-way balanced layout with 5 treatments,
#10 replicates/trtm;
#check df's, sum of squares and mean squares; check F value
#and p-value using a different tolerance
  tol1 <- 0.005
  tol2 <- 0.04
  y <- c(75, 67, 70, 75, 65, 71, 67, 67, 76, 68,
        57, 58, 60, 59, 62, 60, 60, 57, 59,
        61, 58, 61, 56, 58, 57, 56, 61, 60,
        57, 58, 58, 59, 58, 61, 57, 56, 58,
```

```
57, 57, 59, 62, 66, 65, 63, 64, 62,
65, 65, 62, 67)
y.treat <- factor(rep(1:5,
                     c(10, 10, 10, 10, 10)))
y.df <- data.frame(y, y.treat)
y.aov <- aov(y ~ y.treat, data = y.df)
a.tab <- summary(y.aov)
all(c(a.tab$Df == c(4, 45), abs(a.tab$
  "Sum of Sq" - c(1077.32, 245.5)) <
  tol1, abs(a.tab$"Mean Sq" - c(269.33,
  5.46)) < tol1, abs(a.tab$"F Value"[1] -
  49.33) < tol2, a.tab$"Pr(F)"[1] <
  0.001))
}
. . .
All tests PASSED
```

#### VALIDATION TEST SUMMARY:

```
Test Directory: /homes/qpe/spl us/li b/val idate
File anova: All tests PASSED
File regress: All tests PASSED
```

To run customized tests, you must first write a test file. For information on creating a test file, see the section *Creating Your Own Tests* (page 853). To use `validate` with your test files, specify the name of your files with the argument `file` and the directory containing them with the argument `test.loc`.

```
> validate(file=c("anova1", "hypotest1"),
+ test.loc="/homes/mydi r/Spl uswork/val di r")
```

Should any validation test fail, the details of the failed tests are returned, followed by a notification such as the following.

```
[1] "***** Test FAILED *****"
1 test(s) FAILED
```

#### VALIDATION TEST SUMMARY:

```
Test Directory: /homes/mydi r/Spl uswork/val di r
File anova1: All tests PASSED
File hypotest1: 1 test(s) FAILED
```

---

## CREATING YOUR OWN TESTS

The built-in validation test files are set up as loop tests. A loop test consists of a set of S-PLUS commands that are grouped as a braced expression, and returns either TRUE or FALSE. A validation test file contains one or more of these loop-style expressions.

Custom test files can be created either from scratch or by using an existing file as a template. The test files included as part of S-PLUS can be found in `$SHOME/splus/lib/validate`. If the variable `$SHOME` is not set in your environment, use the path returned by calling `getenv("SHOME")` within S-PLUS.

The basic construct of a test file includes an expression, which is defined by a set of commands enclosed in `{}`. The `validate` function expects the first expression to be a comment; if the first expression is a test, this test will not be accounted for in the summary. Each expression is self-contained and evaluates to either TRUE or FALSE. Commonly, a validation test sets up some data and calls one or more S-PLUS functions. Results of the function calls are then compared to accepted results, within some tolerance. The S-PLUS functions `all` and `any` are often used to set up the comparison expression that evaluates to either TRUE or FALSE.



# INDEX

## Symbols

...argument 89  
 .C fz 761  
 .Data directories 770  
 .First function 759  
 .First.lib fz 493, 739  
 .Fortran fz 761  
 .Internal functions 415, 793, 839  
 .Last function 760  
 .Last.lib fz 493, 739  
 .Last.value object 128  
 .Random.seed 832  
 \_Help directory 489  
 \_Help subdirectory 487

## Numerics

2-D Line and Scatter Plots 223  
 3D Contour Plots 231  
 3D Line and Scatter Plots 229  
 3D plot palette 229

## A

abline fz 260, 288  
 About Multipanel Display 361  
 acf fz 178  
 Action argument 476  
 add argument 311  
 Adding 2D Axes 232  
 adding a class 465  
 adding a legend 263  
 Adding an ActiveX control to a dialog 675  
 adding information on object 462  
 adding menu items 475  
 adding new data to a plot 261

adding straight lines to a scatter plot 260  
 adding text to existing plot 262  
 adj parameter 298  
 Advise fz 533  
 aggregate fz 82  
 along argument 48  
 amatch fz 780, 783  
 And and Or operations 126  
 angle argument 266  
 annotation objects 212  
 aov fz 334  
 application objects 510  
 apply fz 164  
 apply fz 818, 821  
 ar fz 776  
 ar.burg fz 191  
 Area Plots 228  
 Arg fz 122  
 argnames fz 773  
 argument ... 89  
 array fz 56  
 arrays 43, 55, 133  
 arrows fz 315  
 as.complex fz 122  
 as.data.frame fz 71  
 as.data.frame.array fz 391  
 as.data.frame.ts fz 392  
 as.double fz 129  
 as.integer fz 129  
 as.matrix fz 129  
 as.single fz 129  
 as.vector fz 129  
 ASCII files 100  
 ASCII:specifying a format string 100  
 aspect argument 339, 376  
 aspect fz 394  
 assign fz 128, 167, 406, 798, 808, 826  
 at argument 302, 354

attach function 758  
 attach fz 337  
 attr fz 171  
 attributes 44  
 attributes fz 45, 90  
 attributes of S-PLUS 812  
 auto.dat data set 105, 159  
 auto.stats data set 287  
 axes objects 212  
 axes parameter 303  
 axis fz 303  
 Axis2dX object 232  
 Axis2dY object 232

## B

Bar Plots 226  
 bar.fill parameter 383  
 barchart fz 348  
 barley data set 361  
 barplot fz 152  
 barplot fz 265  
 batch mode 742  
 batch processing 754  
 between argument 395  
 border argument 389  
 Box Plots 226  
 break statement 146, 147  
 Break.flag 844  
 breaks argument 66, 153, 320  
 browser fz 202, 407, 773  
 building block functions 822  
 built-in vectorized functions 816  
 buttons  
     modifying 640  
 bwplot fz 343  
 by fz 82, 85  
 byrow argument 52

## C

C code 762  
 c fz 152  
 call fz 776

call.ole.method fz 522  
 callback function 694  
 CancelTransaction function object 511  
 car.miles data set 253  
 cat function 548  
 cat fz 108, 109, 166, 167, 180  
 categorical variables 62  
 cbind fz 51, 71, 76, 244  
 cex argument 325, 347  
 cex parameter 297, 377, 380  
 CF\_TEXT format 535  
 Changing the Text in Strip Labels 376  
 channel number 533  
 character data type 88  
 character fz 49  
 character mode 135  
 character values 44  
 circle fz 412  
 city.name data set 316  
 city.x data set 316  
 city.y data set 316  
 class attribute 62, 88, 90, 129, 407  
 class fz 409  
 ClassInfo object 483  
 ClassName 507  
 clear.frame fz 794  
 client 533  
 cloud fz 357  
 codes fz 63  
 col argument 325  
 col parameter 258, 377  
 column-major order 133  
 columns argument 388  
 combining data frames 76  
     by column 76  
     by row 78  
     merging 79  
     rules 88  
 command argument 476  
 command line 742  
     script files 742  
     switches 742, 754  
     tokens 744  
     variables 742



- 
- commands
    - object name 429
  - Commands window 207
  - comments 138, 838
  - Common error conditions when using ActiveX controls in S-PLUS 678
  - Commonly-Used S-PLUS Graphics Functions and Parameters 380
  - complete function instruction 190
  - complete instruction 190
  - complete loop instruction 190
  - complex fz 49
  - complex numbers 837
  - complex values 44
  - composite figures 313
  - conditioned Trellis graphs 219
  - Conditioning 3D Graphs 232
  - Conditioning On Discrete Values of a Numeric Variable 368
  - Conditioning On Intervals of a Numeric Variable 370
  - conditioning variables 361
  - Conj fz 122
  - Connect fz 533
  - contour fz 282
  - Contour Plots 228
  - contourplot fz 354
  - Controlling the Pages of a Multipage Display 367
  - coordinate systems 211
  - copying external GUI files 492, 736
  - copying help files 493, 738
  - corn.rain data set 315
  - CPU time 810
  - create.gui.menuGaussfit fz 492
  - create.menu.gaussfit fz 475
  - create.ole.object fz 521
  - create.toolbar.gaussfit fz 476, 492
  - CreateConditionedPlots 509
  - CreateConditionedPlots – SeparateDataGallery 509
  - CreateConditionedPlotsGallery 509
  - CreateConditionedPlotsSeparateData 509
  - CreateObject fz 505
  - CreatePlots 509
  - CreatePlotsGallery 509
  - creating a list in a loop 820
  - creating a sophisticated tabbed dialog 479
  - creating and modify toolbars 636, 649
  - creating and modifying buttons 640
  - creating directories 491, 735
  - Creating HTML Output
    - Graphs 113
    - Tables 112
    - Text 113
  - creating the help file 487
  - creating toolbars 476
  - csi parameter 298
  - cummax fz 818
  - customized graphical user interface 474
  - customizing the context menu 483
  - customizing the dialog 477
  - cuts argument 355
- 
- ## D
- D fz 766
  - data argument 337
  - data array 279
  - data directory 758
  - data frames 69
    - adding new classes of variables 88
    - applying functions to subsets 82
    - attributes 90
    - combining objects 74
    - dimnames attribute 73
    - row names 73
    - rules for combining objects 88
  - data objects 69
  - data sets
    - growing 819
  - data.class fz 88
  - data.frame data type 88
  - data.frame fz 71
  - databases 795
  - datax horizontal screen axis 356
  - datay vertical screen axis 356
  - dataz fz 354
  - dataz perpendicular screen axis 356
  - date fz 120
  - dBase files 102

dbobjects fz 799, 804  
 dbread fz 804  
 dbremove fz 804  
 dbwrite fz 804  
 debugger fz 203  
 debugging 182  
 default class 407  
 defining attribute 44  
 density argument 266  
 density plot fz 351  
 deparse fz 765, 768  
 Designing ActiveX controls that support S-Plus 678  
 destination program 533  
 destroy. type. library fz 504  
 dev. off fz 334  
 Device. Default fz 288  
 dget fz 167  
 dialog callback 694  
     example 696  
 digits 265  
 digits argument 162, 268  
 digits option 162  
 dim attribute 42, 44, 51, 53, 55, 170  
 dim fz 45, 53, 56  
 dimnames argument 56  
 dimnames attribute 45, 843  
 dimnames fz 45, 53, 161, 172  
 Direct axis 306  
 disk activity 810  
 display properties 215  
 distributing functions 489  
 distributing the library 494, 740  
 do instruction 189  
 do.call fz 778  
 doc fz 171  
 dos function 552  
 DOS interface 547, 552  
 DOS interface, comprehensive examples 554  
 dot plot fz 347  
 dotplot fz 333, 341  
 double backslash 430  
 double numeric values 44  
 down instruction 186  
 dput fz 166, 167  
 draw fz 413

drop argument 134  
 dump fz 166  
 dump fz 489  
 dump.calls option 154  
 dump.frames fz 203  
 dump.frames option 154

## E

ed function 548  
 ed fz 167  
 edit function 548  
 editable graphics 223  
 efficient programming 820  
 elapsed time 810  
 enter instruction 191  
 environment variables 742, 745, 750  
     S\_CMDFILE 748  
     S\_CMDSAVE 749  
     S\_CWD 749  
     S\_DATA 749  
     S\_HOME 751  
     S\_NOAUDIT 751  
     S\_NOSYMLOOK 752  
     S\_PATH 752  
     S\_PREFS 752  
     S\_PROJ 752  
     S\_SCRSAVE 753  
     S\_SILENT\_STARTUP 753  
     S\_TMP 753  
 equal count algorithm 371  
 erase.screen fz 310  
 error argument 154  
 Error Bar Plots 228  
 error.action 198  
 ethanol data set 368, 376  
 eval fz 184, 192, 765, 771  
 eval instruction 196  
 Example functions, archive 554  
 Example functions, copy 554  
 Example functions, matrix.ed 557  
 Example functions, unarchive 555  
 Example functions, write.excel.matrix 556  
 Excel files 102

exclude argument 65  
 Execute fz 533  
 ExecuteStringResult 508  
 exp parameter 304  
 export.data fz 107  
 export.graph fz 111  
 exporting data 107  
 expression frame 788  
 expression fz 764, 771, 776, 841  
 Extended axes label 306  
 extraction functions 120  
 eye argument 284

## F

faces fz 281  
 factor class 64  
 factor fz 64  
 factor generator function 409  
 factors 62, 403  
 FASCII files 101  
 FASCII importing:specifying a format string 101  
 Fibonacci sequence function 822  
 fig argument 325  
 fig parameter 308  
 figure region 293  
 file expansion 744  
 files:importing 94  
 fill argument 109, 166  
 find instruction 185  
 First.lib fz 490, 734  
 fix function 548  
 fix fz 770  
 font parameter 377  
 For fz 827  
 for loops 138, 149, 827, 844  
 format fz 109, 162, 166  
 formula argument 335, 361, 374  
 Fortran code 762  
 fourplot fz 326  
 frame 1 788  
 frame 2 788  
 frame argument 806  
 frame fz 308

frames 788, 790, 811  
 fuel.frame data set 341, 353  
 FUN argument 84  
 func.entry.time fz 195  
 func.exit.time fz 195  
 FunctionInfo object 477, 484  
 functions 838  
 fundef fz 184

## G

gas data set 335  
 gauss data set 354  
 gaussfit fz 469  
 gaussfit1 fz 460  
 gaussfit2 fz 461  
 gaussfit3 fz 463  
 gaussian distribution 458  
 general 300  
 general display function 341  
 general display functions 333  
 get fz 174, 321, 792, 826  
 get.ole.property fz 522  
 GetObject fz 506  
 GetParameterClasses 510  
 GetSAPIObject 509  
 Getting Data from S-PLUS 540  
 glm fz 334  
 grain.size argument 828  
 Graph Measurements with Labels 358  
 Graph Multivariate Data 358  
 Graph sheets 211  
 graphical parameters 325  
 graphics 299  
 graphics objects 211  
 graphics parameters 287  
 graphs  
     2D, 3D, Polar, Matrix, and Text 211  
     Trellis 219  
 grid argument 471  
 group component 59  
 grouping 846  
 GUI objects 427

GUI toolkit 427  
    summary of functions 456  
gui Copy fz 431  
gui Create fz 428  
gui Create fz 213, 475  
gui DisplayDialog fz 444  
gui DisplayDialog fz 215  
guiExecuteBuiltIn function 455  
guiGetArgumentNames fz 440  
guiGetArgumentNames fz 216  
guiGetAxisLabelsName function 217, 452  
guiGetAxisName function 217, 453  
guiGetAxisTitleName function 217, 453  
guiGetClassNames fz 427, 438  
guiGetClassNames fz 223  
guiGetGraphName function 217, 453  
guiGetGSName function 217, 453  
guiGetOption function 451  
guiGetPlotClass function 453  
guiGetPropertyOptions function 442  
guiGetPropertyValue fz 441  
guiGetRowSelectionExpr function 449  
guiGetRowSelections function 449  
guiGetSelectionNames fz 448  
guiModify fz 432  
guiModify fz 475  
guiModifyDialog fz 446  
guiMove fz 433  
guiOpen fz 434  
guiOpenView fz 435  
guiPlot function 452  
guiPrintClass function 438  
guiRemove fz 435  
guiRemove fz 475, 477  
guiRemoveContents function 437  
guiSave fz 436  
guiSetOption function 451  
guiSetRowSelections function 449

## H

halibut data set 244  
help files  
    user written 487

help fz 493, 738  
help instruction 183  
high-level graphics functions 287  
High-Low-Close Plots 228  
hist fz 150, 152, 320  
hist fz 272  
histogram fz 341, 350  
Histograms and Densities 226  
History Log 216  
History log 208  
    recorded scripts 427  
How to Change the Rendering in the Data Region 378  
hstart time series 83  
html.table function 112

## I

I fz 337  
identify fz 259  
if statement 141  
ifelse statement 144, 817  
Im fz 122  
image fz 282  
ImageFileName argument 492  
immediate argument 808, 826  
implicit attributes 44  
import.data fz 96  
importing data 94  
importing:dBase files 102  
importing:Lotus files 102  
Index argument 476  
Informix 803  
inherits fz 129  
inspect fz 182, 183  
integer numeric values 44  
internally labeled axis 306  
interp fz 282  
interpolates 282  
intervals argument 372  
invisible fz 165  
iris data set 55, 58, 279  
is.matrix. fz 129  
is.na fz 127

is.null fz 127  
 is.ole.object.valid fz 523  
 is.vector fz 129

## J

jitter argument 344

## K

key argument 384, 387  
 kyphosis data frame 83, 802

## L

lab parameter 304  
 labels argument 65, 66  
 labex argument 283  
 lapply fz 139  
 lapply fz 821  
 last.dump object 154  
 last.lib fz 490, 734  
 layout algorithm 365  
 layout argument 362  
 left assignment 128  
 length argument 47, 48, 161  
 length attribute 42, 44, 60  
 length fz 46, 131  
 levelplot fz 355  
 levels argument 64, 65  
 levels attribute 62, 90  
 levels fz 371  
 lib.loc argument 490, 734  
 libraries 490, 734  
 library fz 490, 734  
 line types 255  
 Linear Curve Fit Plots 225  
 LineColor property 215  
 lines fz 261, 380  
 LineStyle property 215  
 LineWeight property 215  
 list data type 88  
 list fz 58, 152, 777

list mode 135  
 lists 58, 135  
   creating in loops 820  
 literals 837  
 lm fz 779  
 lm fz 260, 334  
 local argument 793  
 locator fz 411, 412  
 locator fz 263  
 loess fz 334  
 log fz 252  
 logical fz 49  
 logical values 44  
 loops  
   for, while and repeat loops 817  
 Lotus files 102  
 low-level graphics functions 287  
 low-level plotting functions 311  
 ltsreg fz 261  
 lty argument 255  
 lty parameter 380  
 lwd parameter 380

## M

mai parameter 295  
 main argument 250, 374  
 main title of a plot 249  
 main-effects ordering of levels 367  
 make.data.frames fz 184  
 make.groups fz 391  
 make.symbol fz 318  
 mar parameter 295  
 margin 293  
 mark instruction 185, 191, 192  
 Math.factor group method 415  
 matplot fz 246  
 matrices 43, 51, 53, 103, 133, 156  
 matrix data type 88  
 matrix fz 42, 52, 410  
 matrix multiplication 821  
 matrix.ed function 557  
 max fz 86  
 maximum likelihood estimates 460

mean fz 86, 152  
 memory allocation 811  
     arenas 811  
     buckets 811  
 memory fragmentation 814  
 memory requirements 814  
 menu fz 772  
 menuGaussfit fz 479  
 MenuItem objects 475  
 merge fz  
     by.x argument 80  
     by.y argument 80  
 merge fz 71, 79  
 method argument 462  
 mex parameter 295  
 mfcoll parameter 293  
 mfrow argument 325  
 mfrow parameter 248  
 mgp parameter 305  
 mileage.means vector 347  
 missing attribute 781  
 missing fz 142, 841  
 Mod fz 122  
 mode attribute 44, 67  
 mode fz 46  
 model.matrix data type 88  
 more argument 359  
 most useful graphics parameters 327  
 mtext fz 301  
 multiline argument 104, 158  
 multiple plots 251  
     on one graph 218  
 mypanel fz 378

## N

n argument 260  
 names 838  
 names attribute 44, 60, 67, 132  
 names fz 50, 53, 60, 772  
 namesattr attribute 50  
 nclass argument 151, 153, 272  
 ncol argument 52  
 New Dialog Controls In S-Plus 4.5 699

New Toolbar dialog 637  
 next statement 147  
 NextFlag 844  
 NextMethod fz 408, 842, 843  
 nint argument 350  
 nrow argument 52  
 numeric fz 49, 409  
 numeric numbers 837  
 numeric summaries 83  
 numeric values 44, 48

## O

ObjectContainees 507  
 ObjectContainer 507  
 object-oriented programming 23, 43, 403  
 objects fz 800  
 ole.reference.count fz 523  
 oma parameter 294  
 omd parameter 294  
 omi parameter 294  
 on.exit fz 168, 831  
 on.exit instruction 190  
 operators 118, 174  
 Ops.ordered group method 415  
 optional arguments 153  
 options fz 121, 154, 161  
 Oracle 803  
 ordered class 407  
 ordered fz 65  
 orientation of axis labels 304  
 outer margin 293  
 outlier data point 259  
 overlay figures 311  
 ozone data set 282

## P

p argument 382  
 page argument 395  
 page fz 161  
 pairs fz 278  
 panel argument 378, 395  
 Panel functions 333

panel functions 378  
 panel fz 378  
 panel variables 361  
 panel.loess fz 380  
 panel.special fz 379  
 panel.superpose fz 384, 386  
 panel.xyplot fz 378, 380, 381  
 par fz 248, 325  
 par.strip.text argument 377  
 parallel fz 353  
 parent frame 793  
 parse fz 764  
 paste function 548  
 paste fz 53, 179  
 PathName 507  
 pch argument 256, 378  
 pch parameter 380  
 pdf.graph argument 334  
 Pie Charts 226  
 pie fz 269  
 piechart fz 349  
 plot 248  
 plot area 293  
 plot argument 150, 153  
 plot fz 241, 323, 768  
 plot fz 459  
 plot method 465  
 plot types 253  
 plot.gaussfit fz 473  
 plot.line fz 383  
 plot.symbol parameter 383  
 plotting characters 255  
 points fz 261, 379, 380  
 points-type plot 257  
 Poke fz 533  
 polygon fz 380  
 position argument 359  
 postscript argument 334  
 postscript device 326  
 precedence of operators 119  
 preferences 757  
 prepanel argument 394  
 prepanel.loess fz 395  
 pretty fz 320  
 primes fz 200

print fz 63, 180, 403  
 print fz 459  
 print method 465  
 print.atomic fz 165  
 probability argument 150, 153  
 problems 178  
 process of freeing memory 818  
 prod fz 125  
 programmer time 810  
 Projection Planes 232  
 projects 757  
 prompt fz 139  
 prompt fz 487  
 prompt.screen fz 309  
 pscals argument 375  
 pty argument 248  
 pugetN data set 286  
 putting functions in the library 491, 736

## Q

qq fz 345  
 Q-Q Plots 225  
 qqline fz 275  
 qqmath fz 346  
 qqnorm fz 275  
 qqplots 274  
 qqunif fz 275  
 Quantile-Quantile plots 225

## R

RandomNumber fz 772  
 rbind fz 51, 71, 78, 79  
 Re fz 122  
 read.table fz 71, 105, 106, 156, 159  
 real arithmetic 122  
 real numbers 121  
 rebuild.type.library fz 504  
 Recall fz 842  
 rectangular plot shape 248  
 recursion  
     avoiding 822  
 register.all.ole.objects fz 503

- register.ole.object fz 503, 504
- Registering an ActiveX control 677
- remove fz 798, 800
- remove fz 821
- removing menu items and toolbars 477
- reorder.factor fz 367
- rep fz 47
- repeat loops 844
- repeat statement 146
- replacement functions 120
- Request fz 533
- resume instruction 190
- return expression 840
- return statement 147
- Return.flag 844
- Reuse Computations 821
- reusing code 822
- right assignment 128
- rm fz 800
- Rotating 3D Graphs 232
- round fz 125, 162
- rownames fz 173
- Rows fz 387

## S

- S\_CMDFILE variable 748
- S\_CMDSAVE variable 749
- S\_CWD variable 749
- S\_DATA variable 749
- S\_FIRST 750
- S\_FIRST variable 750
- S\_HOME variable 751
- S\_NOAUDIT variable 751
- S\_NOSYMLOOK variable 752
- S\_PATH variable 752
- S\_PREFS variable 752
- S\_PROJ variable 752
- S\_SCRSAVE variable 753
- S\_SILENT\_STARTUP variable 753
- S\_TMP environment variable 167
- S\_TMP variable 753
- SAMPLES/OCX 675
- sapply fz 821

- SAS 803
- save.x argument 470
- scales and labels of graphs 374
- scales argument 375
- scan fz 103, 105, 156, 157, 158
- Scatter Plot Matrices 228
- scatterplot 278
- Scoping rules 793, 805
- screen argument 356
- screen axes 356
- Script window 207
- scripts 36
  - function dialog example 426
  - running 427
- search command 761
- search fz 795
- segments fz 315, 380
- Sending Data to S-PLUS 539
- sep argument 180
- seq fz 48
- server 533
- server name 534
- session frame 470
- set.ole.property fz 522
- SetParameterClasses 510
- SetSAPIObject 509
- shingle fz 371
- show instruction 193
- show.settings fz 381, 382
- ShowDialogInParent 506
- ShowDialogInParentModeless 506
- side effects 165
- signif fz 162
- single backslash 430
- single numeric values 44
- single-symbol operators 336
- skip argument 395
- smooth fz 261
- solder data set 70
- source fz 489
- source program 533
- space argument 387
- span argument 380
- span parameter 390
- split argument 359



split.screen fz 309  
 splom fz 352  
 splus.exe executable 742  
 SPSS 803  
 square plot shape 248  
 standard arena 811  
 Standard axes 306  
 star plot 280  
 start-up and exit actions 493  
 static data visualization 278  
 StatLib 622  
 step instruction 188  
 steps in creating a library 490, 735  
 stop fz 142, 154  
 storage.summary fz 814  
 strings 838  
 strip argument 377  
 strip.names argument 377  
 strip.white argument 105, 159  
 stripplot fz 344  
 structure fz 167  
 sub argument 250, 374  
 subscripting 131  
 subscripts  
     efficiency considerations 816  
     logical 817  
 subscripts argument 380  
 subset argument 337  
 substitute fz 764, 766, 768, 771, 789, 841  
 subtitle of a plot 249  
 sum fz 125  
 summary fz 63  
 summary fz 459  
 summary method 465  
 Summary.data.frame.group method 415  
 superpose.symbol fz 385  
 Surface Plots 230  
 swap space 810

switch  
     COMPILE 755  
     GCOMPILE 755  
     GLOAD 755  
     HKEY\_CURRENT\_USER 756  
     LOAD 755  
     MULTIPLEINSTANCES 755  
     Q 755  
     REGISTEROLEOBJECTS 756  
     REGKEY 756  
     TRUNC\_AUDIT 756  
     UNREGISTEROLEOBJECTS 756  
 switch fz 143  
 switch statement 841  
 switches 754  
 switzerland data set 282  
 Symbol Color property 215  
 symbolic constants 838  
 symbols fz 316  
 Symbol Size property 215  
 Symbol Style property 215  
 synchronize fz 808  
 sys.call fz 769  
 sys.calls fz 840  
 sys.frames fz 791, 840  
 sys.parents fz 840  
 system 548

## T

t fz 108, 166  
 t fz 268  
 tapply fz 86  
 tck parameter 303  
 tempfile fz 167  
 Terminate fz 533  
 text fz 263, 380  
 the GUI function 480  
 the menu function 479  
 times argument 47  
 title fz 250, 289  
 ToolbarButton property dialog 641  
 toolbars and palettes  
     customizing 636

top-level frame 788  
 trace fz 204, 773  
 traceback fz 154, 179  
 track instruction 185, 194  
 Trellis graph 219  
 Trellis settings 381  
 trellis.device fz 333, 381  
 trellis.par.get fz 381  
 trellis.par.set fz 381, 383  
 trim argument 153  
 true.file.name fz 489, 493, 738  
 trunc fz 124  
 ts.lines fz 262  
 ts.plot fz 241  
 ts.points fz 262  
 type argument 242, 325, 385  
 type checking 461  
 Type factor 352

## U

uin argument 325  
 Unadvise fz 533  
 unmark instruction 193  
 unregister.all.ole.objects fz 503  
 unregister.ole.object fz 503, 504  
 up instruction 186  
 usa fz 317  
 UseMethod fz 406, 842  
 user preferences 757  
 using libraries 490  
 using logarithmic scale 252  
 using text files 489  
 usr parameter 298

## V

validate fz  
     coverage areas 851  
     examples 851  
     running a test 851  
 validate fz 847  
 validation test files 853  
 Validation Tests 848

vector data type 88  
 vector fz 48  
 vectors 42, 43, 47, 103, 131, 157, 403  
 virtual memory 810

## W

warning fz 154  
 Watcom C and Fortran 762  
 what argument 103, 105, 157, 158  
 where argument 806  
 Where can the PROGID for the control be found?  
 675  
 where instruction 188  
 while loop  
     replacement for recursion 823  
 while loops 844  
 while statement 148  
 Why only "OCX String"? 678  
 width argument 161, 274, 351  
 widths argument 105, 158  
 Windows applications, Calculator 548  
 Windows applications, Notepad 549  
 Windows interface 547, 548  
 Windows interface, comprehensive examples 554,  
 556  
 WinHelp 487  
 wireframe fz 333, 341, 356  
 write fz 108, 166, 167  
 Writing A Panel Function 378

## X

xaxis argument 252  
 xlab argument 150, 153, 251, 325, 374  
 xlim argument 251, 374  
 xplot fz 333, 335, 342

## Y

yaxis argument 252  
 ylab argument 251, 374  
 ylim argument 251, 374

---

**Z**

zseven class 416, 422

